# Bachelorarbeit

**Stephan Phieler**

## Entwicklung eines Linux-Kernel-Moduls zur Anbindung einer Mensch-Maschinen-Schnittstelle an Echtzeit-Ethernet basierende Netzwerke

*Fakultät Technik und Informatik*
*Studiendepartment Informatik*

*Faculty of Engineering and Computer Science*
*Department of Computer Science*

Stephan Phieler

# Entwicklung eines Linux-Kernel-Moduls zur Anbindung einer Mensch-Maschinen-Schnittstelle an Echtzeit-Ethernet basierende Netzwerke

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

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

Eingereicht am: 04. April 2013

**Stephan Phieler**

**Thema der Arbeit**

Entwicklung eines Linux-Kernel-Moduls zur Anbindung einer Mensch-Maschinen-Schnittstelle an Echtzeit-Ethernet basierende Netzwerke

**Stichworte**

Linux, Kernel, Treiber, Joystick, Echtzeit, Ethernet, TTEthernet, Mikrocontroller, RS-232

**Kurzzusammenfassung**

Ziel dieser Arbeit ist es, eine Verbindung zwischen einer, zu Demonstrationszwecken entwickelten, Steer-By-Wire-Anwendung, dem Demonstrator, und einem PC zu realisieren, um mit dieser, Anwendungen auf dem PC steuern zu können. Dazu werden der Aufbau des Demonstrators und das dort verwendete Time-Triggered-Ethernet-Protokoll betrachtet. Anschließend wird erklärt, wie der Demonstrator als Eingabegerät unter einem Linux-Betriebssystem genutzt werden kann. Nach der Anaylse der Anforderungen und der Erstellung eines Konzeptes, wird beschrieben, wie dieses Konzept umgesetzt wurde. Das Ergebnis der Arbeit, wird am Ende gegen die ermittelten Anforderungen geprüft.

**Stephan Phieler**

**Title of the paper**

Development of a Linux kernel module for interfacing a man-machine interface to real-time-ethernet-based networks

**Keywords**

Linux, Kernel, Driver, Joystick, Real-time, Ethernet, TTEthernet, Microcontroller, RS-232

**Abstract**

The aim of this work is to develop a link between a steer-by-wire-implementation, called Demonstrator, and a pc, to use this implementation for controlling applications on the pc. Therefore, the structure of the Demonstrator and the Time-Triggered-Ethernet-Protocol used there, will be considered. Afterwards it will be described, how to use the Demonstrator as an input device under a Linux operating system. After analysing the requirements and the creation of a concept, will be described, how the concept was implemented. The result of the work is proved against the requirements in the end.

# Inhaltsverzeichnis

# 1 Einleitung und Motivation

Elektronik ist aus dem heutigen Automobil nicht mehr wegzudenken. Sensoren, die im ganzen Auto verteilt sind, nehmen ständig Informationen auf und leiten diese über verschiedene Bussysteme weiter. Kameras an allen Seiten eines Autos, Filme die über eine Multimedia-Plattform gestreamt werden, Statusinformationen über die Umwelt oder neue Technologien wie X-by-Wire[1], treiben den Bedarf an Bandbreite und den sicherheitskritischen Datenverkehr in die Höhe. Für die Datenübertragung werden im Automobil verschiedene Bussysteme mit verschiedenen Aufgabenbereichen genutzt, was die Komplexität der Informationsübertragung zusätzlich erhöht. Dies wird, anhand des VW Phaetons, in Abbildung 1.1 veranschaulicht.

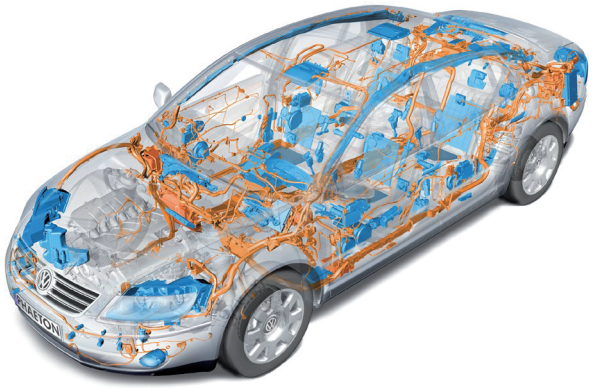

Abbildung 1.1: Bordnetz im VW Phaeton (Quelle: NYFEGA Elektro-Garage AG)

Hinzu kommt, dass die verwendeten Bussysteme, wie MOST, LIN, CAN oder FlexRay, nicht genug Bandbreite für die kommenden Entwicklungen zur Verfügung stellen. Ein weit verbreitetes Bussystem, welches eine hohe Bandbreite bietet, ist das Ethernet. Dieses wurde zu Demonstrationszwecken, in einer Konzeptstudie der BMW-Group, in einem Fahrzeug erfolgreich eingesetzt (vgl. Hammerschmidt, 2007) . Ethernet bietet die in Zukunft gebrauchte

[1]Beschreibt das Bedienen von Aktoren über elektrische Signale, die vorher durch mechanischen Verbindungen bedient wurden. Bekannte Beispiele sind das Steer-by-Wire und Brake-by-Wire.

Bandbreite, ist aber für zeitkritischen Datenverkehr nicht zu verwenden, da keine Vorraussagen über die erfolgreiche Datenübertragung gemacht werden können (vgl. Tanenbaum und Wetherall, 2011) . Eine Technologie die eine hohe Bandbreite und Echtzeitfähigkeit bietet, ist das Time-Triggeret-Ethernet-Protokoll (vgl. Steiner, 2008) , welches eine Erweiterung des Ethernet-Protokolls ist. Dieses wurde von der Uni Wien entwickelt und später von der Firma TTTEch weiterentwickelt.
Das Team von Communication over Real-time Ethernet (CoRE), mit dem diese Arbeit entsteht, befasst sich unter anderem mit der Übertragung von zeitkritischen Datenverkehr über das Ethernet. Der Fokus liegt dabei auf Anwendungen im Automotive-Bereich.
Wenn neue sicherheitskritische Technologien, wie das Real-Time-Ethernet (RTE), getestet werden sollen, können Simulatoren oder virtuelle Prototypen eingesetzt werden. Diese ermöglichen eine sicherere und kostengünstigere Entwicklung. Die traditionellen Vorgehensweisen von Versuch und Irrtum (try and error), werden damit durch realitätsnahe Computersimulationen ersetzt (vgl. Döbler, 2008) . Zusätzlich bietet ein Simulator die Möglichkeit, dem Entwickler und dem Kunden, eine Technologie zu visualisieren. Komplexe und abstrakte Daten können veranschaulicht und so besser verstanden und interpretiert werden. Um die Möglichkeiten von RTE im Automotive-Bereich zu zeigen, wurde vom CoRE-Projektteam ein Demonstrator einer Steer-By-Wire-Lösung entwickelt.
Ziel dieser Arbeit ist es, eine Verbindung zwischen dem Demonstrator und einem PC zu realisieren, um mit diesem, Anwendungen auf einem PC steuern zu können. Dazu muss analysiert werden, welche Schnittstellen der Demonstrator bietet, um mit einem PC kommunizieren zu können und welche Daten für die Steuerung benötigt werden. Aufbauend auf diesem Wissen, wird ein Treiber entwickelt werden, der die Schnittstelle zwischen Demonstrator und Anwendung bildet.

# 2 Grundlagen

In diesem Kapitel werden die Grundlagen vermittelt, die benötigt werden, um den Demonstrator um eine Schnittstelle zu erweitern, welche eine Anbindung an eine Computersimulation erlaubt. Als erstes wird erläutert was der Demonstrator ist und sein Aufbau beschrieben. Weiter wird erklärt was Echtzeit bedeutet und wie der Ethernet-Standard um diese Eigenschaft erweitert werden kann. Da später in dieser Arbeit ein Kommunikationsprotokoll auf Basis des seriellen Busses entwickelt und implementiert wird, folgt ein kurzer Überblick über serielle Kommunikation. Während der Anforderungsanalyse wurde sich auf Linux, als Betriebssystem für die Computersimulation, festgelegt. Als letztes wird daher auf die Themen Linux-Kernel und Linux-Treiberentwicklung eingegangen.

## 2.1 Der Demonstrator

Der von der CoRE-Projektgruppe entwickelte Demonstrator veranschaulicht eine Steer-By-Wire-Lösung auf Basis von Echtzeit-Ethernet. Er soll darstellen, wie ein Echtzeit-Ethernet-Netzwerk im Automobil eingesetzt werden kann und wie dies zur Vereinfachung der Komplexität beiträgt, die in heutigen Automobilen durch eine Vielzahl an unterschiedlichen Bussystemen entstehen kann. Dies wird dadurch erreicht, dass, wie in Abbildung 2.1 dargestellt, alle Teilnehmer nur noch über eine Netzwerkarchitektur kommunizieren, die alle benötigten Anforderungen erfüllt.
Die im Demonstrator verwendeten Komponenten kommunizieren größtenteils über das CAN- oder OpenCAN-Protokoll. Um diese Komponenten über ein Echtzeit-Ethernet zu vernetzen, müssen die Nachrichten, die diese Komponenten versenden und empfangen, in Echtzeit-Ethernet-konforme Nachrichten übersetzt werden. Dazu wurde innerhalb einer weiteren Bachelorarbeit eine RTE-CAN-Bridge entworfen und implementiert (vgl. Kamieth, 2011) . Diese Bridge ist auf einem Mikrocontroller realisiert worden und nutzt den TTE-Protokollstack für eingebettete Systeme, der in der Arbeit von Kai Müller entwickeltet wurde (vgl. Müller, 2011) .
Das Lenkrad und das Rad veranschaulichen das Steer-By-Wire, welches Lenken ohne mechanische Kopplung, rein über Steuersignale, erlaubt. Hierzu werden die Steuersignale als kritische

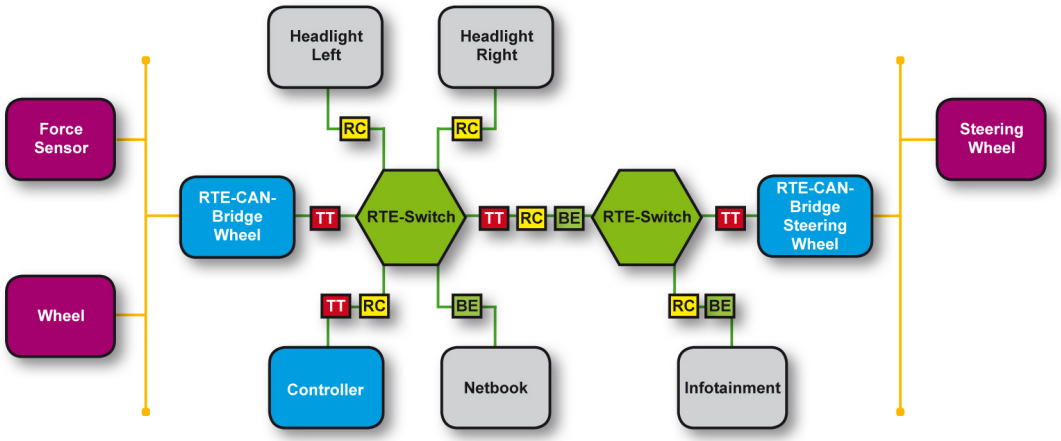


Abbildung 2.1: Netzwerkaufbau des Demonstrators

Daten über das Netzwerk verschickt. Dabei kann man das Rad mit dem Lenkrad steuern, oder Effekte die auf das Rad wirken, an das Lenkrad weitergeben. Dank des softwarebasierten Ansatzes, können das Lenkverhalten oder die Stärke des Force-Feedback[1] vom Rad zum Lenkrad beliebig eingestellt werden. Eine Infotainmentanlage demonstriert durch zwei Videostreams, die viel Bandbreite beanspruchen, dass der kritische Datenverkehr, zwischen Lenkrad und Rad, durch den unkritischen Datenverkehr der Infotainmentanlage, nicht beeinflusst wird. Über sie kann man auch die Scheinwerfer steuern und das Lenkverhalten sowie die Stärke des Force-Feedback einstellen. Weiterhin kann demonstriert werden, was passiert falls kritische Nachrichten nicht rechtzeitig eintreffen.
Ein zentraler Controller übernimmt die Steuer- und Regelung der Komponenten. Er initialisiert alle Komponenten, trägt Informationen über das System zusammen und hält den globalen Zustand des Systems (vgl. Stepanov, 2011) .
Die in Abbildung 2.1 dargestellten Nachrichtenklassen TT, RC und BE, werden auf Seite 6 näher erklärt.

## 2.2 Echtzeit

In Informationssystemen versteht man unter Echtzeit, die zeitlich garantierte Reaktion eines Systems auf ein bestimmtes vorhergehendes Ereignis. Tritt die Reaktion auf ein Ereignis zu spät ein, kann dies, je nach Anforderung, von unkritischen Informationsverlust bis hin zu kritischen

[1] Bezeichnet die Rückkopplung der Kraft an einen Benutzer. Es wird oft dort eingesetzt, wo es keine mechanische Verbindung gibt, die die Kraft an den Benutzer weitergeben kann.

Systemschäden führen. So unterliegen zum Beispiel in einem Automobil der Airbag und die Entertainmentanlage den Echtzeiteigenschaften, allerdings mit eindeutig unterschiedlichen zeitlichen Grenzen (Deadlines). Sollte beim Entertainmentsystem ein Bild eines Video-Streams nicht rechtzeitig ankommen, kann das zu rucklern im Bild und Ton führen, was das System aber nicht in einen kritischen Zustand überführt. Anders ist es, falls die Information *Airbag auslösen* zu spät eintrifft, dies kann den Insassen eines Fahrzeuges, im Falle eines Unfalls, tödliche Verletzungen zufügen. Anhand des vorhergenden Beispiels kann man erkennen, dass es verschiedene Anforderungen an Echzeitsysteme gibt. Man unterscheidet diese an der Schwere der Auswirkung einer verletzten Deadline.

### 2.2.1 Weiche Echtzeitanforderungen

In Systemen mit weichen Echtzeitanforderungen sind die Deadlines als Richtlinienen zu betrachten. Das Überschreiten einer Deadline überführt das System nicht in einen kritischen Zustand. Die Informationen werden aber als ungültig angesehen.

### 2.2.2 Harte Echtzeitanforderungen

Wird eine Deadline in einem System mit harten Echtzeitanforderungen überschritten, hat dies kritische Auswirkungen auf das System. In Systemen die dieser Eigenschaft unterliegen, wird garantiert, dass die Reaktionszeit innerhalb der zeitlichen Grenzen liegt.

## 2.3 Time-Triggered-Ethernet

Ethernet ist eine sehr verbreitete Technologie. Nahezu jeder Haushalt, der einen Zugang zum Internet hat, nutzt Ethernet als Übertragungsmedium. Durch die hohe Verfügbarkeit und den günstigen Anschaffungspreisen für Hardware sowie der hohen Bandbreite (bis 10 Gbit/s (vgl. Tanenbaum und Wetherall, 2011) ), ist Ethernet attraktiv für den Einsatz in lokalen Netzwerken. Es ist allerdings nicht für den Einsatz in Echtzeitsystemen ausgelegt.
Time-Triggered-Ethernet (TTE) stellt eine Echtzeiterweiterung des Ethernet-Standards 802.3 dar. Es vereint verschiedenste Anwendungsgebiete, vom einfachen Mediastream, der weichen Echtzeiteigenschaften unterliegt, bis hin zu X-By-Wire-Anwendungen, die harten Echtzeiteigenschaften unterliegen. Es bietet sich demnach auch für einen Einsatz im Automotive-Bereich an.

### 2.3.1 Nachrichtenklassen

Ein Problem von Ethernet, in Bezug auf die Einhaltung von Zeitgrenzen, sind mögliche Kollisionen von Nachrichten. Tritt auf einem Bus eine Kollision auf, hat der Sender die Möglichkeit, diese durch Abhören des Kanals zu erkennen und gegebenenfalls darauf zu reagieren. In einem Switched-Ethernet würden beide Nachrichten gespeichert und dann nacheinander an ihre Empfänger verschickt werden. Bei beiden Verfahren tritt eine unberechenbare zeitliche Verzögerung in der Übertragung auf. Harte Echtzeitanforderungen können somit nicht erfüllt werden.
Um dieses Problem zu lösen, ist es nötig Nachrichten zu priorisieren und sie damit in Klassen einteilen zu können, so dass bei einer möglichen Kollision hochpriore Nachrichten den Vorrang erhalten und rechtzeitig ankommen. Im TTEthernet sind dazu drei verschiedene Nachrichtenklassen spezifiziert.

#### Time-Triggered-Nachrichten

Diese Klasse wird für zeitkritischen Datenverkehr genutzt. Die Nachrichten unterliegen festen Sende- und Empfangszeiten und besitzen die höchste Priorität. Somit können leicht Vorhersagen über Sende und Empfangszeit gemacht werden.

#### Rate-Constrained-Nachrichten

Nachrichten dieser Klasse unterliegen keinen festen Sendezeiten. Es wird der nutzenden Anwendung eine feste Bandbreite garantiert und man kann den Nachrichten unterschiedliche Prioritäten vergeben. Allerdings haben RC-Nachrichten immer eine geringere Priorität als TT-Nachrichten und können von diesen unterbrochen oder verdrängt werden.

#### Best-Effort-Nachrichten

Diese Klasse entspricht den normalen Ethernet-Nachrichten. Es kann nicht gewährleistet werden, ob und wann eine Nachricht versendet wird, oder ankommt. TT- und RC-Nachrichten haben immer Vorrang vor BE-Nachrichten.

### 2.3.2 Nachrichtenaufbau

Wie eingangs erwähnt, baut TTEthernet auf dem Ethernet-Standard 802.3, genauer Switched-Ethernet, auf. Das bedeutet, dass nur der Standard-Ethernet-Frame ohne Erweiterungen zur Verfügung steht, der in Abbildung 2.2 zu sehen ist.

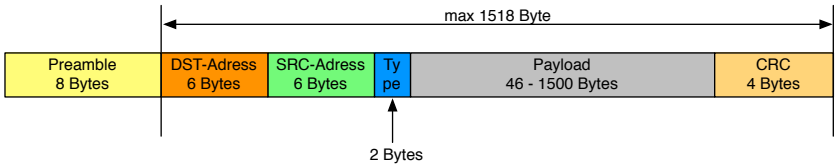


Abbildung 2.2: Standard-Ethernet-Frame (Quelle: Bartols, 2010)

Das bedeutet, dass es kein freies Feld gibt, welches zur Kennzeichnung der Priorität verwendet werden kann. Dafür nutzt man im TTEthernet das 48 Bit Zieladressfeld, wie in Abbildung 2.3 dargestellt.

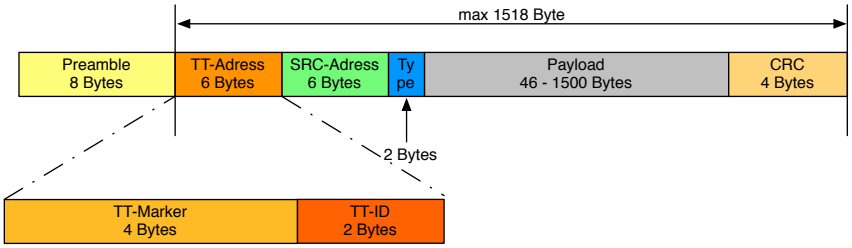


Abbildung 2.3: TTEthernet-Frame (Quelle: Bartols, 2010)

Dieses wird in zwei Bereiche aufgeteilt. Die ersten 32 Bit (der TT-Marker) kennzeichnen die Nachricht als zeitkritische Nachricht. Diese Markierung muss im gesamten Netzwerk identisch und jedem Teilnehmer bekannt sein. Die restlichen 16 Bit (die TT-ID) weisen die Nachricht als TT- oder RC-Nachricht aus.
Eine Unterscheidung zwischen einer zeitkritischen und -unkritischen Nachricht sieht der Ethernet-Standard im Typfeld vor. Hier wurde für zeitkritische Nachrichten der Typ 0x88D7 spezifiziert.

### 2.3.3 Konfiguration

Eine Besonderheit des Echtzeit-Ethernet-Netzwerkes ist seine statische Konfiguration. Vor Inbetriebnahme des Netzwerkes muss bekannt sein, wann und an welche Teilnehmer TT-Nachrichten gesendet werden. Die festen Sende- und Empfangszeiten von TT-Nachrichten setzen zu dem vorraus, dass Sie sich zyklisch wiederholen müssen. Die restliche Bandbreite steht RC- und BE-Nachrichten, die asynchron auftreten können, zur Verfügung. So kann gewährleistet werden, dass genug Bandbreite für kritischen Datenverkehr zur Verfügung steht

und dieser auch rechtzeitig verschickt wird. Das bedeutet aber auch, falls keine kritischen Daten zum senden bereit stehen, Bandbreite ungenutzt bleibt, da die Sendepunkte fest reserviert sind. TT- und RC-Nachrichten besitzen als Zieladresse keine IP-Adresse, sondern, wie zuvor beschrieben, eine Nachrichten-ID. Daher werden in der Konfiguration, statische Netzwerkrouten eingerichtet. Wie bei einem Multicast, sendet ein Teilnehmer seine Nachricht in das Netzwerk an registrierte Teilnehmer. Der Switch weiß anhand der statischen Netzwerkrouten, an welche Teilnehmer die Nachricht gesendet werden soll. Es darf aber nur eine Nachricht, mit der gleichen ID, von einem Teilnehmer kommen. BE-Nachrichten besitzen dagegen eine Zieladresse und werden vom Switch wie normale Ethernet-Nachrichten an die Empfänger weitergeleitet.
Ein Scheduler sorgt dafür, dass TT-Nachrichten rechtzeitig über die konfigurierten Ports verschickt werden. Dazu kann er anstehende RC- oder BE-Nachrichten verzögern oder unterbrechen. Anschließend werden RC- und zum Schluss BE-Nachrichten abgearbeitet. Haben zwei RC-Nachrichten die gleiche Priorität oder stehen mehrere BE-Nachrichten an, werden diese nach dem FIFO-Prinzip abgearbeitet.

### 2.3.4 Synchronisation

Da der Synchronisationsprozess sehr komplex ist, wird hier nur auf die wichtigsten, für diese Arbeit relevanten, Aspekte eingegangen.
Um eine globale Zeit im Netzwerk zu realisieren, muss es einen Mechanismus geben, der eine globale Zeit bildet, auf die sich alle Teilnehmer im Netzwerk synchronisieren können. Nur dann sind feste Sende- und Empfangszeiten realisierbar. Um diesen Synchronisationsprozess zu realisieren, werden alle Komponenten in ein oder mehrere Synchronisationsklassen, mit speziellen Aufgaben, eingeordnet. Als Nachrichtenklasse für die Synchronisationsnachrichten werden Protocol-Contorol-Frames verwendet. Diese basieren auf dem Standard-Ethernet-Frame mit minimaler Länge (vgl. Steiner, 2008, Seite 19) . Er besitzt die höchste Priorität, lässt sich aber nicht als TT-Nachricht deklarieren, denn im unsynchronisierten Zustand, würden die Nachrichten sonst abgewiesen werden. Somit enspricht er eher einer RC-Nachricht. Wie Abbildung 2.4 veranschaulicht, lässt sich der Synchronisationsprozess in zwei Schritte einteilen, die zyklisch wiederholt werden.
In Schritt 1 senden die als $synchronization\ master$ festgelegten Teilnehmer ihre lokale Zeit an den $compression\ master$. Dieser bildet aus den lokalen Zeiten eine globale Zeit und sendet diese, in Schritt 2, an alle $synchronization\ clients$. Es werden also drei verschiedene Synchronisationsklassen unterschieden.

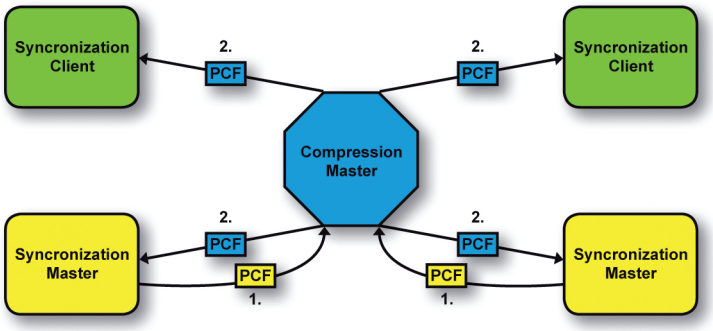


Abbildung 2.4: Synchronisationsprozess

### Synchronization client

Alle Teilnehmer des Netzwerkes sind *synchronization clients*. Ihre einzige Aufgabe ist es, die zyklisch vom *compression master* gesendeten Zeiten, mit ihrer internen Uhr, abzugleichen und diese gegebenenfalls anzupassen.

### Synchronization master

*synchronization master* werden während der Konfiguration des Netzwerkes bestimmt. Sie senden einmal pro Zyklus ihre eigene interne Zeit an den *compression master*.

### Compression master

Der *compression master* verarbeitet die zyklisch eintreffenden Zeiten der *synchronization master* zu einer globalen Systemzeit und schickt diese dann an alle *synchronization clients*. Dabei ist es üblich, dass ein Switch die Aufgabe des *compression master* übernimmt.

## 2.4 Serielle Kommunikation RS-232

Die RS-232-Schnittstelle dient zum Datenaustausch zwischen zwei Geräten. Sie besitzt jeweils eine Datenleitung zum Empfangen (RX) und eine zum Senden (TX) von Daten. (vgl. sprut, 2012) . Die Daten werden dabei asynchron, also ohne eine gemeinsame Taktleitung, übertragen. Somit muss auf beiden Seiten die gleiche Übertragungsgeschwindigkeit eingestellt sein und es muss eine Synchronisation stattfinden, damit der Empfänger weiß wann die nächste Übertragung beginnt. Zum Synchronisieren wird vor den eigentlichen Datenbits ein Startbit angestellt. Danach folgen maximal 8 Datenbits (vgl. BURKHARD, 1994) . Am Ende wird mindestens ein Stoppbit angehangen, welches die minimale Pause zwischen zwei Nachrichten angibt.

Hier werden üblicherweise 1, 1,5 oder 2 Bit genutzt. Theoretisch kann man aber beliebig viele Stoppbits verwenden. Um mögliche Übertragungsfehler aufzudecken, kann zwischen dem letzen Datenbit und dem ersten Stopbit ein Paritätsbit eingefügt werden. Das Paritätsverfahren ist nicht im EIA-232-Standard vorgesehen, wird aber von Linux unterstützt. Die Daten werden Bit für Bit, nacheinander übertragen, wobei die Symbolrate genau 1 Bit beträgt. Das bedeutet, dass die Bitrate gleich der Baudrate ist. Den kompletten Aufbau kann man in Abbildung 2.5 sehen.

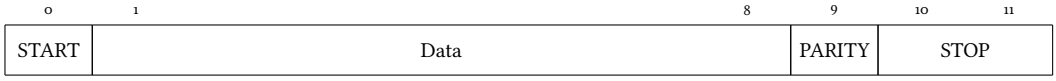


Abbildung 2.5: Möglicher Aufbau einer seriellen Nachricht

## 2.5 Linux - Kernel und Gerätetreiber

In der Anforderungsanalyse wird später festgelegt, welches Betriebssystem auf dem PC laufen wird. Da die Wahl auf Linux gefallen ist, wird hier kurz darauf eingegangen was ein Kernel ist und wo sich der Linuxkernel einordnen lässt. Anschließend wird erläutert, wie Gerätetreiber in Linux umgesetzt sind und was deren Aufgabe ist.

### 2.5.1 Kernel

Der Kernel oder Kern eines Betriebssystems hat die Aufgabe, die Kommunikation mit der Hardware eines System zu übernehmen und der über ihm liegenden Anwendungsschicht, eine API auf diese zur Verfügung zu stellen. Er bildet somit, als unterste Softwareschicht, eine Abstraktionsschicht zwischen Hardware und Applikationssoftware.
Das hat unter anderem den Vorteil, dass Software, unabhänging von der darunter liegenden Hardware erstellt werden kann. Hauptbestandteile eines Kernels sind die Kommunikation mit der Hardware, Funktionen zur Speicher- und Prozessverwaltung sowie das Dateisystemmanagment. Man unterscheidet zwischen zwei Arten von Kerneln, dem monolitischen Kernel und dem Mikrokernel.

#### monolitischer Kernel

Ein strikt monolitischer Kernel hat alle Funktionalität, die zusätzlich zu den Hauptfunktionen gebraucht werden, schon integriert. Das heißt, dass alle Treiber ständig im Speicher vorgehalten werden, was Resourcen kostet, aber auch einen Geschwindigkeitsvorteil bedeuten kann. Falls im

Fehlerfall ein Teil des Kernels ausfallen sollte, kann das allerdings zum Absturz des kompletten Kernels führen, da die Treiber nicht in vom Kernel getrennten Prozessen bzw. Speicher laufen und zur Laufzeit nicht entladen werden können.

#### Mikrokernel

Diese Art von Kernel besteht nur aus den Hauptfunktionen. Weitere Funktionalität kann bei Bedarf, in Form von eigenständigen Modulen, gekapselt in eigene Prozesse, nachgeladen werden. Die Vor- und Nachteile kehren sich dabei, in Bezug zum monolitischen Kernel, um. Man schont die Resourcen, in dem man nur Module lädt die auch gebraucht werden. Dies kann jedoch einen Performanceverlust bedeuten, da jedes Modul als eigener Prozess behandelt wird. Diese Behandlung hat aber auch den Vorteil, dass abgestürzte Module, da Sie vom Rest des Kernels getrennt sind, nicht den kompletten Kernel abstürzen lassen und ggf. zur Laufzeit entfernt werden können.

#### Linux-Kernel

Der Linux-Kernel, für den in dieser Bachelor-Arbeit ein Treiber entwickelt werden soll, hat einen modular monolitischen Ansatz. Er hat zusätzlich zu den Hauptfunktionen noch weitere, aus Sicht der Entwickler, wichtige Funktionen einkompiliert. Dazu gehören unter anderem Funktionen zur seriellen Kommunikation. Es können ihm, wie dem Mikrokernel, zur Laufzeit weitere Module hinzugefügt werden.
Das Linuxbetriebssystem teilt sich grob in zwei Bereiche: dem Userspace und dem Kernelspace. Als Userspace wird der Speicherbereich bezeichnet der von Benutzerprozessen verwendet werden kann. Der Kernelspace ist der Speicherbereich der dem Betriebssystemkern zur Verfügung steht. Beide Bereiche haben somit einen eigenen Adressraum und sind auch sonst strikt voneinander getrennt. Trotzdem müssen User- und Kernelspace miteinander kommunizieren können, damit Userspace-Prozesse, unter anderem, auf Hardware zugreifen können. Dafür gibt es das in Abbildung 2.6 veranschaulichte $ystem\ call\ interface$, welches dem Userspace standardisierte Funktionen bietet, um Daten mit dem Kernelspace auszutauschen (vgl. Quade und Kunst, 2011) .
Wenn nachfolgend vom Kernel gesprochen wird, ist damit der Linux-Kernel gemeint.

### 2.5.2 Gerätetreiber

Treiber stellen eine Abstraktion einer bestimmten Hardware auf wohldefinierte Funktionen dar. Der Treiber selbst hat immer eine spezielle Funktionalität und kann dem Kernel zur Laufzeit

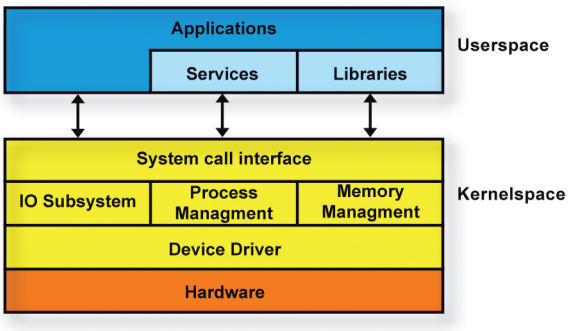


Abbildung 2.6: Kernel und Userspace (Quelle: Schütte, 2012)

(als Modul) oder zur Kompilierzeit (als build-in) hinzugefügt werden. Gerätetreiber werden in verschiedenen Kategorien klassifiziert (vgl. Tanenbaum, 2009) .
In Linux unterscheidet man hauptsächlich zwischen blockorientierten und zeichenorientierten Geräten. Darüber hinaus gibt es noch weitere Arten, wie USB, Netzwerk oder IrDA-Geräte.

### Zeichenorientierte Geräte

Auf ein zeichenorientiertes Gerät wird wie auf einen Stream von Daten zugegriffen. Der Aufruf gleicht dem eines Dateiaufrufes, nur dass man die Position des Datenzeigers (wahlfreier Zugriff) nicht beliebig verändern kann. Ein Treiber der ein solches Gerät unterstützt, muss also einen Strom von Daten verarbeiten können. Nach außen wird über Funktionen wie $open$, $close$, $read$ und $write$ auf das Gerät zugegriffen. Alle verwendbaren Geräte werden als Geräteknoten im Dateiverzeichnis abgebildet. Zu zeichenorientierten Geräten gehört unter anderem die RS-232-Schnittstelle, die unter /dev/ttySX abgebildet wird.

### Blockorientierte Geräte

Geräte die blockorientiert arbeiten, unterscheiden sich nach außen kaum von zeichenorientierten Geräten. Der einzige Unterschied liegt im wahlfreien Zugriff auf die Daten. Während ein Stream von Anfang bis Ende gelesen werden muss, ist es hier mögluch auf einen bestimmten Block an Daten zuzugreifen. Dazu nutzt man dazu die gleichen Funktionen, die auch bei zeichenorientierten Geräten benutzt werden. Auch sie werden im Dateisystem als Geräteknoten abgelegt. Intern liest der Treiber die Daten nicht mehr hintereinander ein, sondern greift über Adressen auf Blöcke von Daten zu. Blockgrößen sind immer Zweierpotenzen, zum Beispiel 8192 Byte. Eines der bekanntesten Blockgeräte ist die Festplatte, welche in Linux unter /dev/sdX zu finden ist.

#### Gerätetreiber für virtuelle Geräte

Zu einem Gerätetreiber muss nicht zwangsweise ein reales Gerät existieren. In Linux sind Gerätetreiber für solche virtuellen Geräte bereits implementiert. Zwei bekannte Treiber sind /dev/zero, welches einen Ausgabestrom erzeugt der nur aus Nullen besteht, und /dev/random, welches Zufallszahlen produziert.

### 2.5.3 Low- und High-Level-Treiber

Man kann Treiber zusätzlich noch zwischen Low- und High-Level-Treibern unterscheiden. Ein Low-Level-Treiber kümmert sich um die Kommunikation mit einer bestimmten Hardwareschnittstelle, wie zum Beispiel einem Controller für den seriellen Bus. High-Level-Treiber werden dagegen für ein bestimmtes Gerät geschrieben, welches Beispielsweise über den seriellen Bus angeschlossen wird. Er implementiert die Protokolle zur Kommunikation mit dem jeweiligen angeschlossenen Gerät. Die zwei Schichten werden, wie in Abbildung 2.7 veranschaulicht, durch eine dazwischenliegende Core-Schicht miteinander verbunden, die Schnittstellen für beide Seiten zur Verfügung stellt und so die modularität erhöht.

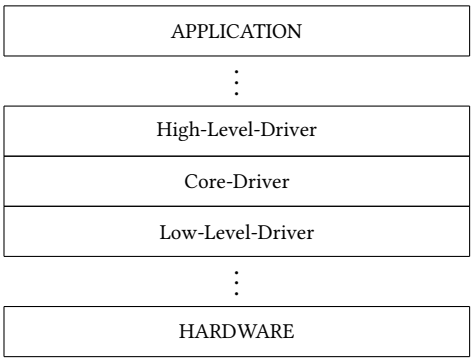


Abbildung 2.7: Schichttreiber

Sie übernimmt unter anderem die Zuweisung von Geräten zu passenden Treibern. Der Vorteil solcher Treiber-Stacks ist, dass es mehrere High-Level-Treiber geben kann, die einen Low-Level-Treiber nutzen. Zudem nimmt man dem Entwickler ab sich, wenn ein Treiber für ein Gerät geschrieben werden soll, welches über eine bereits implemetierte Schnittstelle angeschlossen wird, selbst um die Hardwareansteuerung kümmern zu müssen.

# 3 Anforderungen und Konzept

Ziel der Arbeit ist es, das Lenkrad des Demonstrators mit einem PC zu koppeln und es dort als Eingabegerät zu nutzen.
Der Demonstrator stellt zur Kopplung des Lenkrades mit einem PC verschiedene Möglichkeiten bereit. Die naheliegendste ist die Integration eines neuen Netzwerkteilnehmers in das RTE-Netzwerk, direkt über einen Switch. Man könnte den PC aber auch über eine weitere RTE-CAN-Bridge in das Netzwerk integrieren oder sich an eine bestehende RTE-CAN-Bridge anschließen und die benötigten Daten dort abgreifen. Dazu muss analysiert werden, welche Informationen man von welchem Teilnehmer benötigt und wie man an diese Informationen kommt. In der Anforderungsanalyse wird auch geklärt, welches Betriebssystem zur Umsetzung der Arbeit verwendet werden kann. Es werden nachfolgend mehrere Vorgehensweisen sowie alternative Anbindungen zwischen Demonstrator und PC betrachtet, deren Vor- und Nachteile verglichen und die Komplexität der Umsetzung analysiert.

## 3.1 Anbindungsmöglichkeiten

Der idealste Weg für die Anbindung, ist die direkte Einbindung des PCs in das Netzwerk, über einen Switch. Die Switches, die im Demonstrator eingesetzt werden, kommen von der Firma TTTech, die, wie schon erwähnt, einen eigenen RTE-Standard entwickelt haben (vgl. Steiner, 2008) . Das komplette Netzwerk ist auf dem TTE-Standard aufgebaut und unterscheidet sich daher von anderen, in Industrieanlagen eingesetzten, RTE's. Der PC muss also über eine Netzwerkkarte verfügen, zu der es einen Treiber gibt, der diesen Standard unterstützt. Zur Zeit gibt es nur einen funktionierenden TTE-Protokollstack von der Firma TTTech für das Linux-Betriebssystem. Dieser ist für den Linux-Kernel 2.6.24 entwickelt worden. Parallel zur Recherche dieser Arbeit, entwickelt die CoRE-Projektgruppe einen eigenen TTE-Protokollstack für den neueren Linux-Kernel 3.X. Voraussetung für den Einsatz eines der TTE-Protokollstacks ist ein Kernel mit Echtzeiterweiterung. Für Windows gibt es derzeit keinen TTE-Protokollstack der den RTE-Standard von TTTech unterstützt.
Sollte die direkte Einbindung in das Netzwerk nicht umgesetzt werden können, muss eine alternative Lösung zur Umsetzung des Ziels gefunden werden. Der Demonstrator bietet, über

die angesprochene RTE-CAN-Bridge, eine weitere Anbindungsmöglichkeit. Sie wird derzeit nur zum tunneln der CAN-Nachrichten, über das RTE-Netzwerk, eingesetzt. Man kann die Bridge so erweitern, dass sie die CAN-Nachrichten nicht nur über das RTE-Netzwerk, sondern auch über eine weitere Kommunikationsschnittstelle, an den PC tunnelt. Als einzige freie Schnittstelle steht dafür nur die RS-232-Schnittstelle zur Verfügung. Da das RTE-Netzwerk eine viel höhere Bandbreite bietet als die RS-232-Schnittstelle, muss geprüft werden, ob die RS-232-Schnittstelle ausreichend Bandbreite bietet, um die anfallenden Daten rechtzeitig übertragen zu können. Daher muss analysiert werden, welche Daten überhaupt an den PC übertragen werden müssen und welche Date der PC sendet.
Für beide Varianten der Anbindung wurde ein Konzept entwickelt. Dies war nötig, da nicht abzusehen war, ob es für die Umsetzung der direkten Anbindung einen funktionsfähigen TTE-Protokollstack, für den Linux-Kernel 3.X, gibt.

## 3.2 Vorstellung der Konzepte

Es haben sich durch die Analyse der Anforderungen zwei Konzepte ergeben. Zum einen die direkte Integration über die Ethernet-Schnittstelle des PC's und zum anderen die indirekte Integration über die serielle Schnittstelle an die RTE-CAN-Bridge des Lenkrades. Nachfolgend werden die zwei Konzepte näher erläutert und deren Vor- und Nachteile betrachtet.

### 3.2.1 Direkte Einbindung in das Netzwerk

Für dieses Konzept wird davon ausgegangen, dass es einen funktionierenden TTE-Protokollstack, für den Linux-Kernel 3.X, gibt. Es wird zu erst festgestellt, welche Funktionen der TTE-Protokollstack, zur Kommunikation mit einem anderen Treiber, zur Verfügung stellt. Anschließend wird analysiert, über welche Nachrichten die relevanten Informationen zum Lenken und über den Force-Feedback-Effekt, verschickt werden. Danach wird das TTE-Netzwerk des Demonstrators betrachtet und analysiert, wie dieses für einen neuen Teilnehmer angepasst werden muss. Zum Schluss wird festgestellt, welchen Anforderungen der Joystick-Treiber unterliegt.

#### TTE-Protokollstack

Die nachfolgenden Angaben, sind für die beiden zuvor beschriebenen TTE-Protokollstacks gleich. Der TTE-Protokollstack wird, genau wie das restliche Netzwerk, statisch konfiguriert. Das bedeutet, dass vor dessen Inbetriebnahme bekannt sein muss, wie die Zykluszeit des

Netzwerkes eingestellt ist, welche Nachrichten er verarbeiten soll, wann diese eintreffen, bzw. wann er eigene Nachrichten versenden muss.
Für jede TT- und RC-Nachricht wird ein eigener Buffer angelegt, und es werden die festen Empfangs- und Sendezeiten der TT-Nachrichten eingestellt. Auf alle Buffer kann über standardisierte API-Funktionen zugegriffen werden. Zusätzlich ist es möglich, auf die Input-Buffer eigene Callback-Funktionen anzumelden.
Möchte man eine Nachricht verschicken, schreibt man die Daten, verpackt in einer definierten Struktur, in den passenden Output-Buffer, für die jeweilige Nachricht. Der Scheduler des TTE-Protokollstacks, übernimmt dann das zeitlich abgestimmte Versenden der Nachrichten. Zum Empfangen einer Nachricht muss man, wie erwähnt, eine Callback-Function beim jeweiligen Input-Buffer anmelden. Diese wird aufgerufen, sobald eine neue Nachricht eingetroffen ist.
Voraussetzung für den Betrieb des TTE-Protokollstacks unter Linux, ist die Echtzeiterweiterung für den Linux-Kernel. Nur so kann gewährleistet werden, dass der Scheduler des TTE-Protokollstacks, seine Nachrichten punktgenau verschicken kann. Durch die Erweiterung wird der Kernel vollständig präemptiv. Somit können auch laufende ISR, die zum Beispiel genau dann ausgeführt werden wenn eine Nachricht verschickt werden soll, durch die ISR des TTE-Protokollstacks verdrängt werden. Zusätzlich kann Prozessen somit eine so hohe Priorität verliehen werden, dass damit der Ansatz des „Completely Fair Scheduler“ ausgehebelt wird (vgl. Klinger, 2008) . Für Windows gibt es derzeit keinen TTE-Protokollstack.
Der TTE-Protokollstack der Firma TTTech steht für verschiedene Chipsätze zur Verfügung. Dazu zählen, der e1000e Chipsatz der Firma Intel, der ar8132 Chipsatz der Firma attansic und der r8169 Chipsatz der Firma Realtek. Alle drei stehen ausschließlich für den Linux-Kernel 2.6.24 zur Verfügung. In Abbildung 3.1 ist veranschaulicht, wie der TTE-Protokollstack den Protokollstack der Netzwerkkarte erweitert.

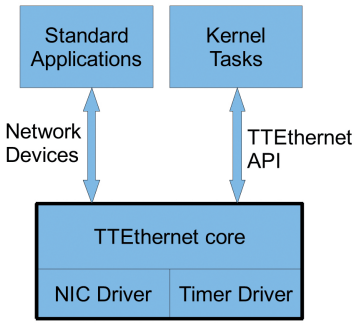


Abbildung 3.1: Das Modell der TTE-Protokollschicht (Quelle: TTTech Computertechnik AG, 2010)

### Nachrichtenanalyse

Die Einbindung eines neuen Netzwerkteilnehmers zieht auch eine Erweiterung der Netzwerkkonfiguration nach sich. Um festzustellen welche Nachrichten an den PC und welche vom PC an andere Teilnehmer gesendet werden sollen, muss bekannt sein, welche Daten relevant für das Lenken und den Force-Feedback-Effekt sind. Die Informationen über das Lenken, den Fahrmodus und den Force-Feedback-Effekt vom Rad, werden mit TT-Nachrichten übertragen. In Abbildung 3.2 ist der Verlauf der Nachrichten abgebildet.

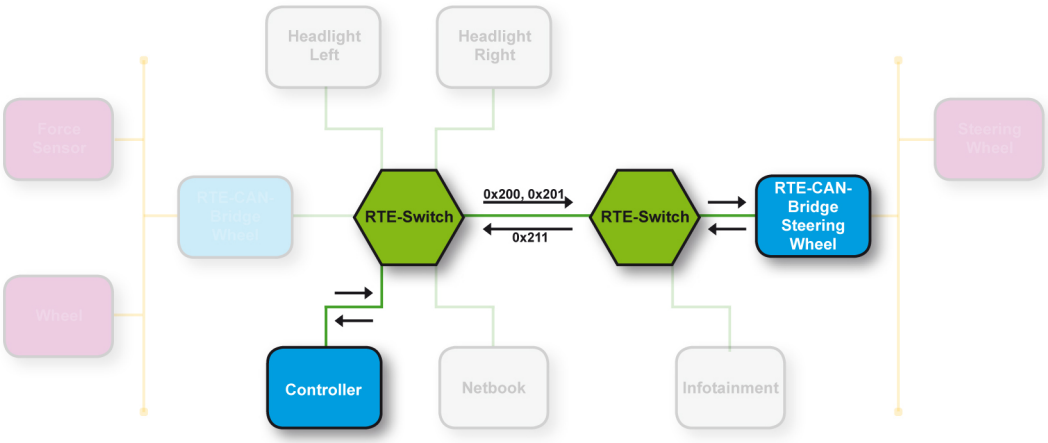


Abbildung 3.2: Aktueller Nachrichtenverlauf für den Austausch von Lenkwinkel, Endanschlägen und Force-Feedback

Die Information, über den aktuellen Lenkwinkel wird, wie in Abbildung 3.2 zu sehen, vom Lenkrad an den Controller gesendet. Es stellt seine aktuelle Position in Inkrementen bereit. Der Wert wird in einem 32Bit-Feld gespeichert und ist vorzeichenbehaftet. In Tabelle 3.1 sind die gemessenen Inkremente für die Winkel $180°$ und $540°$ zu sehen. Dafür wurde das Lenkrad auf die jeweiligen Winkel eingestellt und die Anzahl der Inkremente aus den Nachrichten gelesen.

| Winkel | Anzahl Inkremente | Kommentar |
|---|---|---|
| $\pm 180°$ | $\pm 20061$ | Endanschläge Sport-Modus |
| $\pm 540°$ | $\pm 60184$ | Endanschläge Cruise-Modus |

Tabelle 3.1: Endanschläge der Fahrmodi

Der Lenkwinkel wird, wie in Tabelle 3.2 zu sehen, mit der Nachricht $0x211$ an den Controller übertragen.

| Art | ID | Datenbreite | Beschreibung |
|---|---|---|---|
| TT | $0x200$ | $32Bit$ | Enthält die Endanschläge für das Lenkrad |
| TT | $0x201$ | $16Bit$ | Enthält die Stärke des Force-Feedback-Effektes vom Rad |
| TT | $0x211$ | $32Bit$ | Enthält die Position des Lenkrades |

Tabelle 3.2: Relevante Nachrichten und Daten

Damit das Lenkrad einen definierten Lenkbereich hat, werden Endanschläge definiert. Diese werden, genau wie die Position vom Lenkrad, in Inkrementen angegeben. Sie werden, je nach Fahrmodus, vom Controller eingestellt und an das Lenkrad, mit der Nachricht $0x200$, gesendet. Das Force-Feedback vom Rad, wird vom Controller, mit der Nachricht $0x201$, an das Lenkrad gesendet. Er wird als Kraft in Newton angegeben und ist ein vorzeichenbehafteter 16Bit-Wert. Der Force-Feedback-Effekt muss später mit dem Force-Feedback-Effekt aus vom PC kombiniert werden. Dafür muss der PC entweder eine Nachricht an das Lenkrad oder den Controller senden. Für beide Varianten muss eine neue Nachrichten-ID in das Netzwerk aufgenommen werden. Das liegt daran, dass der PC seinen Force-Feedback-Effekt, genau wie der Controller, über das TTE-Netzwerk senden muss. Dazu kann er aber nicht die gleiche TT-ID verwenden, die der Controller verwendet, da dies laut Spezifikation nicht zulässig ist. Somit muss eine neue Nachrichten-ID, zum Beispiel $0x202$, ins Netzwerk aufgenommen werden. Weitere Daten werden für die Realisierung nicht benötigt.

### Netzwerk

Da alle Netzwerk-Routen für den kritischen Datenverkehr statisch konfiguriert sind, müssen weitere Netzwerk-Routen für den PC eingerichtet werden. Wie schon erwähnt muss auch eine weitere Nachrichten-ID ins Netzwerk aufgenommen werden, die den Force-Feedback-Effekt aus der Anwendung, im Netzwerk transportieren kann.
Diese Nachricht muss an die Komponente geroutet werden, die die Kombination von den beiden Effekten durchführt. Dafür bieten sich zwei Möglichkeiten an. Zum einen kann der Force-Feedback-Effekt aus der Anwendung an den Controller gesendet werden, oder der Force-Feedback-Effekt der Anwendung wird direkt an das Lenkrad, genauer die RTE-CAN-Bridge, gesendet. Da die RTE-CAN-Bridge des Lenkrades über keine eigene Logik verfügt, sondern nur das tunneln der CAN-Nachrichten übernimmt, sollte man die Daten, konsistenter Weise, im Controller zusammenführen. Dabei ist darauf zu achten, dass der Demonstrator auch noch funktionieren muss, sollte die Anwendung keine Effekte generieren oder der PC gar nicht angeschlossen sein.
Das Netzwerk muss somit, wie in Abbildung 3.3 dargestellt, erweitert werden.

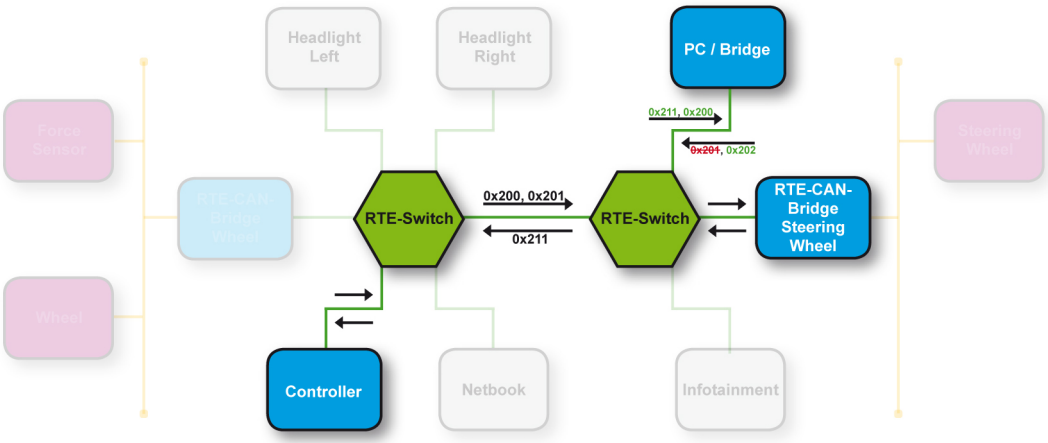


Abbildung 3.3: Nachrichtenverlauf nach Konfiguration des Netzwerkes, bei direkter Anbindung

Dazu müssen am Switch, an dem auch das Lenkrad angeschlossen ist, 3 weitere Netzwerkrouten eingerichtet und am Switch, an dem der Controller angeschlossen ist, eine weitere Netzwerkroute hinzugefügt werden. Dabei müssen die Nachrichten, mit dem Lenkwinkel und mit den Endanschlägen, an den PC und die Nachricht, mit dem Force-Feedback, vom PC an den Controller weitergeleitet werden.

### Betriebssystem

Die Entscheidung, welches Betriebssystem für die Realisierung eingesetzt wird, fällt auf Linux. Aufgrund des sich gerade in der Entwicklung befindlichen Netzwerkkartentreibers für Linux und der leichten Integration der Echtzeiterweiterung für den Standardkernel sowie das Nichtvorhandensein eines Windows-Netzwerkkartentreibers, ist Linux Windows vorzuziehen. Andere Betriebssysteme wurden nicht betrachtet. Auch wenn die direkte Einbindung nicht umsetzbar sein sollte, kann man unkompliziert auf eine alternative Anbindung wechseln.
Als Distribution wird Ubuntu 12.04 verwendet. Die Distribution ist sehr verbreitet und es gibt eine gute Auswahl an Fahrsimulatoren, die man direkt aus dem Ubuntu-Software-Center installieren kann.

### Joystick-Treiber

Um den Demonstrator einer Applikation als Eingabegerät bekannt zu machen, muss dieser beim Betriebssystem durch einen Geräteknoten repräsentiert werden. Dazu muss ein Joystick-Treiber entwickelt werden, der die Kommunikation zwischen Applikation und Netzwerk übernimmt.

Dieser hat die Aufgabe, den Lenkwinkel des Lankrades, einer Anwendung, zur Verfügung zu stellen, bzw. einen Force-Feedback-Effekt aus einer Anwendung, an den Demonstrator, weiterzuleiten. Da der maximale und minimale Lenkanschlag abhängig vom Fahrmodus (Cruise, Sport, Individuell) ist, muss auch diese Information vom Joystick-Treiber verarbeitet werden.
Die Anforderungen, die sich an den Joystick-Treiber stellen, beziehen sich auf die Schwerpunkte: Sicherheit, Zuverlässigkeit und Latenzverhalten. Das Verhalten des Demonstrators darf durch die Anbindung an die Simulation nicht beeinträchtigt werden.
Durch die repräsentative Funktion des Demonstrators, sollte zwingend darauf geachtet werden, dass er durch den Joystick-Treiber, nicht in kritische Zustände überführt werden kann. Es muss also darauf geachtet werden, dass dem Demonstrator keine fehlerhaften Daten gesendet werden. So kann eine zu große Kraft des Force-Feedback-Effektes das Lenkrad zu stark beanspruchen oder die Person verletzen, die gerade das Lenkrad nutzt.
Der Joystick-Treiber sollte die Daten unmittelbar verarbeiten, so dass die Latenzen, die unter anderem durch die Datenübertragung und den Scheduler des Betriebssystems entstehen, gering bleiben. Bei hohen Latenzen zwischen Aktionen am Lenkrad und Reaktionen in der Simulation, kann es sonst zu einem Verzögerungseffekt kommen.
Der Joystick-Treiber nutzt zur Kommunikation den TTE-Protokollstack. Dafür meldet er sich zum Empfangen von Nachrichten, mit einer Callback-Funktion am jeweiligen Eingangsbuffer an. Möchte er Daten versenden, schreibt er sie einfach in den passenden Ausgangsbuffer. Der Scheduler des TTE-Protokollstacks übernimmt dann, wie schon beschrieben, das zeitlich abgestimmte Versenden der Nachricht.
Um einer Anwendung den Lenkwinkel zur Verfügung zu stellen und deren Force Feedback aufzunehmen, müssen zwei Geräteknoten angelegt werden, über die die Anwendung mit dem Joystick-Treiber, Daten austauschen kann. In Linux kann man solch einen Knoten selbst anlegen oder überlässt das dem jeweiligen Subsystem.
Da der Joystick-Treiber ein Input-Gerät repräsentiert, wird hierzu das Input-Subsystem genutzt. Somit muss sich der Joystick-Treiber nicht selbst um die Anmeldung im Kernel und das Anlegen von Geräteknoten kümmern. Eine Anwendung kann dann über die definierten Standardstrukturen des Input-Subsystems mit dem Joystick-Treiber kommunizieren (vgl. Espinosa, 1998) (vgl. Deneux und Hannula, 2001) . Somit wird ein transparenter Zugriff und eine hohe Kompatibilität gewährleistet.

### Vorteile

- es gibt eine direkte Anbindung an das RTE

- Joystick-Treiber ist unabhängig von möglichen Änderungen im Netzwerk
- TTE-Protokollstack kümmert sich um die Kommunikation

**Nachteile**

- es muss eine neue Nachrichten-ID mit festem Sendeslot integriert werden, welche auch Bandbreite belegt, wenn PC nicht angeschlossen ist
- TTE-Protokollstack funktioniert nur für sehr wenige Netzwerkkarten
- Betriebssystem muss echtzeitfähig sein

### 3.2.2 Indirekte Anbindung über die RTE-CAN-Bridge des Lenkrades

Als alternative Anbindung an das Netzwerk, wird die RS-232-Schnittstelle der RTE-CAN-Bridge des Lenkrades, genutzt. Denn wie in Abbildung 3.4 zu sehen ist, kommen hier alle erforderlichen Informationen an. Deshalb bietet es sich an, die RTE-CAN-Bridge so zu erweitern, dass die Nachrichten von hier auch an den Joystick-Treiber getunnelt werden.

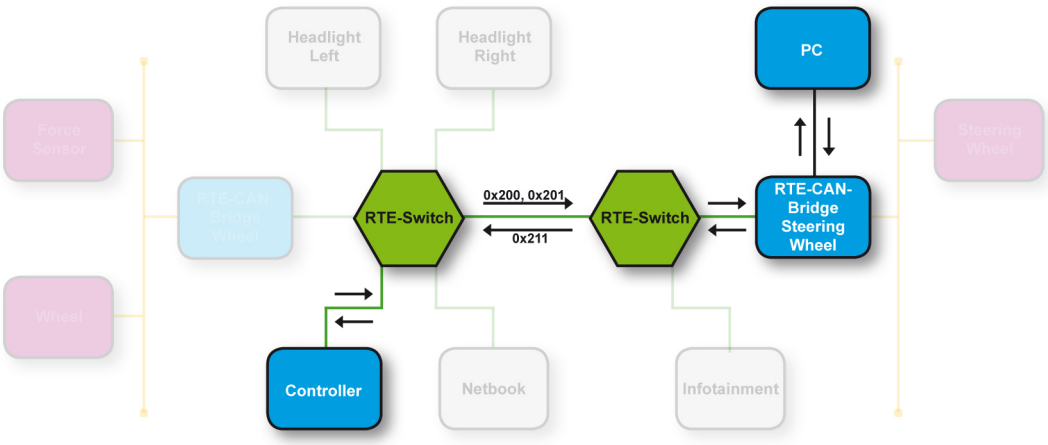


Abbildung 3.4: Nachrichtenverlauf nach Anbindung über die Bridge des Lenkrades

Die aktuelle RTE-CAN-Bridge tunnelt die CAN-Nachrichten zur Zeit nur über das RTE-Netzwerk. Dazu verpackt sie die CAN-Nachrichten in RTE-Nachrichten oder extrahiert sie aus diesen (vgl. Kamieth, 2011) . Die RTE-CAN-Bridge muss also um eine Tunnelung von RTE- bzw. CAN-Nachrichten, über die RS-232-Schnittstelle, erweitert werden. Dazu muss festgestellt werden, ob die Bandbreite der RS-232-Schnittstelle, für die Datenübertragung ausreicht.

Der Joystick-Treiber muss nun so konzipiert werden, das dieser über die serielle Schnittstelle seine Daten austauscht. Damit der Joystick-Treiber die ankommenden Daten unterscheiden kann, muss ein Übertragungsprotokoll entwickelt werden.

### RTE-CAN-Bridge

Um mit dem Joystick-Treiber über die RS-232-Schnittstelle kommunizieren zu können, muss die RTE-CAN-Bridge auf dem seriellen Bus schreiben und lesen können. Zur Zeit ist nur das Schreiben auf der RS-232-Schnittstelle implementiert, somit muss sie noch um das Lesen erweitert werden. Man kann die RTE-CAN-Bridge so konfigurieren, dass ein Interrupt ausgelöst wird, sobald eine Nachricht auf der RS-232-Schnittstelle eintrifft. Auf diesen kann man sich dann mit einer eigenen ISR registrieren und die ankommenden Daten verarbeiten.
An den Stellen, an der die Bridge die CAN-Nachrichten des Lenkrades in RTE-Nachrichten für den Controller verpackt und abschickt, müssen die CAN-Nachrichten auch in serielle Nachrichten für den Joystick-Treiber, verpackt und abgeschickt werden. Der zyklisch eintreffende Force-Feedback-Effekt des echten Rades, muss, bevor er an das Lenkrad weitergeschickt wird, mit dem Force-Feedback-Effekt vom Joystick-Treiber, zusammengeführt werden.

### Serielle Kommunikation

Um die Daten, die über die RS-232-Schnittstelle ausgetauscht werden, identifizieren zu können, muss ein eigenes Übertragungsprotokoll entwickelt werden. Um dies so transparent wie möglich zu machen, orientiert sich das Übertragungsprotokoll zum einen am Aufbau von RTE-Nachrichten. Das bedeutet, dass die Protokollnachrichten anhand einer ID unterschieden werden. Dafür wird die TT-ID aus der RTE- bzw. CAN-Nachricht, ohne den TT-Marker, genutzt.
Zum anderen gibt es unter Linux bereits Joystick-Treiber die für die RS-232-Schnittstelle geschrieben wurden. Fast jeder von ihnen hat ein eigenes Übertragungsprotokoll implementiert, um über die RS-232-Schnittstelle Daten auszutauschen. Für die Entwicklung des Protokolls, wurde sich daher am Iforce-Protokoll orientiert. Dieses ist nicht vollständig bekannt, da es von den Rechteinhabern geheim gehalten wird. Alles was davon bekannt ist, wurde durch *reverse engineerning* herausgefunden (vgl. Deneux, 2011) . Allerdings ist die Idee des Protokolls, eine gute Grundlage für ein eigenes serielles Übertragungsprotokoll. Es zeichnet sich dadurch aus, dass es jeder Nachricht eine Präambel voranstellt und ein Feld für die Länge der Nutzdaten besitzt.
Anhand dieser Informationen, ist ein eigenes Übertragungsprotokoll entwickelt worden, welches das Protokoll der RS-232-Schnittstelle erweitert. Es dient dazu, übertragene Daten unterscheidbar zu machen, in dem sie in einer definierten Struktur übertragen werden. Die

eigentliche Datenübertragung wird aber weiterhin mit dem Protokoll der RS-232-Schnittstelle durchgeführt.

### Nachrichtenaufbau

Um die Daten richtig identifizieren zu können, muss bekannt sein, wann eine Protokollnachricht anfängt und welche ID die Daten haben. Da die zu übertragenden Daten unterschiedlich groß sein können, muss auch die Länge der Daten übertragen werden. Der prinzipielle Aufbau einer Protokollnachricht ist in Abbildung 3.5 zu sehen.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| PREEMBEL | ID | | LENGTH | DATA | | | |
| $0x2b$ | $0x11$ | $0x02$ | $0x04$ | $0x5b$ | $0x43$ | $0xff$ | $0xff$ |

Abbildung 3.5: Prinzipieller Aufbau einer Nachricht am Beispiel des Lenkradwinkels

Um den Start einer neuen Übertragung zu signalisieren, steht am Anfang jeder Nachricht eine Präambel. Diese ist 1 Byte groß und wie beim Iforce-Protokoll, wird sie durch den Wert $0x2b$ gekennzeichnet. Gefolgt wird sie durch ein 2 Byte großes ID-Feld, welches die Nachricht ausweist. Dieses entspricht genau dem ID-Feld für kritische Nachrichten aus dem TTE-Frame (TT-ID), nur ohne den TT-Marker. Dann folgt ein 1 Byte großes Feld mit der Länge der eigentlichen Daten. Damit ist die maximale Länge der Daten auf 255 Byte begrenzt. Das ist nicht weiter problematisch, da, wie später gezeigt wird, die RS-232-Schnittstelle sowieso nicht schnell genug ist, um 255 Byte in einem Zyklus zu übertragen. Als letztes kommen die eigentlichen Nutzdaten.
Über die Präambel ist es zwar möglich den Start einer Protokollnachricht zu erkennen, sollte man aber erst mit dem Lesen beginnen, während die Übertragung einer Protokollnachricht schon begonnen hat, kann es zu Problemen kommen. So kann es vorkommen, dass mitten in einer Protokollnachricht der Wert der Präambel in Form von Daten auftaucht und fehlinterpretiert werden kann. Daher muss bei der Implementierung des Protokolls darauf geachtet werden, dass die Daten einen Sinn ergeben. Zum Beispiel kann der Force-Feedback-Effekt nie über einen bestimmen Wert steigen.
Dazu können die Länge der Daten, die ID und die Nutzdaten einer Plausibilitätsprüfung unterzogen werden. Unterlässt man eine Überprüfung der Daten, kann es, zumindest theoretisch, möglich sein, dass die ganze Zeit falsche Daten gelesen werden.

### Datenanalyse für die serielle Kommunikation

Es muss nun sichergestellt werden, dass die Bandbreite der RS-232-Schnittstelle ausreichend für die anfallenden Daten ist. Begrenzt wird die Bandbreite durch die Baudrate der RS-232-Schnittstelle und den Zyklus des TTE-Netzwerkes.
Das NetX-Board, auf dem die RTE-CAN-Bridge realisiert wurde, unterstützt eine maximale Baudrate von $115200 Baud$ (vgl. Lipfert, 2008) . Der Informationsgehalt eines Zeichens beträgt bei der seriellen Übertragung genau $1bit$. Somit sind Baudrate und Datenraten in diesem Fall identisch.
Wie bereits beschrieben, werden TT-Nachrichten zyklisch verschickt. Die derzeitige Zykluszeit des Demonstratornetzwerkes beträgt $5ms$. Somit ergibt sich, aus der Zykluszeit des Netzwerkes und der maximalen Datenrate der RS-232-Schnittstelle, die folgende maximale Anzahl an übertragbaren Bits pro Zyklus:

$$115200 \frac{Symbol}{s} * 1 \frac{bit}{Symbol} = \underline{\underline{115200 \frac{bit}{s}}} \tag{3.1}$$

$$115200 \frac{bit}{s} * 5 * 10^{-3} s = \underline{\underline{576 \frac{bit}{Zyklus}}} \tag{3.2}$$

Der PC überträgt nur den Force-Feedback-Effekt an die Bridge, wofür 16 Bit an Daten anfallen. Die Bridge überträgt den Lenkwinkel und den Endanschlag an den PC, wofür insgesamt 64 Bit an Daten anfallen. Da die RS-232-Schnittstelle im Full-Duplex-Modus arbeitet, sind Senden und Empfangen unabhängig voneinander. Daher muss nur nachgewiesen werden, dass die Bandbreite ausreichend ist, um die 64 Bit rechtzeitig zu übertragen.
Um 64 Bit zu übertragen, müssen mindestens acht serielle Nachrichten verschickt werden, da eine serielle Nachricht nur acht Datenbits zur Verfügung hat. Mit den Einstellungen: ein Startbit, acht Datenbits, kein Paritätsbit und ein Stoppbit, hat die serielle Nachricht eine Länge von insgesamt 10 Bit.
Zu einer Protokollnachricht kommt nun noch der Protokoll-Header dazu. Dieser ist, mit Präambel, ID und der Längenangabe, 32Bit groß. Fasst man die Werte zusammen, werden für die Übertragung von zwei Protokollnachrichten, sechzehn serielle Nachrichten benötigt, was 160 Bit entspricht.

$$32bit \text{ (Header)} + 32bit \text{ (Nutzdaten)} = 64bit \text{ (Protokollnachricht)} \tag{3.3}$$

$$\frac{64bit}{8\frac{bit}{Nachricht}} = 8Nachrichten \tag{3.4}$$

$$1Startbit + 8Datenbits + 1Stoppbit = 10bit \tag{3.5}$$

$$10bit * 8 \text{ serielle Nachrichten} = 80\frac{bit}{\underline{\underline{\text{Protokollnachricht}}}} \tag{3.6}$$

$$80\frac{bit}{\text{Protokollnachricht}} * 2 \text{ Protokollnachricht} = 160\frac{bit}{\text{Zyklus}} \tag{3.7}$$

$$160\frac{bit}{\text{Zyklus}} < 576\frac{bit}{\text{Zyklus}} \tag{3.8}$$

Die Bandbreite reicht aus um die Daten in einem $5ms$ Zyklus zu übertragen. Wie die Tabelle 3.3 zeigt, darf aber die minimale Zykluszeit $2ms$ nicht unterschreiten.

| Bit | Zyklus |
|---|---|
| $\approx 115$ | $1ms$ |
| $\approx 230$ | $2ms$ |
| $\approx 345$ | $3ms$ |
| $\approx 460$ | $4ms$ |
| $576$ | $5ms$ |

Tabelle 3.3: Anzahl übertragbarer Bits in Abhängigkeit zur Zykluszeit

### Joystick-Treiber

Wie zuvor beschrieben, muss der Treiber, den Lenkwinkel an eine Anwendung weitergeben und einen Force-Feedback-Effekt, aus einer Anwendung, an das Lenkrad oder den Controller weiterleiten. Dies macht er, in dem er sich auch hier, am Input-Subsystem als Joystick mit Force Feedback Funktion, anmeldet. Der große Unterschied liegt darin, dass er keine direkte Anbindung an den Netzwerkkartentreiber hat, sondern über die serielle Schnittstelle kommuniziert.
Um Zugang zur seriellen Schnittstelle zu erhalten, hat der Joystick-Treiber zwei Möglichkeiten. Zum einen kann er direkt mit der Hardwareschnittstelle kommunizieren. Zum anderen kann er sich am Serial-Subsystem anmelden.
Im ersten Fall, muss der Standard-Linux-Treiber für die serielle Schnittstelle entfernt werden, bevor der Joystick-Treiber selbst mit der seriellen Schnittstelle kommunizieren kann. Dafür

muss der Kernel neu kompiliert werden, da dieser als build-in-Treiber realisiert worden ist. Das hätte zum einen den Nachteil, dass die serielle Schnittstelle nur noch zusammen mit dem Demonstrator genutzt werden kann. Zum anderen kann man den Joystick-Treiber so nur auf Systemen einsetzten, die den Standardtreiber nicht integriert haben.
Die andere Möglichkeit, sich am Serial-Subsystem anzumelden, hat den Vorteil, dass die Hardware der seriellen Schnittstelle nicht selbst gesteuert werden muss. Man kann einfach die Schnittstellen des Serial-Subsystems nutzen. Für alle seriellen Ports, sind vom seriellen Linux-Treiber schon Geräteknoten erstellt worden. Somit muss nur bekannt gemacht werden, über welchen dieser Ports das Gerät angeschlossen ist. Ist das Gerät mit einem Port verbunden, sollte sichergestellt sein, dass nur über den Joystick-Treiber mit dem Port kommuniziert werden kann. Es wäre sonst möglich, dem Gerät, beliebige Signale zu senden.
Die Anmeldung am Input- und Serial-Subsystem kann leicht umgesetzt werden, da es eine ausreichende Dokumentation zu den Schnittstellen gibt.

#### Vorteile

- Das Protokoll kann auch zur Datenübertragung an andere Geräte genutzt, bzw. zur Steuerung des Demonstrators eingesetzt werden.
- Eine Echtzeiterweiterung des Kernels ist nicht zwingend notwendig, so lässt sich der Treiber leicht auf anderen Systemen installieren.
- Die Konfiguration des Netzwerks bleibt unverändert.
- Die Funktionalität der Bridge wird um serielles Lesen erweitert, was auch für Debug-Zwecke interessant sein kann.

#### Nachteile

- Eine sehr begrenzte Anzahl an übertragbaren Nachrichten zwischen RTE-CAN-Bridge und Joystick-Treiber, bedingt durch die geringe Bandbreite

# 4 Umsetzung und Integration

Dieses Kapitel beschreibt die Umsetzung des Konzeptes der indirekten Anbindung und dessen Integration in den Demonstrator sowie die Umstände, die dazu geführt haben, dass das Konzept der direkten Anbindung nicht realisiert werden konnte.
In der Anforderungsanalyse und Konzeption wurden zwei Vorgehensweisen beschrieben. Die direkte Anbindung basiert auf dem Einsatz eines Ethernet-Protokollstacks der RTE-fähig ist. Leider wurde der, sich in der Entwicklung befindliche, TTE-Protokollstack, der CoRE-Projektgruppe, nicht rechtzeitig fertiggestellt. Somit konnte nur auf den TTE-Protokollstack der Firma TTTech zurückgegriffen werden.
Eine der Anforderungen ist es, die Arbeit unter dem aktuellen Linux-Kernel 3.X zu realisieren. Um feststellen zu können, ob das erste Konzept überhaupt mit dem TTE-Protokollstack der Firma TTTech umsetzbar ist, wurde ein Driver[1] entwickelt. Dieser nutzt Funktionen der API des TTE-Protokollstacks, um das Senden und Empfangen von Nachrichten anzustoßen.
Bevor der Driver eingesetzt werden kann, muss geprüft werden ob die TTE-Erweiterung der Firma TTTech auch unter dem aktuellen Kernel einsetzbar ist. Dazu wurde der aktuelle Ethernet-Protokollstack, für den r8169 Chipsatz, um die TTE-Erweiterung der Firma TTTech ergänzt, da dieser für Kernel 3.X verfügbar ist.
Dabei hat sich gezeigt, dass, sobald eine Funktion der TTE-Erweiterung aufgerufen wird, ein Fehler im Kernel[2] auftritt. Es ist aus der Logdatei des Kernels[3] zwar ersichtlich, an welchen Stellen die Fehler auftreten, allerdings können diese nicht behoben werden, da der Quellcode der Bibliothek, auf der die TTE-Erweiterung basiert und auf die per API zugegriffen wird, nicht verfügbar ist.
Da somit kein TTE-Protokollstack für den Kernel 3.X existiert, muss vom ersten Konzept abgelassen werden.

[1] Begriff aus dem Bereich Software-Testing, welcher ein Codefragment beschreibt, dass Funktionen eines anderen Codefragmentes zu Testzwecken ausführt.
[2] Auch Kernel-Panic genannt, beschreibt eine Abweichung vom Normalverhalten des Kernels. Sobald ein Fehler in einem Kernelprozess auftritt, wird eine Fehlernachricht erzeugt und der Prozess, wenn möglich, beendet.
[3] Diese ist im Ordner /var/log/ zufinden.

## 4.1 Joystick-Treiber für ein serielles Gerät mit Anbindung an die RTE-CAN-Bridge

Die Umsetzung des Konzeptes der indirekten Anbindung, zeigt eine alternative Einbindung in das RTE-Netzwerk. Es werden nachfolgend alle Schritte beschrieben die dazu nötig sind, um die RTE-CAN-Bridge des Lenkrades mit einem PC zu koppeln und dabei alle gestellten Anforderungen zu erfüllen.

### 4.1.1 Arbeitsschritte

Um die Arbeit strukturiert und aufeinander aufbauend zu gestalten, wurde ein Arbeitsplan erstellt. Anhand der definierten Komponenten aus Abbildung 4.1, wurden 4 Arbeitsschritte festgelegt.

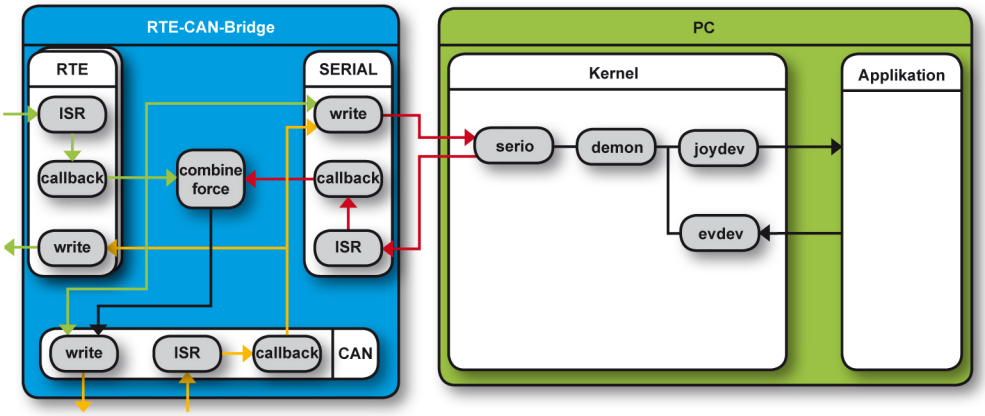


Abbildung 4.1: Komponentenansicht

#### Erweiterung der RTE-CAN-Bridge um serielles Lesen

Als erstes wurde die RTE-CAN-Bridge um serielles Lesen erweitert. Das war notwendig, damit der Force-Feedback-Effekt vom PC an die Bridge gesendet werden kann. Damit wurde die Grundlage für die Kommunikation zwischen beiden Komponenten geschaffen.

#### Weiterleitung der CAN- bzw. RTE-Nachrichten, an die serielle Schnittstelle

Als nächstes wurden die Routinen, die die CAN- und TT-Nachrichten verarbeiten, darum erweitert, dass sie nun auch die Nachrichten über den Lenkwinkel und den Fahrmodus, über

die RS-232-Schnittstelle der Bridge, versenden und der Force-Feedback-Effekt des Rades mit dem vom PC kombiniert wird.

#### Entwicklung des Joystick-Treibers

Den größten Teil nahm die Entwicklung des Joystick-Treibers ein. Er verarbeitet den Lenkwinkel und den Fahrmodus und leitet diesen an eine Applikation weiter. Zudem schickt er den Force-Feedback-Effekt aus dem Spiel an den Demonstrator.

#### Inbetriebnahme des erweiterten Systems

Der letzte Punkt besteht aus der Integrierung des Joystick-Treibers in das Betriebssystem. Anschließend wird der komplette Aufbau des Demonstrators betrachtet und das gesamte System in Betrieb genommen.

### 4.1.2 Erweiterung der RTE-CAN-Bridge um serielles Lesen

Um überhaupt einen Force-Feedback-Effekt vom Joystick-Treiber zu erhalten, muss die RTE-CAN-Bridge von der seriellen Schnittstelle lesen können. Um dies zu realisieren, wurden die Einstellungen, die während der Initialisierungsphase der seriellen Schnittstelle getätigt werden, ergänzt. Anschließend wurde, wie in Abbildung 4.2 zu sehen, eine ISR entwickelt, die durch einen Interrupt, ausgelöst durch eine eintreffende Nachricht, getriggert wird.

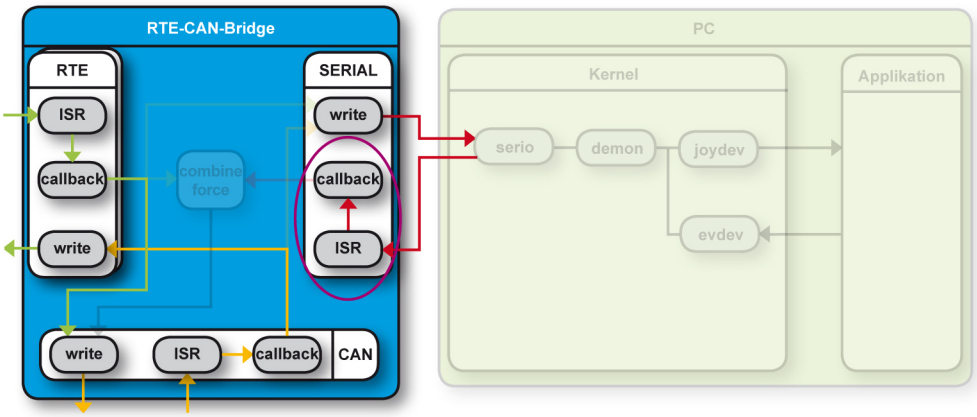


Abbildung 4.2: Aktuelle Komponentenansicht mit Komponente „serielles Lesen “

Um die ISR so modular wie möglich zu halten, ruft diese nur eine Callback-Function[4] auf, die die eigentliche Datenverarbeitung ausführt. So kann später einfach die Callback-Function ausgetauscht werden, wenn eine andere Funktionalität gewünscht ist.

### Initialisierung

Für die Initialisierung der seriellen Schnittstelle gibt es bereits eine Routine, die die Geschwindigkeit, Parität, Startbit und Stoppbits, sowie das Schreiben auf diese einstellt und aktiviert. In dieser Routine wurden auch die Einstellungen zur Initialisierung des seriellen Lesens eingepflegt. Diese bestehen nur aus wenigen Schritten, die nachfolgend erläutert werden.
(1.) Die Bridge löst immer dann einen Interrupt aus, wenn ein Input-Buffer eine bestimmte Anzahl an Zeichen enthält. Die Grenze kann man dabei auf eine beliebige Anzahl einstellen. Die Grenze des Input-Buffers der seriellen Schnittstelle, wurde auf ein Zeichen eingestellt. Das verhindert, dass Daten im Input-Buffer liegen, von denen man nichts weiß, da die Protokollnachricht zwar komplett übertragen wurde, aber die Mindestanzahl an Zeichen noch nicht erreicht ist. (2.) Anschliessend, wurde eine ISR auf den Interrupt registriert, der durch den Input-Buffer der seriellen Schnittstelle ausglöst wird. (3.) Im letzten Schritt wurde der Interrupt aktiviert.

### ISR

In Abbildung 4.3 ist der Ablauf der ISR dargestellt. Sie muss als erstes prüfen ob der Interrupt, der sie getriggert hat, auch der Richtige ist. Da die Interrupt-Leitung auf mehrere Events reagiert (Shared Interrupt), kann es sein, dass der Interrupt für jemand anderen bestimmt ist. Ist der Interrupt als der passende identifiziert, wird der Input-Buffer der seriellen Schnittstelle komplett ausgelesen, was gleichzeitig den Interrupt zurücksetzt. Alle gelesenen Daten werden, Zeichen für Zeichen, an die vorher registrierte Callback-Function weitergegeben.

### Callback-Schnittstelle

Als letztes wurde eine Schnittstelle implementiert, die es erlaubt, eine Callback-Function für eintreffende serielle Nachrichten anzumelden. Die Callback-Function bekommt von der ISR immer ein Zeichen übergeben, wobei die ISR keinen Rückgabewert erwartet. Dementsprechend ist der Typ einer Callback-Function wie in Tabelle 4.1 definiert.

[4] Eine Funktion, die an ein Ereignis gebunden werden kann. Sobald das Ereignis eintritt, wird die Funktion aufgerufen.

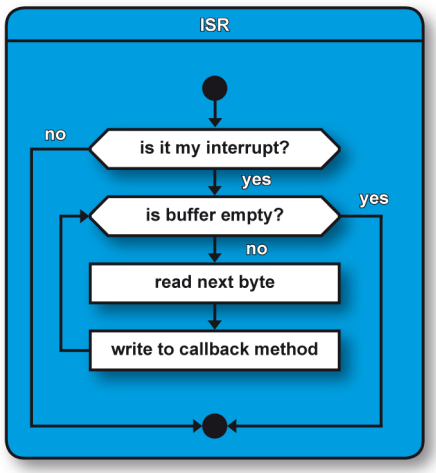


Abbildung 4.3: Ablauf der ISR

| Return value | Type | Parameter |
|---|---|---|
| void | *usartCbFuncPtr | unsigned char |

Tabelle 4.1: Typ der Callback-Function für die RS-232-Schnittstelle

Über die Funktion $usart\_register\_receive\_callback$ wird die Callback-Function registriert. Die Adresse der Callback-Function wird dazu in einem globalen Funktionszeiger abgelegt, wobei jeder serielle Port seinen eigenen Funktionszeiger besitzt. Dieser wird von der ISR verwendet, um die Daten an die Callback-Function weiterzugeben. War die Registrierung nicht erfolgreich, bekommt man dies durch einen Rückgabewert mitgeteilt, der ungleich 0 ist.
Die implementierte Callback-Function kümmert sich um die Verarbeitung der eintreffenden Nachrichten. Eine Protokollnachricht wird vor der Übertragung in kleinere serielle Nachrichten aufgeteilt. Somit kommt jedes Byte einzeln in der Callback-Function an. Um die Protokollnachricht wieder zusammenzusetzen, wurde ein Zustandsautomat entwickelt und implementiert, der die Daten der Protokollnachricht in einer passenden Struktur ablegt. Dieser ist in Abbildung 4.4 zu sehen. Die Aktion, die zu einer Zustandsänderung führt, ist jedesmal das Eintreffen einer neuen seriellen Nachricht. Die Daten werden zu dem einem Plausibilitätstest unterzogen. So wird festgetellt, dass die Längenangabe der Nutzdaten 4 Byte nicht übersteigt. Ist die ermittelte ID unbekannt, wird die Nachricht verworfen. Damit ist die Implementierung der seriellen Kommunikation, auf Seiten der RTE-CAN-Bridge, abgeschlossen.

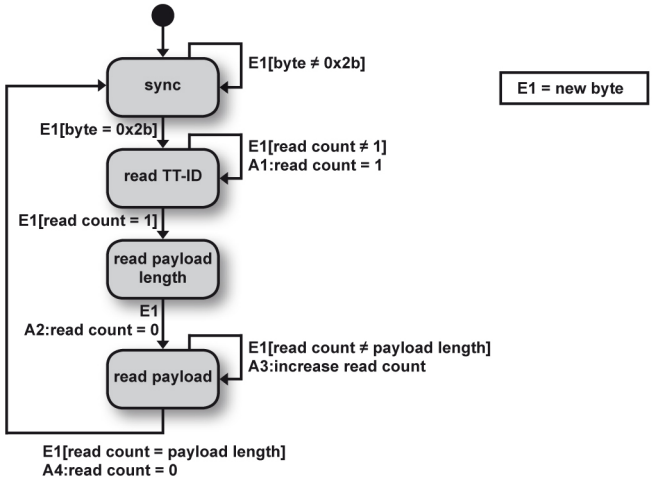


Abbildung 4.4: Zustandsautomat zum Lesen einer Protokollnachricht

### 4.1.3 Weiterleitung der CAN- bzw. RTE-Nachrichten, an die serielle Schnittstelle

Um die Nachrichten über den Lenkwinkel und den Fahrmodus an den Joystick-Treiber weiterzuleiten, bzw. die Force-Feedback-Effekte zu kombinieren, musste erst festgestellt werden, welche Routinen die Nachrichten vom RTE zum CAN-Bus tunneln und umgekehrt. In Abbildung 4.5 ist zu sehen, das es jeweils eine ISR und eine Callback-Function für eintreffende RTE- und CAN-Nachrichten gibt.

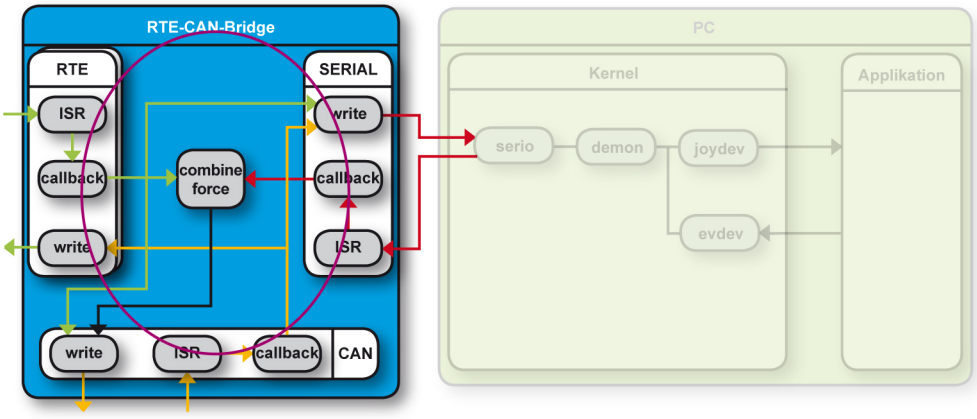


Abbildung 4.5: Verknüpfung der Kommunikationsschnittstellen

Für den Lenkwinkel, der direkt vom Lenkrad über die CAN-Schnittstelle kommt, ist es möglich, in der ISR eine weitere Callback-Function hinzuzufügen, die die Nachricht in den Output-Buffer der seriellen Schnittstelle schreibt. Für das Force-Feedback muss ein anderer Ansatz verfolgt werden, da man zur Zeit nur eine Callback-Function für jede RTE-Nachricht anmelden kann. Deswegen wird die Callback-Function $bridge\_reth\_receive\_low$ selbst erweitert. In Tabelle 4.2 kann man sehen welche Nachrichten von RTE nach CAN und umgekehrt getunnelt werden.

| Coming from | Source | Destination | ID | Content |
|---|---|---|---|---|
| Controller | RTE | CAN | $0x200$ | Winkel Endanschlag |
| Controller | RTE | CAN | $0x201$ | Force-Feedback vom Rad |
| Lenkrad | CAN | RTE | $0x211$ | Winkel Lenkrad |

Tabelle 4.2: Quellen der Nachrichten

### Weiterleitung des Lenkwinkels an den Joystick-Treiber

Die Nachricht mit dem Lenkwinkel wird in dem Datentyp $CAN\_FRAME\_T$ an die Bridge übermittelt. In diesem gibt es bereits, wie in Tabelle 4.3 zu sehen, separate Felder für die TT-ID, die Länge der Nutzdaten und die Nutzdaten selbst.

| Field | Comment |
|---|---|
| ulUniqueID | unique packet identifier (only CAN) |
| ulFrameInfo | Frame Information (length of payload) |
| ulIdentifier | Frame identifier (TT-ID) |
| abData[X] | Frame payload (max 8 Bytes per frame) |
| ulTimestampS | Timestamp in seconds |
| ulTimestampNS | Timestamp in nanoseconds |

Tabelle 4.3: Aufbau CAN_FRAME_T

Der Ablauf der ISR, zur Verarbeitung der Nachricht, entspricht dem aus Abbildung 4.3 auf Seite 31. Um die Daten zu verarbeiten und an die RS-232-Schnittstelle zu übergeben, wurde eine weitere Callback-Function in die ISR der CAN-Schnittstelle integriert. Diese nimmt sich die relevanten Daten aus dem $CAN\_FRAME\_T$ und legt sie in einem angepassten Format ($SER\_FRAME\_T$), im Output-Buffer der seriellen Schnittstelle, ab. Den Aufbau des Datentyps kann man in Tabelle 4.4 sehen.
Der Scheduler übernimmt dann das Versenden der seriellen Nachricht. Wird in einem Zyklus kein Lenkwinkel vom Lenkrad an die Bridge übertragen, wird auch keine serielle Nachricht an den Joystick-Treiber geschickt.

| Field | Comment |
|---|---|
| preamble | Frame preamble ($0x2b$) |
| id[2] | TT-ID |
| length | Length of payload |
| payload[X] | Frame payload (max 255 Bytes per Frame) |

Tabelle 4.4: Aufbau SER_FRAME_T

### Weiterleitung des Fahrmodus an den Joystick-Treiber

Wie schon festgestellt wurde, muss der Fahrmodus, der in diesem Fall nur die Lenkanschläge enthält, auch an den Joystick-Treiber übermittelt werden. Das wird später, bei der Implementierung des Joystick-Treibers, noch klarer. Um dies zu realisieren, wird die Callback-Function, die bereits auf alle für das Lenkrad relevanten Nachrichten registriert wurde, erweitert. In der Callback-Function selbst, wird eine RTE-Nachricht auf eine CAN-Nachricht gebridged. Diese hat das gleiche Format wie schon beim Lenkwinkel ($CAN\_FRAME\_T$). Immer wenn die Callback-Function aufgerufen wird, wird überprüft ob die TT-ID der Nachricht, der TT-ID des Fahrmodus entspricht. Stimmen die TT-ID's überein, wird im Output-Buffer der seriellen Schnittstelle, die Nachricht im passenden Format abgelegt. Auch diese wird zyklisch verschickt.

### Zusammenführen der Force-Feedback-Effekte

Genau wie bei den Endanschlägen für das Lenkrad, kann die Nachricht mit dem Force-Feedback-Effekt nur in der bereits vorhandenen Callback-Function abgehandelt werden. In dieser werden die Daten aus der RTE-Nachricht in eine CAN-Nachricht kopiert und danach verschickt. Vor dem Verschicken der CAN-Nachricht, wird jetzt der Wert des Force-Feedback-Effektes manipuliert.
Die Stärke des Force-Feedback-Effekts vom Joystick-Treiber wird, genau wie die vom Rad des Demonstrators, in Newton angegeben (vgl. Stepanov, 2011) . Das vereinfacht die Kombination der beiden Effekte. Die Stärke des Force-Feedback-Effektes vom Joystick-Treiber, hängt immer von der jeweiligen Applikation ab. So kann es vorkommen, dass die Kraft aus der Applikation zu stark für den Demonstrator ist. Daher wird, zum Schutz vor möglichen Schäden, die Kraft vom Joystick-Treiber begrenzt und zwar auf die maximale Kraft, die vom Rad des Demonstrators gesendet wird, die zur Zeit $5000N$ beträgt.
Um die beiden Effekte zu kombinieren, werden sie nicht einfach addiert. Das könnte sonst dazu führen das zwei gegeneinander gerichtete Kräfte sich aufheben. Vielmehr hat, wie Abbildung **??** zeigt, immer der stärkere Effekt Vorrang, unabhängig vom Vorzeichen. Sollten beide Effekte

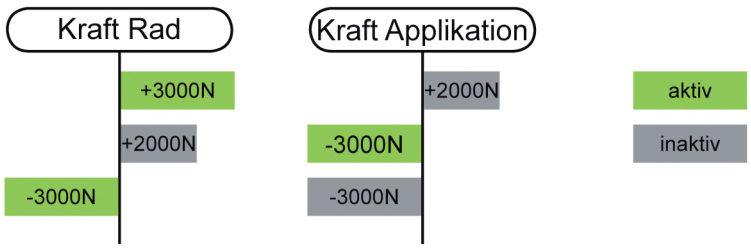


Abbildung 4.6: Stärkeabhängige Priorität der Kräfte

gleich Stark sein, wird immer der Force-Feedback-Effekt des Rades bevorzugt. Ist der PC nicht angeschlossen, oder unterstützt die Applikation kein Force-Feedback, beträgt die Stärke des Force-Feedback-Effekts vom Joystick-Treiber immer $0N$.

### 4.1.4 Entwicklung des Joystick-Treibers

Den größten Teil der Entwicklung, hat der Joystick-Treiber eingenommen. Er soll die Daten von der seriellen Schnittstelle lesen, bzw. Daten über die RS-232-Schnittstelle schreiben. Diese Daten gibt er aufbereitet an Applikationen weiter, oder bekommt sie von diesen. Der erste Schritt war die Registrierung als serieller Treiber. Danach folgte die Verarbeitung und Validierung der eintreffenden Daten und zum Schluss wurde eine Schnittstelle implementiert, die es Applikationen erlaubt, mit dem Joystick-Treiber zu kommunizieren. Die Komponenten sind in Abbildung 4.7 zu sehen, wobei die Komponente *demon*, den Joystick-Treiber repräsentiert.

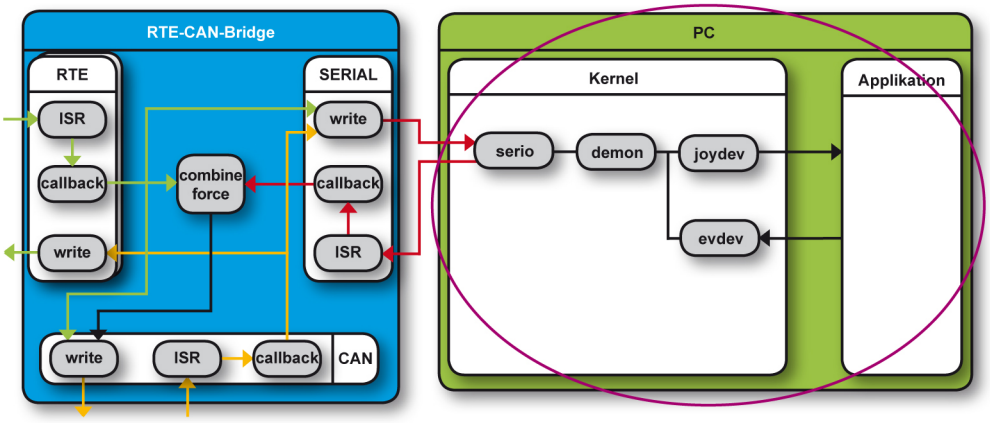


Abbildung 4.7: Erstellung des Joystick-Treibers

### Registrierung als serieller High-Level-Treiber

Eine Möglichkeit wäre es, einen seriellen Low-Level-Treiber zu entwickeln, der genau auf den Demonstrator angepasst wäre. In Linux gibt es allerdings bereits einen Low-Level-Treiber für die RS-232-Schnittstelle. Dieser ist als build-in-Treiber realisiert worden. Möchte man einen eigenen Low-Level-Treiber benutzen, muss vorher der Kernel neu kompiliert werden, um den Standardtreiber zu entfernen. Das ist für die Implementierung des Joystick-Treibers nicht optimal, da dieser somit nur auf Linux-Systemen einsetzbar wäre, die den Standardtreiber nicht im Kernel einkompiliert haben.
Die zweite Möglichkeit ist es, einen High-Level-Treiber zu entwickeln, der den seriellen Low-Level-Treiber von Linux nutzt. Das hat auch den Vorteil, dass man sich nicht selbst um die Hardwareansteuerung kümmern muss. Alle Funktionen zur seriellen Kommunikation werden dann vom seriellen Subsystem zur Verfügung gestellt. Deshalb wurde der Joystick-Treiber, als High-Level-Treiber für die serielle Schnittstelle, realisiert.
Um sich als High-Level-Treiber für die serielle Schnittstelle anzumelden, werden in der Initialisierungsphase des Joystick-Treibers zwei Funktionen des seriellen Subsystems genutzt. Der Aufruf der Funktion $serio\_register\_driver$ meldet den Joystick-Treiber bei der Core-Schicht des seriellen Subsystems an, bindet den Joystick-Treiber aber noch nicht an einen Port. Um sich beim seriellen Subsystem wieder abzumelden, wird die Funktion $serio\_unregister\_driver$ verwendet. Beide Funktionen erwarten als Übergabeparamter einen Zeiger auf die Struktur $serio\_driver$. Sie sorgt dafür, dass Funkionsaufrufe aus der Low-Level-Schicht bis in die High-Level-Schicht hochgereicht werden.
In Tabelle 4.5 sind die vom Joystick-Treiber genutzten Funktionen und Strukturen zu sehen.

| Field | Comment |
|---|---|
| *interrupt | wird aufgerufen, wenn eine neue Nachricht eingetroffen ist |
| *connect | wird aufgerufen, wenn der Treiber an einen seriellen Port gebunden wird |
| *disconnect | wird aufgerufen, wenn der Treiber von einem seriellen Port endbunden wird |
| description | enthält Beschreibung des Treibers |
| id_table | enthält Informationen über seriellen Bus und Protokolle |

Tabelle 4.5: Funktionen und Felder aus der Struktur $serio\_driver$

Die Funktionen *connect* und *disconnect* werden aufgerufen, sobald ein Gerätetreiber an einen Port gebunden wird oder sich abmeldet. Die Funktion *interrupt* wird durch einen Interrupt getriggert und zwar sobald eine Nachricht im Input-Buffer der seriellen Schnittstelle zur Abholung bereit ist. Die *description* enthält einen kurzen Text mit Informationen über den Treiber. Die Struktur *serio_device_id* enthält die unterstützten Geräte, sowie deren Protokolle. Wichtig sind vorallem die beiden Felder *type* und *proto*, sie geben den Typ der seriellen Schnittstelle und das verwendete Übertragungsprotokoll an.
Alle seriellen Bustypen und alle Protokolle die der Linux-Kernel unterstützt, findet man in den Linux-Sourcen für die serielle Schnittstelle. Es kann natürlich auch vorkommen, dass dem Kernel ein Protokoll nicht bekannt ist, wie dies beim Protokoll des Demonstrators der Fall ist. Dann ist es möglich, dem Kernel mitzuteilen, dass der Joystick-Treiber, ein unbekanntes Protokoll unterstützt. Der Typ gibt an, um welchen Standard des seriellen Busses es sich handelt. Die Felder *id* und *extra* werden dafür verwendet, um bestimmte Geräte und deren Klassen zu unterscheiden. Es ist somit möglich, zwei Geräte eines Herstellers, die zwar über den selben seriellen Typ angeschlossen werden und das selbe Protokoll sprechen, unterschiedlichen High-Level-Treibern zuzuweisen. Genauso ist es möglich, durch die Angabe mehrerer *serio_device_id* Strukturen, einen Treiber für verschiedene Geräte bereitzustellen. Für den Demonstrator wurden die Einstellungen aus Tabelle 4.6 vorgenommen.

| **Field** | **Configuration** |
|---|---|
| type | SERIO_RS232 |
| extra | SERIO_ANY |
| id | SERIO_ANY |
| proto | SERIO_UNKNOWN |

Tabelle 4.6: Die Struktur *serio_device_id* und die Einstellungen für den Demonstratortreiber

Die Einstellungen sind nicht spezifisch. Solange kein weiterer serieller Treiber mit den gleichen Einstellungen geladen ist, wird immer der Joystick-Treiber des Demonstrators genutzt. Es hat sich während der Entwicklung gezeigt, dass alle anderen installierten oder geladenen Treiber, bekannte Protokolle nutzen und der Joystick-Treiber für den Demonstrator damit der einzige Treiber ist, der diese Einstellungen nutzt.
Damit sind alle für die Anmeldung notwendigen Schritte ausgeführt worden. Nachfolgend werden die einzelnen Funktionen der Struktur *serio_driver*, die vom Joystick-Treiber genutzt werden, genauer erklärt.

### Bindung des Joystick-Treibers an einen seriellen Port und Anlegen der Geräteknoten

Um einen seriellen High-Level-Treiber an einen seriellen Port zu binden, gibt es das Userspace-Programm $Inputattach$ (vgl. Pavlik, 1996) . Dieses stellt die RS-232-Schnittstelle auf die richtige Geschwindigkeit ein und versucht einen Treiber zu finden, der zum angeschlossenen Gerät passt. Dafür muss beim Aufruf des Programms angeben werden, an welchem seriellen Port man das Gerät angeschlossen hat und mit welchem Protokoll es angesprochen wird. Das Programm kennt das Protokoll des Demonstrators nicht und unterstützt auch nicht alle nötigen Einstellungen. Da der Quellcode des Programms offen ist, wurde dieses Problem durch einen eigenen Geräteeintrag behoben. Der Joystick-Treiber des Demonstrators kann nur mit dieser angepassten Version des Programms an einen seriellen Port gebunden werden. Das Programm muss dabei die ganze Zeit aktiv sein und kann daher als Daemon ausgeführt werden.
Wird der Joystick-Treiber über das Programm an einen seriellen Port gebunden, wird die Funktion $connect$ aufgerufen. Sie hat die Aufgabe den seriellen Kommunikationskanal zu öffnen. Darüber hinaus werden während der Verbindungsphase weitere Initialisierungen durchgeführt. Dazu gehört auch das Registrieren beim Input-Subsystem. Denn erst wenn ein Gerät angeschlossen ist, ist es nötig die Geräteknoten zu erzeugen. Ist ein Treiber erst einmal an einen seriellen Port gebunden worden, hat er exklusiven Zugriff auf diesen. Das bedeutet, dass kein anderer Prozess, direkt Nachrichten an den Demonstrator senden kann, solange der Joystick-Treiber geladen und gebunden ist.
Während der Registrierung als Input-Device muss dem Input-Subsytem mitgeteilt werden, welche Funktionen das Gerät unterstützt. Dazu gibt der Treiber an, wieviele Knöpfe und Achsen das Gerät besitzt, welche Einstellungen diese haben und ob das Gerät Force-Feedback-Fähig ist. In Tabelle 4.7 sind die Einstellungen zu sehen, die nötig sind um einen Geräteknoten anzulegen, der das Lenkrad des Demonstrators als Joystick repräsentiert.

| Type | Mode | Min | Max | Fuzz | Flat |
|---|---|---|---|---|---|
| wheel | absolute | $-32767$ | 32767 | 0 | 0 |

Tabelle 4.7: Einstellungen für den Geräteknoten

Linux unterscheidet zwischen absoluten und relative Axen und ordnet diese in bestimmte Klassen ein, wie Lenkrad, Ruder, Gas und Bremse. Der Wert $fuzz$ dient zur Rauschuntedrückung. Er gibt an, um wieviel sich der Absolutwert verändern muss, bis er als neuer Wert angesehen wird. Flat ist der Bereich der als Totwinkel betrachtet wird. Werte innerhalb dieses Bereiches, werden als 0 interpretiert.

Um das Anlegen des Geräteknotens kümmert sich der $joydev$-Treiber. Er verwaltet die unter $/dev/input/$ abgelegten Geräteknoten für Joysticks. Man erkennt sie am Namen $jsX$. Diese Schnittstelle dient nur zum Lesen des Lenkwinkels. Damit eine Applikation einen Force-Feedback-Effekt an den Joystick-Treiber übergeben kann, muss ein weiterer Geräteknoten erzeugt werden. Dies ist Aufgabe des $evdev$-Treibers. Durch ihn ist es möglich, Events aus dem Kernelspace in den Userspace zu senden und umgekehrt. Die Geräteknoten werden unter dem Namen $eventX$ angelegt und sind ebenfalls unter $/dev/input/$ zu finden. Die Zahl, die den Joystickknoten kennzeichnet, muss dabei nicht mit der Zahl die den Eventknoten kennzeichnet, übereinstimmen. Der Treiber $evdev$ erwartet die Struktur $ff\_event$, die nötig ist, um Effekte aus dem Userspace zu verarbeiten. Dazu sind die in Tabelle 4.8 aufgelisteten Funktionen implementiert und mittels der Struktur $ff\_event$ an den $evdev$-Treiber übergeben worden.

| Field | Comment |
|---|---|
| upload | wird aufgerufen, sobald ein neuer Effekt hochgeladen wird |
| erase | wird aufgerufen, wenn ein Effekt gelöscht werden soll |
| playback | wird aufgerufen, wenn ein Effekt gespielt<br>oder gestoppt werden soll |

Tabelle 4.8: Funktionen zur Verarbeitung von Force-Feedback-Effekten

Zudem erwartet der $evdev$-Treiber Einstellungen hinsichtlich der Unterstützung von Force-Feedback-Effekten. Dazu zählt die Anzahl an gleichzeitig anliegenden Effekten und die Angabe, welche Effekte das Gerät unterstützt. Die Einstellungen dazu sind in Tabelle 4.9 zu sehen. Auf diese wird bei der Verarbeitung von Force-Feedback-Effekten auf Seite 42 eingegangen.

| | |
|---|---|
| **number of simultaneous effects** | 1 |
| **supported effects** | $FF\_CONSTANT$ |

Tabelle 4.9: Einstellungen zur Force-Feedback-Unterstützung

Wird das Programm $Inputattach$ beendet, wird der Joystick-Treiber vom seriellen Port entbunden. Anschließend wird die Funktion $disconnect$ aufgerufen, welche die Kommunikation mit dem seriellen Port schließt und den Joystick-Treiber beim Input-Subsystem abmeldet. Beide Geräteknoten werden dann entfernt.

### Verarbeiten von seriellen Nachrichten

Ist der Joystick-Treiber erfolgreich am seriellen Subsystem angemeldet und an einen seriellen Port gebunden worden, kann er serielle Nachrichten empfangen. Ist eine neue Nachricht im

seriellen Input-Buffer abgelegt worden, wird ein Interrupt ausgelöst. Auf diesen Interrupt ist eine eigene ISR registriert worden, in dem an das Feld $interrupt$, der Struktur $serio\_driver$, der Funktionszeiger der ISR übergeben wurde. Diese wird, genau wie bei der RTE-CAN-Bridge, für jedes eintreffende Zeichen einzeln aufgerufen. Da auch hier das Problem besteht, dass eine Protokollnachricht (mindestens 4 Byte) zu groß für eine serielle Nachricht (maximal 1 Byte) ist, wird sie in mehrere serielle Nachrichten aufgeteilt. Somit muss dafür gesorgt werden, dass die Protokollnachricht in der ISR, wieder zusammengesetzt wird. Dazu wurde der auf Seite 32 dargestellte Zustandsautomat, auch hier implementiert. Und genau wie bei der RTE-CAN-Bridge, wird ein Plausibilitätstest der Daten durchgeführt.
Ist die Protokollnachricht vollständig, wird sie über eine $dispatchfunction$, an die passende Funktion geschickt. Der Ablauf ist in Abbildung 4.8 dargestellt.

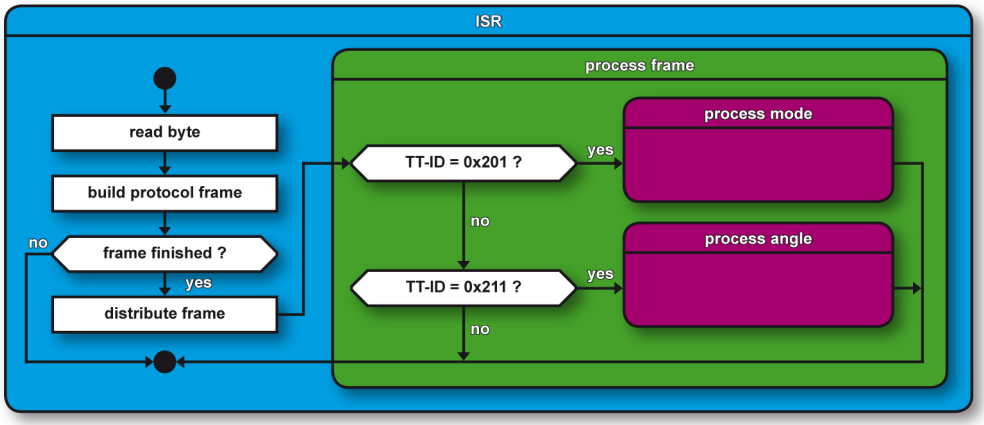


Abbildung 4.8: Verarbeitung von eingehenden seriellen Nachrichten

Über die Funktion $pocess\_mode$ werden die Lenkanschläge gespeichert. Diese können aber, je nach Fahrmodus, unterschiedlich zu den festeingestellten Lenkanschlägen aus Tabelle 4.7 von Seite 38 sein. Daher bilden die festeingestellten Lenkanschläge nur die Basis für den maximalen und minimalen Lenkwinkel. Diese betragen $\pm 32767$, was einem Lankradwinkel von $\pm 294°$ entspricht. Der Wert des Lenkwinkels wird einer Applikation mit der Struktur $js\_event$, über den $joydev$-Treiber, bereitgestellt. Das vorzeichenbehaftete 16 Bit-Feld $value$ enthält dabei den Lenkwinkel des Lenkrades. Somit bildet die Basis den größtmöglichen Wert, den der $joydev$-Treiber verarbeiten kann (vgl. Espinosa, 1998) . Im Fahrmodus $Cruise$ sind die Lenkanschläge bei $\pm 540°$ definiert, im Fahrmodus $Sport$ bei $\pm 180°$. Die äquivalenten Werte sind in Abbildung 4.9 zu sehen.

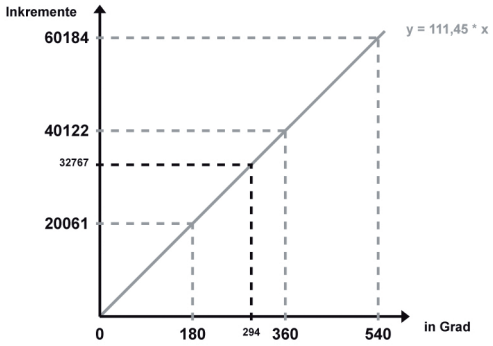


Abbildung 4.9: Maximale und Minimale Lenkanschläge

Wie man erkennen kann, übersteigt der Wert des Lenkwinkels, im $Cruise$-Modus, die Basis und damit den Wertebereich. Um beliebige Lenkwinkel zu unterstützen, werden diese, anhand der Basis, skaliert. Dies geschieht, sobald ein neuer Lenkwinkel an den Joystick-Treiber übertragen wurde und bevor dieser an den $joydev$-Treiber übergeben wird.

### Bereitstellen des Lenkwinkels

Die Funktion $process\_angle$ hat die Aufgabe den aktuellen Lenkwinkel an den Joydev-Treiber weiterzuleiten. Wie schon angemerkt, muss der Lenkwinkel, je nach Fahrmodus, skaliert werden, da die Endanschläge in der Initialisierungsphase statisch festgelegt werden und nicht mit der Anzahl der Inkremente der verschiedenen Fahrmodi übereinstimmen. Dafür werden der Winkel und die Inkremente mit der folgenden Funktion angepasst.

$$\frac{\text{Basis}}{\text{Endanschlag in Inkrementen}} * \text{Inkremente} = \text{skalierte Inkremente}$$

Falls später weitere Fahrmodi hinzukommen sollten, ist gewährleistet, dass die neuen Winkel unterstützt werden.
Um den Winkel einer Applikation bekannt zu machen, muss dieser an das Input-Subsystem übergeben werden. Dafür nutzt der Joystick-Treiber die Funktion $input\_report\_abs$. Der alte Wert wird damit aber nicht automatisch aktualisiert. Erst über die Funktion $input\_sync$, wird dem Input-Subsystem mitgeteilt, dass die Daten übernommen werden sollen.

### Steuerung und Verarbeitung von Force-Feedback-Effekten

Linux unterscheidet diverse Arten von Force-Feedback-Effekten, die selbst noch über Unterarten verfügen können (vgl. Deneux und Hannula, 2001) . Für die Wahl der zu implementierenden Effekte, war ausschlaggebend, welche Force-Feedback-Effekte die Fahrsimulationen für Linux unterstützen und bei welchen es Sinn macht, sie auf ein Lenkrad anzuwenden. Dabei hat sich herausgestellt, dass die meißten Fahrsimulatoren nur den $FF\_CONSTANT$-Effekt unterstützen. Diesen Effekt, kann man auch auf ein Lenkrad anwenden. Daher wurde nur dieser Effekt implementiert.
Der Effekt lässt eine konstante Kraft in eine bestimmte Richtung wirken. Allerdings ist die Angabe der Richtung nur relevant, wenn man einen zweidimensionalen Raum betrachtet. Das Lenkrad verfügt nur über eine Achse, was bedeutet, dass die Angabe der Richtung hier keinen Einfluss hat. Somit ist nur die Stärke und dessen Vorzeichen relevant für das Lenkrad, was in Abbildung 4.10 verdeutlicht wird.

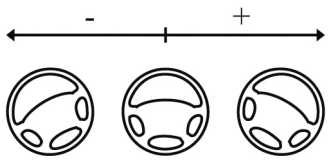


Abbildung 4.10: Auswirkung der Kraft auf das Lenkrad

Es hat sich auch herausgestellt, dass die Richtung von keinem der eingesetzten Fahrsimulatoren verwendet wird, wenn ein Lenkrad als Eingabegerät angeschlossen ist.
Der Aufbau eines Effektes wird durch die Struktur $ff\_effect$ definiert, diese ist in Tabelle 4.10 zu sehen, wobei hier nur die verwendeten Felder dargestellt sind.

| Field | | Comment |
|---|---|---|
| type | | enthält den Effekttyp |
| id | | identifiziert einen Effekt |
| replay | length | enthält Effektdauer |
| | delay | enthält Startverzögerung |
| constant | level | enthält die Kraft |
| | envelope | Ein- und Ausblenden des Effektes |

Tabelle 4.10: Auszug aus der Struktur $ff\_effect$

Das Typfeld enthält dabei die Art des Effektes. Somit kann man, wenn man mehrere Effekte unterstützt, unterscheiden, um welche Art Effekt es sich handelt. Unterstützt man die gleich-

zeitige Aktivität von Effekten, kann man diese anhand des $id$-Feldes unterscheiden. Da bei der Initialisierung des Joystick-Treibers angegeben wurde, dass nur eine Art Effekt unterstützt wird und nie mehr als ein Effekt gleichzeitig aktiv sein darf, sind die Felder hier von geringerer Bedeutung. Das Feld $replay$ gibt an, wie lange ein Effekt aktiv sein und ob dieser verzögert gespielt werden soll.
Da jeder Effekt andere Einstellungsmöglichkeiten bietet, gibt es auch für jede Art von Effekt ein eigenes Feld in der $ff\_effect$-Struktur. Das Feld für den konstanten Effekt enthält, neben einem Feld für die Stärke des Effektes, auch ein Feld für einen Effektumschlag. Mit diesem ist es möglich, Effekte, ähnlich wie mit einem Lautstärkeregler, ein- und ausblenden zu lassen. Es hat sich gezeigt, dass kein, für die Arbeit verwendeter, Fahrsimulator, diese Eigenschaft nutzt. Daher wurde bei der Implementierung auf diese Eigenschaft verzichtet.
Wie schon erwähnt wurde, besitzt die Bridge selbst keine Logik. Das soll auch mit Blick auf das Force-Feedback so bleiben. Daher kümmert sich allein der Joystick-Treiber um die Steuerung der Effekte und das Aktualisieren der Zustände. Die Bridge hat lediglich eine Sicherheitsfunktion implementiert, die dafür sorgt, dass die Effekte nicht unendlich lang anliegen können.
Eine Besonderheit ist, dass die Effekte nur in einem bestimmten Zeitabstand zueinander an die Bridge gesendet werden. Der Grund dafür ist einmal, dass der Force-Feedback-Effekt vom Rad des Demonstrators auch nur zyklisch aktualisiert wird. Zum anderen hat sich gezeigt, dass wenn die serielle Schnittstelle der Bridge ununterbrochen Nachrichten verarbeiten muss, die Bridge dadurch sehr belastet wird. Das kann dazu führen, dass sie kurzzeitig in einen unsynchronisierten Zustand übergeht, da die Synchronisationsnachrichten nicht verarbeitet werden können. Der Zeitabstand zwischen den Nachrichten orientiert sich dabei am Zyklus des Demonstrators. Dieser liegt, wie bereits beschrieben, bei $5ms$.
Damit der Joystick-Treiber nicht jedes Mal angepasst werden muss, nur weil sich die Zykluszeit geändert hat, wird dieser dynamisch ermittelt. Um festzustellen, wie lang ein Zyklus ist, wird der Zeitabstand zwischen zwei eingehenden Protokollnachrichten mit gleicher ID gemessen. Für die Zeitmessung stellt der Linux-Kernel verschiedene Funktionen bereit. Alle haben aber gemeinsam, dass das Auftreten von Unterbrechungen, durch Interrupts, den Scheduler oder anderen Ursachen, zwischen zwei Messpunkten, die Messungen verfälschen können. Um den Fehler zu minimieren, wird daher eine Messreihe aufgenommen und aus dieser der Mittelwert bestimmt. Da die Berechnung innerhalb einer ISR durchgeführt wird, können nicht beliebig viele Werte herangezogen werden.
Zur Zeitmessung wird die Funktion $getnstimeofday$ genutzt. Sie liefert die Zeit, die seit dem 01.01.1970 vergangen ist, in Sekunden und Nanosekunden. Somit hat man eine ausreichend große zeitliche Auflösung, um den Zyklus zu bestimmen. Dafür wird der Abstand zweier

Protokollnachrichten in Nanosekunden gespeichert. Die letzten 10 Messungen sind in einem Ringpuffer hinterlegt, um über diese den Durschnitt zu errechnen.
Mit dieser Maßnahme kann gewährleistet werden, dass die Bridge nicht ununterbrochen mit seriellen Nachrichten konfrontiert wird. Im Gegensatz zu TT-Nachrichten im TTE-Netzwerk, werden auch keine Nachrichten gesendet, wenn keine neuen Force-Feedback-Daten hochgeladen wurden.
Es hat sich im laufenden Betrieb gezeigt, dass die Effekte eher in unregelmäßigen Abständen auftreten und deren Aktualisierungsrate geringer ist, als ein Zyklus. Daher wird auch kein Kernel-Thread dazu verwendet, die Effekte in regelmäßigen Abständen zu aktualisieren. Vielmehr wird solange keine Aktualisierung eines Effektes durch eine Applikation zugelassen, bis sichergestellt ist, dass mindestens ein Zyklus im TTE-Netzwerk vergangen ist.

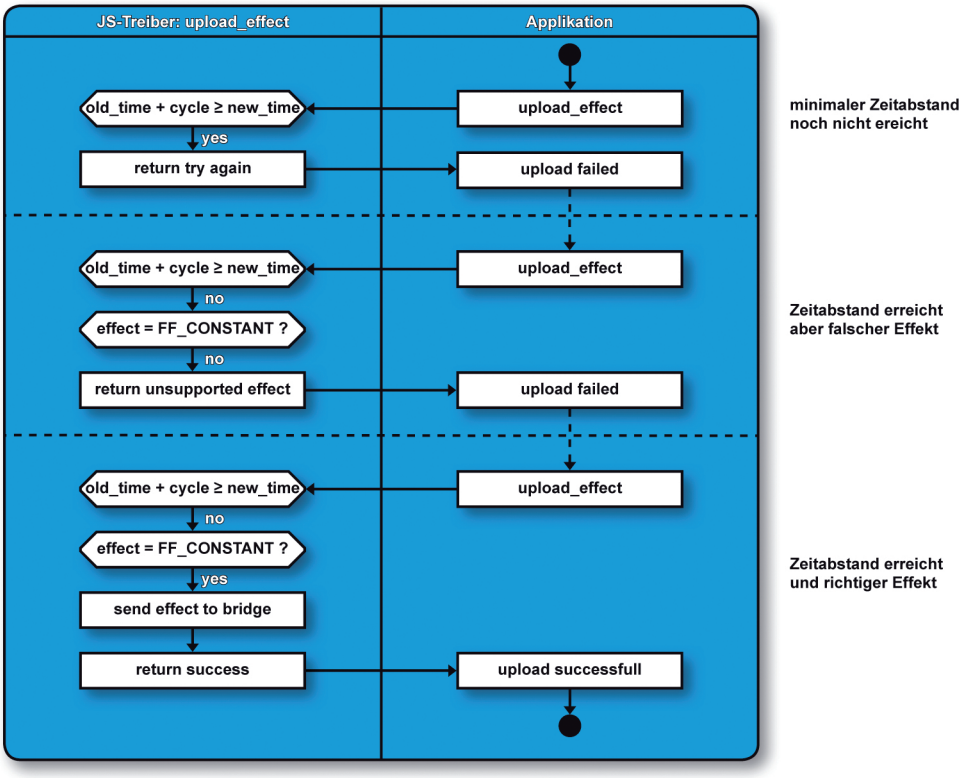


Abbildung 4.11: Mögliche Abläufe bei der Aktualisierung eines Force-Feedback-Effektes zwischen Joystick-Treiber und Applikation

Damit wird sichergestellt, dass nicht zwei Effekte in einem Zyklus gesendet werden. Was, wie schon bemerkt, unter anderem die Bridge entlastet. Es wird damit aber auch erreicht, dass der Applikation mitgeteilt werden kann, dass ihr Effekt nicht aktualisiert worden ist und sie

es noch einmal versuchen soll. Die drei möglichen Szenarien, die beim Aktualisieren eines Effektes auftreten können, sind in Abbildung 4.11 dargestellt.
In Tabelle 4.8 ist zu sehen, welche Funktionen vom Joystick-Treiber für die Force-Feedback-Unterstützung implementiert wurden. Es gibt noch weitere Funktionen, die aber nicht relevant für die Funktionstüchtigkeit der Force-Feedback-Schnittstelle sind.
Im Normalfall wird erst ein Effekt über die Funktion *upload* hochgeladen und dann über die Funktion *playback* gestartet. Dies kann aber auch in umgekehrter Reihenfolge ablaufen. Ist ein Effekt hochgeladen und gestartet worden, läuft dieser solange, bis die Zeit, welche im *replay*-Feld festgelegt wurde, abgelaufen ist oder die Bridge ihn selbst beendet. Es kann auch dazu kommen, dass zwischenzeitlich ein neuer Effekt hochgeladen wird. Dieser überschreibt dann den aktuellen Effekt und aktualisiert damit auch die *replay*-Zeit. Die möglichen Abläufe, um einen Effekt hochzuladen, zu spielen und zu stoppen, sind in den Abbildungen 4.12 und 4.13 dargestellt.
Die Funktion *playback* setzt nur ein Flag, welches signalisiert, dass die Effekte an die Bridge gesendet werden dürfen oder nicht. Wartet schon ein Effekt auf seine Ausführung und wurde das Flag auf play gesetzt, wird dieser umgehend an die Bridge gesendet.
In Abbildung 4.12 wurde veranschaulicht, was passiert, sollte die Applikation erst das Flag setzten und dann einen Effekt hochladen.

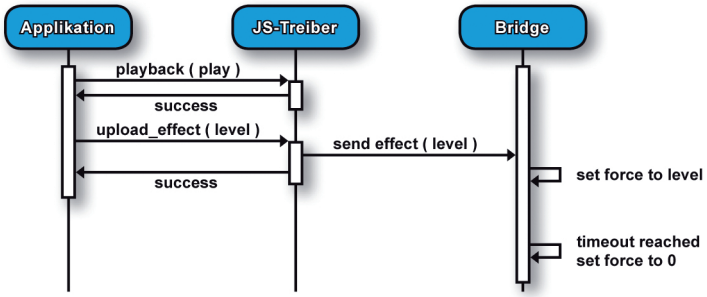


Abbildung 4.12: Force-Feedback wird nicht gelöscht

Zusätzlich ist zu sehen, wie sich die Bridge verhält, sollte ein Effekt nie mehr gelöscht oder angehalten werden. Sie setzt dann selbst die Kraft auf oN. Läuft die *replay*-Zeit vorher ab, setzt der Joystick-Treiber den Effekt zurück. Während der Implementation hat sich gezeigt, dass manche Fahrsimulatoren die *replay*-Zeit durchgehend auf das Maximum eingestellt haben. Daher ist es wichtig, dass die Bridge diesen Selbstschutz implementiert hat.
Der normale Ablauf ist dagegen in Abbildung 4.13 dargestellt.

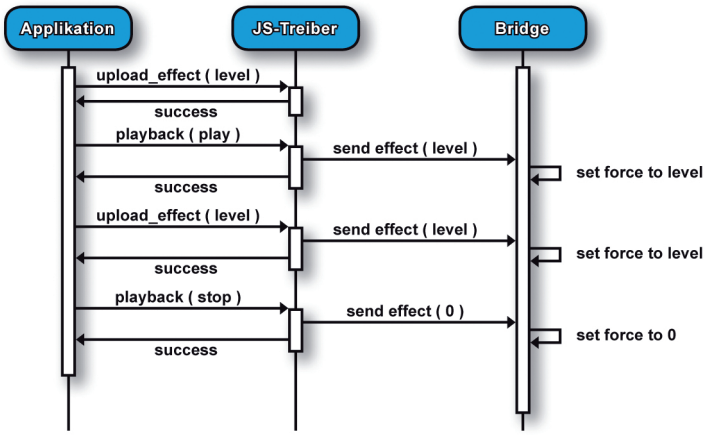


Abbildung 4.13: Force-Feedback wird aktualisiert und gelöscht

Die Funktion *erase*, die nicht abgebildet ist, setzt die Stärke eines Effektes auf oN und schickt diese Änderung umgehend, ohne auf den Zyklus zu achten, an die Bridge. Gibt es keinen Effekt der gelöscht werden soll, schickt sie nichts.

### 4.1.5 Ergänzungen und Inbetriebnahme des Systems

Nun kann man das Lenkrad des Demonstrators als Eingabegerät verwenden. Der Demonstrator selbst verfügt aber über keine Möglichkeit, um mit den Füßen die Geschwindigkeit zu regeln. Daher wurde das Pedalset „CSR Elite Pedals EU“ von der Firma Fanatec gekauft, welches über USB an den PC angeschlossen werden kann. Dieses Modell wurde mit Blick auf eine hohe Belastbarkeit ausgewählt.
Um den kompletten Aufbau flexibel zu halten, wurde zusätzlich ein PC gekauft, der leicht zu transportieren ist. Da Laptops mit aktueller Hardware sehr teuer sind, wurde ein tragbarer PC zusammengestellt. Dieser ist Leistungsstark genug, um die Fahrsimulationen in ansprechender Grafik darstellen zu können. In Tabelle 4.11 sind die eingesetzten Komponenten des PCs aufgelistet.

| | |
|---|---|
| **CPU** | Intel Core i5 @ 3,2 GHz |
| **RAM** | 8GB DDR3 |
| **Graphics card** | AMD HD6570 |
| **Operating system** | Ubuntu 12.04 - Kernel 3.2.38 |

Tabelle 4.11: Typ der Callback-Function für die RS-232-Schnittstelle

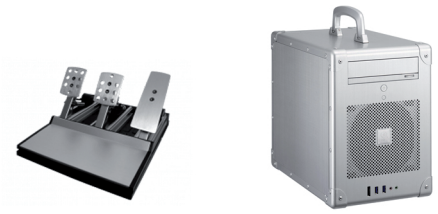

Abbildung 4.14: Pedale und PC für den Demonstrator

Um den Demonstrator und den PC in Betrieb zu nehmen, muss der Demonstrator, bis auf die Scheinwerfer und das Infotainment, komplett aufgebaut und über die serielle Schnittstelle der Bridge mit dem PC verbunden sein. Es ist dabei egal, in welcher Reihenfolge die Komponenten initialisiert werden.
Den Joystick-Treiber kann man einfach mit dem Befehl $insmod$ laden. Wichtig ist, dass man anschließend, mit dem angepassten $Inputattach$-Programm, das Binden an einen seriellen Port durchführt. Sollte der Joystick-Treiber auf einem anderen System, mit einer anderen Kernelversion, eingesetzt werden, kann man ihn mit dem beiligenden Makefile für diese Kernelversion kompilieren.
Damit ist die Inbetriebnahme abgeschlossen und der Demonstrator kann als Eingabegerät genutzt werden.

# 5 Qualitätssicherung

Es muss nun festgestellt werden, ob die gestellten Anforderungen erfüllt werden. Dazu wird als erstes die Funktionsfähigkeit des Joystick-Treibes validiert. Anschließend wird überprüft, ob die Datenübertragung über die RS-232-Schnittstelle und die Datenverarbeitung auf der Bridge und im Joystick-Treiber, keine zu große Latenz aufweisen. Das bedeutet, dass zwischen Lenken am Lenkrad und der Reaktion in der Fahrsimulation, bzw. zwischen dem Auftreten des Force-Feedback-Effekts in der Fahrsimulation und der Wirkung auf das Lenkrad, kein erkennbarer Verzögerungseffekt auftreten darf.

## 5.1 Verifizierung der Funktionsfähigkeit

Das Linuxprogramm $jstest$ kann dazu verwendet werden, festzustellen, ob der Joystick-Treiber funktioniert und die skalierten Lenkwinkel richtig abgebildet werden. Dazu ermittelt es welche Eigenschaften das Eingabegerät hat und wie dessen aktueller Zustand ist. Abbildung 5.1 zeigt das Programm mit den ermittelten Daten über den Joystick-Treiber.

```
aav300@aav300-haw:~/Dokumente/git/serial_demonstrator$ jstest /dev/input/js0
Driver version is 2.1.0.
Joystick (Demonstrator) has 1 axes (Wheel)
and 0 buttons ().
Testing ... (interrupt to exit)
Axes:  0:     0
```

Abbildung 5.1: Das Programm $jstest$

Damit kann validiert werden, dass die in der Initialisierungsphase des Joystick-Treibers vorgenommenen Einstellungen und auch die implementierten Funktionen, korrekt sind. Es zeigt an, dass es einen Joystick erkannt hat der vom Typ $Wheel$ ist und, dass dieser nur über eine Achse und keine Buttons verfügt. In Abbildung 5.2 sind die ermittelten Daten für die Lenkwinkel dargestellt. Wie zuvor beschrieben wurde, wird der Lenkwinkel immer mit der Basis von 32767 Inkremente skaliert. Je kleiner der Winkel, was weniger Inkremente bedeutet, desto eher wird das Maximum erreicht. Anhand der Grafik erkennt man, dass die Inkremente richtig skaliert wurden.

Abbildung 5.2: Ermittelte Inkremente in Abhängigkeit vom Winkel

Um die Force-Feedback-Schnittstelle zu testen, bietet Linux, unter anderem, die Programme $fftest$ und $ffcfstress$ an. Das Programm $fftest$ ermittelt alle unterstützten Effekte eines Gerätes. In Abbildung 5.3 sind die ermittelten Einstellungen des Joystick-Treibers zu sehen. Als einziger Effekt, wurde der $Constant$-Effekt ermittelt. Auch die Anzahl der gleichzeitig anliegenden Effekte wurde richtig eingestellt.
Das Programm zeigt auch an, welche Effekte ein Treiber nicht unterstützt. Dies wird über die Mitteilung „Invalid argument “ angezeigt. Alle erimttelten Effekte lassen sich über die Auswahl einer Ziffer, in diesem Fall die 1 für einen konstanten Effekt, auf das Gerät anwenden. Somit konnte also überprüft werden, dass der Effekt vom Gerät unterstützt wird.

```
aav300@aav300-haw:~/Dokumente/git/serial_demonstrator$ sudo fftest /dev/input/event12
Force feedback test program.
HOLD FIRMLY YOUR WHEEL OR JOYSTICK TO PREVENT DAMAGES

Device /dev/input/event12 opened
Axes query: Wheel
Effects: Constant
Number of simultaneous effects: 1
Upload effects[0]: Invalid argument
Upload effects[2]: Invalid argument
Upload effects[3]: Invalid argument
Upload effects[4]: Invalid argument
Upload effects[5]: Invalid argument
Enter effect number, -1 to exit
1
Now Playing: Constant Force
Enter effect number, -1 to exit
```

Abbildung 5.3: Das Programm $fftest$

Um zu testen was passiert, wenn sich die Kraft, die auf das Lenkrad wirkt, ständig in ihrer Stärke und Richtung ändert, kann das Programm $ffcfstress$ verwendet werden. Dieses lässt eine, sich ständig in ihrer Stärke und Richtung ändernde, Kraft, auf das Gerät wirken. Die Aktualisierungsrate des Effektes kann dabei selbst festgelegt werden.
So kann auch simuliert werden, was passiert, wenn die Aktualisierungsrate unter die Zykluszeit sinkt. Wie erwartet, werden die Anfragen solange abgewiesen, bis mindestens ein Zyklus vergangen ist. In Abbildung 5.4 ist zu sehen, welche Auswirkungen die Kraft auf das Lenkrad hat.
Es werden dazu die Position des Lenkrades, und die anliegenden Kraft angezeigt. Der Wert $center$ ist für den Test des Joystick-Treibers nicht von Bedeutung. Anhand der Sternchen wird gezeigt, wie Stark die Kraft ist und in welche Richtung sie wirkt. Die Sternchen der Position

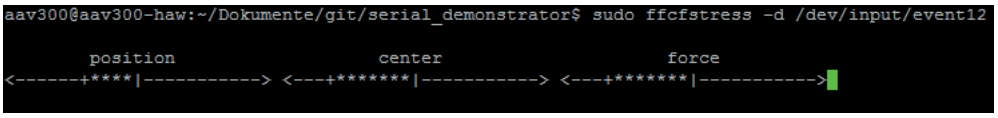


Abbildung 5.4: Das Programm $ffcfstress$

geben die Position des Lenkrades in Abhängigkeit zu den Inkrementen an. Man kann erkennen, das die Kraft das Lenkrad dazu gbracht hat nach links auszuschlagen.

## 5.2 Validierung der Kommunikationsstrecke

Nachdem die Funktionstüchtigkeit des Joystick-Treibers festgestellt wurde, muss nun die zeitliche Verzögerung, zwischen Lenken am Lenkrad des Demonstrators und Lenkbewegung in der Fahrsimulation, festgestellt werden. Dazu müsste man die komplette Strecke, vom eintreffen der CAN-Nachricht mit dem Lenkwinkel an der Bridge, bis zum $input\_sync$-Befehl im Joystick-Treiber, messen können. Das größte Problem dabei ist, dass es, durch die verschiedenen Systeme, keine gemeinsame Sicht auf die Zeit gibt. In Abbildung 5.5 ist die Messstrecke zu sehen.

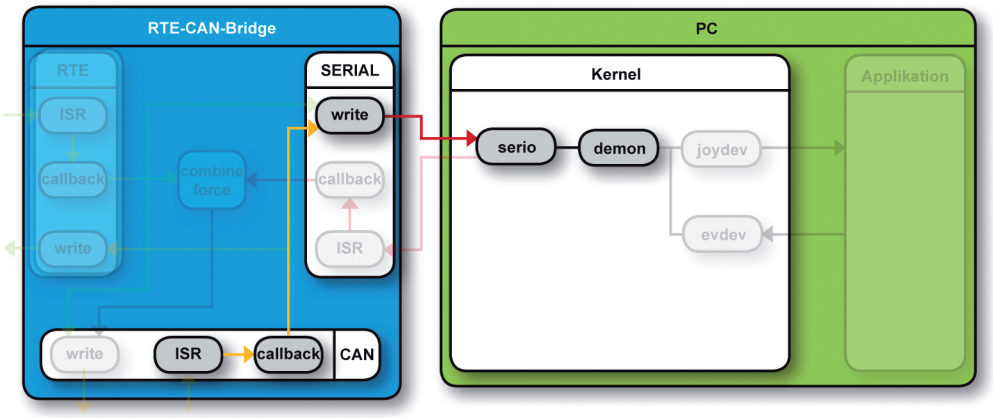


Abbildung 5.5: Messstrecke

Der Zeitpunkt des Eintreffens der CAN- oder TTE-Nachricht, kann nur auf der Bridge gemessen werden. Die Aktualisierung des Winkels oder der Endanschläge, kann man nur im Joystick-Treiber ermitteln. Somit könnte man nur feststellen, ob die Zeitverzögerung der kompletten Verarbeitungskette, konstant bleibt. Das könnte man feststellen, indem man beobachtet, ob die zeitliche Veränderung in beiden Messungen übereinstimmt. Man kann aber keine Aussage über die Bearbeitungszeit machen.

Daher werden die einzelnen Systeme für sich analysiert. Zuerst wird geprüft, ob die Änderungen an den Routinen, die für das Tunneln der Nachrichten zuständig sind, eine Auswirkung auf das Zeitverhalten der Bridge haben. Danach wird analysiert, wie genau die Berechnung des Zyklus im Joystick-Treiber ist. Zum Schluss wird gemessen, wie lange die Verarbeitung einer Protokollnachricht im Joystick-Treiber dauert.
Damit kann zwar keine Aussage über die Verarbeitungszeit der gesamten Strecke gemacht werden, da immernoch unbekannte Faktoren, wie der serielle Low-Level-Treiber von Linux, enthalten sind, aber zumindest kann gezeigt werden, dass die eigene Implementation die zeitlichen Anforderungen erfüllt.

### 5.2.1 Validierung der Verarbeitungszeiten auf der Bridge

Es wurden in weiteren Arbeiten bereits diverse Messungen zum Latenzverhalten der Bridge durchgeführt (vgl. Kamieth, 2011) , (vgl. Müller, 2011) . Dort wurde unter anderem ermittelt, wie lang die Bridge für die Tunnelung der Nachrichten von CAN auf RTE und umgekehrt benötigt. Jetzt muss noch ermittelt werden, wie sich die Einbindung der RS-232-Schnittstelle, auf das Verhalten der Bridge, auswirkt, da die Nachrichten nun über eine zusätzliche Schnittstelle getunnelt werden.
In Abbildung 5.6 ist zu sehen, dass die Ausführungszeit der CAN-Routine, die die CAN-Nachrichten in TT-Nachrichten und nun auch in serielle Nachrichten umwandelt, $\approx 43ms$ beträgt.
Die in der Arbeit von Jan Kamieth gemessene Ausführungszeit lag, ohne die Tunnelung auf RS-232, zwischen $35ms$ und $45ms$. Somit hat sich die Ausführungszeit der Routine kaum verändert. Sie liegt jetzt immer im Bereich zwischen $40ms$ und $45ms$.
Das Gleiche gilt für die TTE-Routine, die die TT-Nachrichten in CAN-Nachrichten und nun auch in serielle Nachrichten umwandelt. Diese braucht nun, wie in Abbildung 5.7 zu sehen ist, $\approx 23ms$, was keine Veränderung zu den vorher gemessenen Werten ist, die bei $20ms$ bis $30ms$ lagen.
Alle Werte wurden im laufenden Betrieb des Demonstrators gemessen. Wobei alle Messungen auf der Bridge des Lenkrades durchgeführt wurden.

### 5.2.2 Validierung der Verarbeitungszeiten des Joystick-Treibers

Bei den Messungen auf dem PC ist darauf zu achten, dass das Linux-Betriebssystem wesentlich komplexer ist, als das Programm der Bridge. Der Scheduler des Betriebssystems, sowie auftretende Interrupts oder weitere Faktoren, können die Messdaten verfälschen. Die Sys-

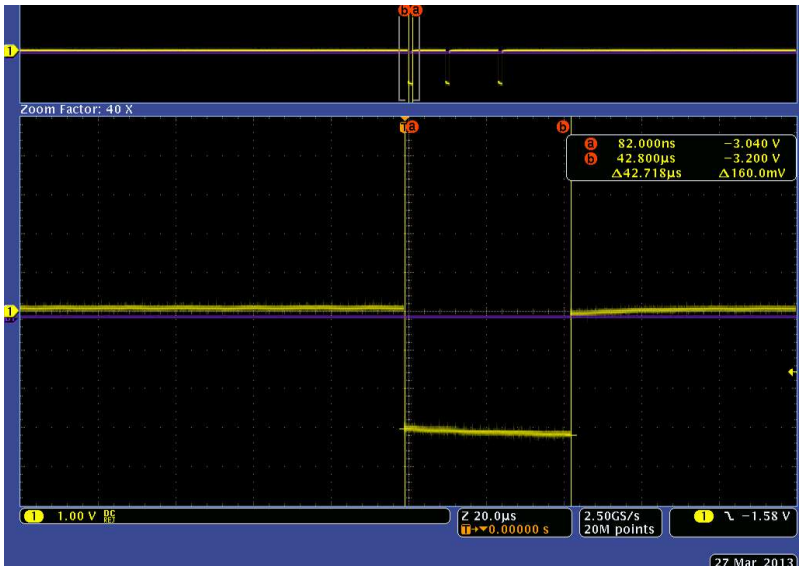


Abbildung 5.6: Ausführungszeit der CAN-Routine

temspezifikation des PCs, auf dem die Messungen durchgeführt wurden, sind in Tabelle 5.1 zu sehen.

| | |
|---|---|
| **CPU** | Intel Core i5 - 3,2 GHz |
| **RAM** | 8GB DDR3 |
| **Operating system** | Ubuntu 12.04 - Kernel 3.2.38 |

Tabelle 5.1: Systemspezifikation

In der Umsetzung wurde beschrieben, dass die Zykluszeit vom Joystick-Treiber dynamisch ermittelt wird. Um festzustellen, ob man mit den gemessenen Werten überhaupt eine Aussage über die Zykluszeit treffen kann, wurde eine Messreihe von 1000 Werten aufgezeichnet. Anhand der Daten dieser Messreihe, die in Abbildung 5.8 dargestellt ist, kann man erkennen, dass es nur kleinere Schwankungen in der Genauigkeit gibt, die dann durch die Durchschnittsberechnung aufgefangen werden. Die blauen Markierungen sind die gemessenen Zykluszeiten zwischen zwei Protokollnachrichten. Die roten Markierungen stellen die errechneten Mittelwerte über die jeweils letzten 10 Messpunkte dar, wobei diese auf Millisekunden, ohne Nachkommastelle, gerundet wurden. Es ist zu erkennen, dass der Zyklus genau ermittelt werden kann, solange sich dieser mindestens im Millisekundenbereich befindet.
Um herauszubekommen, ob die Verarbeitung der Daten, vom Lesen der seriellen Nachricht, bis zum Aktualisieren der Endanschläge oder des Winkels, eine Auswirkung auf die gesam-

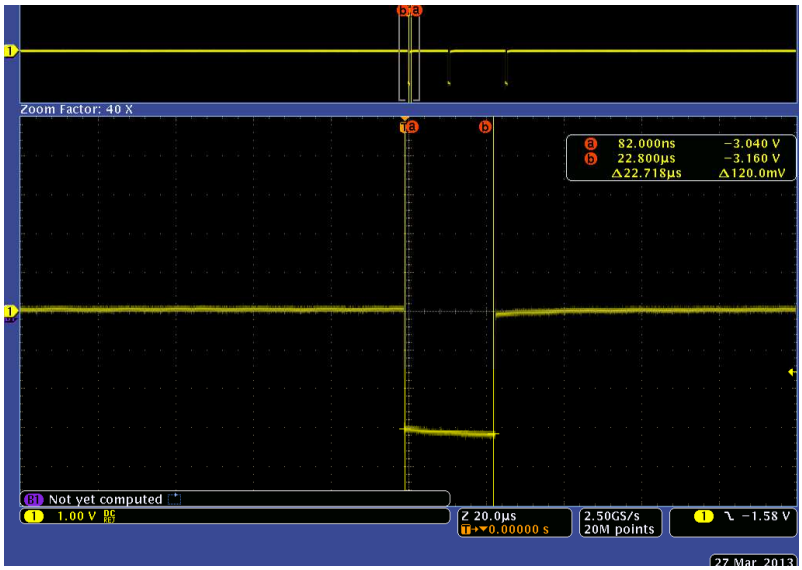


Abbildung 5.7: Ausführungszeit der RTE-Routine

te Latenzzeit hat, wurde eine weitere Messreihe über 2000 Messungen aufgenommen. Die Messwerte sind in Abbildung 5.9 dargestellt. Dazu wurde die Zeit gemessen, die vergeht, wenn die Nachricht über den Lenkwinkel vom Lenkrad, verarbeitet wird. Diese hat die längste Verabeitungskette. Gemessen wurde die Zeit vom Eintreffen der Protokollpräambel, bis zum Synchronisationsbefehl, der den Lenkwinkel aktualisiert.
Dabei ist zu erkennen, dass die Verarbeitungszeit sehr schwankt aber ausreichend klein bleibt. Die Maximalwerte liegen zwichen $1ms$ und $1,1ms$, die Minimalwerte zwischen $400ns$ und $2000ns$. Man kann gut erkennen, wie stark der Scheduler des Betriebssystems und andere Faktoren, die Laufzeiten beeinflussen. Mithilfe der Echtzeiterweiterung für Linux kann man der ISR, die die Nachricht einliest und zum Schluss die Synchronisation anstößt, eine so hohe Priorität zuweisen, dass die Verarbeitungszeit immer im Bereich von $400ns$ bis $2000ns$ liegt.

### 5.2.3 Schlussbetrachtung

Somit wurde nachgewiesen, dass die implementierten Funktionen schnell genug sind. Nicht gemessen werden konnte, wie lange der serielle Low-Level-Treiber von Linux braucht, um die Nachricht einzulesen und sie an den Joystick-Treiber weiterzugeben. Man kann aber davon ausgehen, dass die Verarbeitungszeit sehr gering ist.
Es wurde nachgewiesen, dass die Verarbeitungszeiten der Routinen, auf der Bridge, sehr gering sind. Somit wird die maximale Latenz der Bridge vom Zyklus bestimmt. Ausgehend davon,

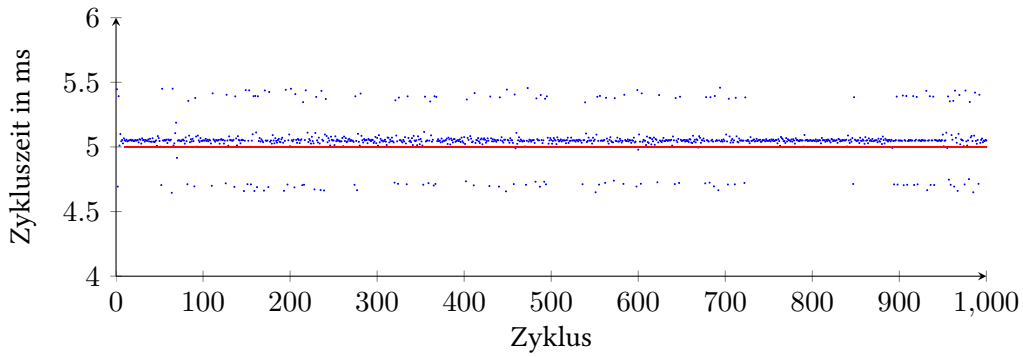


Abbildung 5.8: Messreihe zum Bestimmen des Zyklus im Joystick-Treiber

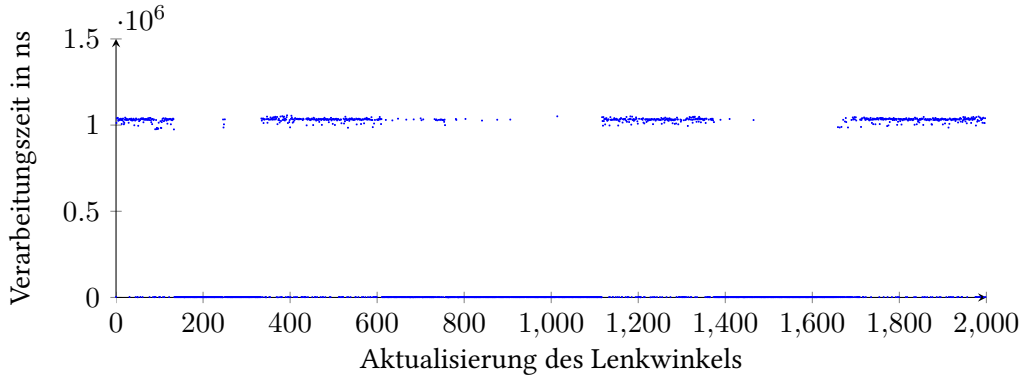


Abbildung 5.9: Messreihe zum Lesen einer Protokollnachricht

dass zwischen dem Eintreffen der CAN-Nachricht mit dem Lenkwinkel und dem Versenden der zugehörigen Protokollnachricht, ein kompletter Zyklus liegt, beträgt die maximale Latenzzeit der Bridge zur Zeit $TB = 5ms$. Um eine Nachricht, die 80 Bit groß ist, über die RS-232-Schnittstelle zu übertragen, braucht man, bei einer Geschwindigkeit von $115200Baud$, $TS \approx 0,7ms$, wie in Rechnung 5.1 dargestellt ist. Der Joystick-Treiber braucht für die Verarbeitung des Lenkwinkels maximal $TJ = 1,1ms$. Das bedeutet, dass wenn man die Verarbeitungszeit des seriellen Standardtreibers von Linux außen unbetrachtet lässt, die gesamte Verarbeitungszeit $T_{gesamt} \approx 6,8ms$ beträgt, was in Rechunung 5.3 zu sehen ist.

$$\frac{80Bit}{115,2\frac{Bit}{ms}} = 0,69\overline{44}ms \tag{5.1}$$

$$TB + TS + TJ = T_{gesamt} \tag{5.2}$$

$$5ms + (\approx 0,7)ms + 1,1ms \approx 6,8ms \tag{5.3}$$

Diese Latenz ist gering genug, so dass kein Verzögerungseffekt entstehen kann. Dies hat sich während des Betriebes auch so wiedergespiegelt.

# 6 Fazit

Diese Arbeit entstand mit der Intention, den Demonstrator mit einer Fahrsimulation auf einem PC zu verbinden. Dazu musste sich in neue Themengebiete, wie das Time-Triggered-Ethernet und die Treiberprogrammierung unter Linux, eingearbeitet werden. Das Ziel wurde innerhalb dieser Arbeit erreicht, so dass das Lenkrad des Demonstrators, als Eingabegerät unter Linux, genutzt werden kann.

## 6.1 Zusammenfassung

Für die Umsetzung musste analysiert werden, wie der Demonstrator aufgebaut ist und welche Schnittstellen er für eine Anbindung zur Verfügung stellt. Daher wurden in den Grundlagen die Themen „Echtzeit“ und „Time-Triggered-Ethernet“ besprochen sowie ein Überblick über den Demonstrator gegeben. Anschließend wurde die Seite des PCs betrachtet. Anhand der zur Verfügung stehenden Anbindungsmöglichkeiten, wurde als erstes das Betriebssystem gewählt. Da die Wahl auf Linux gefallen ist, wurde in den Grundlagen das Thema „Linux - Kernel und Gerätetreiber“ aufgegriffen.
In der Anforderungsanalyse wurden zwei Konzepte vorgestellt. Das erste Konzept beschreibt die direkte Anbindung über einen TTE-Switch, welche einen funktionierenden TTE-Protokollstack auf dem PC vorraussetzt. Im zweiten Konzept, wurde die indirekte Anbindung über die RS-232-Schnittstelle der RTE-CAN-Bridge, vorgestellt. Daher wurde in den Grundlagen kurz auf die serielle Kommunikation eingegangen. In der Umsetzungsphase stellte sich heraus, dass es keinen TTE-Protokollstack für den Linux-Kernel 3.X gibt.
Somit wurde das Konzept mit der indirekten Anbindung umgesetzt. Dafür wurde ein Arbeitsplan erstellt und dessen Punkte, Schritt für Schritt, abgearbeitet. Zuerst wurde die RTE-CAN-Bridge dahingehend erweitert, dass sie nun auch von der RS-232-Schnittstelle lesen kann. Erst dadurch wurde eine Kommunikation in beide Richtungen möglich. Der zweite Schritt bestand in der Erweiterung der Routinen, die auf der RTE-CAN-Bridge, das Tunneln der TT- bzw. CAN-Nachrichten, übernehmen. Diese wurden so erweitert, dass sie nun auch serielle Nachrichten verarbeiten können. Zu dem wurden die beiden Force-Feedback-Effekte, vom

Rad und vom PC, miteinander kombiniert. Der dritte Schritt beschreibt die Entwicklung des Joystick-Treibers.
Um Zugriff auf die serielle Schnittstelle zu bekommen, registriert sich der Joystick-Treiber beim seriellen Subsystem. Die Einstellungen für die serielle Kommunikation, führt das externe Programm $Inputattach$ durch. Dieses wurde dahingehend erweitert, dass der Joystick-Treiber mit diesem, an einen seriellen Port gebunden werden kann. Anschließend wurde die Applikationsschnittstelle implementiert. Dafür wurde das Input-Subsystem verwendet, was sich automatisch um das Anlegen der Geräteknoten kümmert. Im letzten Arbeitsschritt wurde die Inbetriebnahme beschrieben, und welche weiteren Komponenten zur Steuerung der Fahrsimulation eingesetzt werden.
In der Qualitätssicherung wurden die Funktionen, den gestellten Anforderungen, gegenübergestellt. Es wurde gezeigt, dass der Joystick-Treiber voll funktionsfähig ist und auch die zeitlichen Anforderungen erfüllt werden.

## 6.2 Ausblick

Der Joystick-Treiber wurde so entwickelt, dass ein Wechsel der Kommunikationsschnittstelle möglich ist. Wenn der TTE-Protokollstack der CoRE-Projektgruppe fertiggestellt ist, kann dieser die serielle Schnittstelle ersetzen. Das ist dadurch möglich, weil der Joystick-Treiber so konzipiert wurde, dass er auf die zyklische Verarbeitung der Daten vorbereitet ist.
Wie schon angesprochen, könnte die serielle Schnittstelle nun auch dafür genutzt werden, um der Bridge Daten zu senden, die zu Testzwecken genutzt werden können. So könnte man die Bridge, während des Betriebes, dynamisch in verschiedene Zustände überführen, um etwa den Verlust von Nachrichten zu simulieren, oder Nachrichten zu manipulieren.

# 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

[BURKHARD 1994] BURKHARD, Kainka: *Messen, Steuern, Regeln über die RS-232-Schnittstelle : Meẗdatenerfassung und Prozeẗsteuerung mit dem PC.* Germany : Franzis, 1994. – ISBN 3-7723-6057-2

[Deneux 2011] Deneux, Johann: *Iforce Protocol.* Juli 2011. – URL `https://www.kernel.org/doc/Documentation/input/iforce-protocol.txt`. – Zugriffsdatum: 2012-09-03

[Deneux und Hannula 2001] Deneux, Johann ; Hannula, Anssi: *Force Feedback for Linux.* April 2001. – URL `https://www.kernel.org/doc/Documentation/input/ff.txt`. – Zugriffsdatum: 2012-10-25

[Döbler 2008] Döbler, Thomas: Simulation und Visualisierung in der Produktentwicklung. In: *Fazit Forschung* 12 (2008). – ISSN 1861-5066

[Espinosa 1998] Espinosa, Ragnar H.: *Joystick API Documentation.* August 1998. – URL `https://www.kernel.org/doc/Documentation/input/joystick-api.txt`. – Zugriffsdatum: 2012-10-25

[Hammerschmidt 2007] Hammerschmidt, Christoph: BMW brings Internet protocol under the hood. In: *EE Times Europe* (2007). – URL `http://www.eetimes.com/showArticle.jhtml?articleID=204300325`. – Zugriffsdatum: 2010-12-10

[Kamieth 2011] Kamieth, Jan: *Entwurf einer Mikrocontroller basierten Bridge zur Kopplung von CAN Bussen über Time-Triggered Realtime Ethernet.* August 2011. – Bachelorthesis

[Klinger 2008] Klinger, Andreas: *Echtzeit unter Linux mit dem RT-Preemption-Patch.* 2008. – URL `http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/implementierung/articles/149175/`. – Zugriffsdatum: 2012-11-21

[Lipfert 2008] Lipfert, Jan: *Technical Data Reference Guide - netX500/100.* Hilscher GmbH. Dezember 2008. – URL `http://www.hilscher.com`

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

[NYFEGA Elektro-Garage AG ] NYFEGA Elektro-Garage AG: *Phaeton Bordnetz.* – URL `http://www.nyfega.ch/bilder/phaeton_bordnetz.jpg`. – Zugriffsdatum: 2011-02-23

[Pavlik 1996] Pavlik, Vojtech: *Linux Joystick driver v2.0.0.* 1996. – URL `https://www.kernel.org/doc/Documentation/input/joystick.txt`. – Zugriffsdatum: 2012-09-16

[Quade und Kunst 2011] Quade, Jürgen ; Kunst, Eva-Katharina: *Linux-Treiber entwickeln : eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung.* 3., aktualisierte und erw. Aufl. Heidelberg : dpunkt.verl., 2011. – ISBN 978-3-89864-696-3

[Schütte 2012] Schütte, Alois: *Betriebssysteme: Eingabe Ausgabe.* 2012. – URL `http://www.fbi.h-da.de/~a.schuette/Vorlesungen/Betriebssysteme/Skript/6_EingabeAusgabe/EingabeAusgabe.pdf`. – Zugriffsdatum: 2013-01-03

[sprut 2012] sprut: *RS232 Interface.* Oktober 2012. – URL `http://www.sprut.de/electronic/interfaces/rs232/rs232.htm`

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

[Stepanov 2011] Stepanov, Vitalij: *Mikrocontroller und CAN-basierte verteilte Regelung einer Steer-by-Wire Lenkung mit harten Echtzeitanforderungen.* August 2011. – Bachelorthesis

[Tanenbaum 2009] Tanenbaum, Andrew S.: *Moderne Betriebssysteme - 3., aktualisierte Auflage.* Pearson Studium, Oktober 2009. – ISBN 978-3-8273-7342-7

[Tanenbaum und Wetherall 2011] Tanenbaum, Andrew S. ; Wetherall, David J.: *Computer Networks.* 5. Boston : Pearson, 2011. – ISBN 978-0-13-255317-9

[TTTech Computertechnik AG 2010] TTTech Computertechnik AG: *TTEProtocol Layer Manual*, September 2010

# Tabellenverzeichnis

# Abbildungsverzeichnis

# Inhalt der beigelegten CD

- **Digitale Version dieser Arbeit** im PDF.
  - Stephan_Phieler_Bachelorarbeit.pdf
- **Sourcecode:** Der Quellcode des Joystick-Treibers und der angepassten RTE-CAN-Bridge
  - **/demonstrator_driver/** Joystick-Treiber
  - **/TTE_NXHX500/** RTE-CAN-Bridge

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

Hamburg, 04. April 2013 Stephan Phieler