Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Kai Kurt Müller

Time-Triggered Ethernet für eingebettete Systeme: Design, Umsetzung und Validierung einer echtzeitfähigen Netzwerkstack-Architektur

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Franz Korf
Zweitgutachter: Prof. Dr. Hans H. Heitman

Abgegeben am 30. August 2011

Kai Kurt Müller

Thema der Bachelorarbeit
Time-Triggered Ethernet für eingebettete Systeme:
Design, Umsetzung und Validierung einer echtzeitfähigen Netzwerkstack-Architektur
Stichworte
Echtzeit-Ethernet, Fahrzeug-Netzwerke, Netzwerk-Synchronisierung,
IEEE 1588, Design, Umsetzung, Validierung, time-triggered, NetX, TTEthernet, Netzwerkstack, Architektur
Kurzzusammenfassung
Automotive-Anwendungen steigen in Zukunft stetig in ihrer Anzahl und Komplexität. Es ist absehbar, dass momentane Ansätze an ihre Grenzen stoßen werden. Ethernet hat sich in der Computertechnik etabliert und als flexibles und hoch skalierbares Protokoll erwiesen. TTEthernet ist eine Neuentwicklung und richtet sich dabei speziell an die Anforderungen des Automotive-Bereichs. In dieser Arbeit wurde unter der Wahl geeigneter Hardware ein Konzept entwickelt und umgesetzt, welches der TTEthernet Spezifikation gerecht wird. Der entwickelte Prototyp erfüllt realistische Zeitanforderungen des Automotive Bereichs und ist somit der erste Ansatz seiner Art. Durch gründliche Analyse des Systems unter Last konnte seine Korrektheit nachgewiesen werden. Weiterhin wurden typische Metriken des Prototyps mit Bezug auf Echtzeitanforderungen mit Hilfe unterschiedlicher Messverfahren bestimmt und verglichen.

Title of the paper
Time-Triggered Ethernet for embedded systems:
Design, realisation and validation of real-time capable network stack architecture
Keywords
Real-time Ethernet, In-vehicle network, network synchronisation
IEEE 1588, design, realisation, validation, time-triggered, NetX, TTEthernet, network stack, architecture
Abstract
The established automotive communication technologies reach their performance limits due to the increase of electronic systems, especially in driver assistance and comfort applications. Ethernet is a new approach for the communication between control units in cars. Real-time extensions like TTEthernet widen the application of standard switched Ethernet into the time-critical domain. This work demonstrates a prototype which fulfils the TTEthernet specification. It complies with typical requirements of automotive applications and therefore is the first known approach of its kind. Careful analysis has shown the reliability of the system even under heavy load. Furthermore, measurements of real-time metrics has proven the compliance of the given requirements.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Motivation . . . 2
- 1.2 Zielsetzung und Problemstellung . . . 3
- 1.3 Aufbau der Arbeit . . . 4

2 Hintergrund **5**
- 2.1 Jitter und Latenz . . . 5
- 2.2 Echtzeitsysteme . . . 5
  - 2.2.1 Harte Echtzeitanforderungen . . . 6
  - 2.2.2 Weiche Echtzeitanforderungen . . . 6
- 2.3 TT-Ethernet . . . 6
  - 2.3.1 Eigenschaften . . . 7
  - 2.3.2 Nachrichtenklassen . . . 8
  - 2.3.3 Synchronisation . . . 9

3 Hardware **14**
- 3.1 Anforderungen und Analyse . . . 14
- 3.2 Vergleich . . . 15
- 3.3 Aufbau des NXHX500-ETM . . . 16
  - 3.3.1 NetX500 CPU . . . 18
- 3.4 Hardware Abstraction Layer . . . 19
  - 3.4.1 Voraussetzungen . . . 20
  - 3.4.2 Funktionsweise . . . 20
  - 3.4.3 Anwendung . . . 21

4 Konzept und Architektur **22**
- 4.1 Anforderungen und Analyse . . . 22
- 4.2 Aufstellung der grundlegenden Modellfunktionen . . . 22
- 4.3 TTEthernet als prioritätengetriebener Ansatz . . . 25
  - 4.3.1 non-preemtive . . . 26
  - 4.3.2 preemtive . . . 27

4.4 Erweitertes Modell 30
4.4.1 Buffer-Modul 30
4.4.2 Kommunikationseinheit 30
4.4.3 Planung von Ereignissen 31
4.5 Fehlerbehandlung und Signale 33

5 Realisierung **35**
5.1 Ressourcenausnutzung 36
5.1.1 Verteilung der Prioritäten 36
5.2 Scheduling 39
5.2.1 Erfassung der Zeit 39
5.2.2 Architektur 41
5.3 Bufferpool 43
5.3.1 Doublebuffer 46
5.3.2 Queued Buffer 46
5.3.3 Interface zur Definition von Buffertypen 48
5.4 Synchronisation 48
5.4.1 Rx-Modul 49
5.4.2 Tx-Modul 52
5.5 Dropping of Frames 52
5.6 Umgesetztes Konzept 53
5.7 Terminal Funktion 54

6 Test und Ergebnisse **55**
6.1 Bestimmung des Zeitverhaltens 55
6.1.1 Angewandte Messverfahren 55
6.1.2 Verwendete Hardware 56
6.1.3 Bestimmung systemspezifischer Delays 57
6.2 Funktionstest 57
6.3 Validierung der Synchronisation 59
6.3.1 Bestimmung des Jitters 59
6.3.2 Messverfahren zur Bestimmung des Jitters 60
6.4 Fehleranalyse und -bewertung 62
6.4.1 Besonderheiten und Einschränkungen 63

7 Zusammenfassung, Fazit und Ausblick **64**
7.1 Alternativen in der Realisierung 65
7.2 Offene Punkte und Verbesserungsmöglichkeiten 65

Literaturverzeichnis **66**

Abbildungsverzeichnis **68**

Tabellenverzeichnis **69**

Kapitel 1

Einführung

Heutige Kraftfahrzeuge übernehmen eine Vielzahl von assistierenden Aufgaben, die dazu beitragen den Fahrer in Sachen Sicherheit und Komfort zu unterstützen und zu entlasten. Z.B. trägt das elektronische Stabilitätsprogramm (ESP) dazu bei, Übersteuern oder Untersteuern des Fahrzeugs durch gezieltes Abbremsen einzelner Räder entgegenzuwirken und so die Sicherheit der Insassen zu gewährleisten. Mehr Komfort bietet z.B. ein adaptives oder aktives Fahrwerk (Active Body Control), welches unter anderem durch Kennungsvariation der Stoßdämpfer Bodenwellen und Unebenheiten der Fahrbahn ausgleicht um die Karosserie von der Fahrstrecke zu entkoppeln. Dies hat zur Folge, dass die Menge an benötigten Steuergeräte eines Fahrzeugs steigt und somit die Komplexität des gesamten Systems stetig zunimmt (vgl. Abbildung 1.1 auf der nächsten Seite). Weiterhin ist in Zukunft damit zu rechnen, dass der vermehrte Einsatz von videounterstützte Anwendungen wie kamerabasierte Assistenzsysteme die Nachfrage an leistungsfähigen Übertragungsmechanismen weiter verschärft.
Um dieser Problematik entgegenzuwirken wird ein neuer Architekturansatz verfolgt, der ein Backbone-Netzwerk vorsieht. Es handelt sich hierbei um eine Infrastruktur, die Gruppen von Steuergeräten sternförmig zusammenfasst und diese über ein Kabel, dem Backbone, miteinander verbindet. Dieser Backbone muss jeglichen Datenverkehr verarbeiten können und gleichzeitig ein Deterministisches Verhalten in Übertragungszeiten von Nachrichten aufweisen, selbst im Falle von spontanen, kurzfristigen Belastungsspitzen. Neben FlexRay, welches bereits erfolgreich im Einsatz ist, wird auch die Nutzung von Ethernet erwogen. Es hat sich bereits in der Computertechnik als flexibles und hoch skalierbares Protokoll erwiesen, ist aber in seiner ursprünglichen Form aufgrund seiner fehlenden Vorhersagbarkeit und Zuverlässigkeit nicht geeignet um in echtzeitfähigen Netzwerkstrukturen im Automobil eingesetzt zu werden.
In der Prozessautomatisierung haben sich bereits erweiterte Ethernet Protokolle etabliert, welche eine echtzeitfähige Kommunikation ermöglichen, die vor allem in der Produktion eingesetzt werden. Time Triggered Ethernet (TTEthernet) ist ein neuer Ansatz, diese Techno-

Abbildung 1.1: Verteilung der Steuergeräte im Automobil

logie auch auf den Bereich Automotive zu adaptieren (vgl. Kopetz u. a., 2005). TTEthernet setzt dazu einen Zeitplan ein, der für jeden einzelnen Teilnehmer definiert, wann dieser eine Nachricht senden darf. So wird das Auftreten von Kollisionen im Netzwerk vollständig aufgehoben. Um dies zu erreichen werden alle Teilnehmer mit Hilfe eines ausfallsicheren Synchronisationsprotokolls auf eine einheitliche netzwerkweite Zeitbasis gebracht.
Das TTEthernet-Protokoll befindet sich zur Zeit in der Standardisierungsphase. In dieser Arbeit wird eine TTEthernet fähige Architektur entwickelt und umgesetzt. Diese ist in der Lage ein für den Einsatz im Bereich Automotive realistisches Zeitverhalten aufzuweisen und ist somit der erste bekannte Mikrocontroller-basierte Ansatz dieser Art.

1.1 Motivation

Der Ethernet Standard ist weder an eine Übertragungsrate noch an das Übertragungsmedium gebunden. Somit ließe sich in der Theorie sämtlicher in Zukunft anfallender Datenverkehr des Automotive-Bereichs mit Hilfe dieser Technologie übertragen. Weiterhin ist die Homogenität des Ethernet-Netzwerkes ein großer Vorteil im Verhältnis zur jetzigen Lösung. Ein Automobil ist der Zusammenschluss von verschiedenartigen Bussystemen. Jede Anwendung wurde mit einem speziell für sie optimierten Bussystem umgesetzt, was zu Heterogenität im Nachrichtenverkehr führt. Ist eine Bussystem-übergreifende Kommunikation gewünscht, so sind spezielle Gateways von Nöten, die die Nachrichten von einem Bussystem ins nächste übersetzen können (vgl. Abbildung 1.2 auf der nächsten Seite). Dieser Ansatz lässt das Fahr-

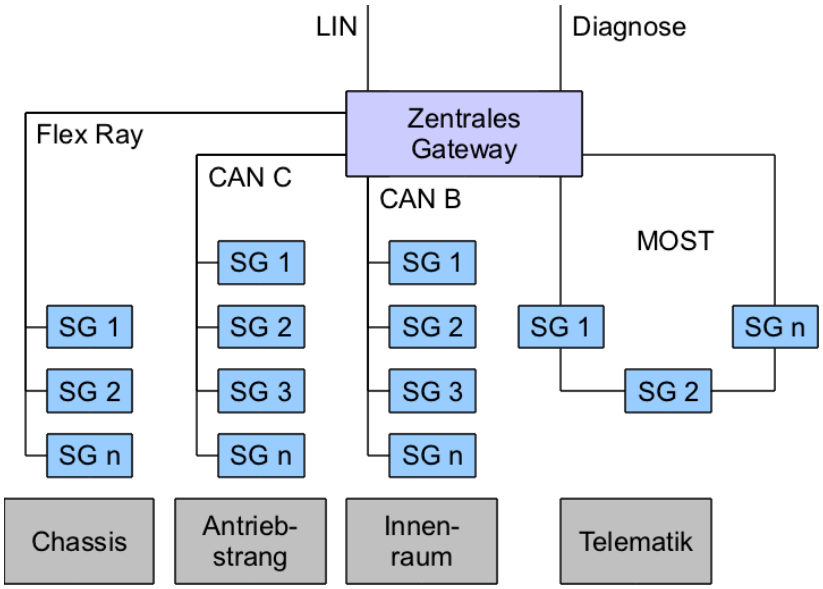

Abbildung 1.2: Zentrales Gateway verschiedener Bussysteme

zeugnetz zu einem komplizierten und schwierig zu beherrschenden System werden. Daher ist TTEthernet eine vielversprechende Variante der Echtzeitübertragung. Es nutzt bewährte Verfahren aus der Automotive-Branche wie FlexRay (vgl. FlexRay Consortium, 2005) und Time-triggered Protocol (vgl. TU Wien, 1997) und kombiniert diese mit Verfahren wie das Avionics Full DupleX Switched Ethernet (AFDX) aus der Flugzeugtechnik. Dadurch ermöglicht es die Übermittlung von Nachrichten mit unterschiedlichen zeitlichen Anforderungen über den selben Kommunikationslink.
Die zur Zeit angebotenen, auf dem Betriebssystem Linux basierenden, Evaluierungssysteme sind aufgrund von eingeschränkter Funktionalität, ungenügendem Zeitverhalten und ungeachteter Energieeffizienz nicht geeignet um im Automobil Verwendung zu finden. Um diesen Anforderungen gerecht zu werden muss ein Konzept zur Umsetzung eines Prototyps entwickelt werden. Es sollte auf geeigneter Hardware basieren, da diese ein ausschlaggebender Faktor in der Verarbeitungszeit und der Bestimmung der Datentransferraten darstellt. In Zukunft kann dieses System als Basis genutzt werden um Automotive-Anwendungen zu entwickeln, welche TTEthernet als Kommunikation nutzen. Die Endsysteme können anschließend beliebig in bestehende TTEthernet-Netzwerken eingesetzt werden.

1.2 Zielsetzung und Problemstellung

Das Ziel dieser Arbeit besteht darin einen TTEthernet basiertes Endsystem als Prototyp zu entwickeln. Dieser soll eine festgelegte API (vgl. TTTech Computertechnik AG, 2008) der

Firma TTTech erfüllen um somit eine Tool-unterstützende Konfiguration des Netzwerkes zu ermöglichen. Zudem bietet dies Portabilität für bestehende und zukünftige, zu TTEthernet konforme, Anwendungen. Über den Funktionsumfang der API hinaus soll der entwickelte Prototyp auch Funktionalität bieten, die zur Zeit nur durch die Spezifikation definiert und noch nicht in den Betriebssystem basierenden Endsystemen umgesetzt wurde. Zudem ist die Erfüllung von Zeitanforderungen von Interesse, welche an ein TTEthernet basierendes Endsystem gestellt werden. Anwendungen, die auf ein solchem Endsystem arbeiten werden hochgradig von dessen Zeitverhalten beeinflusst, was z.B. signifikante Auswirkungen auf die Güte eines als Anwendung realisierten Regelalgorithmus hat. Als Plattform soll hierzu ein Mikrocontroller dienen, um einem realistischen Aufbau im Automobil nahe zu kommen.
Für die Entwicklung ist es notwendig, die Anforderungen, die sowohl an die Hardware eines solchen Systems gestellt werden, als auch die des Konzeptes, zusammenzustellen. Hierzu zählen funktionale Anforderungen zur Festlegung der Arbeitsweise als auch nicht-funktionale, welche das Zeitverhalten beschreiben. Aufbauend hierauf ist ein Konzept zu entwickeln, welches den zuvor gestellten Anforderungen genügt und eine Unterstützung aller gewünschten Funktionen bietet. Während der Umsetzung ist mit Hilfe effizienter Nutzung der Hardware-gegebenen Ressourcen darauf zu achten ein Zeitverhalten zu erreichen, welches den nicht-funktionalen Anforderungen entspricht. Abschließend muss unter verschiedenen Testszenarien die Arbeitsweise des Gesamtsystems validiert werden. Hierzu zählt der Nachweis der allgemeinen Funktionsweise unter Einhaltung der Spezifikation und der Nachweis über über die Erfüllung des geforderten Zeitverhaltens.

1.3 Aufbau der Arbeit

Diese Arbeit gliedert sich wie folgt: Im Kapitel 2 wird ein Überblick über die Grundlagen gegeben. Des weiteren wird das Protokoll TTEthernet näher erläutert, welches die Basis für diese Arbeit darstellt. Die Wahl der verwendeten Plattform wird im Kapitel 3 diskutiert. Hier wird nicht nur der Aufbau näher beschrieben, sondern auch die Vorzüge und Besonderheiten genannt, unter Beachtung einer zeiteffizienten Verarbeitung des Datenverkehrs. Anschließend werden im Kapitel 4 alle Anforderungen an das Endsystem aufgestellt. Das Konzept zur Realisierung des Prototyps wird unter Beachtung dieser Anforderungen erarbeitet. Kernpunkte der Realisierung werden im Kapitel 5 aufgezeigt, sowie Vor- und Nachteile unterschiedlicher Umsetzungsmöglichkeiten diskutiert. Die Evaluierung des Prototyps findet im Kapitel 6 statt. Es werden unterschiedliche Testszenarien diskutiert sowie die Ergebnisse dargestellt. Abschließend wird im Kapitel 7 ein Überblick über die Kernpunkte dieser Arbeit gegeben. Zudem werden zukünftige Arbeiten vorgestellt, die eine Erweiterung des Prototyps bilden.

Kapitel 2

Hintergrund

Dieses Kapitel gibt einen Überblick über die verwendeten Begriffe und Techniken. Weiterhin wird das TTEthernet Protokoll vorgestellt und seine Eigenschaften und Besonderheiten erläutert. Diesbezüglich wird genauer auf Nachrichtenklassen, ihre Priorisierung und die Synchronisation eingegangen, da diese Themen Kernstücke des in dieser Arbeit vorgestellten Konzeptes dastellen.

2.1 Jitter und Latenz

Jitter und Latenz sind Begriffe der Nachrichtentechnik und werden oft im Bereich der Signal- oder Nachrichtenübertragung verwendet. Die Latenz bezeichnet üblicherweise die Übertragungsdauer einer Nachricht. Der Jitter ist die Variation dieser Verzögerung. Im weiteren Verlauf dieser Arbeit werden die beiden Begriffe auf Ausführungszeiten von Programmteilen hin ausgedehnt. Somit lässt sich das Verhalten des Programms und seiner Metriken beschreiben.

2.2 Echtzeitsysteme

Ein TTEthernet-Netzwerk ist ein in sich geschlossenes Echtzeitsystem. Unter einem solchen System versteht man, dass es zu einer vordefinierten Zeit eine Antwort bereit stellt. Die Länge der vorgegebenen Zeit spielt hierbei keine Rolle. Wird die Zeit bis zu einer Reaktion überschritten oder sind in der angegebenen Zeit nicht alle Daten vollständig, so wird das gelieferte Ergebnis als falsch definiert. Im Automobil sind klassische Vertreter von echtzeitfähigen Systemen z.B. das Auslösen des Airbags, ein adaptives Fahrwerk oder auch eine Rückfahrkamera. Das adaptives Fahrwerk unterscheidet sich in seinen Anforderungen zu dem der Rückfahrkamera signifikant. Das adaptives Fahrwerk hat durch seine typische Regelungsfrequenz von 1 kHz eine Mindestreaktionszeit von unter 1 ms. Für die Anwendung der

Rückfahrkamera hingegen ist es völlig ausreichend innerhalb von 60 ms neue Bildinformationen zu erhalten um als flüssiger Ablauf interpretiert zu werden. Selbst ein vorübergehendes Ausbleiben der Bildinformationen würde beim Fahrer nur als kurzes Ruckeln wahrgenommen werden und das System nicht in Gefahr bringen. Es ist zu erkennen, dass verschiedene echtzeitfähige Systeme unterschiedliche Anforderungen aufweisen können. Hierbei ist eine Abstufung der verschiedenen Echtzeitsysteme anhand dieser Anforderungen möglich (vgl. Tanenbaum, 2009).

2.2.1 Harte Echtzeitanforderungen

In Systemen mit harten Echtzeitanforderungen wird eine nicht Einhaltung der Antwortzeit immer als Fehler gewertet. Diese Systeme sind in ihrer Funktionsweise auf den Erhalt der Daten in der vordefinierten Zeit angewiesen. Die Abstufung von harten zu weichen Echtzeitanforderungen ist nicht eindeutig spezifiziert. Dennoch lässt sich zu harten Echtzeitanforderungen sagen: Tritt ein Fehlerfall ein so ist Sach- oder gar Personenschaden möglich. Z.B. würde ein Ausbleiben oder verspäten von Informationen das adaptive Fahrwerk des Autos nicht mehr zuverlässig arbeiten lassen, was zu einem erhöhten Unfallrisiko und somit auch zu einer direkten Gefahr für die Insassen führen würde.

2.2.2 Weiche Echtzeitanforderungen

Von weichen Echtzeitanforderungen wird allgemein dann gesprochen, wenn die Verletzung der Antwortzeit keine bleibenden Auswirkungen auf ein System oder Personen hat. Meist ist es diesem echtzeitfähigen System möglich diesen Fehlerfall mit Hilfe von Mittelwertberechnungen oder statistischen Verfahren zu überbrücken. Als Beispiel sei hier die Anwendung der Rückfahrkamera genannt, die bei Datenabriss zu Ruckeln beginnt, aber dennoch keinen direkten Schaden am System verursacht.

2.3 TT-Ethernet

Bei dem TTEthernet Protokoll (TTEthernet) (vgl. Steiner, 2008) handelt es sich um eine zeitgesteuerte und echtzeitfähige Erweiterung des Standard Ethernets. Es wurde bei der Entwicklung speziell darauf geachtet den stetig steigenden Anforderungen des Automotive-Bereichs und der der Avionic gerecht zu werden. Das Protokoll entstand aus einem Projekt der Real-Time Systems Group ((vgl. Real Time Systems Group (RTS))) der TU Wien aus dem Jahre 2004. Die österreichische Firma TTTech (vgl. TTTech Computertechnik AG) hat dieses in Kollaboration mit Honeywell (vgl. Honeywell International) weiterentwickelt und Spezifiziert (vgl. Steiner, 2008). Zudem wird es zur Zeit zur Standardisierung durch die So-

Abbildung 2.1: Switch zur Evaluierung eines TTEthernet-Netzwerks

ciety of Automotive Engineers (vgl. SAE - AS-2D Time Triggered Systems and Architecture Committee, 2009) vorgeschlagen.

2.3.1 Eigenschaften

TTEthernet basiert auf Switched Ethernet. Dies bedeutet, dass jede Topologie auf speziellen Switches aufbaut, die die Nachrichten an die jeweiligen Teilnehmer weiterleiten. Abbildung 2.1 zeigt einen Evaluierungsswitch der Firma TTTech. Zudem wird eine Sicherheit im Fehlerfall durch Redundanz erreicht. Hierzu werden beliebig viele redundante Ports bis hin zu ganzen Netzwerken eingerichtet. Das Protokoll nutzt Time Devision Multiple Access (TDMA), eine Zeitscheibentechnik, um die zur Verfügung stehende Bandbreite für die zu übermittelnden Nachrichten aufzuteilen. Dazu wird ein periodischer Zyklus definiert, in dem festgelegt wird, wer zu welchem Zeitpunkt welche Nachricht senden darf. Die Festlegung dieser Zeitslots geschieht offline, das heißt, zu einer Phase in der das Netzwerk konfiguriert wird aber noch nicht aktiv wurde. Ist das Netzwerk einmal hochgefahren so sind keine Änderungen am Zeitplan (Schedule) mehr erlaubt, weshalb man von einem statischen Ansatz spricht. Dies hat zum Vorteil, dass die Übertragungsverzögerung der Nachrichten vorhersagbar und somit Latenz und Jitter gering sind. Um jedem Knoten zu ermöglichen jeden explizit für ihn definierten Versendezeitpunkt nutzen zu können verfügen alle Teilnehmer im Netzwerk über eine lokale Zeit und einen eigenen Sende-Schedule. Diese lokalen Zeiten erfordern eine Synchronisation zueinander um eine netzwerkweite globale Zeitbasis zu erhalten. Hierzu definiert TTEthernet ein ausfallsicheres Synchronisationsprotokoll. Da TTEthernet noch niederpriorisierte Nachrichtenklassen definiert, werden diese in den freien Bereichen des Zyklus untergebracht.

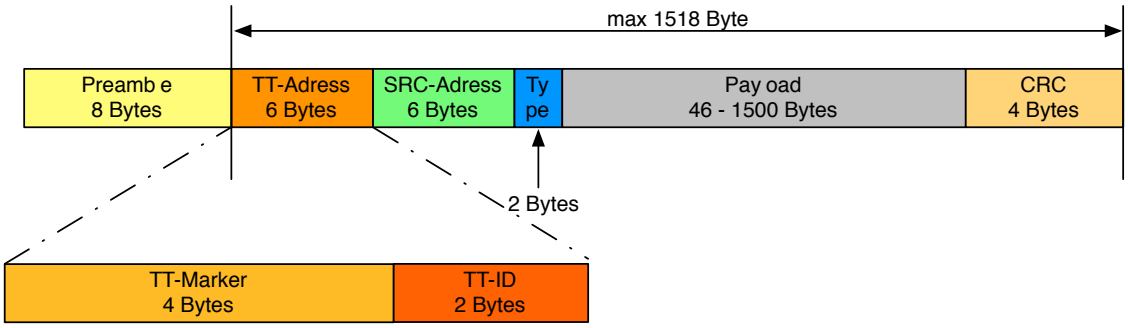

Abbildung 2.2: Aufbau eines TT-Ethernet Frames

2.3.2 Nachrichtenklassen

Ein TTEthernet Frame unterscheidet sich in erster Betrachtung nicht von dem eines Standard Ethernet Netzwerkes (vgl. Abbildung 2.2). Das Protokoll nutzt ein inhaltsorientiertes Adressierungsformat, welches vergleichbar mit dem des Ethernet Multicast ist. Anstelle über die Zieladresse einen Knoten im Netzwerk anzusprechen wird über diese Adresse die Art der Nachricht festgelegt. Die ersten 32 Bit der 48 Bit Zieladresse dienen zur Differenzierung zwischen echtzeitfähigem und unkritischem Datenverkehr. Sie werden *Critical Identifier* genannt

Echtzeit Nachrichten

Um echtzeitfähige Nachrichten (CT) identifizieren zu können wird ein *Critical Marker* (CT-Marker) und eine *Critical Mask* (CT-Mask) festgelegt. Über die CT-Mask wird festgelegt welche Bits des Identifiers einer Nachricht auf den CT-Marker hin überprüft werden. Findet eine Übereinstimmung der selektierten Bits statt so wird die Nachricht als *Critical Traffic* (CT) klassifiziert. Die Gleichung 2.1 veranschaulicht den Identifizierungsvorgang von echtzeitfähigen Nachrichten.

$$CT\,Identifier \wedge CT\,Mask = CT\,Marker \wedge CT\,Mask \tag{2.1}$$

Die letzten 16 Bit der Zieladresse enthalten die *Critical Traffic ID* (CT-ID). Sie wird verwendet um alle gesendeten echtzeitfähigen Nachrichten eindeutig voneinander unterscheidbar zu machen. Dies ist notwendig, da nicht alle Nachrichten den gleichen Zyklus aufweisen müssen und auch nicht zwangsläufig in jedem Zyklus gesendet werden. Die Spezifikation legt nur die letzten 12 Bit als CT-ID fest. Die verbleibenden 4 Bits bleiben ungenutzt bzw. sind unspezifiziert. Somit liegt in einem Netzwerk die maximale Anzahl bei 4096 identifizierbaren Nachrichten. Alle Routingentscheidungen basieren auf der CT-ID und sind so für jede Nachricht definiert.

Echtzeitfähige Nachrichten unterteilen sich weiterhin in Time-Triggered und Rate-Constrained. Time-Triggered Nachrichten sind an den Zyklus gebunden. Der Zyklus definiert für jeden Knoten eindeutig Zeitpunkte zu denen sie gesendet bzw. empfangen werden können. Diese Nachrichtenklasse hat die höchste Priorität und gewährleistet, dass zu jedem definierten Zeitpunkt keine weitere Übertragung den Time-Triggered Nachrichtenverkehr behindern kann. Hierdurch wird eine Zustellung der Nachricht immer garantiert. Rate-Constraind Verkehr ist kompatibel zum AR-136 INC 664 Part 7 Standard (vgl. Aeronautical Radio Incorporated, 2002). Es wird lediglich garantiert, dass den Nachrichten eine bestimmte Bandbreite zur Verfügung steht. Diese ist an die jeweilige CT-ID der Nachricht gebunden. Sie sind vollkommen unabhängig vom Schedule und werden gesendet sobald keine Time-Triggered Nachricht ansteht. Um sicher zu stellen, dass Rate-Constrained Nachrichten ihr Kontingent an Bandbreite nicht überschreiten werden *Bandwidth Allocation Gaps* (BAG) definiert. Diese legen die minimale Zeitspanne fest, die zwischen zwei zu sendenden Nachrichten mit der gleichen CT-ID liegen darf. Wird diese Zeit unterschritten, wertet der Switch die Nachricht als ungültig und verwirft sie. Eine Verzögerung der Nachricht über die für sie definierte BAG hinaus ist jederzeit erlaubt. Für Rate-Constrained Nachrichten ist ebenso eine Zustellung garantiert und es kann eine obere Grenze für die Latenz angegeben werden (vgl. Charara u. a., 2006). Diese Nachrichten stellen die mittle Priorität dar und werden von Time-Triggered Verkehr verdrängt.

Best-Efford Nachrichten

Best-Efford Nachrichten (BE) sind äquivalent zu Standard Ethernet Nachrichten. Dies ermöglicht das Einhängen von beliebiger Standard Ethernet Hardware in ein bestehendes TTEthernet Netzwerk. Best-Efford Nachrichten werden in Ruhephasen des Netzwerkes versendet. Es können keine Aussagen über eine maximale Latenz oder den Jitter getroffen werden. Zudem kann nicht garantiert werden, dass eine Übertragung der Nachrichten erfolgreich ist. Diese Nachrichtenklasse hat die niedrigste Priorität und wird von allen anderen Nachrichtenklassen im Netzwerk verdrängt.

2.3.3 Synchronisation

Die Synchronisation erfolgt separiert vom eigentlichen Datenverkehr über spezielle Frames, den *Protocol Control Frames* (PCF). Der PCF gleicht vom Aufbau einer CT-Nachricht mit einer minimalen Länge. Prinzipiell kann jede beliebige CT-Nachricht von einem Endsystem als PCF interpretiert werden. Die Spezifikation schlägt vor das Typen-Feld des Ethernetframes fest auf 0x891D zu setzen um eine globale Identifikation des Frames zu ermöglichen. Jeder PCF enthält den aufsummierten Delay aller statischen und dynamischen Verzögerungen von seiner Erzeugung bis hin zum empfangenen Endsystem und bildet so eine transparente

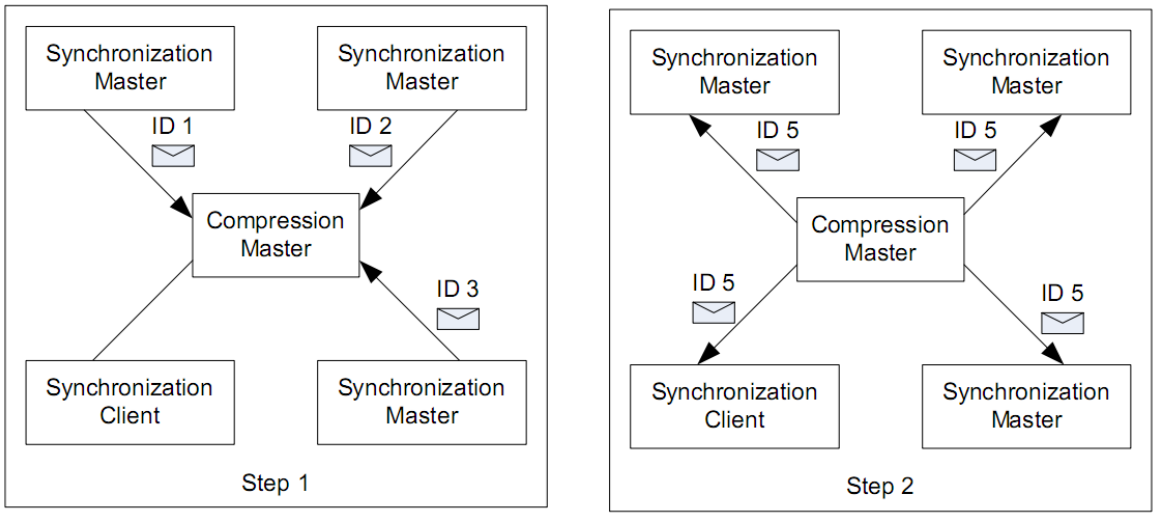

Abbildung 2.3: Ablauf der zweistufigen Synchronisation

Zeitbasis des Netzwerks. Die Einheit beträgt 2^{-16} ns, was bedeutet, dass eine Nanosekunde durch 0x10000 h repräsentiert wird.

Rollen der Knoten

TTEthernet definiert drei Synchronisationsrollen: Synchronisationsclient, Synchronisationsmaster und Compressionmaster. Jedes Endsystem und jeder Switch nimmt genau eine dieser Rollen an. Die Verteilung bleibt dabei dem Netzwerkdesigner überlassen und unterliegt prinzipiell keiner Einschränkung, mit der Ausnahme, dass zur erfolgreichen Synchronisation mindestens ein Synchronisationsmaster Teil des Netzwerks sein muss. Ist ein Compressionmaster im Netzwerk vorhanden so findet jede Periode ein zweistufiger Synchronisationsprozess, welcher in Abbildung 2.3 anhand einer Beispielkonfiguration, bestehend aus einem Compressionmaster, drei Synchronisationsmastern und einem -client dargestellt wird. Zunächst senden alle Master innerhalb des Zyklus einen PCF mit ihrer eigenen Zeitbasis an den Compressionmaster. Dieser wertet die übertragenen Zeiten aus und berechnet einen geeigneten Synchronisationszeitpunkt. An diesem sendet er an alle Teilnehmer des Netzwerkes einen PCF mit der gewünschten Zeitbasis auf die sich anschließend alle Teilnehmer synchronisieren. Sollte ein Netzwerk über keinen Compressionmaster verfügen so ist eine Synchronisation auch mit Hilfe genau eines Masters möglich. Dieser sendet im Zyklus einen PCF an alle beteiligten Clients und Synchronisiert so das Netzwerk auf seine eigenen Zeitbasis.

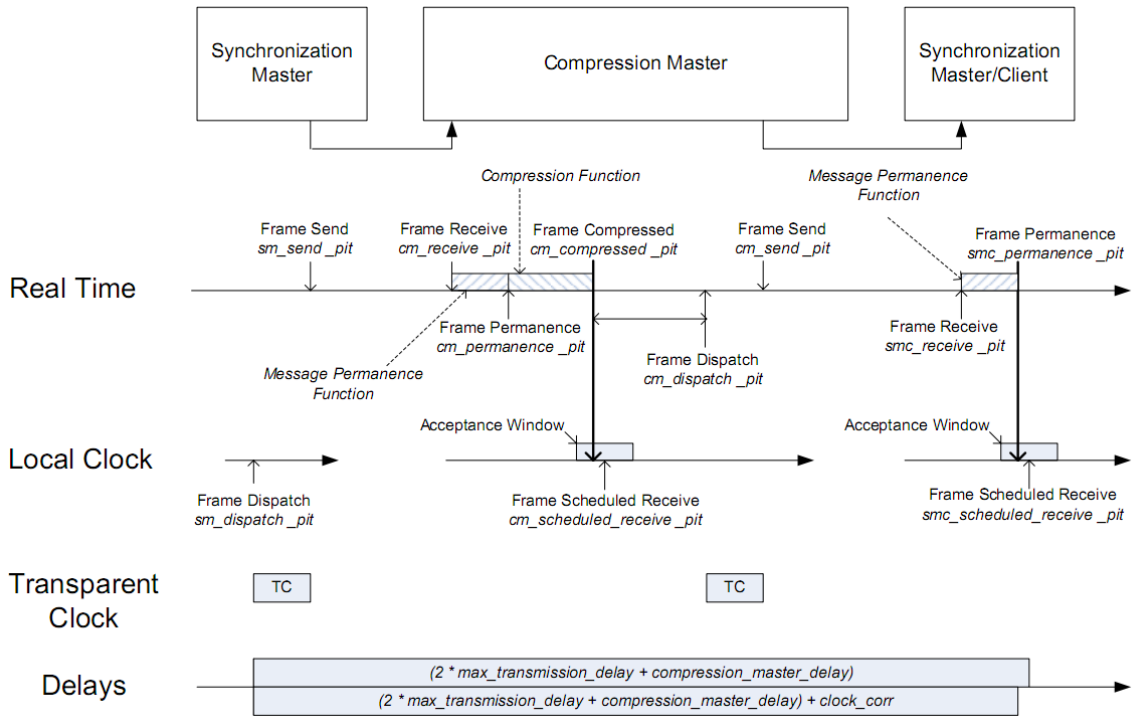

Abbildung 2.4: Kontrollfluss des Synchronisationsalgorithmus

Kontrollfluss

Abbildung 2.4 verdeutlicht den Synchronisationsprozess zwischen einem Synchronisationsmaster, einem Compressionmaster und einem beliebigen Endsystem im Detail. Es handelt sich hierbei also ebenfalls um einen zweistufigen Synchronisationsprozess. Das Beispiel geht davon aus, dass Synchronisationsmaster und -client Endsysteme darstellen, während der Compressionmaster auf einem Switch umgesetzt wurde. Das Echtzeitsystem bindet alle asynchronen Ereignisse wie z.B. das Empfangen von Frames an die lokale Zeit. Die Abbildung zeigt die netzwerkweite Zeit (Real Time) im Vergleich zur lokalen Zeit (Local Time) eines jeden Knotens. Das Konzept der Synchronisation legt eine Reihe charakteristische Zeitpunkte fest, welche im Folgenden näher erläutert werden.

- **Synchronisationsmaster Dispatch Point in Time** (*sm_dispatch_pit*) stellt den Zeitpunkt dar, an welchem das lokale Scheduling das Ereignis zum Senden des PCF beginnt auszuführen.
- **Synchronisationsmaster Send Point in Time** (*sm_send_pit*) legt fest, wann der Sendevorgang eines Frames auf dem Kommunikationslink beginnt. Der Synchronisations-

master bestimmt unmittelbar vor dem Absenden die zeitliche Differenz zwischen dem sm_send_pit und dem sm_dispatch_pit und schreibt diese in den PCF.

- **Compression Master Receive Point in Time** (*cm_receive_pit*) ist der Zeitpunkt an dem ein Start-of-Frame auf dem Kommunikationslink empfangen wurde und der eigentliche Empfangsprozess des Frames beginnt.

- **Compression Master Permanence Point in Time** (*cm_permanence_pit*) definiert, wann ein PCF *permanent* wurde. Das bedeutet, dass er zu diesem Zeitpunkt im Schedule stabil ist und zur weiteren Verarbeitung bereit steht. Zur Ermittlung dieses Zeitpunkts wird die größtmögliche Verzögerung eines PCFs als Referenz genommen. Dies wird anschließend um die Verzögerung des aktuell empfangenen PCFs reduziert. Die aktuelle Verzögerung bestimmt sich hierbei aus der im PCF enthaltenen Verzögerung summiert mit möglichen, zusätzlichen Verzögerungen in der Verarbeitung.

- **Compression Master Compressed Point in Time** (*cm_compressed_pit*) stellt den Zeitpunkt dar, an dem die *Commpression* des PCFs abgeschlossen wurde. Die Compression-Funktion bestimmt anhand aller PCFs den sm_dispatch_pit jedes einzelnen in diesem Synchronisationszyklus beteiligten Masters. Durch diese wird mit Hilfe einer Durchschnittsberechnung der cm_compressed_pit gebildet. Diese Methode gleicht mögliche Schwankungen der lokalen Uhren der Synchronisationsmaster aus. Der cm_compressed_pit ist nicht an den Schedule des Compression Masters gebunden.

- **Compression Master Scheduled Receive Point in Time** (*cm_scheduled_receive_pit*) legt den Zeitpunkt im Schedule fest, an welchem PCFs erwartungsgemäß empfangen werden. Während des Synchronisationsvorgangs wird ein Zeitfenster über diesen Zeitpunkt gespannt, welches bestimmt, ob der zuvor ermittelte cm_compressed_pit innerhalb des Schedules liegt oder nicht.

- **Compression Master Dispatch Point in Time** (*cm_dispatch_pit*) ist der Zeitpunkt im Schedule des Compression Masters, an welchem der Weiterleitungsvorgang des PCFs eingeleitet wird. Dieser Zeitpunkt ist ein Offset zum cm_compressed_pit, welcher sich frei konfigurieren lässt. Zusätzlich werden zwei Modi in der weiteren Vorgehensweise eingesetzt: Sollte der Compression Master so konfiguriert sein, dass er im *Clique Detection*-Modus arbeitet so werden alle Frames weitergeleitet, sollten sie sich im Zeitfenster des cm_scheduled_receive_pit befinden oder nicht. Dies ermöglicht die Definition von Subclustern im Netzwerk, die eine andere Periode als Basis nutzen als das globale System. Dies führt jedoch dazu, dass auch fehlerhafte PCF weitergeleitet werden. Im Falle keiner Clique Detection wird der Frame nicht weitergeleitet sollte er sich außerhalb des Schedules befinden.

- **Compression Master Send Point in Time** (*cm_send_pit*) definiert, wann der Sendevorgang eines Frames vom Compression Master auf dem Kommunikationslink beginnt.
- **Synchronization Master / Synchronization Client Receive Point in Time** (*smc_receive_pit*) stellt den Zeitpunkt dar, an dem ein Start-of-Frame auf dem Kommunikationslink von einem Synchronisationsmaster oder -client empfangen wurde und der eigentliche Empfangsprozess des Frames beginnt.
- **Synchronization Master / Synchronization Client Permanence Point in Time** (*smc_permanence_pit*) legt den Zeitpunkt fest, wann der PCF permanent wurde. Zur Ermittlung dieses Zeitpunkts erfolgt ähnlich wie die des cm_permanence_pit. Es wird die größtmögliche Verzögerung des PCFs als Referenz genommen. Anschließend wird die Differenz zwischen der Verzögerung des aktuell empfangenen PCFs und der Referenz gebildet. In die Bestimmung der aktuellen Verzögerung fließen die im PCF enthaltene Verzögerung und mögliche, zusätzliche Verzögerungen in der Verarbeitung ein.
- **Synchronization Master / Synchronization Client Scheduled Receive Point in Time** (*smc_scheduled_receive_pit*) ist der Zeitpunkt im Schedule, an welchem PCFs erwartungsgemäß vom Master und Client empfangen werden. Dieser Zeitpunkt ist mit einem konfigurierbaren Zeitfenster versehen. Liegt der zuvor ermittelte smc_permanence_pit innerhalb dieses Zeitfensters so ist der Frame in Schedule empfangen worden. Andernfalls liegt er außerhalb des Schedule.

Abschließend erfolgt die zuvor angesprochene Anpassung der lokalen Zeit des Knotens womit er synchronisiert ist und auf der netzwerkweiten Zeitbasis arbeitet.

Kapitel 3

Hardware

Die Hardwareplattform beeinflusst die Metriken des umzusetzenden TTEthernet Stacks und ist somit ein essentieller Teil des Endsystems. Sie bestimmt die Reaktion des Systems und nimmt somit direkten Einfluss auf den Delay aller Ereignisse. Weiterhin muss die Hardware ein reproduzierbares Verhalten in der Verarbeitungszeit aufweisen, um störende Schwankungen, welche im Jitter des Systems resultieren können, möglichst gering zu halten. Es gilt jedoch darauf zu achten, dass bei echtzeitfähigen Systemen dazu geneigt wird eine beliebig kleine Reaktionszeit zu fordern, die mit reeller Hardware nicht umzusetzen ist. In diesem Kapitel werden Anforderungen an die Hardware aufgestellt. Anschließend wird eine Auswahl an Hardware getroffen und eine Hardware, welche die Anforderungen erfüllt, ermittelt und diese vorgestellt.

3.1 Anforderungen und Analyse

Es existieren eine Reihe von Anforderungen, die ein TTEthernet fähiger Stack an eine Hardware stellt. In Tabelle 3.1 auf der nächsten Seite sind diese Anforderungen aufgelistet und werden anschließend auf ihre Umsetzbarkeit hin untersucht.

Zusatzanmerkungen

Anforderung 2 tritt nicht nur in speziellen Situationen auf, sondern bildet die normale Arbeitsweise des Systems. Daher ist hier besonders auf eine, in Bezug auf Zeitaufwand und Datendurchsatz, effiziente Lösung zu achten. Zu Anforderung 4 sei anzumerken, dass mindestens eine Speicherkapazität von vier Nachrichten pro Port vorhanden sein muss. Dieses Minimum fordert die Situation in der jeweils eine Nachricht geschrieben und gelesen wird. Zur gleichen Zeit ist es möglich, dass Nachrichten über das Kommunikationsmodul gesendet oder empfangen werden, was unabhängig von Lese- oder Schreibzugriffen möglich sein

Tabelle 3.1: Anforderungen der Hardware

#	Anforderung
1	Die Plattform sollte über mindestens 2 Ethernetschnittstellen verfügen um Failsave durch Redundanz zu unterstützen
2	Mehrere Ports müssen zur selben Zeit fähig sein Frames zu empfangen bzw. zu senden
3	Alle Ports müssen eine Übertragungsrate von 100 MBits Full-Duplex unterstützen
4	Es muss ausreichend Speicherkapazität vorhanden sein um multiple Frames speichern zu können (mindestens vier Frames pro Port)
5	Das Board muss über einen Mechanismus verfügen, welcher erlaubt den Zeitpunkt eines eingehenden Frames mit einer Genauigkeit von unter 1 µs zu ermitteln.
6	Es müssen für weiterführende Projekte Bussysteme, wie z.B. CAN (vgl. Robert Bosch GmbH) Unterstützt werden.

muss. Die geforderte Genauigkeit aus Anforderung 5 resultiert in der Betrachtung realistischer Automotive Anwendungen. Um eine entsprechende Unterstützung zu liefern darf kein Jitter größer 1 µs auftreten. Anforderung 6 bezieht sich auf zukünftige Projekte, die darauf abzielen, unter Nutzung dieses TTEthernet Stacks, Anbindungen zu bestehenden Bussystemen zu realisieren.

3.2 Vergleich

Tabelle 3.2 zeigt einen Vergleich von möglicher Hardware zu den in Tabelle 3.1 gestellten Anforderungen. Jeder erfüllte Anforderung wird durch ein 'X' Markiert. Nicht erfüllte Anforderungen bleiben leer. Es wird nur eine Auswahl der Hardware präsentiert, welche eine maximale Anzahl an Anforderungen erfüllen konnte. Die Tabelle zeigt, dass ausschließlich das Entwicklungsboard NXHX500-ETM der Firma Hilscher (vgl. Lipfert, 2008) alle Anforderungen erfüllen konnte. Weiterhin ist die Hardware auf Umsetzung von Echtzeitfähigen Netzwerkstacks ausgelegt und unterstützt bereits eine Vielzahl von Protokollen, was ebenfalls für die Wahl dieses Boards spricht.

Tabelle 3.2: Vergleich von Entwicklungsboards mit den Anforderungen

Anbieter	Board	1	2	3	4	5	6
Texas Instruments	LM3S8962 Evaluation Board			X	X	X	X
Keil	MCBXC167-NET Evaluation Board			X	X	X	X
Atmel	Atmel SAM9X	X		X	X	X	X
NXP	LPC1850 Evaluation Board			X	X	X	X
Hilscher	NXHX500-ETM Evaluation Board	X	X	X	X	X	X

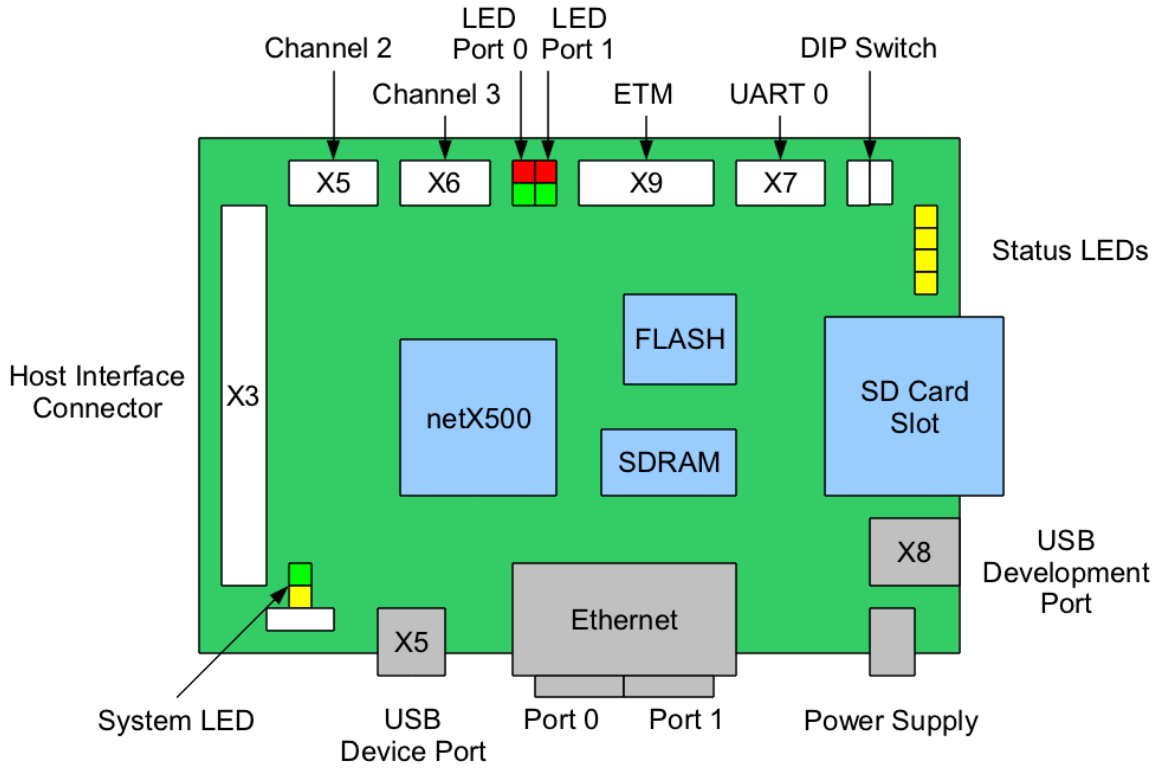

Abbildung 3.1: Stilisiertes NXHX500-ETM

3.3 Aufbau des NXHX500-ETM

Kernstück der Architektur ist das System on Chip Design der NetX CPU. Es ermöglicht die unabhängige Kommunikation über mehrere Kanäle zur gleichen Zeit. Die Kompatibilität auf Physical Layer Basis wird mit Hilfe von Modulen erreicht. Das NXHX500-ETM besitzt zwei fest installierte Ethernetschnittstellen (vgl. Abbildung 3.1). Weitere Kommunikationsschnittstellen sind über zwei Modulsteckplätze erweiterbar. Weiterhin Verfügt es über eine serielle Schnittstelle mit einer maximalen Übertragungsrate von 3.125 MBaud, einem USB Port mit 12 Mb/s und Kompatibilität zum USB 2.0 Standard, sowie einer Dual Port Memory Schnittstelle für hohe Übertragungsbandbreiten an ein Endsystem. Das Evaluierungsboard verfügt über 8 MByte SDRAM und 16 MByte Flashspeicher. Zur Kommunikation mit Entwicklungswerkzeugen stehen ein zusätzlicher USB Port, an welchen ein interner Debugger gekoppelt ist, sowie eine ETM Schnittstelle für externes Debugging zur Verfügung. Zudem können über ein MMC/SD-Card Einschub Programme auf das Board geladen und zur Ausführung gebracht werden werden (vgl. GmbH, 2009).

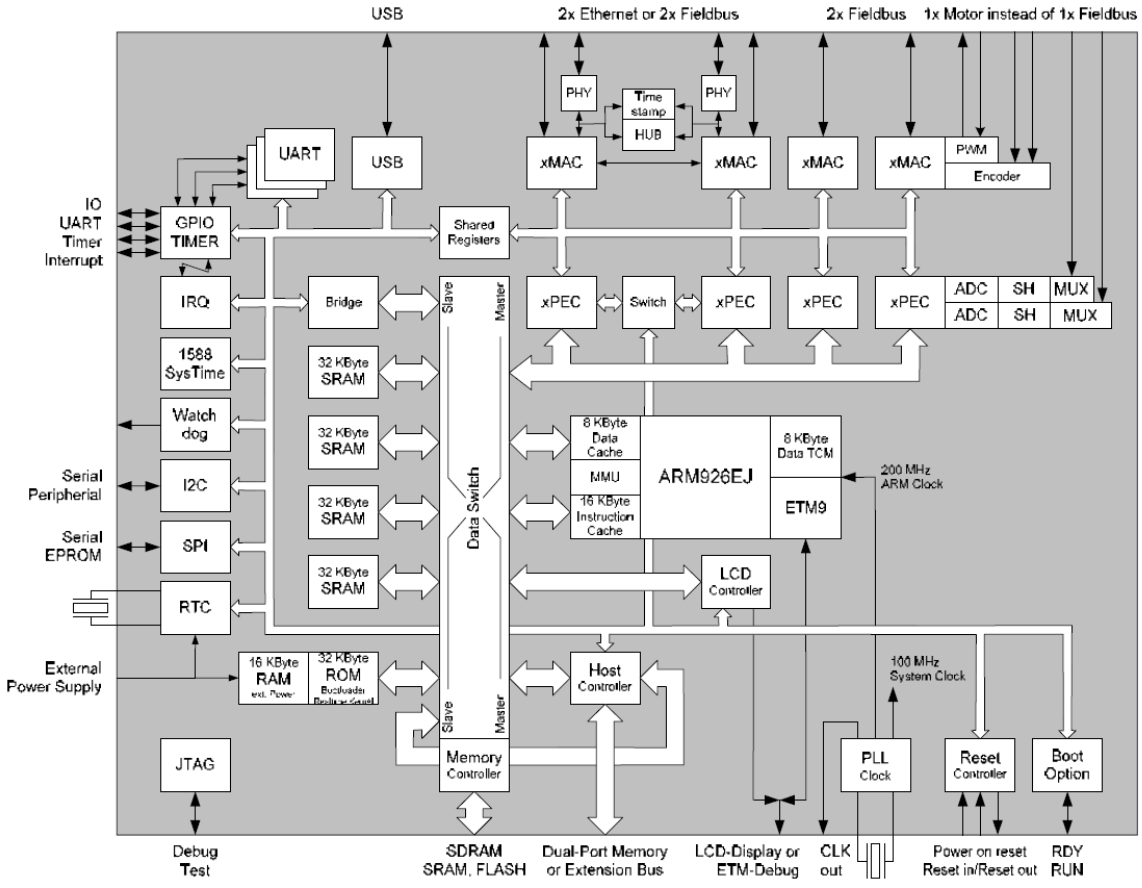

Abbildung 3.2: Aufbau der NetX500 CPU

3.3.1 NetX500 CPU

Die NetX CPU zeichnet sich durch vier unabhängige Kommunikationskanäle, einen internen SRAM für ein- und ausgehende Nachrichten sowie einer integrierten 32 Bit ARM9 CPU aus (vgl. Abbildung 3.2 auf der vorherigen Seite). Der ARM926EJ wird mit einer Taktrate von 200 MHz betrieben und ist an eine Memory Management Unit (MMU), 8 KByte Datencache Cache, sowie 16 KByte Instructionscache gebunden. Weiterhin verfügt der NetX über einen 8 KByte großen Tightly Coupled Memory, welcher als interner Speicher realisiert wurde und in direkter Verbindung mit der ARM CPU steht. Aus diesem Grund erlaubt er eine Verarbeitung ohne Waitstates und dient somit zur Ausführung von zeitkritischen Programmteilen wie z.B. Interruptserviceroutinen. Die weiteren Peripheriegeräte der NetX CPU arbeitet mit einem Systemtakt von 100 MHz. Der Interne SRAM unterteilt sich in vier 32 KByte große Bereiche für die Verarbeitung von Nachrichten über die Kommunikationskanäle sowie einen 16 KByte Block mit externer Spannungsversorgung. Dieser eignet sich z.B. zur Rettung von sicherheitsrelevanten Daten vor einem Neustart des Controllers.
Der NetX eigene Interruptvectorcontroller unterstützt bis zu 32 Interruptquellen, von denen sich 16 vektorisieren lassen. Die ARM-CPU lässt eine Verarbeitung von zwei getrennten Interruptquellen zu, den General Purpose Interrupt (IRQ) und den Fast Interrupt (FIQ). Der FIQ besitz die Höchste Priorität und niedrigste Latenz, solange er über nur eine Quelle getrieben wird. Eine Vektorisierung sollte daher nur für IRQ-Quellen vorgenommen werden.

Kommunikationskanäle

Alle 4 Kommunikationskanäle werden jeweils durch drei ebenfalls voneinander unabhängige, frei konfigurierbare ALUs betrieben, was gleichzeitiges Senden und Empfangen erlaubt und somit den Kanal full-duplex-fähig macht. Es handelt sich bei den ALUs um eine *Receive Processing Unit* (RPU) zum Empfangen, einer *Transmit Processing Unit* (TPU), welche gleichzeitiges Senden erlaubt und des *Protocol Execution Controllers* (xPAC). Der xPAC formt aus den empfangenen Bytes das protokollspezifische Paket und schreibt dieses über einen Dataswitch in den internen SRAM. Die ALUs können beliebig konfiguriert werden um verschiedenen Protokollen gerecht zu werden. Zwei der Kommunikationskanäle verfügen über jeweils eine interne PHY, welche den Physical Layer des Ethernet Protokolls bis zu einer Übertragungsrate von 100 MBits umsetzt. Dies ermöglicht eine direkte Ethernet Kommunikation über die ersten beiden Kommunikationskanäle. Weiterhin sorgt eine Timestamping-Einheit dafür, dass jeder empfangende Start-of-Frame eine Zeit erhält, die der Systemtime-Einheit entnommen wird.

Dataswitch

Der Dataswitch (vgl. Abbildung 3.2 auf der vorherigen Seite) des NetX ersetzt die klassischerweise im Embedded-Bereich Verwendung findende *Advanced Microcontroller Bus Ar-*

chitecture (AMBA). Diese Architektur besteht aus zwei Bussystemen: Dem *Advanced High Performance Bus* (AHB) und dem *Advanced Peripherial Bus* (APB). Nutzen zwei oder mehr Master den AHB, so ist eine Arbitrierung des Busses notwendig, was bei gleichzeitiger Anfrage des Busses zu einem Flaschenhalsverhalten führt und den Übertragungsdurchsatz einschränkt. Zudem ist die Zugriffszeit auf den Bus nicht in jedem falle Deterministisch. Dieses Verhalten umgeht der Dataswitch. Er kann bis zu fünf Übertragungskanäle zwischen je einem Master und einem Client zur selben Zeit aufrecht erhalten solange jeder Übertragungskanal vollkommen unabhängig zu den jeweils anderen ist. Dies ist der Fall, solange alle Master und Clients individuell adressierbar sind. Um diese Konstellation in möglichst vielen Fällen zu erreichen wurde der interne SRAM in 4 Bänke aufgeteilt. So ist bei einer Übertragung von 32 Bit auf allen fünf Kanälen und einer Taktrate von 100 MHz eine maximale Transferrate von bis zu 2 GBytes möglich. Ist keine vollkommene Unabhängigkeit zwischen den Übertragungskanälen gegeben so ist auch hier eine Arbitrierung von Nöten, die jedoch ohne Taktverlust vonstatten geht.

Systemzeit

In einem echtzeitfähigem Netzwerk ist es erforderlich, dass sich Knoten auf Mikrosekundengenauigkeit Synchronisieren. Da dies nicht mit einem realistischen Zeitverhalten durch Austausch von Nachrichten direkt über das Netzwerk erfolgen kann hängt die Synchronisationsgenauigkeit von der internen Zeit eines jeden Knotens ab. Um diesen hohen Grad an Genauigkeit gerecht zu werden wurde das Precise Time Protocol (PTP) entwickelt (vgl. Ademaj und Kopetz, 2007). Der Knoten mit der höchsten lokalen Genauigkeit wird als Master definiert und Synchronisiert alle Slaves mit der Hilfe von unterschiedlichen Nachrichten.
Das SystemTime Modul erfüllt die Anforderungen des PTP. Es gehört zur Peripherie und wird daher mit einer Taktrate von 100 MHz betrieben. Dies kann eine konstante Abweichung durch Herstellungsprozesse sowie dynamische Abweichungen durch äußere Einwirkungen wie z.B. Alterung oder Temperatureinflüsse beinhalten. Um eine Ungenauigkeit kompensieren zu können wurde die SystemTime nicht als ein weiterer Timer konzipiert, sondern lässt sich in ihrer Schrittweite variieren. Diese Variation erfolgt in einer Auflösung von 2^{-28} ns und ermöglicht so eine präzise Anpassung der Systemzeit des Netzwerkknotens.

3.4 Hardware Abstraction Layer

Für die NetX System-on-Chip Prozessorfamilie wird bereits zu vielen echtzeitfähigen Ethernet-Protokollen ein entsprechender Hardware Abstraction Layer (HAL) angeboten (vgl. J., 2009). Diese lassen sich jedoch nicht für das TTEthernet wiederverwenden, da sie entweder auf einem Tokenverfahren basieren oder eine eigene Struktur der Ethernet-Frames definieren, die mit dem Standard Ethernet nicht mehr kompatibel ist. Daher wird für die-

se Umsetzung der Standard Ethernet Mac HAL (vgl. Pfrommer, 2008) verwendet, welcher ebenfalls ein Produkt der Firma Hilscher ist. Dieser beinhaltet unter anderem Routinen zum Versenden und Empfangen von Ethernet-Nachrichten und stellt somit alle grundlegenden Funktionen der Sicherungsschicht (vgl. Data Link des ISO-Referenzmodells) bereit. Dieser Ansatz erhöht zudem die Portabilität des in dieser Arbeit umgesetzten TTEthernet-Stacks auf beliebige Hardware, welche über einen Hardware Abstraction Layer für Standard Ethernet verfügt.

3.4.1 Voraussetzungen

Jeder HAL bringt kompilierten Microcode in Form eines Byte-Arrays für alle drei ALUs, der RPU, TPU und des Xpecs mit sich, der unter Closed-Source steht. Weiterhin wird C-Code bereit gestellt, welcher erlaubt, den jeweiligen Microcode einer ALU auf diese zu laden. Dies muss in der Initialisierungsphase eines Programms, welches eine Kommunikation mit Hilfe des Standard Ethernet HALs aufbaut, geschehen. Anschließend erhält jeder Kommunikationskanal des NetX direkten Zugriff auf die jeweilige Speicherbank des internen SRAMs. Ab diesem Zeitpunkt arbeitet jeder Kommunikationskanal autonom, ohne die ARM CPU während eines Sende- oder Empfangsvorgangs zu belasten.

3.4.2 Funktionsweise

Der Abstraction Layer wird auf ARM-Seite ausgeführt und arbeitet unabhängig von jedem Kommunikationskanal. Er ist nach dem Prinzip von Queues aufgebaut und stellt somit eine FIFO-Datenstruktur da. Die Queues sind aufgeteilt in eine Datenstruktur für genutzte Nachrichten und eine für freie Plätze. Wird ein Frame durch die ALU der RPU oder die der TPU reserviert so wird er aus der Queue für freie Plätze entfernt und in die Belegt-Queue eingehangen. Die Freigabe eines Frames erfolgt auf genau dem entgegengesetzten Wege. Alle Datenstrukturen sind direkt auf die jeweilige Bank des internen SRAMs gemapped und verwalten so den gemeinsam genutzten Speicher für den Empfangs- und Sendevorgang. Das bedeutet auch, dass gleichzeitiges Reservieren von zu sendenden Frames die Empfangskapazität des gleichen Kommunikationskanals um die reservierte Anzahl einschränkt und umgekehrt.

Speicherausnutzung des HAL:

$$Offset + (Framesize + 2 * Timestamp + Reserved + Management) = Banksize \quad (3.1)$$

$$1560\,Bytes + (1518\,Bytes + 2 * 4\,Bytes + 18\,Bytes + 16\,Bytes) = 32760\,Bytes$$

Jeder Kommunikationskanal hat genau eine Speicherbank des internen SRAMs mit einer Größe von 32768 Bytes zur Verfügung. Ein Frame belegt 1560 Bytes, welche sich aus den

1518 Bytes eines Standard Ethernet-Frames, 2 * 4 Bytes der Timestamping-Einheit, 18 Bytes reservierten Daten sowie 16 Bytes der Queue-Verwaltungsstrukturen zusammensetzen. Jede Bank des SRAMs wurde somit in 21 * 1560 Bytes große Blöcke unterteilt. Der erste Block der Bank 0 kann jedoch nicht für das Protokoll genutzt werden, da er sich im Adressbereich der Interruptvektortabelle befindet. Um die Bank 0 nicht als strukturelle Anomalie zu den übrigen Bänken, die besondere Beachtung in der Verarbeitung erfordern würde, sehen zu müssen, wurde diese Aufteilung auch auf die höheren Bänke übertragen. Somit stehen für jeden Kommunikationskanal effektiv 20 Frames zur Kommunikation über Ethernet zur Verfügung.

3.4.3 Anwendung

Um eine Nachricht abzusenden ist es notwendig zuvor einen leeren Frame vom HAL anzufordern. Somit erhält man direkten Zugriff auf einen explizit für diesen Frame reservierten Teil des Speichers. Ist das beschreiben des Speichers abgeschlossen so wird dem HAL über eine entsprechende Funktion signalisiert, dass der Frame zum Senden bereit steht und von diesem anschließend abgeschickt. Das Empfangen von Frames geschieht autonom und erfordert keinen Einfluss durch den ARM-Core. Der HAL bietet für alle Ereignisse einen Interrupt an. Somit ist es möglich auf alle zur Kommunikation relevanten Zeitpunkte, wie Beginn eines Sendevorgangs, Abschluss des Sende oder Empfangsvorgangs, zu reagieren. Die einzige Ausnahme bildet der Beginn eines Empfangsvorgangs, welcher jedoch durch die Integrierte Timestamping-Einheit (vgl. Kapitel 3.3.1 auf Seite 18) festgehalten und somit zurückverfolgt werden kann.
Durch einfache Modifikation des Codes des HALs ist es möglich die Zuordnung der Speicherbänke neu zu definieren. Hierdurch lässt sich z.B. realisieren, dass ein Port nicht nur eine, sondern beliebig viele Bänke zum Ablegen und Reservieren von Nachrichten zur Verfügung hat. Dies erhöht die maximale Aufnahmekapazität eines Ports drastisch. Dies hat wiederum zum Nachteil, dass nicht mehr alle Kommunikationskanäle des NetX genutzt werden können, da eine eigene Speicherbank für jeden Kommunikationskanal fehlt. Es ist nicht möglich multiple Kommunikationskanäle auf den selben Speicherbereich zu verweisen. Zudem ist es nicht möglich einem Kanal einen anderen Speicher als eine Interne SRAM-Bank wie z.b. den externen SDRAM zuzuweisen, da in diesem Fall keine Möglichkeit einer parallelen Datenverteilung bestünde. Aus diesem Grunde wird in dieser Arbeit auf diese Möglichkeit verzichtet, da in aufbauenden Arbeiten weitere Kommunikationskerne z.B: zur gleichzeitigen Kommunikation über CAN eingesetzt werden sollen.

Kapitel 4

Konzept und Architektur

In diesem Kapitel werden zunächst die Anforderungen, welche an das Konzept gestellt werden aufgezeigt. Anhand derer und der Arbeitsweise des TTEthernet Protokolls wird anschließend der grundlegende Aufbau entwickelt. Weiterhin werden verschiedene Strategien zur Arbeitsweise des Prototyps aufgezeigt und diskutiert. Darauf aufbauend lassen sich Module definieren, anhand derer ein konzeptionelles Modell erstellt wird. Abschließend wird auf die Fehlerabhandlung und deren Darstellung eingegangen.

4.1 Anforderungen und Analyse

Die TTEthernet API sowie die Arbeitsweise des Protokolls stellen eine Reihe von Anforderungen an das Konzept. Diese lassen sich in einer hierarchischen Struktur gliedern und werden in Tabelle 4.1 auf der nächsten Seite zusammengefasst.

4.2 Aufstellung der grundlegenden Modellfunktionen

Aus den aufgestellten Anforderungen des Protokolls lassen sich grundlegende Module über die Funktionalität ableiten. Abbildung 4.1 auf Seite 24 zeigt den grundlegenden Aufbau des TTEthernet-Stacks.
Die Anbindung zum Netzwerk erfolgt über ein Hardware Abstraction Layer zu Ethernet fähiger Hardware. Dieser ermöglicht eine Kommunikation über das Standard Ethernet Protokoll IEEE 802.3, welches die Basis für das TTEthernet Protokoll bildet und zusätzlich eine Kommunikation mit möglichen Standard Ethernet Geräten innerhalb des Netzwerkes gewährleistet. Weiterhin wird so eine hohe Portabilität erreicht, da jede beliebige Standard Ethernet fähige Hardware genutzt werden kann, solange sie das Interface des HALs unterstützt. Die Module Transmitter und Receiver des TTEthernet-Stacks bauen hierauf auf.

Tabelle 4.1: Anforderungen des Konzepts

#	Anforderung
1	Der Prototyp soll alle Funktionen des TTEThernet-Protokolls unterstützen
1.1	Eine Konfigurationsdatei (config.c) soll interpretiert werden können
1.2	Der Prototyp muss als Endsystem in jedem TTEthernet-fähigem Netzwerk kommunizieren können
1.2.1	Alle Funktionalität als Synchronisationsclient soll unterstützt werden
1.2.2	Alle Funktionalität als Synchronisationsmaster soll unterstützt werden
1.2.3	Alle Funktionalität als Compression Master soll unterstützt werden
1.3	Die Kommunikation muss das Senden und Empfangen von Time-Triggered Nachrichten unterstützen
1.4	Die Kommunikation muss das Senden und Empfangen von Rate-Constrained Nachrichten unterstützen
1.5	Die Kommunikation muss das Senden und Empfangen von Best-Efford Nachrichten unterstützen
1.6	Das System muss die Abarbeitung von Time-Triggered Tasks ermöglichen
1.7	Es müssen Callback-Routinen an Empfangende Nachrichten gebunden werden können
1.8	Ein Mechanismus soll das System an eine netzwerkweite Zeit synchronisieren können
1.8.1	Der Zeitpunkt von eingehenden Nachrichten muss möglichst genau zu bestimmen sein
2	Die Konfiguration des Prototypen soll konform mit der Toolchain aus dem Hause TTTech sein
2.1	Die API muss erfüllt werden
2.1.1	Eine Konfigurationsdatei (config.c) muss vom Endsystem interpretiert werden können
3	Das Zeitverhalten des Prototyps soll eine realistische Automotive Anwendung widerspiegeln können
3.1	Der Jitter des Systems soll kleiner 1 µs betragen
3.2	Die Länge eines geschedulten Zyklus muss im einstelligen Millisekundenbereich konfigurierbar sein
3.3	Die zeitliche Auflösung der geschedulten Ereignisse soll möglichst hoch sein. Eine Schrittweite von 10 µs ist zu unterschreiten
4	Die Stabilität des Systems darf unter keinen Umständen gefährdet sein
4.1	Bursts von Nachrichten dürfen das System nicht in seiner Arbeitsweise beeinflussen
5	Eine Portabilität des Endsystem auf beliebige Hardware ist wünschenswert
5.1	Der Ansatz muss auf einer Standard-Ethernet fähigen Plattform aufbauen

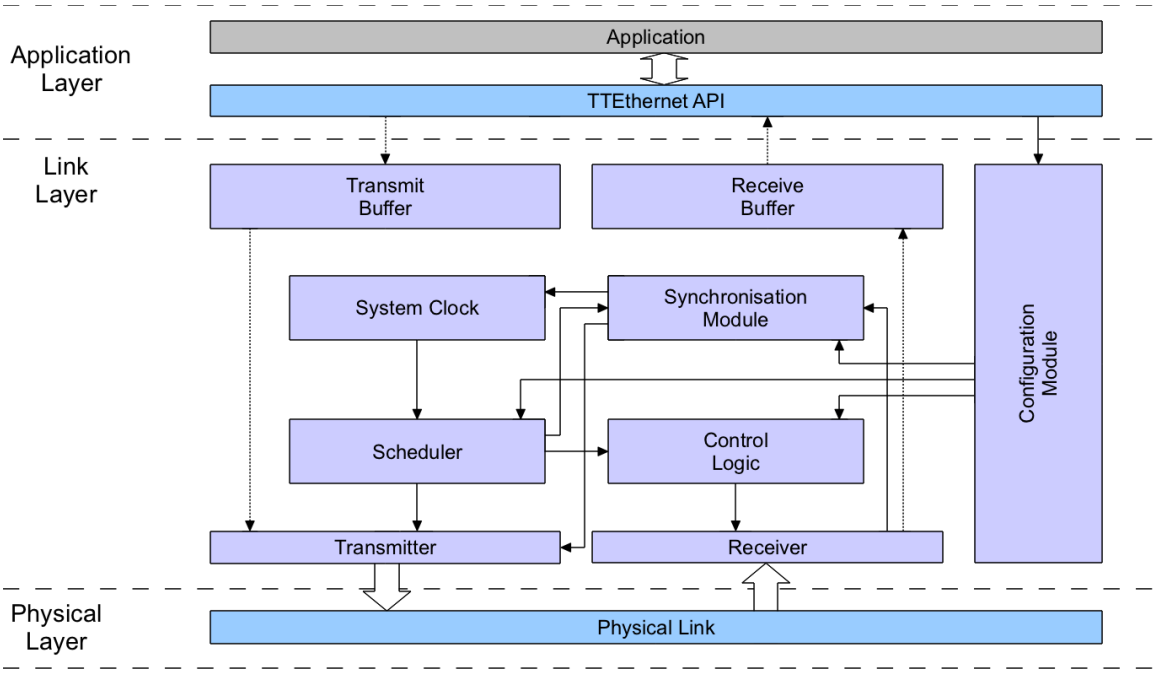

Abbildung 4.1: Basiskonzept des TTEthernet-Stacks

Der Receiver überprüft jede ankommende Nachricht ob es sich um einen Time-Triggered, Rate-Constrained oder Best-Efford Frame handelt. Anschließend leitet er ihn an den entsprechenden Buffer weiter. Diese Buffer können von einer beliebigen Anwendung heraus über die TTEthernet API abgerufen werden. Jedes Öffnen eines Empfangs-Buffers liefert die aktuelle Nachricht. Hier ist bei der Umsetzung darauf zu achten, dass ein multipler Zugriff oder Überschneidungen in den Anfragen vom Receive-Modul und einer Anwendung nicht zur Herausgabe von ungültigen oder leeren Nachrichten führt. Ein Zugriff auf einen Buffer darf jedoch nie in einer Zugriffsverweigerung resultieren sondern muss immer erfolgreich sein um den größtmöglichen Determinismus zu erreichen. Hierzu ist erlaubt, dass ein Buffer immer das nächst ältere Element heraus gibt solange der Schreibvorgang des aktuellsten noch nicht abgeschlossen ist. Sollte kein älteres Element vorhanden sein so quittiert der Buffer dies mit einer "BufferemptyMeldung. Ein besonderer Fall bilden empfangende Protocol Control Frames. Diese werden an das Synchronisationsmodul weitergeleitet und somit der Algorithmus zur Synchronisation angestoßen. Das Synchronisationsmodul enthält alle Algorithmen der Synchronisationsrollen. Zur Initialisierungsphase des Prototyps wird das Synchronisationsmodul über das Konfigurationsmodul auf Synchronisationsmaster -client oder Compressionmaster eingestellt. Die Realisierung der Algorithmen erfolgt nach Spezifikation (Steiner, 2008) und beeinflussen die Systemzeit durch das entsprechende Modul, gleichen so Ungenauigkeiten des Quarzes aus und beheben Verschiebungen der Periode des Schedulers im Bezug zur netzwerkweiten Zeit. Das Endsystem bleibt somit stets Synchronisiert.

Ein Öffnen eines Transmit-Buffers ermöglicht einer Anwendung eine Nachricht TTEthernet konform zu senden. Jeder Buffer ist immer speziell für einen Nachrichtentyp ausgelegt und definiert somit, ob die zu sendende Nachricht als Time-Triggered, Rate-Constrained oder Best-Efford Verkehr interpretiert wird. Hier sind ebenfalls Zugriffsverweigerungen aufgrund multiplem Öffnens des selben Buffers von verschiedenen Anwendungen zu vermeiden. Ebenso darf es nicht zum Konflikt kommen sollten Anwendungen Nachrichten in einen Buffer schreiben, während der Transmitter auf diesen zugreift, um aktuell anstehende Nachrichten zu senden. Neue Nachrichten dürfen von jedem Buffer erst als gültig angesehen und nach außen weiter gegeben werden, sollte der Schreibvorgang vollständig abgeschlossen sein. Der Transmitter wird ebenfalls von dem als Synchronisations- oder Compression-Master konfiguriertem Synchronisationsmodul genutzt um eigene PCFs zu senden. Es ist anhand der Konfiguration ausgeschlossen und durch die Spezifikation nicht erlaubt, dass Time-Triggered Nachrichten zum Synchronisationszeitpunkt geschedult, somit Versendet werden und den Synchronisationsablauf stören könnten. Dennoch sind Rate-Constrained und Best-Efford Nachrichten nicht an den Schedule gebunden und können das Versenden eines PCFs behindern. Diese Verzögerung wird durch den Synchronisationsalgorithmus kompensiert, indem nicht der Zeitpunkt der Synchronisation im Schedule in den PCF geschrieben wird, sondern der tatsächliche Absendezeitpunkt des Frames.
Um Nachrichten sowie Tasks zeitabhängig ausführen zu können ist ein Scheduling notwendig, welches das Scheduler-Modul realisiert. Dieses wird durch die Systemzeit gespeist, das nicht nur die Geschwindigkeit der Zeiteinheit festlegt sondern auch den Nullpunkt des Schedules. Das Scheduling wird in der Initialisierungsphase, durch das Konfigurationsmodul, aus der Konfigurationsdatei (confi.c) ausgelesen und definiert. Im Betriebszustand ist eine dynamische Änderung des Schedules nicht notwendig, was den Prozess vereinfacht. Weiterhin hat das Scheduler-Modul die Aufgabe die Kontrolllogik anhand des aktuell aktiven Ereignisses zu konfigurieren. Die Kontrolllogik steuert den Datenfluss der empfangenen Nachrichten. Das Konfigurationsmodul liest in der Initialisierungsphase alle Strukturen der Konfigurationsdatei ein. Die Informationen werden genutzt um alle notwendigen Module zu parametrisieren. Weiterhin ist es jeder Anwendung über die TTEthernet API möglich zusätzliche Parameter zur Laufzeit festzulegen oder auszulesen.

4.3 TTEthernet als prioritätengetriebener Ansatz

Das Konzept des Prototyps sieht das Auftreten und die Verarbeitung von unterschiedlich klassifizierbaren Ereignissen vor. Ereignisse können direkt an den Schedule gebunden sein und so, regelmäßig zu definierten Zeitpunkten, ausgelöst werden. Zudem ist das Auftreten eines Ereignisses unabhängig dieses Zeitplans möglich. Unabhängige Ereignisse können durch das Empfangen einer Nachricht auf dem Kommunikationskanal oder das Ablaufen einer BAG einer Rate-Constrained Nachricht ausgelöst werden. Sollte zu diesem Zeitpunkt

ein geschedultes Ereignis eintreten so kommt es zu Überschneidungen in der Ausführung. Dies würde zu unvorhergesehenen Schwankungen in der Verarbeitung führen, was in einem nicht-deterministischen Zeitverhalten der Ereignisse resultiert. Durch eine geeignete Strategie kann das unvorhergesehene Auftreten von Ereignisüberschneidungen vermieden oder kontrolliert und ein reproduzierbares Verhalten erreicht werden.

4.3.1 non-preemtive

Ein Ansatz besteht darin alle Ereignisse zu planen. Um bei der Aufstellung des Zeitplans sicherzustellen, dass keine Überschneidungen auftreten können ist die Worst-Case Verarbeitungszeit jedes geschedulten Ereignises nötig. Zudem sind die Ereignisse, welche nicht an den Schedule gebunden sind in den Zeitplan zu integrieren. Hierzu gehört die Bestimmung der Sendezeitpunkte aller aktiven Rate-Constrained Nachrichten sowie das Zyklische abrufen des Empfangsbuffers des Receiver-Moduls.
In einer zyklischen Konfiguration des Receiver-Moduls empfängt und puffert die Hardware alle ankommenden Frames. Nach Ablauf eines definierten Zyklus wird der Inhalt des Empfangspuffers ausgelesen und verarbeitet. Hierbei ist die Periode des Zyklus klein genug zu wählen, damit keine Nachrichten durch einen Bufferüberlauf des Buffers verloren gehen können. Jedoch ist sie ebenfalls groß genug zu wählen um einen möglichst kleinen Overhead an Verarbeitungszeit durch regelmäßiges Abrufen in Kauf nehmen zu müssen. Die in dieser Arbeit gewählte Hardware hat die Möglichkeit in der Standardkonfiguration maximal 20 Frames (vgl. Kapitel 3.4.2 auf Seite 20) zu puffern. Da die Anzahl unabhängig von der Größe eines einzelnen Frames ist muss für eine verlustfreie Abarbeitung von der minimalen Framegröße zur Bestimmung der maximalen Periodendauer des Receiver-Moduls ausgegangen werden. Die Übertragungszeit eines minimalen Frames bei einer Übertragungsrate von 100 MBits beträgt $64 * 8 * 0,01 = 5,12$ µs. Somit beläuft sich die maximale Periodendauer mit Beachtung von Inter-Frame-Gaps (IFG) auf $(5,12 + 0,96) * 20 = 121,6$ µs.
Für Rate-Constained Nachrichten ist es nicht möglich eine lückenlose statische Berechnung aller Zeitpunkte des Schedules im Voraus zu bestimmen. Hierzu wäre es nötig, dass jede BAG ein Teiler der Periodendauer ist. Dies könnte in der Konfiguration erzwungen werden, indem nur echte Teiler der Periodendauer als Zeitbasis für eine BAG zugelassen werden, was jedoch eine Restriktion darstellen würde, die nicht konform zum TTEthernet-Protokoll ist. Somit käme nur eine dynamische Berechnung in Frage. In diesem Fall würde für jede Rate-Constained Nachricht nur der unmittelbar anstehende Ablaufzeitpunkt einer BAG in das Scheduling eingetragen. Ist dieser erreicht und konnte die Nachricht erfolgreich gesendet werden, so wird der nächste Ablaufzeitpunkt registriert. Dieser Ansatz ist jedoch nicht mehr mit einem schleifenlosen Scheduler realisierbar, da Suchalgorithmen benötigt werden, welche wiederum zu Unregelmäßigkeiten in der Verarbeitungszeit führen.

4.3.2 preemtive

Ein weiterer Ansatz besteht darin, aktive Ereignisse zu unterbrechen, sollte ein geplantes Ereignis anstehen. Hierzu ist eine hirachische Einteilung aller betroffenen Ereignisse notwendig. Diese wird meist über Prioritäten ausgedrückt. Diese Ermöglichen es dem Scheduler niederpriorisierte, unabhängig vom Schedule eintretende Ereignisse, wie das Ablaufen einer BAG einer Rate-Constraint Nachricht, durch höherpriorisierte geplante Ereignisse, wie das Senden einer Time-Triggered Nachricht, zu unterbrechen. Ist die Abarbeitung des höher priorisierten Ereignisses abgeschlossen so wird die Ausführung des niederpriorisierten fortgesetzt. Dieser Ansatz verlangt die Eintragung der Startzeitpunkte von ausschließlich geplanten Ereignissen in den Schedule, was diesen reduziert. Zudem sind alle Eintragungen zur Initialisierungsphase möglich, was einen statischen Schedule erlaubt. Somit vereinfacht sich der Algorithmus des Schedulers, indem er Schleifenfrei realisierbar ist und zu einer vorhersagbaren Verarbeitungszeit führt. Daher wird im Folgendem der preemtive Ansatz bevorzugt.
Die Abbildungen 4.2 und 4.3 zeigen Beispiele für die Verdeutlichung der Unterbrechungsstrategie. Hierbei wird zunächst ein störungsfreier Ablauf eines Ereignisses präsentiert (vgl. Abbildung 4.2 auf der nächsten Seite). Durch die Systemzeit angestoßen wird das System aus der Idle-Task gehoben und der Scheduler aktiviert. Dieser besitzt die höchste Priorität um gewährleisten zu können, dass geplante Ereignisse immer zu ihrem jeweiligen Zeitpunkt ausgeführt werden können. Er überprüft zunächst, welches geplante Ereignis zur Ausführung gebracht werden soll und aktiviert dieses. Durch die Beendigung des Scheduling-Vorgangs wird die Ausführung an das anstehende Ereignis weitergeleitet und die Senderoutine zum Absenden eines Time-Triggered Frames wird abgearbeitet. Somit bleiben eventuelle Verzögerungen, zwischen dem erreichen der Systemzeit und dem auslösen des entsprechenden Ereignisses, konstant. Anschließend fällt durch Beendigung der aktive Prozess wieder an die Idle-Task zurück. Die Ausführung eines Background-Tasks ist Schedule-unabhängig. Sie wird bearbeitet, sofern eine Anwendung eine Background-Task definiert hat und deren Ausführung fordert. Erreicht nun die Systemzeit den Zeitpunkt eines geplanten Ereignisses so wird die Ausführung der Background-Task vom höher priorisierten Scheduler unterbrochen. Dieser prüft erneut, welches Ereignis ausgeführt werden muss und aktiviert dieses. Nach Beendigung des Algorithmus beginnt das aktivierte Ereignis mit der Ausführung, da Time-Triggered Tasks höher priorisiert sind als Background Tasks, jedoch nicht so hoch wie das Scheduling selbst. Ist die Bearbeitung abgeschlossen so fällt die Ausführung an den unterbrochenen Background-Task zurück, der nun selbst zur Beendigung gebracht werden kann und die Aktivität an die Idle-Task zurück gibt.
In Abbildung 4.3 auf Seite 29 wird die rekursive Vorgehensweise des Ansatzes Vorgestellt. Eine Anwendung hat an dieser Stelle eine Background-Task definiert und zur Ausführung gebracht. Sie wird so lange ausgeführt bis die Systemzeit einen Zeitpunkt eines geplanten Ereignisses erreicht. Dies führt zur Unterbrechung der Background-Task und zur Aktivierung

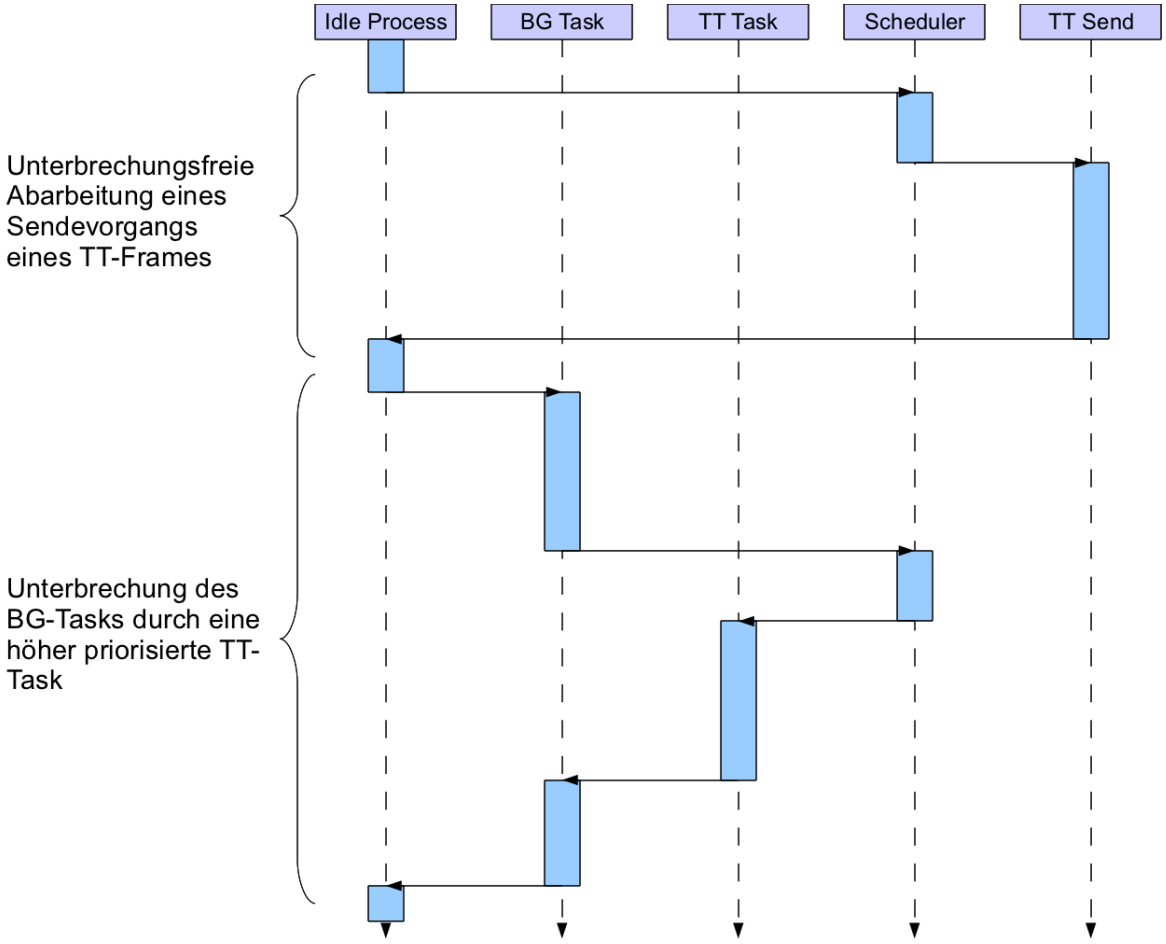

Abbildung 4.2: Beispiel für eine priorisierte Unterbrechung

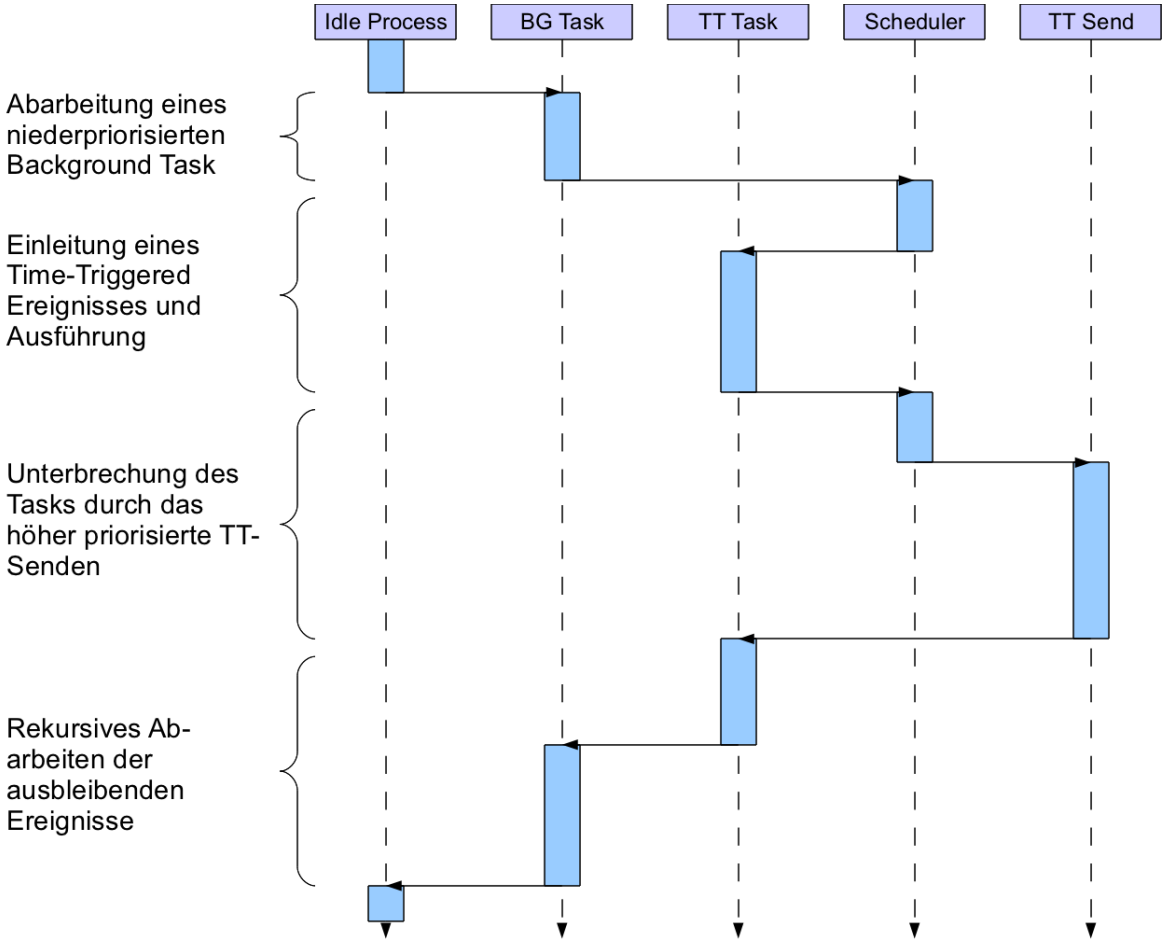

Abbildung 4.3: Beispiel für eine rekursive Unterbrechung

des höher priorisierten Scheduling-Algorithmus. Dieser prüft, welches Ereignis zur Ausführung gebracht werden muss und aktiviert eine Time-Triggered Task. Nach Beendigung des Algorithmus wird diese aktiviert, da sie in der Priorisierung höher liegt als die ausstehende Background-Task. Wird jedoch in dieser Zeit ein weiteres geplantes Ereignis von der Systemzeit erreicht, so wird das in der Bearbeitung liegende Ereignis unterbrochen, erneut das Scheduling gestartet und das entsprechende Ereignis aktiviert. Diese wird nach Abschluss des Schedulings bearbeitet, da die Priorität des Sendens einer Time-Triggered Nachricht höher liegt als die einer Time-Triggered Task. Nachdem das Versenden der Nachricht abgeschlossen werden könnte wird die Ausführung der Time-Triggered Task fortgesetzt. Die Aktivierung fällt rekursiv von höherwertiger zu niederwertiger Priorität zurück, bis wieder die Idle-Task erreicht wurde.

4.4 Erweitertes Modell

Aus den zuvor definierten Anforderungen und dem aufgestellten Prioritäten basierenden Ansatz lässt sich nun ein erweitertes Modell (vgl. Abbildung 4.4 auf Seite 32) erstellen. Zu erkennen ist hier die Aufteilung der Grundmodule in Subkomponenten, welche die jeweiligen Aufgaben der Module widerspiegeln. Das Konfigurationsmodul ist aus Gründen der Übersicht nicht in die Abbildung mit aufgenommen worden, da es alle Module durch die jeweilige Konfiguration beeinflusst. Seine Funktionsweise ist aus Kapitel 4.2 zu entnehmen.

4.4.1 Buffer-Modul

Transmit- und Receivebuffer sind in einen Bufferpool übergegangen, welcher auf den HAL-eigenen Buffer referenziert. Dies ermöglicht einen hohen Datendurchsatz bei möglichst niedrigem Verarbeitungsaufwand. Der Bufferpool ist wiederum in eine Reihe von spezialisierten Bufferpoole unterteilt. Diese sind für empfangende sowie zu sendende Nachrichten mit Best-Efford- Rate-Constrained sowie Time-Triggered-Verhalten zuständig. Dies ermöglicht das Klassifizieren einer Nachricht über den entsprechenden Buffer, was einen Zugriff, ohne aufwändige Suchroutinen ermöglicht. Die TTEthernet API sieht jedoch nur eine Differenzierung zwischen Critical Traffic und Background Traffic vor, weshalb ein Mapping der Buffer Rate-Constrained und Time-Triggered auf Critical Traffic Buffer erfolgt. Der Buffer für Best-Efford Nachrichten werden direkt als Background Traffic Buffer interpretiert.

4.4.2 Kommunikationseinheit

Der Transmitter erhält seine Daten aus allen drei Bufferpoolgruppen für zu sendende Nachrichten. Hierzu wird er über eine Weiterleitungslogik gesteuert, die bestimmt, welche Nachricht der Priorität nach versendet werden soll. Diese Information erhält er einerseits aus

dem Scheduling selbst sowie andererseits aus dem BAG-Counting Algorithmus der Rate-Constrained Nachrichten. So wird sichergestellt, dass zum Sendezeitpunkt einer Time-Triggered Nachricht das Interface nicht durch das Versenden einer anderen Nachricht blockiert ist. Eine Blockade würde zum Jittern der geplanten Nachrichten führen, womit nicht mehr garantiert werden kann, dass diese Nachricht vom Empfänger erhalten wird. Für Rate-Constrained Nachrichten wird das Interface nicht speziell reserviert, da dies zu einem starken Einbruch in der zur Verfügung stehenden Bandbreite für Best-Efford führen würde. Die BAG einer Rate-Constrained Nachricht definiert den frühsten Sendezeitpunkt der Nachricht, was daher eine zusätzliche Verzögerung erlaubt. Es werden vom Transmitter Rate-Constrained Buffer lediglich bevorzugt behandelt.
Der Receiver differenziert beim Empfang von Nachrichten zwischen Critical Traffic und Background Traffic anhand des Headers. Während Background Nachrichten direkt in die jeweiligen Best-Efford Buffer transferiert werden, müssen Critical Traffic Nachrichten noch in Time Triggered und Rate-Constrained unterteilt werden. Dies ist nicht über den Aufbau der Nachrichten selbst, sondern nur anhand des Schedulings möglich. Indem Sollankunftszeiten von Time-Triggered Nachrichten mit der empfangen Nachricht verglichen wird, kann entschieden werden, ob es sich um eine Time-Triggered oder Rate-Constrained Nachricht handelt. Ein besonderer Fall bilden die PCF, welche anhand ihrer ID direkt Klassifizeirt werden können und entsprechend an das Synchronisationsmodul weitergeleitet werden.

4.4.3 Planung von Ereignissen

Der Scheduler aktiviert das Senden von Time-Triggered Nachrichten sowie die Ausführung von Tasks. Hierbei ist es durch den preemtiven Ansatz erlaubt, dass sich Events überschneiden. Sie werden anhand ihrer Priorität unterbrochen und nach Abarbeitung aller höherpriorisierten Ereignisse weiter bearbeitet. So bleibt gewährleistet, dass der Zeitpunkt zum Senden einer Nachricht nicht durch die Ausführung einer Task beeinflussbar ist. Der Schedukler ist ausschließlich für die Aktivierung von Scheduleabhängigen Events zuständig. Das bearbeiten von BAGs der Rate-Constrained Nachrichten ist nicht direkt im Zeitplan festgelegt und wird somit auf ein externes Modul ausgelagert, welches für die Einhaltung der BAGs zuständig ist. Dieses signalisiert dem Transmitter jede Bereitschaft einer Rate-Constarined Nachricht, der diese dann Versendet, sollte kein geschedultes Ereignis anliegen.
Der Algorithmus des Synchronisationsmoduls wird durch das Empfangen eines PCFs ausgelöst. Dieser wird ausgewertet und die Synchronisation der Systemzeit dementsprechend durchgeführt. Ist das Synchronisationsmodul als Synchronisationsmaster oder Compressionmaster konfiguriert so wird zum geschedulten Synchronisationszeitpunkt ein PCF zum versenden an das Transmitter-Modul geleitet. Durch Einflussnahme des Scheduls ist an dieser Stelle garantiert, dass zu diesem Zeitpunkt keine anderen zu sendenden Nachrichten das Modul blockieren.

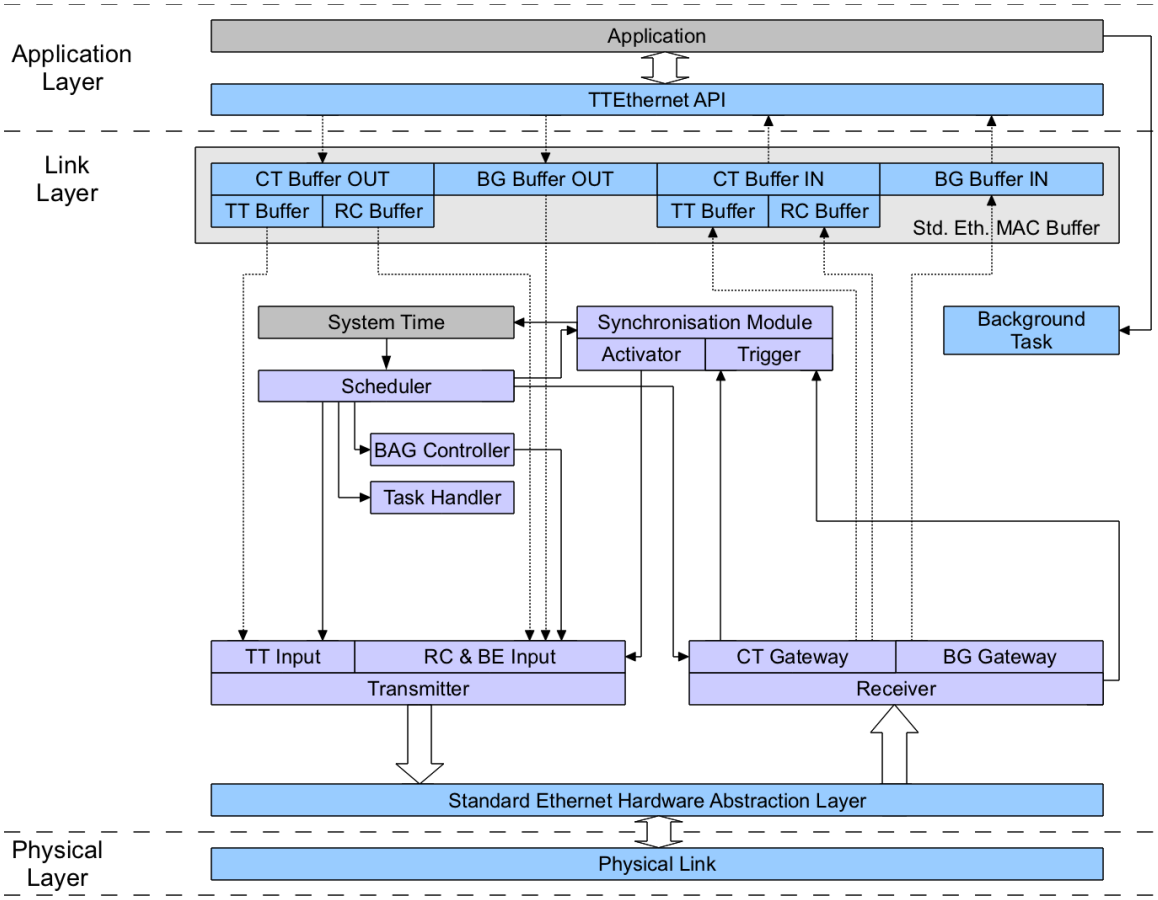

Abbildung 4.4: Konzept des TTEthernet Stacks

Tabelle 4.2: System-LED

Farbe	Aktivität	Zustand
orange	leuchten	Statusanzeige aktiv: System in der Initialisierungsphase
		Statusanzeige inaktiv: System ist in Betrieb. Scheduler ist inaktiv
orange	blinken	Das System ist in Betrieb. Scheduler ist aktiv
grün	blinken	Das System ist in Betrieb. Scheduler ist aktiv. System synchronisiert
		Die Statusanzeige zeigt den Genauigkeitslevel der Synchronisation
		Je ausgefüllter die Statusanzeige, desto genauer wird Synchronisiert

Tabelle 4.3: Port-LED

Farbe	Port	Zustand
aus	0	Modul für Port 0 ist hochgefahren. Kein Link
grün	0	Modul für Port 0 ist hochgefahren. Link OK. Modul ist aktiv
rot	0	Ein Fehler wurde im Modul für Port 0 festgestellt
		Ein Entsprechender Fehlercode ist an der Statusanzeige abzulesen
aus	1	Modul für Port 1 ist hochgefahren. Kein Link
grün	1	Modul für Port 1 ist hochgefahren. Link OK. Modul ist aktiv
rot	1	Ein Fehler wurde im Modul für Port 1 festgestellt
		Ein Entsprechender Fehlercode ist an der Statusanzeige abzulesen

4.5 Fehlerbehandlung und Signale

Das System erzeugt in der Initialisierungsphase einen Statusreport und gibt diesen am Ende der Phase über die serielle Schnittstelle aus. Der Report beinhaltet eine Versionierung des aktiven Stacks, sowie der TTEthernet API. Weiterhin wird eine Gesamtübersicht über die Interpretierung der Konfigurationsdatei gegeben. Fehler in der Konfiguration lassen sich somit zurückverfolgen. Zur Erkennung des aktuellen Status des Prototyps und zur Fehlerdiagnose ist eine Menge an zusätzlichen Signalen vorgesehen. Diese werden zur Laufzeit in der Idle-Task aktualisiert um den Gesamtablauf des Systems nicht zu beeinflussen. Die Signale unterscheiden sich in der System-LED (vgl. Tabelle 4.2, der Port-LED (vgl. Tabelle 4.3) und der Statusanzeige (vgl. 4.4 auf der nächsten Seite sowie 4.5 auf der nächsten Seite).
Es wurde bei der Statusanzeige bewusst darauf geachtet keine Zustände zu verwenden, die in ihrer Darstellung mit einer Synchronisierungsphase übereinstimmen und somit zu Fehldeutungen führen könnten.

Tabelle 4.4: Statusanzeige im Normalbetrieb

V12	V13	V14	V15	Zustand
0	0	0	0	System hochgefahren. System nicht oder nur schlecht synchronisiert
0	0	1	0	Initialisierung die Hardware IO-Lines
0	1	0	0	Initialisierung des Standard Ethernet Physical Layers
0	1	0	1	Initialisierung der Interruptlines und des IVCs
0	1	1	0	Initialisierung des Schedulers
1	0	0	0	Einlesen der TTEthernet Konfigurationsdatei (config.c)
1	0	0	1	Starten der Ethernet Port Module 0 und 1
1	0	1	0	Initialisierung und Öffnen des COM Ports
1	0	1	1	TTEthernet-Engine Startet
1	1	0	0	Erzeugung einer Systemkonfiguration

Tabelle 4.5: Statusanzeige im Fehlerfall

V12	V13	V14	V15	Zustand
0	0	1	0	Portmodul konnte nicht initialisiert werden
0	1	0	0	Fehler beim festlegen der MAC-Adresse des Controllers
0	1	0	1	Ethernetport konnte nicht auf Promisc Mode geschaltet werden
0	1	1	0	Fehler beim Zuweisen von Interrupt Request Leitungen
1	0	0	0	Portmodul konnte nicht gestartet werden
1	0	0	1	Das System hat einen Bufferüberlauf festgestellt
1	0	1	0	Nicht registrierte CT-Nachricht an Port empfangen. Nachricht verworfen
1	0	1	1	Nicht genug Systemspeicher zur Verarbeitung einer Nachricht
1	1	0	0	Zu viele registrierte Events im Schedule
1	1	0	1	Schwerer Systemfehler. System angehalten.

Kapitel 5

Realisierung

Dieses Kapitel zeigt die Realisierung des Konzeptes. In Detailbetrachtungen werden erneut Subkonzepte vorgestellt, die verschiedene Strategien der Umsetzung präsentieren. Diese werden verglichen und auf Effektivität geprüft. Es wird eine Geschwindigkeitsoptimierung erreicht, um eine problemfreie Ausführung des Systems während starker Belastung zu gewährleisten. Im Folgenden wird zunächst ein Überblick über die Konfiguration der Hardware gegeben. Dabei wurde verstärkt auf Faktoren wie Verarbeitungszeit, Sicherheit bei Speicherüberlauf und niedrigen Zeitlichen Varianzen geachtet.

Befehls- und Datencache

Aktiviertes Caching erzeugt einen signifikanten Geschwindigkeitszuwachs. Es ist jedoch darauf zu achten, dass Ausführungszeiten zyklischer Programmteile zu einem gewissen Grad vorhersagbar bleiben. Somit ist ein aktives Caching immer ein Kompromiss zwischen Geschwindigkeit und Determinismus der Zeiten. Alle Einstellungen bezüglich des Befehls- und Datencaches der NetX-CPU werden durch den Startupcode vorgenommen. Hier wird die Erneuerungsstrategie *Round-Robin* gewählt um ein möglichst vorhersagbares Verhalten der Caches zu erzeugen. Während die Aktivierung des Befehlscaches einen Geschwindigkeitszuwachs mit sich bringt erzeugt das Aktivieren des Datencaches starke Schwankungen in der Ausführungszeit. Dies führt zu Ungenauigkeiten in jeder geschedulten Aktion sowie den Synchronisationsroutinen von mehreren Mikrosekunden, was die Zuverlässigkeit des Gesamtsystems gefährdet und ist somit keine Alternative. Daher wird in der Umsetzung auf den Einsatz des Datencaches verzichtet.

Aufteilung des Speichers

Eine Aufteilung der Programmteile auf die zur Verfügung stehenden unterschiedlichen Speicherbereiche hat einen entscheidenden Einfluss auf die Ausführungsgeschwindigkeit.

Der schnellste Speicher des Boards ist der NetX interne *Tightly Coupled Memory*, da er ohne Waitestates auskommt. Ein weiterer Vorteil dieses Speichers gegenüber z.B. einem Datencache ist sein Determinismus in immer gleich bleibenden Abarbeitungszeiten. Da er mit 8 KByte nicht ausreicht um das gesamte Programm zu fassen, werden nur die Zeitkritischen Module welche zyklisch aufgerufen werden auf ihm ausgelagert. Diese sind der Scheduler, der Bufferpool und das Ethernet-Modul. Zudem werden die Stacks auf ihn verlagert, da auf ihnen die meisten Zugriffe pro Zeit erfolgen.
Einen weniger gravierenden, aber dennoch deutlich messbaren, Einfluss hat die Verteilung der restlichen Programmdaten und Instruktionen innerhalb des externen SDRAMs auf verschiedene Bänke. Der auf den NXHX500-ETM verwendete SDRAM MT48LC2M32B2 verfügt über 4 separate Bänke (vgl. Micron, 2008). Ein Zugriff auf eine Bank erfolgt immer über Waitstates. Während dieser Wartezeit ist es jedoch möglich bereits eine weitere Bank zu adressieren und so Teile der Verarbeitung zu parallelisieren. Hierzu muss der Programmcode über die Bänke verteilt werden. Die hier vorgestellte Umsetzung sieht vor, jedes Segment auf eine eigene Bank zu legen um so Instruktionszugriffe von Datenzugriffen zu entkoppeln. Auch andere Verteilungen, die sich z.B. an Modulen orientiert, wären denkbar. Die gesamte Verteilung des Programms lässt sich wie in Abbildung 5.1 auf der nächsten Seite dargestellt beschreiben.

5.1 Ressourcenausnutzung

Es wurde bei dieser Arbeit darauf geachtet eine möglichst effiziente Nutzung der Ressourcen der Hardware zu erreichen. Um das System sinnvoll nutzen zu können müssen genügend weitere Ressourcen für Anwendungen zur Verfügung stehen. Dieses Unterkapitel gibt einen Überblick über die Nutzung des Systems. Weiterhin werden in den folgenden Unterkapiteln der jeweiligen Module Möglichkeiten aufgezeigt, wie durch spezielle Konfigurationen, Ressourcen von ungenutzten Funktionen zusätzlich freigegeben werden können um die Realisierung spezieller Anwendungen zu ermöglichen. Zudem ist die Umsetzung zum Evaluierungsboard NXHX500-ETM sowie dem Schwesterboard NXHX500-RE kompatibel. Die Tabelle 5.2 auf Seite 38 zeigt die Nutzung der Ressourcen in der Standardkonfiguration des TTEthernet Stacks.

5.1.1 Verteilung der Prioritäten

Da es sich bei der Architektur um einen Prioritäten getriebenen Ansatz handelt und zudem alle potentiell aufkommenden Ereignisse und zur konzeptionellen Phase bekannt sind, bietet es sich an diese Struktur über die Interruptbehandlung des Controllers abzubilden. Jedem Ereignis wird somit eine eigenen IRQ Leitung zugeordnet, welche die Priorität repräsentiert. Anschließend ist es möglich den Ereignis treibenden Geräten jeweils eine Leitung zuzuwei-

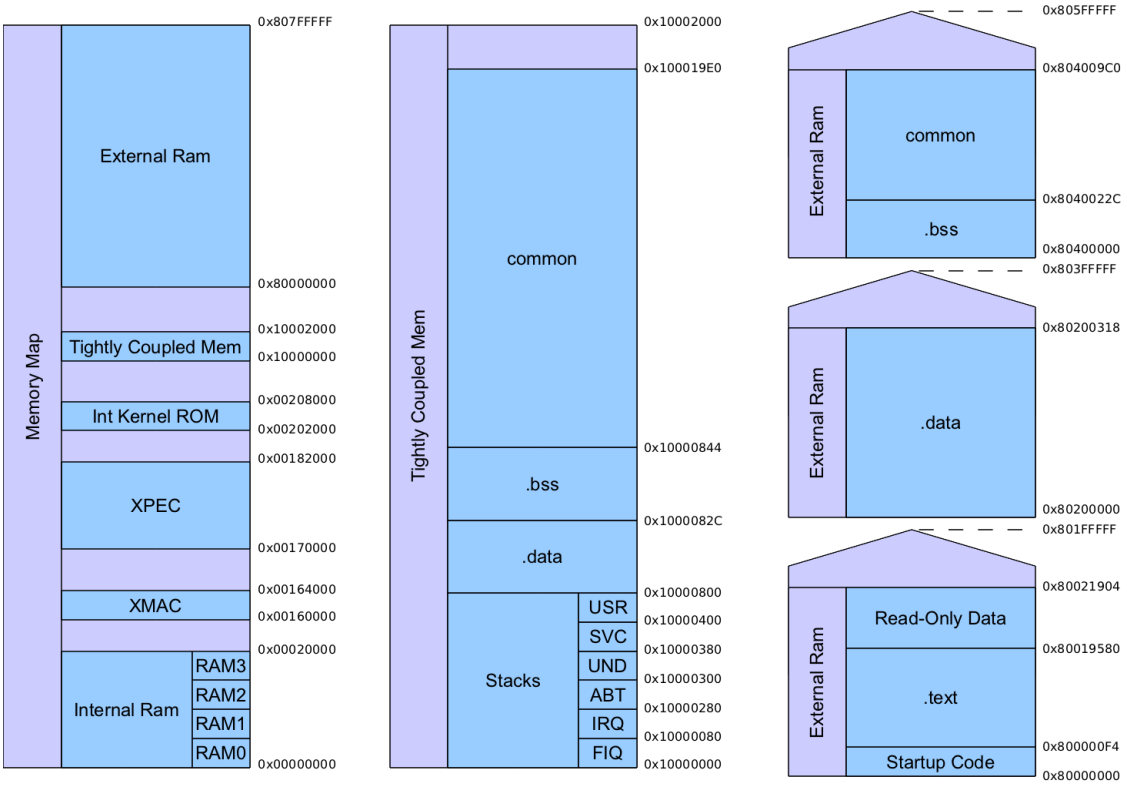

Abbildung 5.1: Übersicht der Speicherverteilung

Abbildung 5.2: Ressourcenverteilung des Prototyps

GPIO	Nutzung
0	UART 0 (Tx)
1	UART 0 (Rx)
2	UART 0 (CTS)
3	UART 0 (RTS)
4	Timer 0
5	Timer 1
6	Timer 2
7	Timer 3
8	LED V15
9	LED V14
10	LED V13
11	LED V12
12	open
13	Switch S30 1
14	Switch S30 2
15	Timer 4

Interrupt	Modul	Nutzung
1	Ethernet	Receive Port 0
2	Ethernet	Receive Port 1
3	Synchronisation	Trigger
4	Synchronisation	Activator
5	Synchronisation	Permanence
6	Scheduler	Send TT
7	Scheduler	Send BG
8	Scheduler	TT Task
9	System	Callback Task
10	System	Background Task
11	System	GPIO

Device	Modul	Nutzung
System Time	Scheduler	Timebase
Timer 0	Scheduler	IRQx Source
Timer 1	Ethernet	receive Port 0
Timer 2	Synchronisation	permanance point
Timer 3	open	
Timer 4	open	

sen. Es ist nicht zwingend erforderlich, dass diese statisch sind, sondern dürfen im Laufe des Betriebs wechseln, was z.B. beim Scheduling ausgenutzt wird um unterschiedlich priorisierte Ereignisse mit der Hilfe eines einzelnen echten Hardware-Gerätes zu treiben. Dies vermeidet unnötige Belastungen der stark begrenzten Hardwareressourcen. Zur Initialisierungsphase des Systems werden die Prioritäten der einzelnen Module, wie in Tabelle 5.2 auf der vorherigen Seite beschrieben, definiert. Eine Anwendung kann diese, während die TTEthernet Engine nicht gestartet ist, durch Neuzuweisung beliebig verändern um Priorisierungen anzupassen oder ungenutzte Funktionalität vollständig abzuschalten und die hierdurch freigewordenen Interruptleitungen für sich frei zu nutzen.

Im Folgenden wird auf die Implementierung der einzelnen Module innerhalb des Konzeptes eingegangen. Die Beschreibung beinhaltet den Aufbau sowie die Arbeitsweise und Besonderheiten in der Umsetzung.

5.2 Scheduling

Ein Scheduling beinhaltet die Einhaltung eines vordefinierten zeitabhängigen Plans. Er definiert, zu welchem Zeitpunkt ein bestimmtes Ereignis ausgelöst wird. Somit steht für ein präzises und direktes Scheduling das Erfassen der aktuellen Zeit sowie das punktgenaue Auslösen von Ereignissen im Vordergrund.

5.2.1 Erfassung der Zeit

Die Erfassung der Systemzeit ist über verschiedene Ansätze möglich. Der gängigste Ansatz eines Schedulers nutzt Micro- und Macroticks zur Erfassung der Zeitbasis (vgl. Kopetz (2004) und Grillinger u. a. (2006)). Die gewünschte Periode wird hierbei in äquidistante Abstände eingeteilt, welche die Macroticks repräsentieren. Ein Zähler wird eingesetzt dessen Periode genau einem Macrotick entspricht. Seine Zählschritte werden als Microticks bezeichnet und sind direkt von der Frequenz der betreibenden CPU abhängig. Der Interrupt bei Erreichen des Vergleichswertes des Zählers dient als Ereignis, in welchem eine Überprüfung statt findet, ob zu diesem Macrotick ein Eintrag in der Schedulingtabelle registriert wurde. Existiert ein Eintrag so wird dieser ausgeführt. Ist dies nicht der Fall so wird die Interruptserviceroutine beendet und der Zähler beginnt von vorne.
Dieser Ansatz fordert keine spezielle Hardware und lässt sich auf allen Mikrocontrollern, welche über ein Zählermodul verfügen realisieren. Ein Nachteil ist jedoch, dass sich gewünschte Ereignisse ausschließlich zu jedem Macrotick registrieren lassen. Die Auflösung der Schedulingeinträge ist somit direkt von der Verteilung der Macroticks, in einer Periode,

abhängig. Um eine möglichst hohe Auflösung zu erreichen ist es nötig eine hohe Dichte an Macroticks zu erzeugen. Dies wiederum bringt, selbst im Leerlauf, eine hohe Rechenlast mit sich, da zu jedem potentiellen Zeitpunkt eines Ereignisses die Tabelle des Schedulings auf einen gültigen Eintrag hin überprüft werden muss. Die gemessene Bearbeitungszeit der Interruptserviceroutine liegt bei 4,6 µs, was bei einer Auflösung von 100 µs der Auslastung des Systems von 4,6 % entspricht. Bei einer Auflösung von 10 µs beträgt die Auslastung bereits 46 % , selbst im Falle einer leeren Scheduletabelle. Bedenkt man, dass die minimale Paketgröße eines Ethernet Frames eine Übertragungszeit von 5,12 µs zuzüglich des Inter-Frame-Gaps von 0,96 µs aufweist, so müsste man selbst bei einer Auflösung von 10 µs einen Verlust der möglichen Übertragungsbandbreite von 39,2 % in Kauf nehmen.
Um diesem Problem entgegenzuwirken wird das NetX interne *Systemtime*-Modul verwendet (vgl. GmbH, 2008). Dieses ermöglicht Ereignisse, direkt mit ihrer, der Periode relativen Zeit zu registrieren, zu ausschließlich denen die Interruptserviceroutine gestartet wird. Hierdurch entsteht keine hohe Rechenlast in einer Leerlaufphase. Zwar sind alle ausgeführten Ereignisse stets mit einem Bearbeitungsaufwand (vgl. Tabelle 6.1 auf Seite 57) und somit mit einer Verzögerung bis zur eigentlichen Bearbeitung des Ereignisses behaftet, was sich jedoch herausrechnen lässt. Nach dieser Korrektur wird jedes Ereignis an exakt dem konfigurierten Zeitpunkt bearbeitet. Das Modul ist so konfiguriert, dass alle 16 ns das höherwertige Register um 1 inkrementiert wird. Die API verlangt eine Angabe der Zeiten in Nanosekunden, was ohne Umrechnung nicht auf die Systemzeit übertragbar ist. Diese word über den 100 MHz Systemtakt gespeist wird, was eine maximale Auflösung von 10 ns erlaubt. In dem die nächst höhere Zweierpotenz als Inkrement gewählt wurde ist es dem Compiler möglich Umrechnungen mit der Hilfe von Schiebeoperationen auf Geschwindigkeit hin zu optimieren. Weiterhin bleibt die Auflösung granular genug um die volle Bandbreite in der Übertragung nutzen zu können, obwohl es nur möglich ist, auf Zeiten des höherwertigen Registers Ereignisse auszulösen.
Ein Nachteil des Moduls gegenüber eines normalen Zählers liegt in einer fehlenden automatischen Rücksetzung nach erreichen der Periode. Ein hieraus nötiger manueller Eingriff in die Systemzeit erfordert einen Schreibzugriff auf den internen Systembus der NetX-CPU, da das Modul kein Teil des ARM-Cores ist (vgl. Kapitel 3.3.1 auf Seite 18). Eine statische Verzögerung ließe sich für einige Perioden bei der Rücksetzung herausrechnen. Jedoch würden sich über längere Zeit gesehen kleinste Abweichungen akkumulieren. Das System könnte zwar, konfiguriert als Client, über eingehende PCF-Frames eine Korrektur vornehmen doch ein Master (in einem Netzwerk ohne Compression-Master) oder ein Compression-Master würde das gesamte Netzwerk mit seiner Abweichung treiben, da es keine Referenzzeit gibt an der sich das System orientieren könnte. Zudem zeigen Tests, dass bei unterschiedlicher Auslastung des Systembusses schwankende Verzögerungszeiten auftreten, was keine zuverlässige und deterministische Festlegung der momentanen Systemzeit möglich macht. Dies würde zu unvorhersehbaren Schwankungen der Zeitbasis und somit aller auf ihr basierenden Aktionen führen.

Aus diesem Grunde wird der Ansatz gewählt die Systemzeit in keiner Weise manuell zu beeinflussen. Um jedoch weiterhin die Zeitpunkte der Ereignisse, welche relativ zur aktuellen Periode vorliegen, in allen Perioden registrieren zu können ist eine Umrechnung der relativen Zeiten in absolute notwendig. Dies lässt sich durch Zählen der aktiven Periode oder Mitführung des zeitlichen Offsets erreichen. Letzteres bietet eine Umrechnung ohne Operationen der Division oder Multiplikation. Weiterhin führt ein Überschlag des höherwertigen Registers der Systemzeit bei diesem Ansatz zu keinem Problem, da der Offset in einer Speicherzelle der gleichen Bitbreite gespeichert wird und so selbst den Überschlag nachahmt. Auch kann es erst bei der Verrechnung mit der relativen Zeit zum Überschlag kommen, was wiederum zur korrekten absoluten Zeit führt, weswegen die Mitführung eines Offsets die bevorzugte Umsetzung ist.
Somit wird ein präzises Scheduling erreicht, welches keine Last im Leerlauf erzeugt und eine Auflösung der zu planenden Ereignisse von 16 ns erlaubt.

5.2.2 Architektur

Aufbereitung der Schedulingtabelle

Wie im Architekturteil angesprochen wurde bei der Implementierung darauf geachtet einen schleifenlosen Scheduler umzusetzen. Ziel ist es Algorithmen, welche einen nicht linearen Aufwand aufweisen, aus der Betriebsphase auszulagern und somit einen möglichst geringen Overhead an Verarbeitungsaufwand von eintretenden Ereignissen zu erreichen. Um dies zu erreichen ist eine chronologisch geordnete Ringstruktur erforderlich, in der alle Ereignisse vorzuliegen haben. So ist es zu jeder Zeit möglich durch entsprechende Zeigerarithmetik das Folgeereignis direkt zu bestimmen und es muss nicht auf Suchalgorithmen zurückgriffen werden. Um die Ordnung der Struktur aufrecht zu erhalten ist jedoch vor jeder Registrierung eines neuen Ereignisses in das Scheduling die richtige Position zu ermitteln, was zu einem rechnerischen Mehraufwand hingegen des traditionellen Ansatzes mit Schleifen führt.
TTEthernet arbeitet mit Hilfe eines statisches Zeitplans, was bedeutet, dass zur Laufzeit keine Einträge dem Scheduling hinzugefügt werden müssen. Alle Ereignisregistrierungen werden somit zum Start des Systems durchgeführt. Der Arbeitsaufwand wird somit von der Betriebsphase in die Initialisierungsphase verlagert.
Registrierte Ereignisse weisen die Eigenschaften *Ereignisklasse*, *Ausführungszeitpunkt*, *Deadline*, *Priorität* sowie zwei Nutzparameter auf. Alle Eigenschaften sind durch die TTEthernet-API frei definierbar, während die Nutzparameter durch die Ereignisklasse spezifiziert werden.
Die *Ereignisklasse* beschreibt die Art des zu planenden Eintrags. Ein *Ausführungszeitpunkt* ist zur Periode des Schedulings relativ und definiert den exakten Zeitpunkt zu dem das Ereignis eintritt. Da das System trotz aller Optimierungen aufgrund von Bearbeitungslatenzen immer eine statische Verzögerungszeit aufweist wird diese, bezogen auf die jeweilige Ereig-

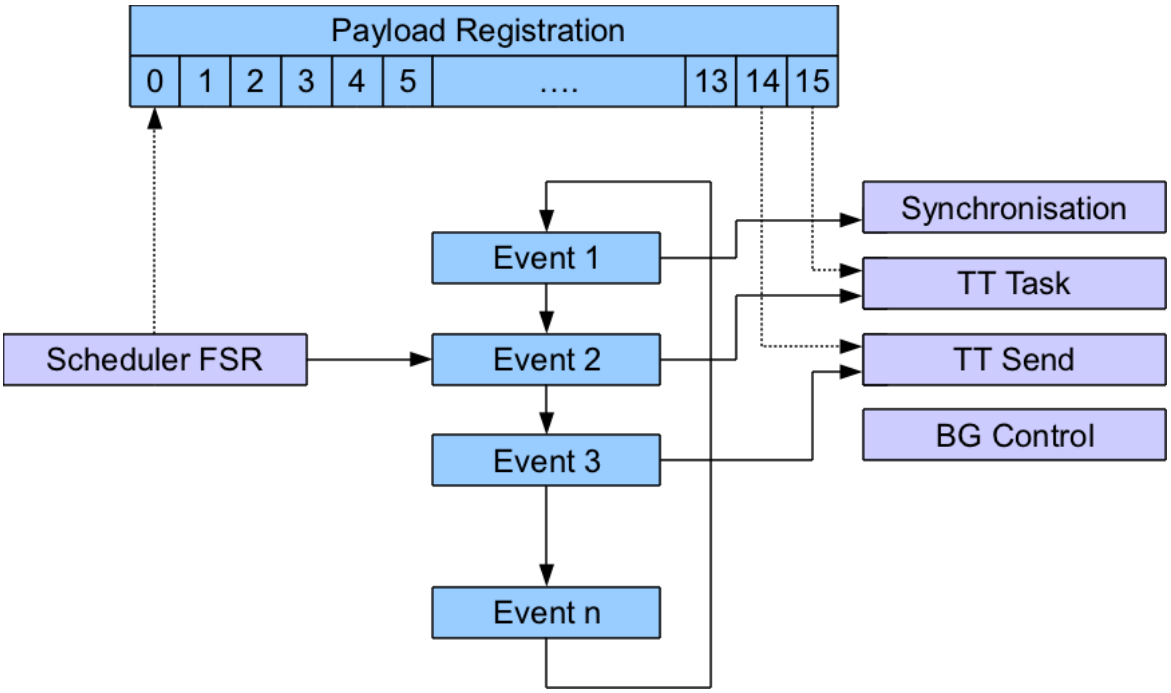

Abbildung 5.3: Aufbau des Schedulings

nisklasse, eingerechnet und der Zeitpunkt zum auslösen des Ereignisses dementsprechend in der Registrierung vorgezogen. Die *Deadline* bestimmt wie lange ein Ereignis das System für sich beanspruchen darf. Wird diese Zeit z.B. von einer Task überschritten so kann eine Warnung an den Programmierer der Anwendung abgegeben werden. Ein Ereignis wird jedoch nicht unterbrochen, egal wie lange es in der Bearbeitung benötigt. Welche Ereignisse sich unterbrechen dürfen wird durch die *Priorität* festgelegt. Dieser Wert ist äquivalent mit der Priorität des Interrupts (vgl. Abbildung 5.2 auf Seite 38).
Die Nutzparameter beziehen sich jeweils auf die Ereignisklasse und sind wie in Tabelle 5.1 beschreibt definiert.

Tabelle 5.1: Nutzparameter in Abhängigkeit der Ereignisklasse

Ereignisklasse	Parameter A	Parameter B
Synchronisation	Bufferelement	
Send TT	Bufferelement	
TT Task	Task	Argument
BG Control	Port	Messagecount

Arbeitsweise im Betriebszustand

Das Scheduling nutzt die Architektur der Interrupts des Controllers zur Verteilung der Arbeitszeiten aus. Hierbei wird die FIQ-Leitung an das Modul der Systemzeit gebunden. Die Interruptserviceroutine des FIQs kann nicht direkt zum Ausführen geplanter Ereignisse genutzt werden, da in diesem Falle alle Ereignisse die gleiche Priorität aufweisen würde. Dies hätte zur Folge, dass kein Ereignis ein anderes unterbrechen könnte. Die FSR generiert daher einen Interrupt mit der jeweiligen Priorität des anstehenden Ereignisses um somit die Ausführung an die jeweilige ISR zu delegieren (vgl. Abbildung 5.3 auf der vorherigen Seite). Hierbei wird ein Hardwaregerät verwendet, welches auf die Interruptleitung des jeweils kommenden Ereignisses umgeschaltet wird. So kann gewährleistet werden, dass Jedes Ereignis in seiner spezifizierten Priorität bearbeitet wird. Weiterhin werden alle Informationen, die zum Verarbeiten des anstehenden Interrupts notwendig sind in einen Buffer geladen, aus dem sich jeder Interrupt zu Beginn seiner Ausführung die für ihn relevanten Daten läd. So ist selbst bei gleichzeitiger Aktivierung verschiedener Interrupts gewährleistet, dass jede Routine ihre Informationen erhält. Auch ist so eine Verkettung von mehreren gleichartigen Ereignissen möglich, sollte vor dem Abschluss einer Verarbeitung schon eine erneute Anfrage auf das gleiche Ereignis erfolgen. Erst nachdem der FSR vollständig abgeschlossen ist werden die anstehenden ISRs ihrer Priorität nach entsprechend abgearbeitet, da die Interruptleitungen solange aktiv bleiben, bis die entsprechende ISR ihn zurücksetzt. Dies dient als Sicherheit im Fehlerfall, sollten sich Abläufe im Scheduling überlagern so werden die anstehenden Ereignisse soweit verzögert bis ein geeigneter Zeitslot vorhanden ist. Die FSR registriert zudem das nächst anstehende Event für direkte Verarbeitung bei erreichen der entsprechenden Systemzeit.
Weiterhin wird die Ausführung von extern eingeleiteten Ereignissen unterstützt, den Callbacks, was den Schedulingansatz zu einem hybriden Ansatz führt. Diese Ereignisse reihen sich in der Verarbeitung des bestehenden schedules ein und unterliegen der gleichen prioritäten-getriebenen Verarbeitung. Das heißt, sollte ein höher priorisiertes Ereignis noch in der Verarbeitung sein so wird das externe Ereignis so lange verzögert, bis nur noch niederpriorisierte Scheduleinträge, welche hierdurch in ihrem Ausführungszeitpunkt verzögert werden, anstehen.

5.3 Bufferpool

Architektur

Der Bufferpool beinhaltet getrennte Datenstrukturen für eine Menge von Input- und Outputbuffer. Die Abbildung 5.4 auf der nächsten Seite verdeutlicht das Zusammenspiel der verschiedenen Ebenen des Bufferpools, bezogen auf konkrete in einem bestehenden Datenverkehr einbezogene Buffer.

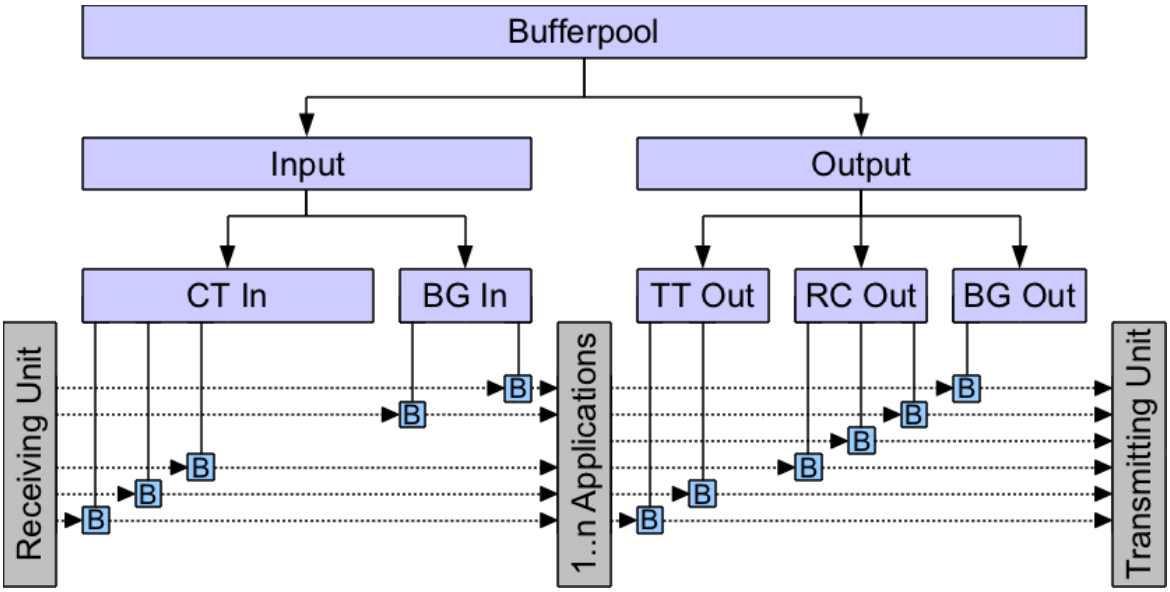

Abbildung 5.4: Hierarchie der Bufferpoole

Der Pool der Outputbuffer ist wiederum in einen für jede Nachrichtenart unterteilt. Somit stehen getrennte Bufferpools für TT- RC- sowie BG-Nachrichten zur Verfügung. Auf der Seite des Inputbufferpools stehen nur getrennte Pools für CT- und BG-Nachrichten zur Verfügung, da zum Zeitpunkt des Empfangs einer Nachricht anhand des Frames nicht zwischen TT- und RC-Nachricht unterschieden werden kann. Dies wäre nur anhand des Empfangszeitpunkt und im Vergleich zum aktuell aktiven Schedule möglich (vgl. Hintergrund, TTEthernet, Echtzeitnachrichten). Durch diese hierarchische Trennung wird eine vollkommene Unabhängigkeit der beiden Buffertypen erreicht, was unter anderem ermöglicht, gleiche Buffer für eingehende sowie ausgehende Nachrichten zu definieren. Weiterhin kann eine Suche nach bestimmten Buffern in unterschiedlichen Ebenen angesetzt werden, je nach dem wie umfangreich die zu durchsuchende Menge maximal sein muss. Dies reduziert den Zeitaufwand einer Suche der Anfrage entsprechend auf ein Minimum.
Jeder Buffer ist einem der fünf Pools zugeordnet. Alle Buffer werden zur Initialisierungsphase erzeugt. Dies ist Möglich, da für das TTEthernet-Protokoll zur Konfigurationszeit bereits bekannt sein muss welche CT-Nachrichten ein Knoten senden bzw. empfangen wird. Für jeden gültigen Regestrierungseintrag einer Nachricht wird ein Buffer mit allen dort definierten Parametern erzeugt. Nachrichten, die nicht in der Konfigurationsdatei des Endsystems registriert wurden werden verworfen, um das System gegen einen störenden Einfluss von außen abzusichern. Dies kann im Falle einer Fehlkonfiguration zum Verlust von kritischen Daten führen, weshalb dieses Verhalten über eine Warnmeldung bekannt gemacht wird. Zudem werden je ein eingehender sowie ausgehender Buffer für BG-Nachrichten erzeugt. Die Aufnahmekapazität dieser Buffer kann beliebig konfiguriert werden. Ein Spezialfall bildet eine Kapazität

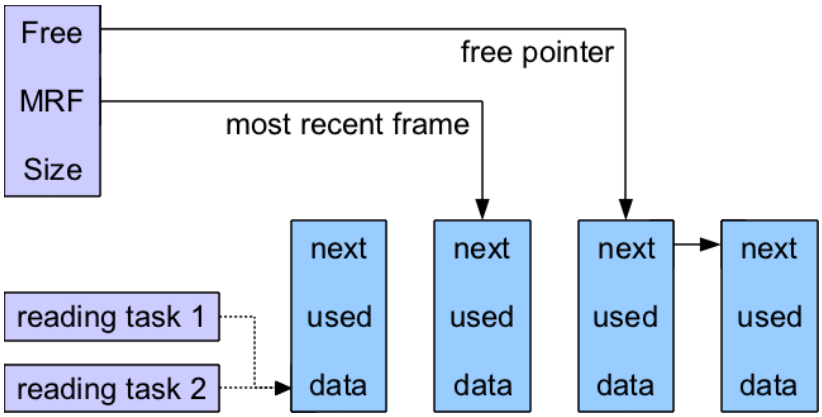

Abbildung 5.5: Aufbau eines Cyclical Asynchronous Buffers

von ’0’, welche diese Buffer deaktivieren würde, womit jeglicher BG-Verkehr verworfen wird. Nach der Erzeugung aller CT-Buffer sind diese, bezogen auf die Kommunikationsrichtung, eindeutig über ihre CT-ID adressierbar. Über das TTTech-API ist es nun möglich zu jeder CT-ID einen Handler auf den jeweiligen Buffer anzufordern. Dies sollte ebenfalls in der Initialisierungsphase der Anwendung erfolgen, da Suchalgorithmen die eindeutige Zuordnung von CT-ID und Bufferhandler ermitteln. Ist diese Zuordnung einmal bekannt so bleibt sie die gesamte Laufzeit des Stacks über erhalten. Alle weiteren Zugriffe auf den durch den Handler definierten Buffer erfolgen Schleifenfrei und sind somit Linear im Aufwand und der Komplexität. Diese Vorgehensweise erreicht die kleinstmögliche Verarbeitungszeit.
Ein Buffer kann jederzeit von mehreren Anwendungen gleichzeitig benutzt werden. Somit kann er ebenfalls als Synchronisationsmittel der Kommunikation zwischen multiplen Anwendungen eingesetzt werden. Jeder Buffer sieht eine Verarbeitung vor, die Threadsave ist. Wird auf einen Buffer mehrfach zugegriffen so ist es vom Typ des Buffers abhängig, wie seine Reaktion ausfällt (vgl. Kapitel 5.3.1 auf der nächsten Seite und 5.3.2 auf der nächsten Seite).
Sobald ein Buffer eine Schreibanfrage erhält fordert er einen Systemframe vom Memorypool an. Ist dieser Frame nicht mehr in Benutzung und wird aus dem Buffer entfernt so gibt er den Systemframe wieder an den Memorypool frei. Der Memorypool verwaltet eine Menge von vorinitialisierten Systemframes, die alle Bufferstrukturen unterstützen. Das erzeugen des Pools mit allen Systemframes findet ebenfalls in der Initialisierungsphase statt, sodass Allokation und Deallokation während der Betriebsphase gering im Zeitaufwand sind. Der Memorypool stellt ein Abbild des für alle Buffer gemeinsam zur Verfügung stehenden Speichers dar. Durch die Nutzung des Standard Ethernet HALs ist es nicht möglich Nachrichten anhand ihrer CT-ID oder des CT-Markers in unterschiedliche Bereiche des Speichers aufzunehmen. Um dennoch für jeden Buffer sicherzustellen, dass nur Nachrichten seines eigenen Typs in ihm enthalten sind wird diese Zuordnung über die Systemframes hergestellt, indem sie über einen Handler der Speicherbänke des Standard Ethernet HALs verfügen.

5.3.1 Doublebuffer

Ein Doublebuffer gibt auf eine Leseanfrage immer die aktuellste Nachricht zurück. Das Element wird nicht aus dem Buffer entfernt, sodass mehrfaches Lesen der Gleichen Nachricht möglich ist. Erst wenn die aktuelle Nachricht durch eine neue ersetzt wird kann die nun veraltete freigegeben werden. Die Realisierung erfolgt über einen Cyclical Asynchronous Buffer (CAB), dessen Aufbau durch Abbildung 5.5 auf der vorherigen Seite verdeutlicht wird.
Der Buffer verfügt über eine Kopfstruktur. Diese enthält einen Zeiger auf ein aktuell beschreibbares leeres Element, einen weiteren Zeiger auf das zur Zeit aktuelle Element sowie die Größe des Buffers. Erfolgt ein Schreibzugriff auf den Buffer so fordert dieser einen leeren Frame aus dem Memorypool an. Dieser wird jedoch noch nicht in den Buffer eingehangen um einen Konflikt zwischen gleichzeitigem Schreiben und Lesen zu vermeiden. Ist der Schreibprozess abgeschlossen so wird der neue Frame als *Most Recent* markiert und alle weiteren Lesezugriffe werden auf ihn referenziert. Weiterhin wird geprüft ob der vorherige Most Recent Frame keine aktiven Lesezugriffe mehr besitzt und anschließend ggf. wieder an den Memorypool freigegeben. Existieren noch offene Zugriffe so ist gewährleistet, dass der letzte sich beendende Zugriff die Freigabe des Frames veranlasst. Lesezugriffe werden aufaddiert um einen Zugriff auf den Most Recent Frame zu signalisieren. Hat eine Anwendung einen Lesezugriff signalisiert so kann sie die Referenz auf den Frame nutzen ohne Gefahr zu laufen, dass dieser zwischenzeitlich wieder freigegeben wird. Erst nachdem sie den Lesezugriff aufhebt kann, sollte es sich um den letzten Lesezugriff gehandelt haben, der Frame vom Buffer wieder freigegeben werden. Prinzipiell besitzt ein Doublebuffer eine feste Größe von zwei Feldern, da er nicht wächst. In dieser Umsetzung ist es jedoch möglich einen Doublebuffer mit einer beliebigen Größe zu initialisieren um einen Zugriff aus multiplen Anwendungen heraus zu unterstützen. Da jede einzelne Anwendung mit einem aktiven Lesezugriff auf den Buffer ein Element blockieren kann und es somit von einer Wiederverwendung abhält, sollte die Größe eines Doublebuffers immer die Anzahl der Anwendungen um 1 inkrementiert betragen. Dies stellt sicher, dass alle Anwendungen zu jeder Zeit erfolgreich gleichzeitig auf den Buffer zugreifen können, selbst wenn zudem noch ein aktives Schreiben oder Lesen durch das Ethernet Interface erfolgt.
So wird sichergestellt, dass der Buffer immer den aktuellsten Frame als Rückgabe liefert. Die einzige Ausnahme, in der ein Doublebuffer keine Elemente enthalten kann, ist zum Erstellungszeitpunkt. In diesem Fall wird ein Lesezugriff auf den Buffer verweigert.

5.3.2 Queued Buffer

Ein Queued Buffer liefert das aktuellste Element und entfernt bei einer Leseanfrage immer die gelesene Nachricht. Multiple Leseanfragen auf den selben buffer liefern somit nie die gleiche Nachricht. Durch eine Schreibanfrage neu eingehangene Frames werden erst wieder gelesen wenn alle aktuelleren Frames bereits ausgelesen wurden. Es handelt sich somit um

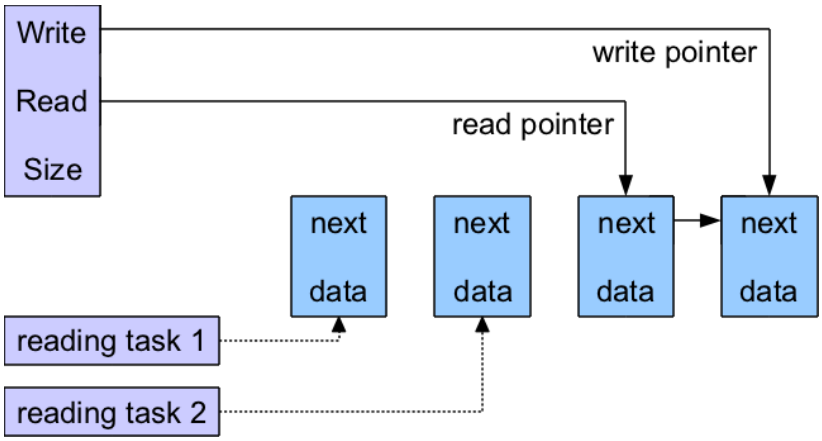

Abbildung 5.6: Aufbau eines Queued Buffers

eine FIFO-Struktur. Abbildung 5.6 verdeutlicht den Aufbau der Datenstruktur. Sie wird durch eine Queue realisiert.
Es wurde bei der Umsetzung bewusst darauf verzichtet die Struktur über einen Ringbuffer abzubilden. Ein Ringbuffer würde zwar durch eine Vorinitialisierung der gesamten Ringstruktur in der anschließenden Verarbeitung einen Geschwindigkeitsvorteil mit sich bringen, aber im Laufe des Betriebs fragmentieren. Eine solche Fragmentierung entsteht wie folgt: Jeder Lesezugriff blockiert die Wiederverwendung des jeweiligen Frames. Werden aus einer Reihe von blockierten Frames segmentweise Lesezugriffe aufgehoben so entstehen freie Bereiche zwischen den Blockierten Frames. Auf schleifenlose Arithmetik begrenzt, ist es sehr schwierig diese Bereiche zu identifizieren und einem Schreibvorgang zur Wiederverwendung zur Verfügung zu stellen. Es wäre notwendig für jede mögliche Fragmentierung genügend Adressierungsstrukturen bereit zu stellen um alle Frames bei einer Schreibanfrage direkt verwenden zu können. Dies stellt einen Mehraufwand an Ausführungszeit dar, der den erstgenannten Geschwindigkeitsvorteil nicht mehr rechtfertigen würde.
Die Realisierung als Queue erfolgt über einen Schreibe- und Lesezeiger. Bei einer Schreibanfrage wird überprüft ob sich noch leere Frames an die Queue hängen lassen. Gründe für das Fehlschlagen dieser Prozedur könnten z.B. ein voller Memorypool oder das erreichen der maximalen Grösse der Queue sein. Bei einem Fehlschlag wird der älteste Frame überschrieben, auf dem kein aktiver Lesezugriff herrscht. Mehrfacher Zugriff aus verschiedenen Anwendungen heraus könnte mit diesem Ansatz zu einem Blockieren des Buffers führen, so dass kein neues Element mehr geschrieben werden kann, bis mindestens eine Anwendung ihren Zugriff wieder frei gibt. Daher ist es zu empfehlen einen Queued Buffer mindestens mit der Grösse der Anzahl der Anwendungen um 1 inkrementiert zu definieren. So ist zu jeder Zeit sichergestellt, dass alle Zugriffe auf den Buffer erfolgreich bleiben. Nachdem ein Schreibvorgang vollständig abgeschlossen ist wird der neue Frame an die Queue angehan-

gen um eine Verarbeitung zu garantieren, die Thread-Sicher ist. Damit ist es für eine lesende Anwendung zu dieser Zeit erst möglich einen Zugriff auf den neuen Frame zu erhalten. Eine Leseanfrage auf den Buffer hängt den aktuellen Frame aus der Queue aus und bietet anschließend den Handler auf den Frame an. Damit ist er für andere Anwendungen nicht mehr sichtbar. Der Handler bleibt so lange gültig bis die Anwendung den Lesezugriff beendet, womit die Gültigkeit des Frames erlischt und er wieder an den Memorypool zurückgeben wird.
Diese Bufferart wächst und schrumpft somit mit der Anzahl der Frames. Sie kann jedoch nie überlaufen, da alte Frames in diesem Fall durch neue ersetzt werden.

5.3.3 Interface zur Definition von Buffertypen

Der Bufferpool bietet ein eigenes Interface an, mit dessen Hilfe es Anwendungsprogrammierern möglich ist eigene Bufferstrukturen zu definieren, sollten Queued- oder Double Buffer die Anforderungen der jeweilige Anwendung nicht erfüllen. Alle Bufferstrukturen, die dieses Interface nutzen, bedienen sich des Memorypools um ihre Systemframes anzufordern, genau wie Queued- und Doublebuffer.
Ein neuer Buffer verfügt über Parameter wie Größe, eine Kommunikationsrichtung die festlegt ob er eingehende oder ausgehende Nachrichten verwaltet, einen Handler auf den zuletzt bearbeiteten Frame, sowie alle Parameter, die über die Nachricht festgelegt werden, für deren Verarbeitung er zuständig ist. Für den neuen Buffer müssen fünf Routinen implementiert werden. Eine Initialisierungsroutine, die ausgeführt wird sobald der Stack startet und in der Konfiguration Nachrichtendefinitionen findet, die auf den neuen Buffertyp verweisen. Weiterhin muss jeweils eine Routine vorhanden sein, welche eine Schreibanfrage auf den Buffer einleitet, und eine, die diese wieder beendet. Zudem muss eine Routine definiert werden, die fähig ist eine Leseanfrage einzuleiten und schließlich auch eine, die die Leseanfrage wieder beendet. Diese können dann vom System genutzt werden um alle Zugriffe auf einen Buffer des neuen Typs umzusetzen. Es wird ausdrücklich empfohlen alle Funktionen Threadsicher zu implementieren, um eine Beeinflussung verschiedener Anwendung auf einander über gleichzeitige Bufferzugriffe zu verhindern. Eine detaillierte Definition über alle Funktionen und ihrer Parameter findet sich in der Headerdatei zum Modul Bufferpool.

5.4 Synchronisation

Die Implementierung der Synchronisation unterscheidet sich in ihrem Ansatz von dem Konzept und der Spezifizierung durch TTTech. Die Spezifikation sieht für den Client und den Master unterschiedliche Automaten als Lösung vor. Ein solcher Ansatz ist im Bezug auf Wiederverwendbarkeit des Codes und Modularität ungeeignet. Daher wird hier eine Modulare Umsetzung präsentiert, die die Funktionsweise des Masters als zusätzliches Modul ansieht,

die der Clientfunktionen hinzugefügt werden. Ein Master muss über Clientfunktionalität verfügen, um eine Möglichkeit zu haben, sich mit anderen Synchronisationsmastern mit der Hilfe eines Compressionmasters auf eine einheitliche Zeitbasis einstellen zu können. Alle Clientfunktionen werden über ein Rx-Modul abgehandelt, welches empfangene Synchronisationsframes verarbeitet und die lokale Zeit entsprechend anpasst. Ein zusätzliches Tx-Modul ist für das zusammenstellen und Versenden von Synchronisationsframes zuständig. Dies ermöglicht eine dynamische Umschaltung von Client- und Masterfunktionalität während eines hochgefahrenen TTEthernet Stacks. Im Fehlerfall kann somit ein dynamisches Downgrade eines Endsystems von Masters zu Client durchgeführt werden womit der Knoten weiterhin erfolgreich an der Netzwerkkommunikation teilnehmen kann. Ebenso wäre es Denkbar, dass ein Client den Ausfall eines Masters kompensiert indem er das Mastermodul aktiviert und somit die Funktionalität des ausgefallenen Knotens übernimmt.

5.4.1 Rx-Modul

Die Synchronisation der lokalen Zeit auf die netzwerkweite Zeit wird durch den Empfang eines PCFs eingeleitet. ID und Typ dieses Frames sind frei über die Konfiguration bestimmbar. Es werden nur Synchronisationsframes ausgewertet, die dem definierten Profil entsprechen. Die Synchronisationsroutine ist als Callback-Funktion an den Synchronisationsbuffer gebunden. Wurde ein neuer PCF in den Buffer eingetragen so wird das Rx-Modul aktiviert. Die Daten des Frames werden ausgewertet und die übertragene Verzögerungszeit bestimmt. Anschließend wird die dynamische Empfangsverzögerung ermittelt. Diese bestimmt sich aus der Zeitdifferenz zwischen der aktuellen Systemzeit und dem Empfangszeitpunkt des Frames, welcher anhand der Framebezogenen Timestamps ermittelt wird. Eine dynamische Bestimmung der Empfangsverzögerung bezogen auf jeden Synchronisationsvorgang ist notwendig, da die Ausführung des Callbacks nicht jede Periode zum gleichen Zeitpunkt erfolgen muss, was eine Variation in der benötigten Zeit vom Erhalt des Frames bis hin zur eigentlichen Synchronisationsdurchführung hervorrufen kann. Dieser Vorgang kompensiert die Variation. Zudem fließen statische Verzögerungszeiten in die Ermittlung der Gesamtverzögerung mit ein. Zum einen die Empfangsverzögerung, welche durch die Ausführungszeit des vergangenen bis hin zur Aktualisierung der Lokalen Zeit noch kommenden Codes bedingt ist. Zum anderen die Signalverzögerung, die aus der Länge des Kabels resultiert, das für die Übertragung der Nachricht genutzt wurde. Die Gleichung 5.1 beschreibt die vollständige Zusammenstellung der Gesamtverzögerungszeit:

$$\begin{aligned} TransparentClockNew = {} & TransparentClockOld \\ & + DynamicReceiveDelay \\ & + StaticReceiveDelay \\ & + WireDelay \end{aligned} \tag{5.1}$$

Ist die Gesamtverzögerung bekannt so wird bis zum Permanence Point der Nachricht mit der weiteren Ausführung der Synchronisation gewartet (vgl. Kapitel 2.3.3 auf Seite 9). Dieser bestimmt sich aus der Differenz der maximalen bisher aufgetretenen Verzögerung und der aktuell errechneten. Ist der Zeitpunkt erreicht, so ist sichergestellt, dass für den Synchronisationsframe jeder Periode das System die gleiche Verzögerung in der Verarbeitung aufweist.

Permanence Point

Nach erreichen des Permanence Points wird die Lokale Uhr an die netzwerweite Zeit angepasst. Hierzu lässt die gewählte Hardware zwei Möglichkeiten zu. Die Systemzeit kann direkt gesetzt oder indirekt beeinflusst werden, indem die Schrittweite der Lokalen Uhr verändert wird. Mit Hilfe des letzten Ansatzes ist es somit möglich die Systemzeit auf einem Controller langsamer oder schneller ablaufen zu lassen. Das direkte Setzen der Systemzeit bringt den Vorteil mit sich, dass keine Verzögerungszeiten auftreten bis eine Synchronisation unter den Netzwerkteilnehmern stabil ist. Der Nachteil ist dabei jedoch, dass ein fehlerhafter Synchronisationsframe direkt zu einem Unsynchronisierten Netzwerk führt, was zur Folge hätte, dass in dieser Phase keine TT-Nachrichten mehr übermittelt werden könnten. Wird nur die Geschwindigkeit der Systemzeit angepasst, so bringen vereinzelte Fehler in der Übermittlung von Synchronisationsframes das Endsystem nicht direkt in einen nicht synchronisierten Zustand. Ein auf dies weise arbeitendes System müsste sich nach einem Kaltstart, oder sollte es durch äußere Einflüsse dennoch in einen nicht synchronisierten Zustand geraten sein, über eine gewisse Zeit wieder an die netzwerkweite Zeit anpassen. Im folgenden wird eine Variante vorgestellt, die beide Lösungswege in sich vereint.
Die Abweichung der Systemzeit zur netwerkweiten Zeit wird in verschiedene Bereiche aufgeteilt, welche die schwere der Abweichung repräsentieren (vgl. Abbildung 5.7 auf der nächsten Seite). Die Bereiche bilden in der Standartkonfiguration eine Schrittweite von 2,5 µs und können zur Initialisierungsphase durch eine Anwendung angepasst werden. In welchem Synchronisationsbereich sich zur Zeit das System befindet wird auf der LED-Statusleiste herausgeführt, welche die jeweiligen Bereiche repräsentieren.
Je weiter sich die Ist-Zeit des Systems von der Soll-Zeit entfernt, desto kritischer ist die Synchronisation gefährdet und desto härter muss ein Gegensteuern erfolgen. Gegengesteuert wird mit Hilfe eines Regelalgorithmus um die Vorteile einer Trägheit des Systems auszunutzen. Hierzu ist es nicht ausreichend nur die Differenz von Ist-Wert zu Soll-Wert zu betrachten und anhand dieser die Schrittweite der Systemzeit zu erhöhen oder zu verringern. Eine entsprechende Regelung würde gleichmäßig um den Sollwert pendeln, da sie mit dem gleichen Faktor überschwingt wie sie den Ausgleich herstellt. Selbst eine Dämpfung in Abhängigkeit zum Abstand von Soll- und Ist-Wert verbessert nur marginal das Verhalten, da eine Schwingung auf diesem Wege nicht unterdrückt werden kann. Es ist daher notwendig eine Dämpfung in Abhängigkeit der Geschwindigkeit, also der Ableitung des Weges nach

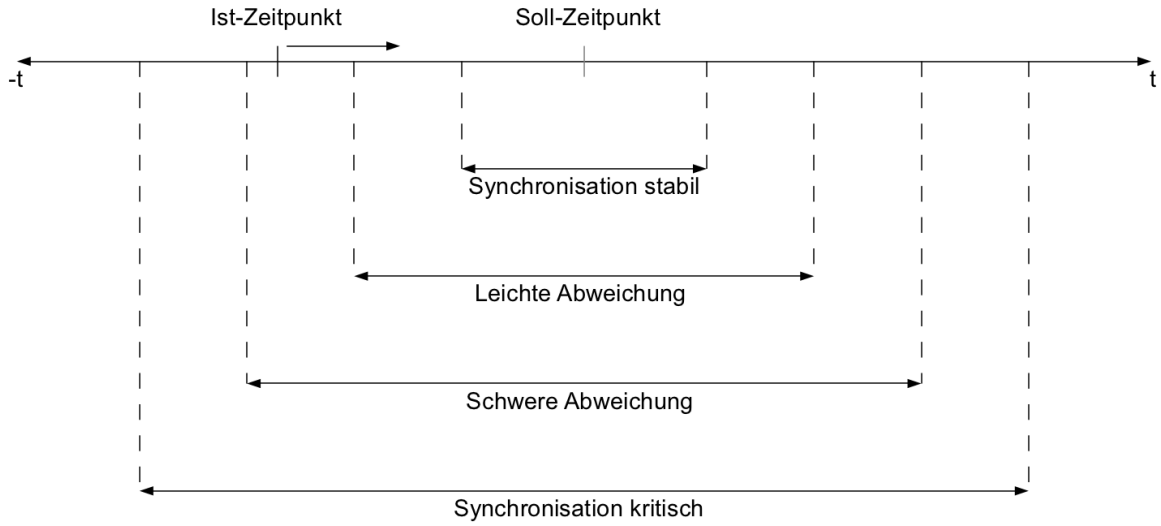

Abbildung 5.7: Abweichungszonen im Verhältnis zum Grad der Synchronisierung

der Zeit, einzubeziehen. Dies bringt eine Schrittweite mit sich, die nicht nur von der Position der Ist-Zeit auf dem Zeitstrahl abhängig ist sondern auch von der Bewegungsrichtung und ebenso der Bewegungsgeschwindigkeit. Sie ist wie folgt definiert:

$$Schrittweite = (Abweichung * \alpha) - (Geschwindigkeit * \beta) \tag{5.2}$$

Abweichung und Geschwindigkeit sind direkt sowie indirekt von der Schrittweite abhängig, womit ein Regelkreis entsteht. α und β stellen Faktoren dar, mit deren Hilfe die Regelung parametrisiert werden kann. Mit ihnen kann der Einfluss der Abweichung bzw. der Geschwindigkeit auf das Regelergebnis konfiguriert werden. Für alle definierten Bereiche der Abweichung können unterschiedliche Regelalgorithmen mit eigener Parametrisierung verwendet werden. Die Implementierung nutzt für alle Stufen den oben beschriebenen Algorithmus, jedoch mit unterschiedlichem Härtegrad. Der Härtegrad der Regelung nimmt mit steigender Annäherung zum Soll-Wert zu. Somit wird gewährleistet, dass sporadische Abweichungen das System je weniger beeinflussen können desto besser es sich auf die netzwerkweite Zeit synchronisiert hat. Dieses Verhalten wird über die Parameterkombination festgelegt. Die Tabelle 5.2 auf der nächsten Seite zeigt alle Parameter, welche in den jeweiligen Abweichungsstärken gelten.
Gerät der Grad der Abweichung außerhalb der spezifizierten Bereiche so wird keine Regelung eingesetzt sondern eine direkte Korrektur der Systemzeit durchgeführt. Dies reduziert die Einschwingzeit des Systems bei Kaltstarts und liefert dennoch eine entscheidende Stabilität bei sporadischen fehlerhaften Synchronisationsframes. Somit werden durch diesen Kompromiss die Vorteile beider Ansätze kombiniert.

Tabelle 5.2: Parameter in Abhängigkeit der Abweichungsstärken

Abweichungsstärke	α	β
Stabile Synchronisation:	1	10
leichte Abweichungen:	2	15
starke Abweichungen:	5	30
Synchronisation Kritisch:	10	100

5.4.2 Tx-Modul

Das Tx-Modul besteht aus zwei Komponenten. Einem Ereignis im Schedule, welches das Versenden des PCFs auslöst und einem an dieses Ereignis gebundene Callbackroutine, die für die Vorbereitungen eines neuen, im nächsten Zyklus zu versendenden, Synchronisationsframes zuständig ist. So kann ein präziser Sendezeitpunkt garantiert werden. Dieser ist unabhängig zu der Laufzeit der Algorithmen, welche die Vorbereitung des Frames durchführen. Das Sendeereignis nutzt nicht das SendTT-Ereignis des Schedulers sondern definiert ein eigenes mit eigener Priorität. Somit hat das Absenden eines Synchronisationsframes die höchste zu dieser Zeit auftretende Priorität und garantiert eine konstante Latenz und einen möglichst geringen Jitter. Die Callbackroutine stellt alle Parameter des Frames zusammen, so dass zum Sendezeitpunkt ausschließlich die Permanence Clock eingetragen werden muss. Diese Eintragung ist mit einer statischen Verzögerung behaftet, welche sich direkt in den einzutragenden Wert einrechnen lässt. Dieser Ansatz hilft, einen möglichst präzisen Protocol Control Frame für die Synchronisation aller Teilnehmer des Netzwerks zu erzeugen.

5.5 Dropping of Frames

Da der zur Verfügung stehende Speicher stark begrenzt ist, müssen spezielle Mechanismen verhindern, dass es durch überlaufen der Buffer zu einem Verlust von Nachrichten und somit von zeitkritischen Daten kommen kann. Hierzu werden zwei Mechanismen durch Konfiguration bereit gestellt. Ein Ansatz lässt sich über die Konfiguration der Buffergrößen realisieren. Alle Buffer verfügen über einen internen Dropping-Mechanismus, für ungültig gewordene Nachrichten. Dieser Verwirft die älteste Nachricht, sollte der Buffer auf eine Schreibanfrage hin keine neuen Nachrichten mehr aufnehmen können. So ist es möglich durch Konfiguration der Buffergrößen, den zur Verfügung stehenden Speicher in keinem Fall überschreiten zu können, was einen fehlerfreien Ablauf garantiert. Ein weiterer Ansatz besteht darin Zyklisch nach veralteten Nachrichten zu suchen und diese wieder frei zu geben, sollte der aktuell Verwendete Speicher zu ausgelastet sein. Hierbei ist zu beachten, dass die Periode der Überprüfung möglichst groß gewählt ist um möglichst wenig Overhead an Verarbeitungszeit zu erzeugen, dennoch es zwischenzeitlich nicht zu einem Speicherüberlauf kommen kann.

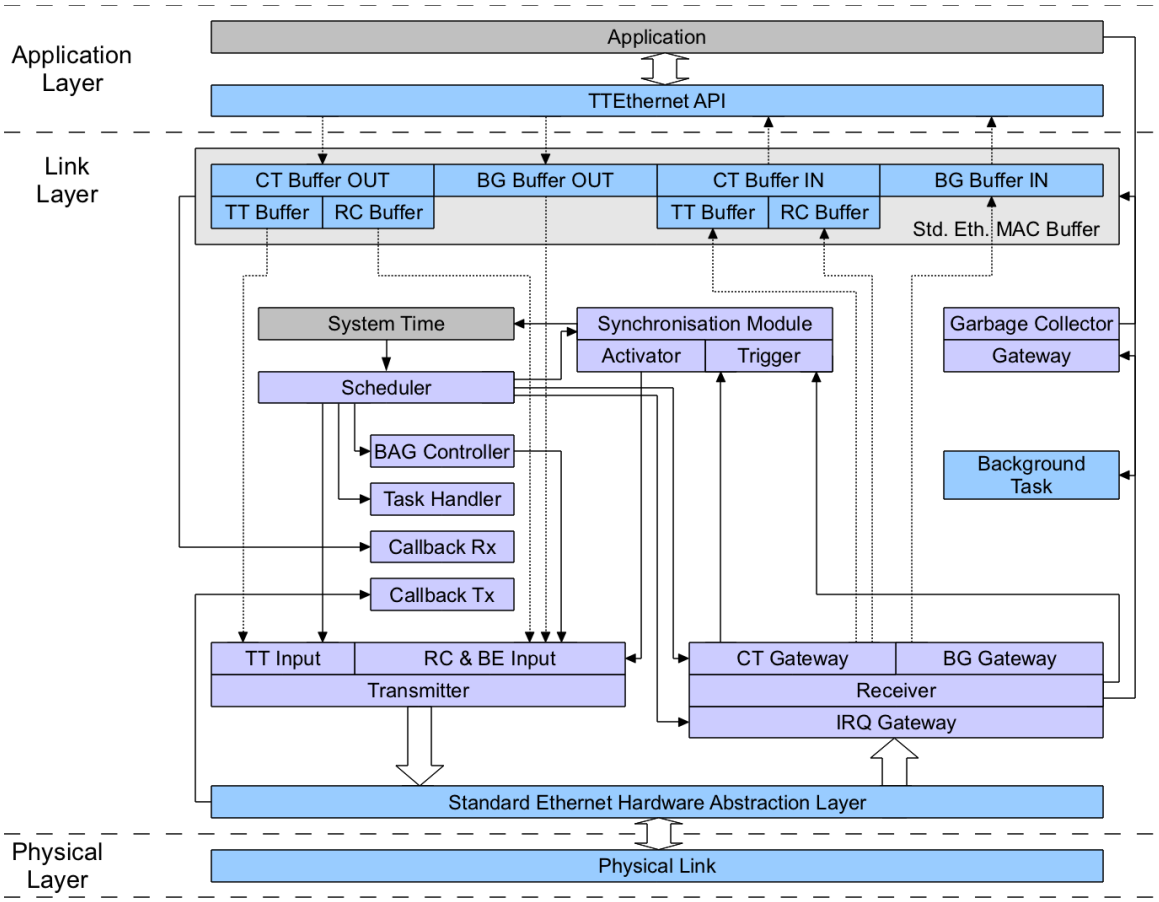

Abbildung 5.8: Aufbau des Systems

die Standardkonfiguration sieht hier eine Periode von 60 µs vor und wird eingeleitet, sollte mehr als die Hälfte des Speichers belegt sein. Dies garantiert, dass zwischen zwei Perioden kein Verlust von kritischen Daten durch Überschreiben belegter Speicherbereiche auftreten kann. Alle Parameter lassen sich durch eine Anwendung auf die jeweiligen Gegebenheiten anpassen.

5.6 Umgesetztes Konzept

In diesem Kapitel wurden viele Teile des bestehenden Konzeptes weiterentwickelt und verfeinert. Aus diesem iterativen Vorgang aus Umsetzung und Konzeptbildung entsteht der endgültige Aufbau des Systems. Die Behandlung von Callback Routinen ist in die Interruptverarbeitung mit eingeflossen (vgl. Kapitel 5.2.2). Zudem ist es dem Receiver nun möglich über ein IRQ-Gateway zwischen einer eventbasierenden oder zyklischer Abfrage der HAL-

Tabelle 5.3: Verhältnis von Empfangszeit zur empfangenen Framegröße

angewendete Empfangsstrategie	Ausführungszeit pro Frame	Systemauslastung bei max. Framegröße	Systemauslastung bei min. Framegröße
ereignisorientiert:	5,43 µs	4,4 %	89,3 %
zyklisch:	4,67 µs	3,8 %	76,8 %

eigenen Empfangsbuffer zu wechseln. Der Zyklische Abruf ermöglicht in Phasen, in denen kein Zeitkritischer Verkehr zu erwarten ist eine Verringerung der Ausführungszeiten des Systems und ermöglicht so die Effiziente Verarbeitung von Best-Efford Bursts. Die Tabelle 5.3 verdeutlicht die Geschwindigkeitsverteilung im Verhältnis zur Empfangsmethode. Zudem ist ein Dropping-Mechanismus in das Konzept mit eingeflossen um zu verhindern, dass ein Überlaufen des Systemspeichers zum Verlust von kritischen Daten führt. Diese Erweiterungen führen zum abschließenden konzeptionellen Aufbau des Prototyps, dargestellt in Abbildung 5.8 auf der vorherigen Seite.

5.7 Terminal Funktion

Der Zugang zu Terminalfunktionen ermöglicht jeder Anwendung Informationen über das Terminal zu empfangen oder nach außen weiter zu geben. Über die einfache Ausgabe hinaus stehen zudem Routinen zum Auslesen von definierten Speicherabbildern oder auch gesamter Frames zur Verfügung. Eine genaue Definition aller Funktionalitäten beschreibt das entsprechende Interface. Die Standardparameter der Kommunikation zeigt die Tabelle 5.4.

Tabelle 5.4: Parameter der Kommunikation über RS232

Baudrate	Datenbits	Parität	Stoppbits
115200	8	keine	1

Kapitel 6

Test und Ergebnisse

In diesem Kapitel werden angewandte Messverfahren vorgestellt und diskutiert. Weiterhin werden Verschiedene Testszenarien Entwickelt um das System auf seine Eigenschaften hin zu untersuchen. Abschließend werden die Ergebnisse Präsentiert und diskutiert. Zudem zählt eine Fehleranalyse zur Eingrenzung möglicher Ungenauigkeiten und eine Abschließende Bewertung.

6.1 Bestimmung des Zeitverhaltens

Das Zeitverhalten eines echtzeitfähigen Systems stellt eine wichtige Aussage dar. An ihr lässt sich die Güte eines Systems bestimmen, welches eine Planung von Zeitpunkten vorsieht um diese zur Ausführung zu bringen. Zur Bestimmung des Zeitverhaltens sind für eingebettete Systeme eine Reihe von verschiedenen Methoden möglich, die im Folgenden vorgestellt und diskutiert werden.

6.1.1 Angewandte Messverfahren

Eine Möglichkeit der Bestimmung von Abarbeitungszeiten besteht darin, einen hardwareeigenen Zähler zu verwenden. Dieser wird so konfiguriert, dass er ein volles 32 Bit Register nutzt, bis es zum Überlauf kommt. Somit ist sichergestellt, dass die größtmögliche Periode für einen Zählvorgang zur Verfügung steht. Bei Beginn des Messvorgangs wird der Zähler gestartet und sobald die zu messende Menge an Zyklen erreicht ist wieder gestoppt. Anhand des Zählerstandes lässt sich anschließend im Verhältnis zur Taktrate des Zählers die benötigte Zeit bestimmen.
Eine weitere Methode ist die Nutzung von General Purpose IOs (GPIO). Hierzu wird ein GPIO genutzt um durch Pegelwechsel die Positionen in der Verarbeitung nach außen sichtbar werden zu lassen. Diese Pegelwechsel lassen sich mit einem beliebigen Oszilloskop

Abbildung 6.1: Tektronix TDS 544A

Abbildung 6.2: Pico ADC-200

abtasten und visualisieren. Dies hat bei periodischen Ereignissen den Vorteil, dass dynamische Schwankungen in der Verarbeitung wie Jitter zur Laufzeit angezeigt werden. Weiterhin ist diese Methode unabhängig vom Quarz des System-under-Test. Sind gemessene Schwankungen in der Verarbeitung auf einen unsauberen Quarz zurückzuführen so wäre dies bei einer zählerbasierten Messung nicht zu erfassen, da der Zähler selbst vom gleichen Quarz getrieben wird. Ein weiterer Vorteil liegt in der Möglichkeit mehrere Ereignisse über entsprechende GPIOs gleichzeitig zur Laufzeit zu betrachten um Einflüsse untereinander nachvollziehen zu können. Dieses Vorgehen ist mit Hilfe einer zählerbasierten Messung verhältnismäßig schwierig und kann nicht zur Laufzeit durchgeführt werden. Bei einer Messung über GPIOs ist jedoch darauf zu achten, dass ein Pegelwechsel ebenso Verarbeitungszeit kostet, welche das Messergebnis verfälscht. Diese Zeit lässt sich jedoch ausmessen und anschließend aus jeder Messung herausrechnen. Sie ist statisch und eventueller Jitter liegt unterhalb des messbaren Bereichs, was somit nicht ins Gewicht fällt.

6.1.2 Verwendete Hardware

Zur Erfassung der Pegelwechsel der herausgeführten GPIOs wurde das Oszilloskop TDS-544A der Firma Tektronix (vgl. Abbildung 6.1) sowie das Picoskop ADC-200 (vgl. Abbildung 6.2) der Firma verwendet. Das ADC-200 verfügt über zwei unabhängige Kanäle mit einer Abtastrate von 10 MHz. Es hat damit eine maximale Auflösung von 100 ns und liegt damit im erwarteten Messbereich von unter 1 µs. Für aufwendigere Abhängigkeiten wird das TDS-544A herangezogen, welches vier unabhängige Kanäle besitzt. Es erreicht eine Abtastrate von 250 MHz pro Kanal, was in einer maximalen Auflösung von 4 ns resultiert.

Tabelle 6.1: Systemspezifische Delays in der Standardkonfiguration

Typ	Verzögerungszeit
Execute Task	8,2 µs
Send BG-Traffic	13,5 µs
Send CT-Traffic	13,5 µs
Synchronisation	6,8 µs

6.1.3 Bestimmung systemspezifischer Delays

Die Tabelle 6.1 beschreibt alle statischen Verzögerungszeiten des Systems. Diese können über die Konfiguration in der Initialisierungsphase von einer Anwendung nachträglich angepasst werden. Dies dient zur Neukalibrierung des Systems sollten sich die Delays, z.B. durch eine anders gewählte Aufteilung des Programms im Speicher, verschieben.
Der Delay *Execute Task* bezeichnet die Verzögerungszeit vom erreichen des Zeitpunktes im Schedule bis zur eigentlichen Aktivierung der Time-Triggered Task. Diese Verzögerungszeit wird bei der Ereignisregistrierung des Schedules verrechnet. Dadurch verschiebt sich der geplante Zeitpunkt um den Delay nach vorn, womit die Ausführung des Tasks genau am konfigurierten Zeitpunkt erfolgt. *Send BG-Traffic* und *Send CT-Traffic* bezeichnen jeweils die Verzögerung einer Sendeaufforderung von BG- sowie CT-Nachrichten bis zum Beginn des Sendevorgangs auf der Ebene des Standard Ethernet Ports. Somit wird ebenfalls gewährleistet, dass z.B. eine Time-Triggered Nachricht exakt zum konfigurierten Zeitpunkt versendet wird. Wäre dies nicht der Fall so würden alle Nachrichten verspätet den Switch erreichen, dessen Empfangsfenster daraufhin nicht mehr ausreicht um den Frame noch zu akzeptieren. Sie würde vom Switch verworfen. Ein weiterer Delay wird von der Synchronisierung gefordert. Dieser bezieht sich auf die Verzögerungszeit des Sendevorgangs eines Protocol Control Frames. Der Synchronisationsalgorithmus des TTEthernet Protokolls sieht eine Aktualisierung der Transparent Clock des Frames zu Beginn des Sendevorgangs vor, was durch die verwendete Hardware nicht unterstützt wird. Daher ist es nötig, im Voraus einen Statischen Delay in die Transparent Clock mit einzubeziehen. Für empfangene Frames ist es nicht nötig einen eigenen Delay zu definieren, da jeder ankommende Frame seinen Zeitstempel erhält. Anhand dessen ist es zu jeder Zeit im System möglich die Verzögerung eines Frames zurückzuverfolgen.

6.2 Funktionstest

Um einen umfangreichen Funktionstest durchführen zu können wurde eine Testumgebung geschaffen, in der die Teilnehmer des Netzwerks Time-Triggered und Rate-Constrained un-

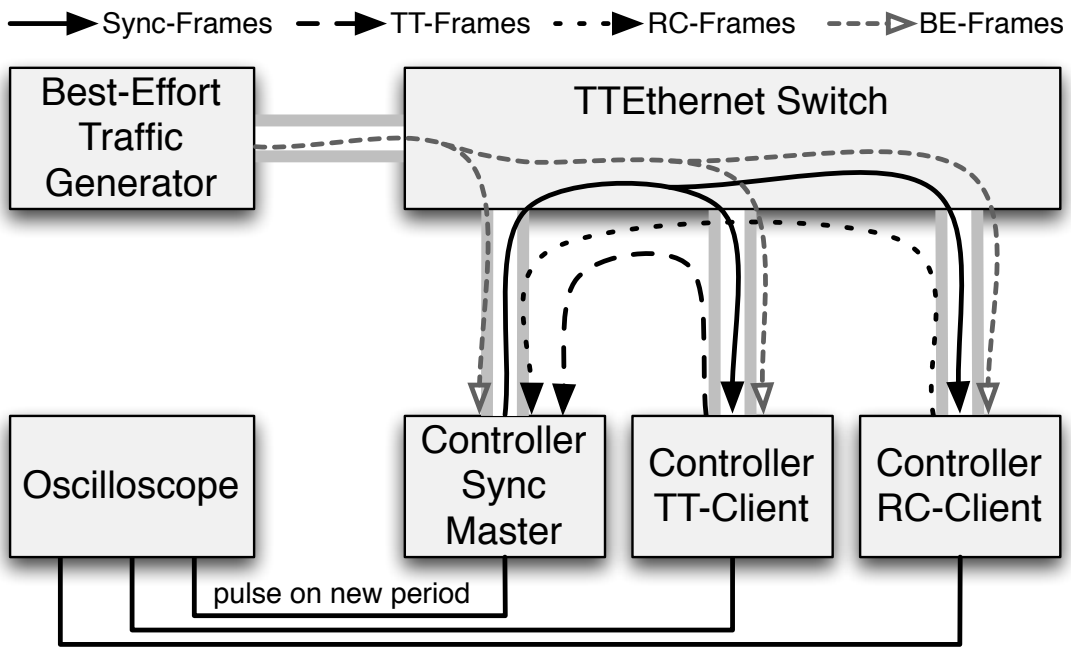

Abbildung 6.3: Aufbau der Testumgebung

tereinander austauschten (vgl. Abbildung 6.3). Zudem wurde das Netzwerk mit einem Best-Efford Traffic Generator belastet.

Versuchsaufbau

Das Netzwerk bestand aus vier Teilnehmern. Drei Prototypen wurden jeweils als Synchronisationsmaster, Time-Triggered Client und Rate-Constrained Client konfiguriert. Weiterhin wurde ein Standard PC zur Erzeugung von zufälligen Best-Efford Verkehr in das Netzwerk integriert. Der Zyklus des Netzwerkes betrug 3 ms. Die Synchronisation durch den Master erfolgte zum Zeitpunkt 0 µs des Schedules worauf sich der Switch und die beiden Clients synchronisierten. Ein Client wurde so konfiguriert, dass er eine Time-Triggered Nachricht an den Master sendete. Diese hatte den Sendezeitpunkt 1,5 ms im Schedule des Clients und wurde vom Switch bei 1,6 ms an den Master weitergeleitet. Der zweite Client hatte die Aufgabe Zyklisch Rate-Constained Nachrichten an den Master zu senden. Die BAG der Rate-Constrained Nachrichten betrug in der Konfiguration des Clients sowie in der des Switches 2 ms.
Zudem wurde jeder Prototyp für Überprüfungszwecke so modifiziert, dass er an einem GPIO zu Beginn jeder neuen Periode einen Pegelwechsel erzeugte. Über das Oszilloskop TDS-544A wurden diese Pegelwechsel aller Prototypen parallel zur Laufzeit beobachtet um Ab-

weichungen der Schedules der Teilnehmer feststellen zu können. Der Stack des Masters erfuhr eine weitere Modifikation in der es dem Prototyp möglich war die Zeitstempel aller empfangenen zeitkritischen Frames zu registrieren. Diese Struktur konnte nach Beendigung des Testlaufs über ein Terminal abgerufen werden. Mit Hilfe dieser Daten war es nicht nur möglich festzustellen ob alle Nachrichten empfangen werden konnten, sondern lieferte auch Auskunft über ihren relativen Empfangszeitpunkt.

Ablauf und Ergebnisse

Der Testlauf verlief über 10000 Zyklen, in denen keine Anomalien festzustellen waren. Die über das Oszilloskop beobachteten Abweichungen aller Zyklenwechsel der Teilnehmer lag unter 0,5 µs. Weiterhin ergab die Auswertung des Strukturinhaltes des Masters eine Empfangsrate der zeitkritischen Nachrichten von 100%, womit eine funktionelle Korrektheit bestätigt werden konnte.

6.3 Validierung der Synchronisation

Die globale Zeit über alle Teilnehmer eines Netzwerks ist direkt von der Synchronisation abhängig. Je präziser diese Synchronisation erfolgt, desto genauer ist es jedem Teilnehmer möglich sich an die netzwerkweite Zeit anzupassen. Dies ermöglicht eine Reduzierung aller Zeitfenster eines Schedules womit seine Periode effizienter genutzt werden kann. Perioden können dadurch kleiner gewählt werden und somit die Aktualisierungsfrequenz der Time-Triggered Nachrichten zu erhöhen, was die Reaktionsfähigkeit des Gesamtsystems verbessert. Weiterhin können Scheduls mit mehr Ereignissen konfiguriert werden. Dies wäre zwar auch durch eine Verbreiterung der Periode zu erreichen, was dennoch nicht in jeder Anwendung möglich ist. Z.B. würde ein Watchdog dies verbieten, da er sonst ausläuft und das System in den Fehlerfall zwingt. Eine genaue Synchronisation ist somit ein essentieller Bestandteil des Systems und wird daher im Folgenden näher betrachtet.

6.3.1 Bestimmung des Jitters

Hierzu wurde ein minimales Netzwerk bestehend aus zwei Prototypen aufgebaut. Sie wurden jeweils als Synchronisationsmaster und -client konfiguriert. Um die Genauigkeit des Schedules beider Prototypen zueinander zu bestimmen wurde eine Anwendung entwickelt, welche über einen GPIO den Beginn der Ausführung durch einen Pegelwechsel signalisiert. Dies Anwendung wurde anschließend jeweils zum gleichen Zeitpunkt in den Zeitplan beider Teilnehmer registriert. Beide GPIOs konnten nun mit Hilfe des ADC-200 im Vergleich zueinander beobachtet werden (vgl. Abbildung 6.4 auf der nächsten Seite).

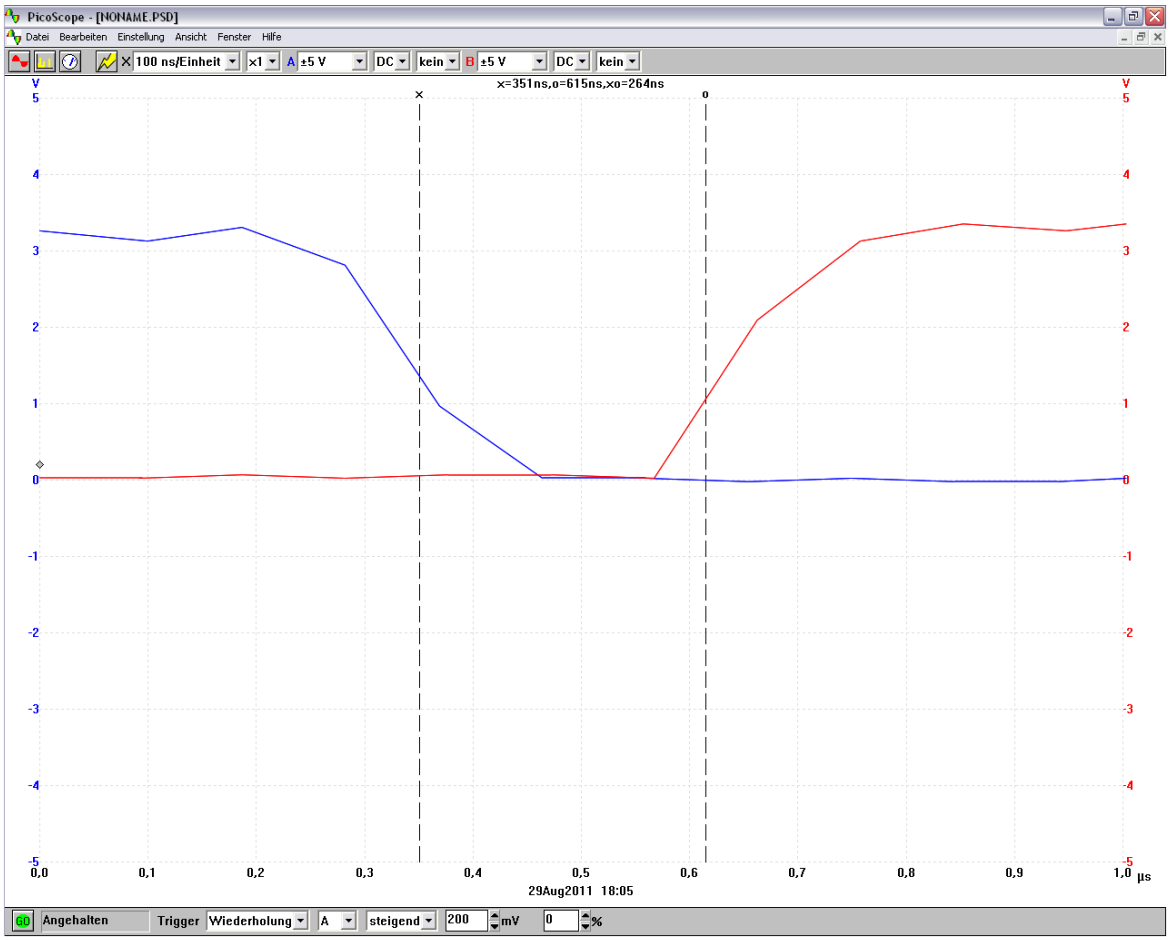

Abbildung 6.4: Messung der Synchronisationsungenauigkeit unter ADC-200

Zu beobachten ist ein Jitter von unter 300 ns. Da das ADC-200 nur mit einer Auflösung von 100 ns abtastet wird hier ein Jitter von unter 0,5 µs angegeben, was innerhalb der geforderten Genauigkeit von unter 1 µs liegt.

6.3.2 Messverfahren zur Bestimmung des Jitters

Die Erfassung von einer Metrik wie Jitter, welche über die Zeit bestimmt wird, ist mit Hilfe eines Oszilloskops nur bedingt möglich, da dieses immer nur eine aktuelle Momentaufnahme der Messung widerspiegeln kann. Daher wurde ein Messverfahren zur Validierung der bisherigen Messergebnisse verwendet, welches auf einem Synchronisierten TTEthernet fähigen Knoten alle eingehenden und ausgehenden Nachrichten durch Zeitstempel registriert und zur eignen Systemzeit ins Verhältnis setzt (vgl. Groß, 2011). Nach Abschluss der Messung lässt sich das Ergebnis auf einem Standard-PC betrachten.

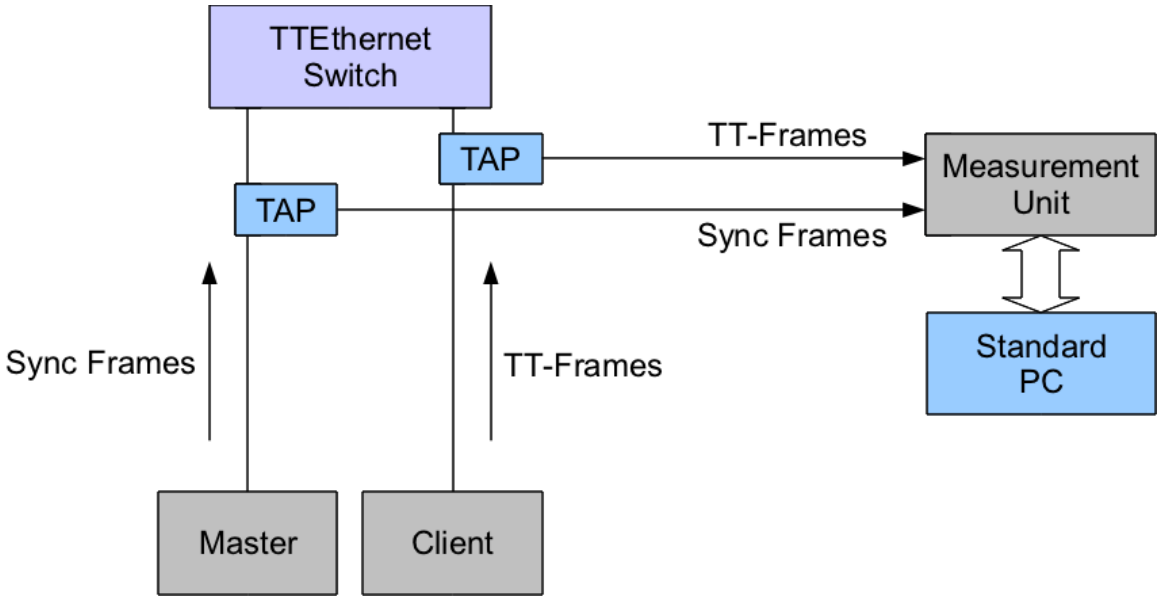

Abbildung 6.5: Messaufbau zur Bestimmung des Jitters

Zur Erfassung des Jitters wurde das Netzwerk aus Kapitel 6.3.1 bestehend aus einem Synchronisationsmaster und -client als Basis verwendet und um die Messeinheit erweitert (vgl. Abbildung 6.5). Der Client Sendete zu einem definierten Zeitpunkt Time-Triggered Nachrichten an den Master. Die über den Synchronisationsvorgang gesendeten Protocol Control Frames sowie die Time-Triggered Nachrichten wurden der Messeinheit zugeführt. Dies machte die Verwendung von zwei aktiven Ethernet TAPs möglich. Eine TAP erlaubt es, einen Port in eine definierte Richtung zu spiegeln. Die Spiegelung erzeugt weder zusätzliche Latenzen noch Jitter. Das Messgerät synchronisiert sich ebenso auf die ankommenden Synchronisationsframes um in der gleichen Zeiteinheit messen zu können in der sich das *Network under Test* befindet. Der Empfangszeitpunkt eintreffender Nachrichten wird in Referenz zur internen Zeit des Messgerätes registriert. Dadurch ist es möglich, die Genauigkeit eintreffender Nachrichten im Verhältnis zum Schedule zu messen. Dieses spiegelt wiederum die Genauigkeit des Schedules des zu untersuchenden Gerätes und schließlich seiner Synchronisation wieder. Diese Informationen werden auf einen Standard-PC transferiert und können genutzt werden um das Synchronisationsverhalten des Prototyps zu Visualisieren.
Die Abbildung 6.6 auf der nächsten Seite zeigt die Verteilung der Empfangszeitpunkte der Time-Triggered Nachrichten. Es ist der relative Empfangszeitpunkt über die betrachtetet Periode aufgetragen. Für jede Periode existiert somit genau ein Zeitpunkt, was den Empfang aller Time-Triggered Nachrichten bestätigt. Der Empfangszeitpunkt ist relativ zum registrierten Zeitpunkt des Schedule aufgelöst womit jede Abweichung direkt abzulesen ist. Hierbei lässt sich ein deutliches Schwingungsverhalten beobachten, welches durch den Synchronisati-

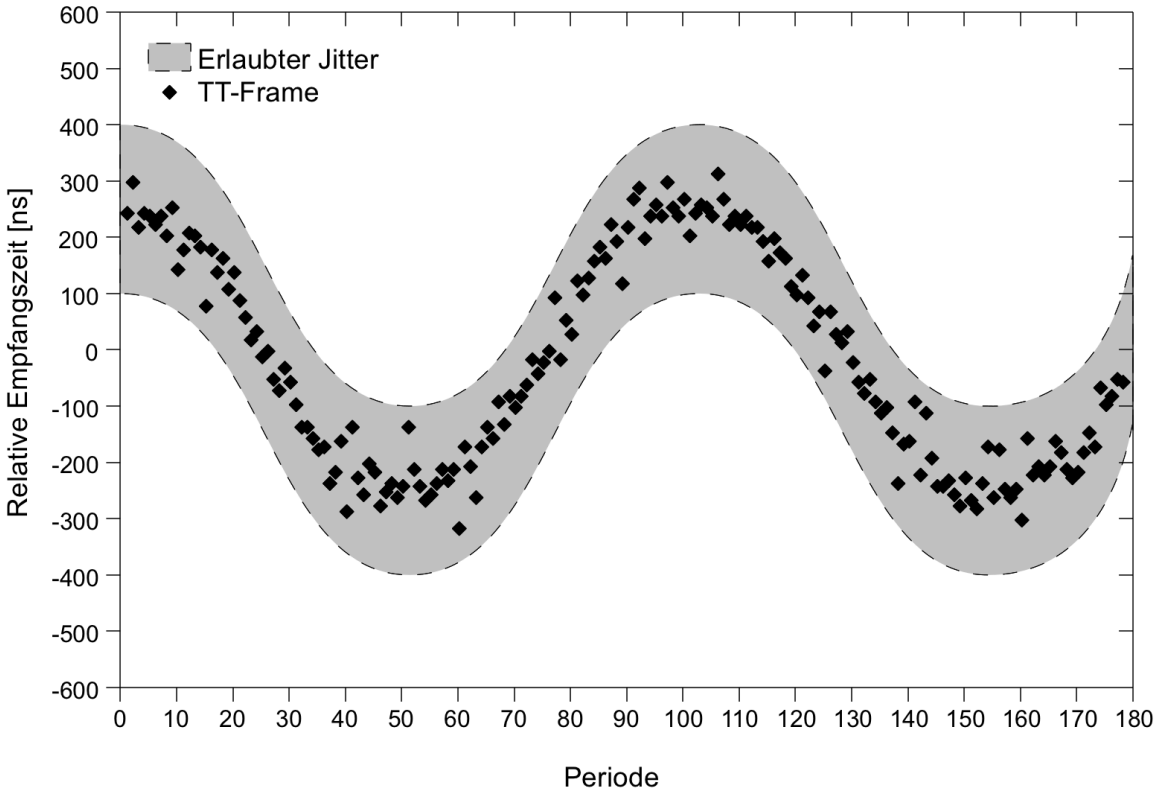

Abbildung 6.6: Empfangsverteilung der Test-Pakete

onsalgorithmus hervorgerufen wird. Je länger die Synchronisierung durch den Algorithmus nachgeregelt wird, desto geringer wird die Amplitude dieser Schwingung. Es handelt sich daher um eine systematische Abweichung, welche vom eigentlichen Jitter überlagert wird. Bestimmt man die Sollverteilung der Ankunftszeiten der Time-Triggerd Nachrichten so lässt sich ein Bereich über den zuvor bestimmten Jitter von 300 ns definieren. Es ist zu erkennen, dass kein Empfangszeitpunkt aus diesem Bereich herausfällt, was den über das Oszilloskop bestimmten Jitter verifiziert.

6.4 Fehleranalyse und -bewertung

Eine Bestimmung der statischen Delays zum Senden von Nachrichten ist nur erschwert möglich und wurde durch iteratives Nachregeln über unterschiedliche Konfigurationen ermittelt. Es existieren keine definierten Zeitangaben über statische Verzögerungen von der Sendeaufforderung des HALs bis zum Sendevorgang auf dem Ethernet-Kommunikationskanal. Hierzu ist ein Ethernetfähiges Oszilloskop von Nöten, welches innerhalb des Bearbeitungszeitpunkt dieser Arbeit nicht zur Verfügung stand. Mit dieser Hardware wäre eine Neukonfi-

gurierung, welche z.B. durch Veränderung der Speicheraufteilung notwendig werden würde, mit weniger Aufwand möglich.

6.4.1 Besonderheiten und Einschränkungen

Die Funktion des Stacks ist zu diesem Zeitpunkt nur auf Port 0 beschränkt. Damit wird der Failsaveansatz durch Redundanz noch nicht unterstützt. Dieser erfordert Methoden zur eindeutigen Bestimmung von doppelten Frames. Ein Vergleich aller Felder resultiert in einem hohen Rechenaufwand. Es reicht nicht aus im Falle einer Änderung den Vergleich abzubrechen, da es dem Regelfall entspricht, dass doppelte Nachrichten empfangen werden und somit vollständig überprüft werden müssten.

Kapitel 7

Zusammenfassung, Fazit und Ausblick

Automotive-Anwendungen steigen in Zukunft stetig in ihrer Anzahl und Komplexität. Es ist absehbar, dass momentane Ansätze über CAN oder FlexRay in ihrer Leistungsfähigkeit und erreichten Bandbreite an ihre Grenzen stoßen werden. Diese Situation wird dadurch verschärft, dass neuere Anwendungen zunehmend Kamera-basiert arbeiten, was in einer hohen Datenlast resultiert. Ethernet hat sich in der Computertechnik etabliert und als flexibles und hoch skalierbares Protokoll erwiesen. Jedoch konnte es bisher wegen seines mangelnden Determinismus nicht in echtzeitfähigen Systemen eingesetzt werden. Aus der Fertigungstechnik etabliert, wurden verschiedene Ansätze, Ethernet als Alternative für echtzeitfähige Bereiche einzusetzen, verfolgt. TTEthernet ist eine Neuentwicklung und richtet sich dabei speziell an die Anforderungen des Automotive-Bereichs. Es existieren eine Reihe von Evaluierungssystemen, die jedoch in ihren zeitlichen Genauigkeiten sowie der verwendeten Plattform nicht einem realistischen Einsatz im Automobil entsprechen. Es wurde in dieser Arbeit daher unter der Wahl geeigneter Hardware ein Konzept entwickelt, welches der TTEthernet Spezifikation und dem API gerecht wird. Dieser Prototyp ist in der Lage beliebig vielen Anwendungen Zugang zu TTEthernet fähigen Netzwerken zu liefern. Hierbei stehen Verschiedene Klassen an Nachrichten, je nach Echtzeitanforderung, zur Verfügung. Der Prototyp Synchronisiert sich mit Hilfe eines Modularen Ansatzes. Dieser ermöglicht einen dynamischen Wechsel seiner Rolle im Netzwerk und hilft so Ausfälle von Knoten zu überbrücken. Tests haben gezeigt, dass selbst unter starker Belastung eine Echtzeitfähigkeit garantiert werden kann. Weiterhin konnten statische Delays durch Vorausbestimmung unterdrückt werden. Der Jitter des Schedules und damit jedes geplanten Ereignisses konnte mit einem Wert unter 0,5 µs bestimmt werden. Damit entspricht das System den gestellten Anforderungen eines im Automotive Bereich eingesetzten eingebetteten Systems zur echtzeitfähigen Übertragung von Ethernet Nachrichten.

7.1 Alternativen in der Realisierung

Die Alternative in der Realisierung bestünde in der Nutzung eines nicht-preemtiven Ansatzes, für die Verteilung der Systemressourcen auf beliebig viele Prozesse. Hierbei gilt es die aufgezeigten Probleme in der Planung und Verarbeitung der Ereignisse zu erfassen und Lösungen anzubieten. Für diesen Ansatz wäre eine Beweisführung der Zuverlässigkeit in allen Fällen denkbar, was eine zukünftige Adaption des Systems vereinfachen würde.
Mitunter auch konzeptionelle Änderungen erfordert der Ansatz der Trennung des TTEthernet Stacks vom ausführenden Host. In dieser Lösung würde eine Anwendung nicht von höherpriorisierten Ereignissen wie dem Scheduling oder dem Versenden einer Time-Triggered Nachricht in ihrer Verarbeitungszeit beeinflusst. Ebenso denkbar wäre ein Multikern für parallele Ausführung von Anwendungen mit Anbindung an einen gemeinsamen TTEthernet Stack.
Weiterhin wäre eine Realisierung auf der Basis eines FPGAs denkbar. Dieser hätte durch schnellere und direkte Verarbeitung die Eigenschaften von geringeren Latenzen und Jitter. Zudem wäre eine Bandbreite von 1 GBit oder mehr realistisch, was mit Hilfe einer eingebetteten Lösung zu dieser Zeit noch nicht zu erreichen ist.

7.2 Offene Punkte und Verbesserungsmöglichkeiten

Der Failsaveansatz durch Redundanz wird in dieser Arbeit noch nicht unterstützt. Dies erfordert geeignete Algorithmen in der effizienten Überprüfung von doppelten Frames (vgl. Kapitel 6.4.1). Zudem ist die Implementierung eines Compression-Masters notwendig, da der Ansatz ebenso den Einsatz von multiplen Master im selben Netzwerk fordert. Ohne Compression-Master ist keine gemeinsame Synchronisation des Netzwerkes möglich.
Zudem wäre der Einsatz des Hardwaregegebenen Watchdogs interessant, welcher Ausfälle des Echtzeitfähigen System detektieren und anschließend einen gesteuerten Reset des Stacks einleiten kann. Hierzu wäre auch der Einsatz des Speicherbereichs denkbar, welcher von der Hauptstromversorgung unabhängig ist, um aus dem Reset wieder in den ursprünglichen Zustand zurückzufinden.

Literaturverzeichnis

[Ademaj und Kopetz 2007] ADEMAJ, Astrit ; KOPETZ, Hermann: Time-Triggered Ethernet and IEEE 1588 Clock Synchronization. In: *Precision Clock Synchronization for Measurement, Control and Communication, 2007. ISPCS 2007. IEEE International Symposium on*, Oktober 2007, S. 41–43

[Aeronautical Radio Incorporated 2002] AERONAUTICAL RADIO INCORPORATED: Aircraft Data Network / ARINC. Annapolis, Maryland, 2002 (664). – Standard

[Charara u. a. 2006] CHARARA, Hussein ; SCHARBARG, Jean-Luc ; ERMONT, Jérôme ; FRABOUL, Christian: Methods for bounding end-to-end delays on an AFDX network. In: *18th Euromicro Conference on Real-Time Systems*, 2006. – ISSN 1068-3070

[FlexRay Consortium 2005] FLEXRAY CONSORTIUM: Protocol Specification / FlexRay Consortium. Stuttgart, Dezember 2005 (2.1). – Specification

[GmbH 2008] GMBH, Hilscher: *Program Reference Guide - Preliminary*. Hilscher GmbH. Dezember 2008. – URL `http://www.hilscher.com`

[GmbH 2009] GMBH, Hilscher: *Device Description - NXHX 500-ETM*. Hilscher GmbH. Mai 2009. – URL `http://www.hilscher.com`

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

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

[Honeywell International]

[J. 2009] J., A.: *netX - The Insider's Guide to netX*. Hilscher GmbH. Februar 2009. – URL `http://www.hilscher.com`

[Kopetz 2004] KOPETZ, Hermann: *Real-time Systems: Design Principles for Distributed Embedded Applications*. 8. Boston : Kluwer Academic, 2004 (The Kluwer international series in engineering and computer science: real-time systems). – ISBN 0-7923-9894-7

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

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

[Micron 2008] MICRON: *Synchronous DRAM - Datasheet*. Micron. Mai 2008. – URL `http://www.micron.comlsdramds.com`

[Pfrommer 2008] PFROMMER, H.: *HAL - Ethernet Media Access Controller*. Hilscher. Juli 2008. – URL `http://www.hilscher.com`

[Real Time Systems Group (RTS)] REAL TIME SYSTEMS GROUP (RTS): *TTEthernet*. – URL `http://ti.tuwien.ac.at/rts`. – Zugriffsdatum: 2010-12-10

[Robert Bosch GmbH] ROBERT BOSCH GMBH: *Controller Area Network*. – URL `http://www.semiconductors.bosch.de/`. – Zugriffsdatum: 2011-02-03

[SAE - AS-2D Time Triggered Systems and Architecture Committee 2009] SAE - AS-2D TIME TRIGGERED SYSTEMS AND ARCHITECTURE COMMITTEE: *Time-Triggered Ethernet (AS 6802)*. 2009. – URL `http://www.sae.org`. – Zugriffsdatum: 2010-12-11

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

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

[TTTech Computertechnik AG]

[TTTech Computertechnik AG 2008] TTTECH COMPUTERTECHNIK AG: *TTEthernet Application Programming Interface*. TTTech Computertechnik AG. Dezember 2008. – URL `http://www.tttech.com`

[TU Wien 1997] TU WIEN: *The TTP Protocols*. 1997. – URL `http://www.vmars.tuwien.ac.at/projects/ttp/ttpmain.html`

Abbildungsverzeichnis

1.1 Verteilung der Steuergeräte im Automobil 2
1.2 Zentrales Gateway verschiedener Bussysteme 3

2.1 Switch zur Evaluierung eines TTEthernet-Netzwerks 7
2.2 Aufbau eines TT-Ethernet Frames 8
2.3 Ablauf der zweistufigen Synchronisation 10
2.4 Kontrollfluss des Synchronisationsalgorithmus 11

3.1 Stilisiertes NXHX500-ETM 16
3.2 Aufbau der NetX500 CPU 17

4.1 Basiskonzept des TTEthernet-Stacks 24
4.2 Beispiel für eine priorisierte Unterbrechung 28
4.3 Beispiel für eine rekursive Unterbrechung 29
4.4 Konzept des TTEthernet Stacks 32

5.1 Übersicht der Speicherverteilung 37
5.2 Ressourcenverteilung des Prototyps 38
5.3 Aufbau des Schedulings 42
5.4 Hierarchie der Bufferpoole 44
5.5 Aufbau eines Cyclical Asynchronous Buffers 45
5.6 Aufbau eines Queued Buffers 47
5.7 Abweichungszonen im Verhältnis zum Grad der Synchronisierung 51
5.8 Aufbau des Systems 53

6.1 Tektronix TDS 544A 56
6.2 Pico ADC-200 56
6.3 Aufbau der Testumgebung 58
6.4 Messung der Synchronisationsungenauigkeit unter ADC-200 60
6.5 Messaufbau zur Bestimmung des Jitters 61
6.6 Empfangsverteilung der Test-Pakete 62

Tabellenverzeichnis

3.1 Anforderungen der Hardware 15
3.2 Vergleich von Entwicklungsboards mit den Anforderungen 15

4.1 Anforderungen des Konzepts 23
4.2 System-LED 33
4.3 Port-LED 33
4.4 Statusanzeige im Normalbetrieb 34
4.5 Statusanzeige im Fehlerfall 34

5.1 Nutzparameter in Abhängigkeit der Ereignisklasse 42
5.2 Parameter in Abhängigkeit der Abweichungsstärken 52
5.3 Verhältnis von Empfangszeit zur empfangenen Framegröße 54
5.4 Parameter der Kommunikation über RS232 54

6.1 Systemspezifische Delays in der Standardkonfiguration 57

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 30. August 2011 Kai Kurt Müller