Bachelorarbeit

Flemming Bunzel

Hardware-Abstraktion eines Open Source Echtzeit Ethernet Stacks - Entwurf, Umsetzung und Evaluation

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Flemming Bunzel

Hardware-Abstraktion eines Open Source Echtzeit Ethernet Stacks - Entwurf, Umsetzung und Evaluation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Korf
Zweitgutachter: Prof. Dr. Fohl

Eingereicht am: 15. August 2013

Flemming Bunzel

Thema der Arbeit

Hardware-Abstraktion eines Open Source Echtzeit Ethernet Stacks - Entwurf, Umsetzung und Evaluation

Stichworte

Portierbarkeit, Hardware Abstraction Layer, Echtzeit-Ethernet, TTEthernet-Stack, Open Source Software

Kurzzusammenfassung

Die "Communication over Real-Time Ethernet"-Projektgruppe arbeitet an auf Echtzeit-Ethernet basierenden Lösungen für die Kommunikation von zeitkritischen Anwendungen im Automobil. Im Rahmen der Projektarbeit ist der Prototyp eines Echtzeit-Ethernet-Stacks entwickelt worden, welcher die Umsetzung eines Time-Triggered Ethernet fähigen Endsystems ermöglicht. Durch die hardwarenahe Implementierung des Stacks war dieser bisher fest an einen bestimmten Microcontroller gebunden. In dieser Arbeit wird der Stack durch die Einführung eines Hardware Abstraction Layers portierbar gemacht und somit die Nutzung des Stacks auf anderen Microcontrollern ermöglicht. Durch die entstandene Portabilität und die Veröffentlichung als Open Source Software soll die Nutzung des Stacks auch in anderen Projekten ermöglicht werden.

Title of the paper

Hardware-abstraction of an open source real-time Ethernet stack - Design, realisation and evaluation

Keywords

Portability, Hardware Abstraction Layer, Real-time Ethernet, TTEthernet-Stack, Open Source Software

Abstract

The "Real-Time Communication over Ethernet" project group works on real-time Ethernet-based solutions for communication of time-critical automotive applications. As part of the project work, the prototype of a real-time Ethernet stack has been developed, which enables the implementation of a Time-Triggered Ethernet-capable end-system. Due to the low-level implementation of the stack, it was previously firmly tied to a particular microcontroller. In this thesis the stack is made portable via the introduction of a hardware abstraction layer and thus allows the use of the stack on other microcontrollers. Through the resulting portability and publication as open source software, the use of the stack should be possible also in other projects.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **4**
- 2.1 Software Portierung . . . 4
- 2.2 Time-Triggered Ethernet . . . 5
- 2.3 TTEthernet-Stack . . . 8
- 2.4 Open Source Software . . . 9

3 Anforderungen **11**
- 3.1 Anforderungen TTEthernet-Stack . . . 11
- 3.2 Verwendete Hardware . . . 12

4 Konzept **16**
- 4.1 Stack Architektur . . . 16
- 4.2 Strategien Software Portierung . . . 17
- 4.3 Hardware Abstraction Layer . . . 21
- 4.4 HAL TTEthernet-Stack . . . 22

5 Umsetzung **26**
- 5.1 Realisierung . . . 26
- 5.2 Timer-Funktionalität des TTEthernet-Stacks . . . 27
- 5.3 Interruptbehandlung des TTEthernet-Stacks . . . 29
- 5.4 Ethernet-Schnittstelle des TTEthernet-Stacks . . . 31
- 5.5 Dokumentation . . . 35

6 Ergebnisse **36**
- 6.1 Zeitverhalten . . . 36
- 6.2 Funktionstest . . . 39

7 Zusammenfassung und Ausblick **45**
- 7.1 Zusammenfassung . . . 45
- 7.2 Ausblick . . . 46

Literatur **49**

Tabellenverzeichnis **50**

Abbildungsverzeichnis **51**

Glossar **52**

1 Einleitung

Wie in den meisten Bereichen des täglichen Lebens nimmt die Vernetzung auch beim Auto immer weiter zu. Zum einen durch die in Zukunft eingesetzte "Car-to-X Kommunikation", bei der das Fahrzeug mit seiner Umgebung kommuniziert. Dabei findet der Datenaustausch entweder mit anderen Verkehrsteilnehmern oder zwischen dem Auto und Teilen der Verkehrsinfrastruktur wie Verkehrsleitsystemen oder Ampelanlagen statt. Dadurch sollen vor allem Sicherheit, Effizienz und Komfort erhöht werden. In Praxistest werden jetzt schon vielversprechende Ergebnisse erzielt (Vgl. VW (2013)).
Zum anderen wird im Fahrzeug von heute schon ein erheblicher Anteil an Elektronik verbaut, welcher in den nächsten Jahren noch weiter steigen soll. Dies sind zum Beispiel Fahrerassistenzsysteme wie das elektronische Stabilitätsprogramm (ESP), das Antiblockiersystem (ABS), Rückfahrkameras oder Einparkhilfen. Eine weitere Technik ist die "x-by-wire" Technologie, welche Teile der Mechanik durch elektronische Lösungen ersetzt. Damit ist es möglich Lenkung und Bremsen rein elektronisch zu steuern oder das Fahrwerk anzupassen. Dabei ersetzt ein Bussystem die mechanische Verbindung.
Bisher gibt es unterschiedliche Bussysteme, welche je nach Anwendung verschiedene Ansprüche in Übertragungsgeschwindigkeit und Echtzeitfähigkeit erfüllen müssen. In aktuellen Fahrzeugen eingesetzte Bussysteme sind z.B. CAN (Controller Area Network), LIN (Local Interconnect Network) oder MOST (Media Oriented System Transport). Diese bisher eingesetzten Systeme haben auf Grund der neuen Ansprüche an Bandbreite und Echtzeitanforderungen die Grenzen ihrer Leistungsfähigkeit erreicht. Des Weiteren ist die Verbindung der einzelnen Bus-Systeme untereinander sehr aufwendig und führt zu einem sehr komplexen Kabelbaum (Vgl. Abbildung 1.1), welcher sich durch seinen erheblichen Umfang zusätzlich Kosten und Gewicht des Autos erhöht. Gerade für die x-by-wire-Anwendungen wird ein verlässliches Kommunikationssystem benötigt, da eine Fehlfunktion bei sicherheitsrelevanten Funktionen wie Bremse und Lenkrad fatale Folgen hätte. Das Bussysteme muss garantieren, dass eine Nachricht innerhalb einer bestimmten Zeit den Empfänger erreicht.
Eine Netzwerktechnik, welche den Ansprüchen an Echtzeitfähigkeit und Bandbreite genügt, ist Time-Triggered Ethernet (TTEthernet) (Vgl. TTTech Computertechnik AG). Time-Triggered

Ethernet ist eine Erweiterung des standard Ethernet Protokolls und wurde von der Firma TTTech entwickelt. Durch die hohe Bandbreite und Flexibilität des TTEthernet kann das Autobordnetz erheblich vereinfacht werden. So kann durch die Nutzung eines auf Ethernet basierenden Backbone-Netzwerks die Anzahl der Kabel stark reduziert werden, was sich wiederum positiv auf Übersichtlichkeit, Gewicht und Kosten auswirkt.

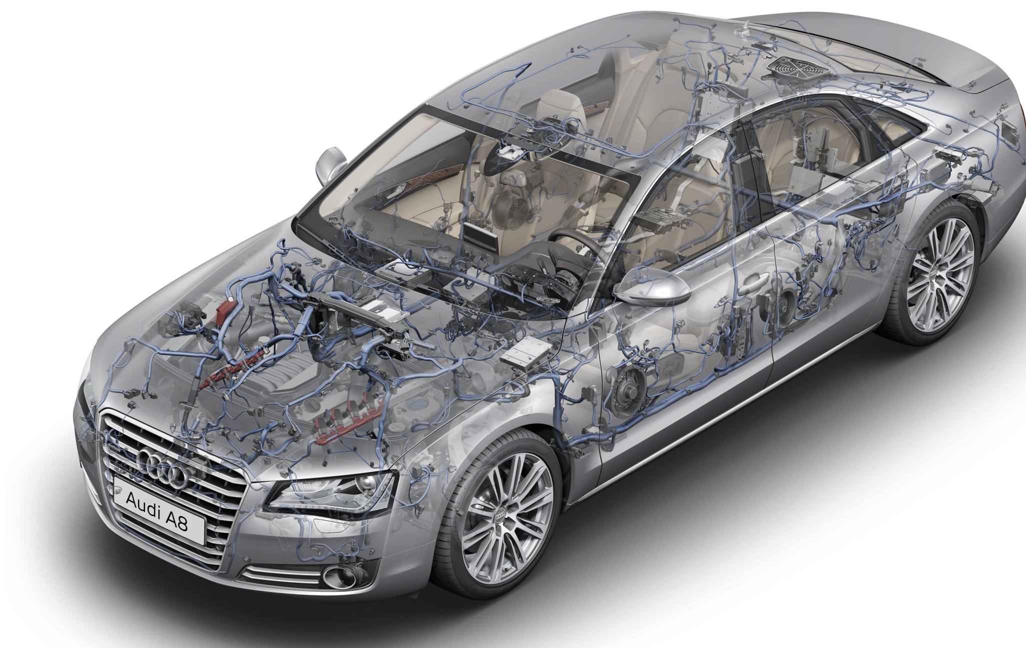

Abbildung 1.1: Bordnetz Audi A8 (Quelle: VDIWissensforum/AudiAG)

Diese Bachelorarbeit entsteht im Rahmen der CoRE (Communication over Real-time Ethernet) Projektgruppe an der Hochschule für Angewandte Wissenschaften Hamburg. Die CoRE-Gruppe arbeitet an neuen Lösungen für die Kommunikation von zeitkritischen Anwendungen über Ethernet. Der Schwerpunkt liegt hier auf der Kommunikation im Auto mit dem Ziel, das herkömmlich verbaute Bordnetz durch ein auf Ethernet basierendes Backbone-Netzwerk zu ersetzen.
Im Rahmen einer früheren Bachelor Arbeit ist ein auf TTEthernet basierender Netzwerk-Stack entwickelt worden, welcher die von TTTech vorgegebene API für das TTEthernet erfüllt (Vgl. Müller (2011)). Mit dem Stack wurde der Prototyp eines Backbone-Netzwerks realisiert, um zu

demonstrieren, dass der Einsatz von TTEthernet im Automobilbereich möglich ist. Dabei wird der Stack auf einem ARM basierten Microcontroller ausgeführt, welcher speziell für Echtzeitkommunikation optimiert ist.
Das Ziel dieser Arbeit ist es den TTEthernet-Stack portierbar zu machen. Im Hinblick auf weitere Projekte des CoRE-Gruppe wie dem RECBAR Projekt (Vgl. CoRE (b)), bei dem der Prototyp eines Ethernet basierten Backbones im Auto realisiert werden soll, wird es notwendig sein, den Stack auf weitere Steuergeräte zu portieren. Durch die Einführung eines Hardware Abstraction Layers (HAL) soll es möglich sein, den Stack auch mit anderer Hardware als dem oben genannten Microcontroller zu nutzen (Vgl. Abbildung 1.2). Um den Erfolg der Umsetzung zu überprüfen, wird der Stack dann abschließend auf einen ausgewählten Microcontroller portiert.
Des Weiteren soll der Stack unter einer Open Source Lizenz veröffentlicht werden, um den Einsatz des TTEthernet-Stacks in anderen Projekten zu ermöglichen.

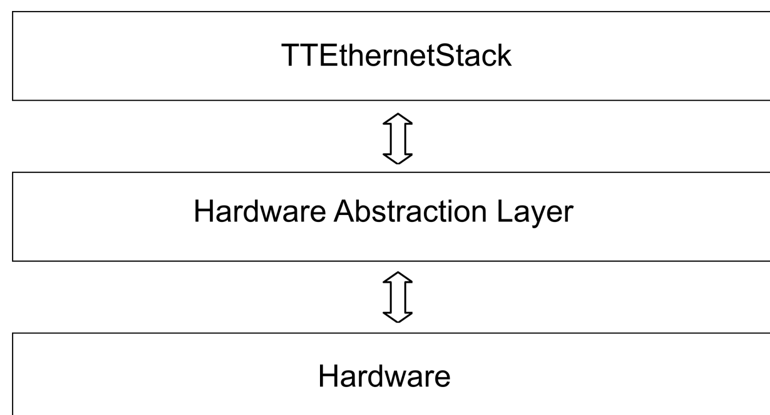

Abbildung 1.2: Einordnung des HAL in Stack Architektur

Inhaltliche Gliederung der Arbeit

Im Kapitel 2 werden einige Grundlagen dieser Arbeit erläutert. Zum einen wird sich mit dem Thema Software-Portierung auseinandergesetzt, eine kurze Einführung zu TTEthernet gegeben und auf das Thema Open Source Software eingegangen. Kapitel 3 geht auf die Anforderungen, welcher der Stack an die Hardware stellt, ein. In Kapitel 4 werden mögliche Vorgehensweisen besprochen und das Konzept entwickelt. Das Vorgehen bei der Umsetzung wird in Kapitel 5 beschrieben und in Kapitel 6 werden Ergebnisse und Probleme bei der Umsetzung dargestellt. Am Schluss werden in Kapitel 7 nochmal alle Ergebnisse zusammengefasst und ein Ausblick auf eine mögliche Fortführung der Arbeit gegeben.

2 Grundlagen

In diesem Kapitel werden die Themen besprochen, welche einen Bezug zum Inhalt der Arbeit haben und zum Verständnis dieser beitragen. Zunächst wird eine kurze Einführung in das Thema Software Portierung gegeben. Dann wird das Time Triggered Ethernet Protokoll vorgestellt und zuletzt ein Einblick in das Thema Open Source Software gegeben.

2.1 Software Portierung

Portabilität bedeutet, dass Software auf unterschiedlichen Plattformen lauffähig ist. Das ist vor allem dann notwendig, wenn die Software von möglichst vielen Anwendern genutzt werden soll. Das bekannteste Beispiel sind Betriebssysteme, welche mit unterschiedlichster Hardware zusammenarbeiten müssen. Das Betriebssystem Linux beispielsweise läuft nicht nur auf den bei PCs üblichen x86- und x64-Architekturen, sondern wurde auch auf zahlreiche andere Prozessorarchitekturen wie z.B. ARM oder PowerPC portiert. Möglich macht dies der anpassungsfähige Kernel, welcher die Basis jeder Linux-Distribution ist. Der Kernel bildet eine hardwareabstrahierende Schicht, welche der darüber liegenden Anwendungsschicht eine Schnittstelle für den Hardwarezugriff zur Verfügung stellt. Der Zugriff auf die Hardware erfolgt über die im Kernel integrierten Gerätetreiber, welche je nach Zielplattform an die genutzte Hardware angepasst werden müssen. Der Kernel wird auf die Zielarchitektur portiert und bietet unabhängig von der Architektur eine identische Schnittstelle für die Softwareprogrammierung. Ein Beleg für den Erfolg dieser Strategie ist das auf dem Linux-Kernel basierende Betriebssystem Android, welches vor allem auf Smartphones und Tablet-Computern eingesetzt wird und inzwischen das meist genutzte Smartphone-Betriebssystem ist (Vgl. Canalys (2013)). Android ist frei verfügbar und kann auf nahezu jede Plattform, die vom Linux-Kernel unterstützt wird, portiert werden.

Eingebettete Systeme

Auch im Bereich der Eingebetteten System wird die Software Portierung genutzt. Durch die Entwicklung der letzten Jahre sind die Kosten für Hardware gesunken. Gleichzeitig steigt die

Leistungsfähigkeit immer weiter an und dank neuer Fertigungstechniken ist es möglich, benötigte Bauteile immer kleiner und kompakter zu fertigen. So kann es durchaus sinnvoll sein, bei einem bestehenden System die Hardware-Komponenten durch günstigere und leistungsfähigere Hardware zu ersetzen. Ein weiterer Grund dafür, Komponenten zu ersetzen, ist die Erweiterung des bestehenden Systems um neue Funktionen, für die das aktuelle System über nicht ausreichende Leistung verfügt. Eine zu schwache CPU, zu wenig Speicher oder nicht verfügbare Peripherie würden so eine vielleicht notwendige Erweiterung verhindern (Vgl. Marcondes u. a. (2006)). Durch das hardwarenahe Design macht der Austausch der Hardware oft auch eine Anpassung der Software notwendig. Diese Anpassung kann je nach Aufwand den Vorteil/Nutzen der neuen Hardware zunichte machen und ist vom jeweiligen Projekt abhängig. Doch für jedes Projekt gilt: Eine portierbare Software würde den Aufwand der Anpassung erheblich reduzieren, zusätzlich die Wiederverwendbarkeit der Software erhöhen und zukünftige Änderungen am System stark vereinfachen. Um die Portabilität der Software zu implementieren, muss eine hardwareunabhängige Schnittstelle definiert werden, welche zwischen Hardware und Software vermittelt. Wenn die Software jetzt auf eine andere Plattform übertragen werden soll, müssen lediglich die Funktionen der Schnittstelle angepasst werden welche den Zugriff auf die Hardware regeln. Angewandte Techniken für die Portierung von Software werden später in Abschnitt 4.2 diskutiert.

2.2 Time-Triggered Ethernet

Ein System ist echtzeitfähig, wenn es eine Reaktion innerhalb eines definierten Zeitrahmens garantiert. Die Antwort des Systems muss innerhalb einer vorgegeben Zeit erfolgen, sonst wird das Ergebnis als falsch interpretiert. Ein Time-Triggered Ethernet Netzwerk ist ein Echtzeitsystem, welches diesen Ansprüchen genügt.
Time-Triggered Ethernet (im Folgenden TTEthernet) ist ein echtzeitfähiges Netzwerkprotokoll, welches für den Einsatz in sicherheitsrelevanten Anwendungen entwickelt wurde (Vgl. TTTech Computertechnik AG). Dabei ist TTEthernet eine Erweiterung des normalen Ethernet Protokolls, welches im IEEE 802.3 Standard (Vgl. IEEE) definiert ist. Ethernet ist ein weit verbreiteter Netzwerkstandard für lokale Netzwerke (LAN) und wird den ersten beiden Schichten des OSI-Models zugeordnet. Die erste Schicht ist die Bitübertragungsschicht (Physical Layer) und definiert die Übertragung auf physischer Ebene wie Verkabelung, Signalerzeugung so wie Signalkodierung (Vgl. Plate (2013)). In Schicht 2 - der Sicherungsschicht (Data Link Layer) - definiert das Ethernet Protokoll einem Rahmen (Frame), in welchem die übertragenen Datenbits zusammengefasst werden. Dieser Frame enthält zusätzlich zu den Daten die Präambel zur Synchronisation der Bitübertragung, Ziel- und Quelladresse, den Frame-Typ, die Länge und eine Prüfsumme zur

Erkennung von Übertragungsfehlern. Der genaue Aufbau des Frames nach IEEE 802.3 Standard ist in der Abbildung 2.1 zu sehen.

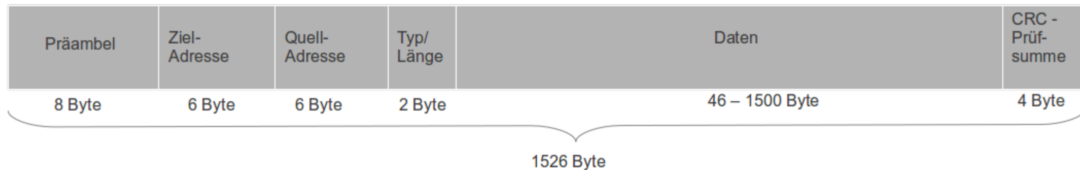

Abbildung 2.1: Aufbau eines Ethernet Frame nach IEEE 802.3 (Vgl. IEEE)

TTEthernet erweitert das Ethernet Protokoll in Schicht 2 des OSI-Referenzmodells und ermöglicht damit die Echtzeit-Kommunikation. Dabei wird das Protokoll so erweitert, dass es zum IEEE 802.3 Standard kompatibel bleibt und ermöglicht so die parallele Nutzung von Echtzeit- und standard Ethernet-Verkehr im gleichen Netzwerk. Es garantiert für den zeitkritischen Datenverkehr deterministische Paketlaufzeiten und eine synchronisierte, staufreie Kommunikation (Vgl. SAE (2011)).
In den TTEthernet Spezifikationen werden drei Nachrichtenklassen definiert, welche nachfolgend vorgestellt werden:

Time-Triggered-Traffic (TT) ist die Nachrichtenklasse mit der höchsten Priorität und wird für den zeitkritischen Datenverkehr genutzt. TT-Nachrichten haben ein deterministisches Zeitverhalten und kurze Paketlaufzeiten, da sie nicht von anderen Nachrichten verdrängt werden können. Sie werden zu statisch definierten Zeitpunkten und synchron zur globalen Uhr verschickt.

Rate-Constrained-Traffic (RC) RC-Pakete haben die zweit höchste Priorität und können nur von TT-Nachrichten verdrängt werden. Ihnen wird mit Hilfe von sogenannten BAGs (Bandwidth Allocation Gap) eine feste Bandbreite zugesichert. Diese legen die minimale Zeit zwischen zwei nachfolgenden Frames sowie ihre maximale Länge fest. Sie haben höhere Schwankungen (Jitter) als TT-Nachrichten, da sie eventbasiert verschickt werden und keine Synchronisation mit der globalen Uhr statt findet.

Best-Effort Traffic (BE) BE-Nachrichten haben die niedrigste Priorität und entsprechen dem klassischen Ethernet-Verkehr. Es kann weder garantiert werden wann die Nachricht übertragen wird, noch ob die Nachricht ankommt.

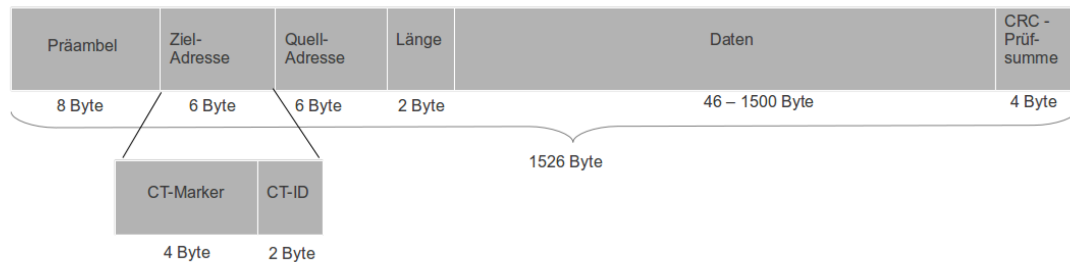

Abbildung 2.2: Aufbau eines TTEthernet Frame

Um die Priorisierung der Frames zu ermöglichen, wird die Ziel-Adresse der Frames anders interpretiert als beim Standard-Ethernet Frame. Bei einem TTEthernet-Frame setzt sich die Ziel-Adresse aus einem 32 Bit großen CT-Marker und der 12 Bit großen CT-ID zusammen. Der CT-Marker definiert den Cluster im Netzwerk und kennzeichnet den Frame als zeitkritischen Verkehr. Die CT-ID identifiziert die Nachricht an sich und muss eindeutig sein. Der Aufbau des TTEthernet-Frames ist in Abbildung 2.2 zu sehen.

Die Identifizierung der Nachrichten als CT-Verkehr wird durch Maskierung des Ziel-Adressfeldes realisiert. Damit das Paket an den richtigen Empfänger zugestellt werden kann, werden im Netzwerk spezielle TTEthernet-Switches genutzt. Die Zustellung der Nachrichten erfolgt anhand statischer Routen, welche im Switch definiert sind. So entscheidet das Switch nach Nachrichtenart (CT-Marker) und CT-ID über, welchen Port das Paket weitergeleitet wird (Vgl. Bartols (2010)). Zusätzlich wird in dem Switch ein Schedule festgelegt, welcher definiert wann eine TT-Nachricht weiter geschickt wird. Dafür wird ein Zyklus mit festen Zeitschlitzen definiert, in welchem jede TT-Nachricht einen Zeitslot zugeordnet bekommt. Damit ein TT-Frame zum geplanten Zeitpunkt verschickt werden kann, muss er innerhalb eines bestimmten Zeitfensters beim Switch ankommen. Der Zyklus richtet sich nach einer globalen Zeit und ist in jedem Netzwerkteilnehmer realisiert.
Um sicher zu stellen, dass die Zykluszeit auf jedem Gerät gleich ist, müssen sich die Teilnehmer auf die globale Zeit synchronisieren. Die Synchronisation geschieht mit den Protocol Control Frames (PCF). Die Frames enthalten Synchronisationsdaten, welche es den Netzwerkteilnehmern ermöglichen, ihre Uhren der globalen Zeit anzupassen und so den korrekten Ablauf des Datenverkehrs im Netzwerk sicher zu stellen. Für die Synchronisation nehmen die Netzwerkknoten unterschiedliche Rollen an: Synchronisationsmaster, Synchronisationsclient und Compressionmaster. Jeder Synchronisationsmaster sendet einmal im Zyklus ein PC-Frame mit seiner Zeitbasis.

Der Compressionmaster wertet die empfangenen Frames aus und berechnet einen geeigneten Synchronisationszeitpunkt. Zu dem berechneten Zeitpunkt sendet der Compressionmaster PC-Frames mit der gemeinsamen Zeitbasis, auf die sich alle Netzwerknoten synchronisieren. Eine Synchronisation ist auch ohne Compressionmaster möglich. Dafür muss genau ein Synchronisationsmaster im Netzwerk vorhanden sein, welcher alle Synchronisationsclients auf seine Zeitbasis synchronisiert.

2.3 TTEthernet-Stack

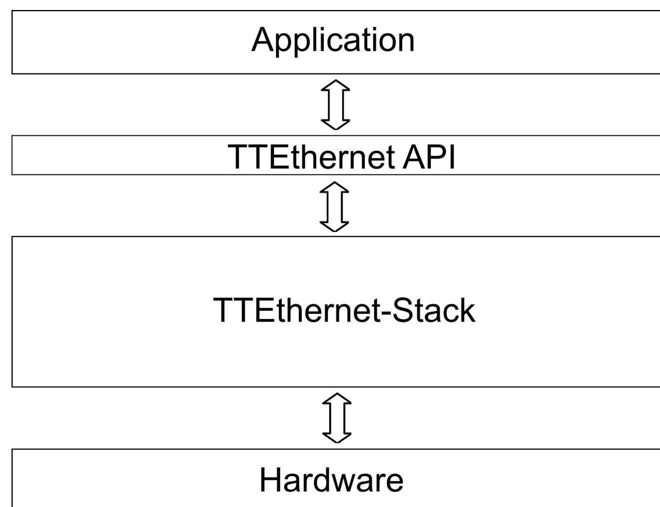

Abbildung 2.3: Softwarearchitektur des TTEthernet Stacks

Mit dem Echtzeit Ethernet Stack, welcher die Grundlage dieser Arbeit bildet, ist es möglich ein TTEthernet konformes Endsystem in einem TTEthernet Netzwerk zu realisieren. Der Stack unterstützt alle Funktionen des TTEthernet Protokolls und erfüllt die API der Firma TTTech (Vgl. TTTech Computertechnik AG (2008)). Die API bietet bestimmte Funktionen an, welche Anwendungen das Senden und Empfangen von TTE-Nachrichten ermöglicht. Die Anwendungen greifen über die API auf die Funktionen des TTEthernet Stacks zu (Vgl. Abbildung 2.3). Der Stack vermittelt zwischen der Netzwerk- und Anwendungsschicht und übernimmt die TTEthernet spezifischen Funktionen wie die Kommunikation oder die Synchronisation des Endsystems mit der Netzwerkzeit. Die Anbindung an das Netzwerk erfolgt über eine Standard Ethernet Schnittstelle und ermöglicht so die Nutzung des Stacks mit jeder standard Ethernet-fähigen Hardware. Auf die genauen Anforderungen, welche der Stack an die Hardware stellt, wird in Abschnitt 3.1 eingegangen.

Eine detaillierte Ausführung über Architektur und Funktion des Stacks ist in der Bachelorarbeit von Kai Müller zu finden, in deren Verlauf der Stack entwickelt wurde (Vgl. Müller (2011)).

2.4 Open Source Software

Nach der Umsetzung dieser Arbeit soll der TTEthernet-Stack als Open Source Software (OSS) veröffentlicht werden. Daher wird im folgenden der Begriff Open Source Software kurz erläutert und mögliche Lizenzen für die Veröffentlichung des Stacks vorgestellt.
Eine Definition, welche Open Source Software beschreibt, stammt von der 1998 gegründeten Open Source Initiative (OSI) (Vgl. OSI) und ist in der Open Source Definition festgehalten. Diese enthält die folgenden Kriterien:

Freier Zugang zum Quellcode: Der vollständig Quellcode der Anwendung ist offengelegt und öffentlich verfügbar gemacht worden.
Freie Weitergabe der Software: Die Anwendung darf beliebig oft ohne Einschränkung des Urhebers kopiert und weiter gegeben werden.
Freie Modifikation der Software: Die Anwendung darf beliebig modifiziert werden. Die veränderte Anwendung muss aber unter den gleichen Bedingungen wie die Original-Software verbreitet werden.
Freie Nutzung: Die Software darf für beliebige Zwecke ohne Einschränkungen genutzt werden.

Software, welche all diese Kriterien erfüllt, ist nach OSI Definition Open Source Software (Vgl. (Renner u. a., 2005, S.12 f.)). Damit Software als OSS veröffentlicht werden kann, wird diese unter eine Lizenz gestellt, welche das Urheberrecht regelt. Die Lizenz schreibt die Freiheit der Software rechtsverbindlich fest. Es existiert eine große Anzahl unterschiedlicher Lizenzen, aus denen der Entwickler die für seine Zwecke am besten geeignete auswählen kann. Die drei meistgenutzten Lizenzen werden im Folgenden kurz vorgestellt:

General Public License(GPL): Hauptsächlich genutzt werden die beiden neuesten Versionen GPLv2 und GPLv3. Beide schränken die Anpassung für die interne Nutzung der Software nicht ein. Für die Weitergabe an Dritte ist allerdings festgelegt, dass Modifikationen der Software ebenfalls unter der GPL lizensiert sein müssen (Copyleft). Der Copyleft-Effekt erlaubt es nicht, Programmcode mit proprietärer Software zu verbinden und als proprietäre Software zu vertreiben.

Lesser General Public License(LGPL): LGPL ist eine Copyleft abgeschwächte Variante der GPL und erlaubt die Verbindung mit proprietärer Software. Modifikationen müssen nur Veröffentlicht werden, wenn der Quellcode der OSS verändert wurde.

Berkeley Software Distribution License(BSD): Verfolgt einen anderen Ansatz als die GPL und hat keinen Copyleft-Effekt. Die BSD Lizenz enthält keine Vorschrift dafür, unter welcher Lizenz modifizierter Quellcode weitergegeben wird. Es ist möglich, den Quellcode der OSS mit proprietären Code zu verbinden und als proprietäre Software zu vertreiben.

Die meisten der OS-Lizenzen enthalten den vollständigen Ausschluss von Haftungs-und Gewährleistungsansprüchen; das bedeutet, die Nutzung der Software erfolgt auf eigene Gefahr (Vgl. (Keßler, 2013, S.31 f.)).

3 Anforderungen

In diesem Kapitel werden die Anforderungen an die Hardware beschrieben. Anschließend werden die genutzten Hardwareplattformen vorgestellt und geprüft ob die aufgestellten Anforderungen erfüllt werden.

3.1 Anforderungen TTEthernet-Stack

Die Hardware ist gerade im Embedded-Bereich ein wichtiger Bestandteil des Gesamtsystems, welcher großen Einfluss auf die erfolgreiche Umsetzung einer Anwendung hat. Besonders bei zeitkritischen Systemen wie einem TTEthernet-fähigen Netzwerkstack spielt die Hardware eine entscheidende Rolle. Denn die Hardware definiert die obere Grenze dafür, mit welcher Latenz Aufgaben ausgeführt werden, wie schnell das System auf Anfragen reagiert und wie konstant Ausführungszeiten vorhersagbar sind. Bei einem zeitkritischen Echtzeitsystem ist eine reproduzierbare Verarbeitungszeit essentiell, da zeitliche Schwankungen im System möglichst gering sein müssen. Deswegen stellt der TTEthernet-Stack bestimmte Anforderungen an die Hardware, welche im Folgenden besprochen werden sollen. Die Anforderungen wurden der Bachelorarbeit von Kai Müller entnommen, in dessen Verlauf der Stack entstanden ist (Vgl. Müller (2011)).

Anforderungen an die Hardware:

1. Die Plattform sollte über mindestens 2 Ethernetschnittstellen verfügen, um Ausfallsicherheit durch Redundanz zu unterstützen
2. Mehrere Ethernet-Ports müssen fähig sein, parallel Frames zu empfangen bzw. zu senden
3. Alle Ethernet-Ports müssen eine Übertragungsrate von 100 MBit/s Full-Duplex unterstützen
4. Es muss ausreichend Speicherkapazität vorhanden sein um multiple Frames speichern zu können (mindestens vier Frames pro Port).
   Framegröße = 1518Byte max. Framegröße + 8Byte Zeitstempel + 18Byte reserviert = 1544Byte

1544 Byte * 4 Frames = 6176 Byte
Die Speicherkapazität sollte mindestens 6176 Byte pro Port betragen.

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 unterstützt werden.
7. Da der Stack auf einem prioritätengetriebenen Ansatz basiert, müssen Nested-Interrupts unterstützt werden, um ein konstantes Zeitverhalten zu garantieren.

Zu Anforderung 1 und 2 ist zu sagen, dass der Stack in der aktuellen Implementierung noch keine Redundanz unterstützt und nur ein Ethernetport genutzt wird. Aus diesem Grund kann der Stack gegenwärtig auch auf Hardware ausgeführt werden, welche über nur eine Ethernetschnittstelle verfügt. Daher wird auf diese Anforderungen bei der Hardware Auswahl im Abschnitt 3.2 nicht mehr eingegangen. Trotzdem soll die Anforderung bei der Umsetzung des Hardware Abstraction Layers berücksichtigt werden, da die Unterstützung von Redundanz in einer späteren Version des Stacks vorgesehen ist.

3.2 Verwendete Hardware

Der TTEthernet-Stack wurde auf dem Microcontroller-Entwicklungs-Board NXHX500 der Firma Hilscher entwickelt, welches speziell für echtzeitfähige Anwendungen konzipiert wurde (Vgl. Hilscher (b)). Ein weiteres Ziel dieser Arbeit ist, neben der allgemeinen Portierbarkeit des Stacks die Portierung auf das LPC1769-LPCXpresso Board der Firma NXP. Das LPCXpresso Board ist ein low cost Entwicklungs-Board, das als benutzerfreundliche Hardwareplattform für die Software Entwicklung entworfen wurde (Vgl. NXP).
Nachfolgend werden beide Hardwareplattformen exemplarisch gegenüber gestellt und das LPCXpresso Board in Bezug auf die Anforderungen in Abschnitt 3.1 untersucht.

NXHX500

Der Kern des Microcontrollers bildet die NETX500 CPU welche auf einer 32 Bit ARM9 CPU basiert und mit 200Mhz getaktet ist. Der Controller besitzt vier 32KByte große SRAM Bänke welche für die Verarbeitung der Kommunikationsdaten genutzt werden. Des Weiteren besitzt die CPU einen 8KByte großen Tightly Coupled Memory, welcher als interner Speicher direkt mit der CPU verbunden ist und für die Ausführung möglichst schnell abzuarbeitender Programmteile

gedacht ist. Eine Besonderheit des Prozessors sind die vier voneinander unabhängigen Kommunikationskanäle, welche full-duplex fähig sind und damit gleichzeitiges Senden und Empfangen von Daten unterstützen (Vgl. Abbildung 3.1). Die Kanäle werden von konfigurierbaren ALUs betrieben. Diese übernehmen unabhängig von der CPU das Senden und Empfangen von Daten sowie das Schreiben der Daten in den SRAM. Dabei kodieren diese die empfangenen Daten je nach konfiguriertem Protokoll bevor die Daten in den Speicher geschrieben werden. Zwei der Kanäle können zur Ethernet-Kommunikation genutzt werden und unterstützen eine Geschwindigkeit von 100 MBit/s. Außerdem wird von einer IEEE 1588 konformen Zeitstempel-Einheit jeder ankommende Frame mit einem Zeitstempel versehen.(Vgl. Hilscher: Gesellschaft für Systemautomation mbH (2008))

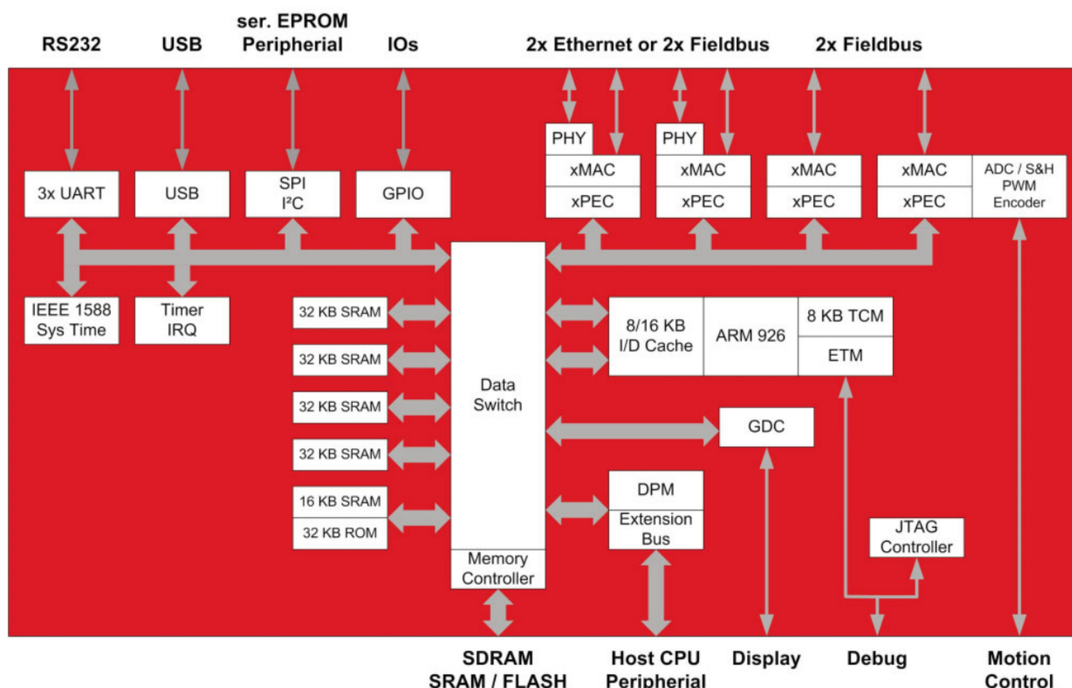

Abbildung 3.1: Blockdiagramm NXHX500 (Quelle: Hilscher (a))

LPC1769 - LPCXpresso

Der LPC1769 ist ein auf dem ARM Cortex-M3 basierender Microcontroller. Die 32 Bit ARM CPU kann bis zu 120Mhz schnell getaktet werden. Als Programmspeicher stehen 512 KByte Flash-Speicher zur Verfügung. Der Controller besitzt 64 KByte SRAM, welcher in drei Blöcke aufgeteilt ist. Zum einen eine 32 KByte große Speicherbank, welche über einen speziellen Bus mit

der CPU verbunden ist, was einen schnelleren Zugriff auf den Speicher ermöglicht. Der restliche Speicher setzt sich aus zwei 16 Kbyte-Bänken zusammen, auf die, um einen höheren Durchsatz zu erzielen, über zwei separate Buskanäle zugegriffen wird (Vgl. Abbildung 3.2). Deswegen sollten diese beiden Speicherbänke bevorzugt von der Peripherie oder als Datenspeicher genutzt werden. So kann der Speicher zum Beispiel für die Paketdaten des Ethernetblocks genutzt werden. Dieser greift dann direkt über seinen zugehörigen DMA-Controller auf den Speicher zu und entlastet so die CPU. Der LPC1769 verfügt über eine Ethernetschnittstelle, welche mit bis zu 100 MBit/s betrieben werden kann und full-duplex unterstützt. Der integrierte CAN-Controller ermöglicht die Kommunikation über das CAN Protokoll und ist mit der CAN Spezifikation 2.0B kompatibel (Vgl. NXP Semiconductors (2010)).

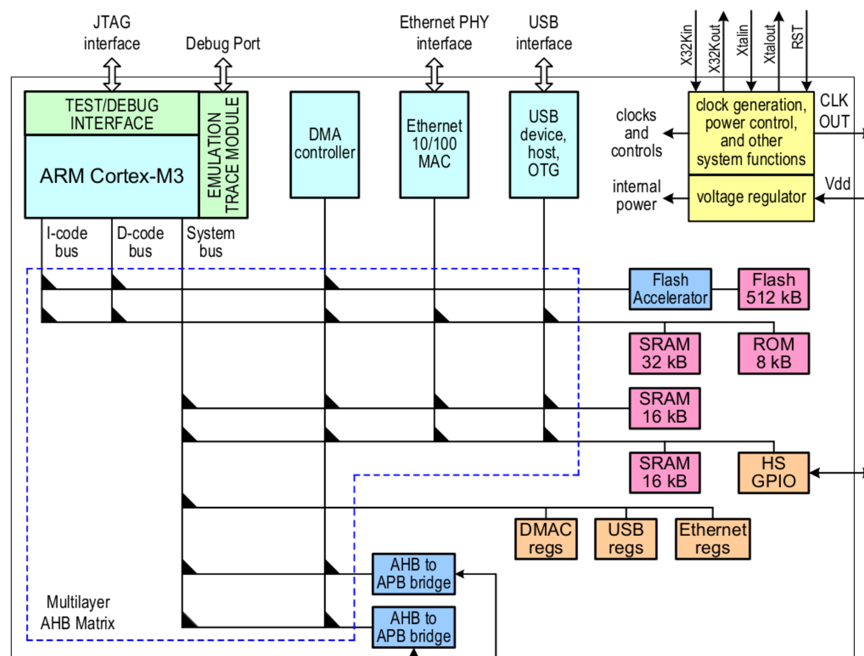

Abbildung 3.2: Ausschnitt Blockdiagramm LPC1769 (Quelle: NXP Semiconductors (2010)

Vergleich beider Microcontroller

Vergleicht man die beiden Microcontroller miteinander ist sofort offensichtlich, dass der NX-HX500 das LPCXPressoBoard bezüglich Leistung und Ausstattung übertrifft. Da der NXHX500 für den Einsatz als Netzwerkcontroller entwickelt wurde, ist dementsprechend viel Wert auf eine schnelle Verarbeitung der Paketdaten gelegt worden. Dies wird zum einen durch die leistungsstarke ARM CPU, eine auf Geschwindigkeit ausgelegte Speicherarchitektur und die

CPU-unabhängige Kommunikationsschnittstelle realisiert. Das LPCXpressoBoard wurde als kostengünstige Entwicklungsplattform konzipiert. Dabei wurde der Fokus auf einen niedrigen Stromverbrauch und einen einfachen und benutzerfreundlichen Entwicklungsprozess gelegt. Der LPC1769 wurde für den Einsatz in Eingebetteten Systemen wie z.B. zur Motorsteuerung oder für den Einsatz in Weißer Ware entwickelt (Vgl. (NXP Semiconductors, 2010, S.6)). Vor allem die geringe Leistungsaufnahme und der extrem niedrige Preis machen den Einsatz des LPCXPressoBoards interessant.
Vergleicht man die Hardware in Bezug auf die oben aufgestellten Anforderungen kann das XPressoBoard nicht alle geforderten Spezifikationen erfüllen(Vgl. Tabelle 3.1). Anforderung 5 fordert einen Mechanismus, mit dem die Empfangszeit eines Frames auf 1µs genau bestimmt werden kann. Da das XpressoBoard keine Zeitstempeleinheit für eingehende Frames besitzt, kann es diese Anforderung nicht erfüllen. Um das Board trotzdem für die Ausführung des Stacks zu nutzen, muss im nächsten Kapitel bei der Erarbeitung des Konzepts eine Lösung für die fehlende Stempeleinheit gefunden werden.
Trotz der nicht komplett erfüllten Anforderungen soll der Stack später auf das LPCXpressoBoard portiert werden. Dies dient einerseits zur Überprüfung der Portierbarkeit, zum Anderen soll die Arbeit auch ein Test für den Einsatz des Stacks auf einem low-cost Microcontroller sein.

Tabelle 3.1: LPCXPresso1769: Vergleich mit Anforderungen aus Abschnitt 3.1

Anforderung	LPCXPresso1769
1. (Paralleles Senden/Empfangen)	(X)
2. (2 Ethernetschnittstellen)	(X)
3. 100 MBits Full-Duplex	√
4. Speicherkapazität	√
5. Zeitstempel	X
6. CAN	√
7. Nested-Interrupts	√

4 Konzept

Dieses Kapitel stellt das Konzept für die Umsetzung der Stack-Portierung von möglichen Techniken der Software-Portierung bis hin zur angewandten Vorgehensweise bei der Portierung vor.

Die Portierung im Bereich Eingebetteter Systeme ist durch die hardwarenahe Programmierung und begrenzten Hardwareresourcen eine besondere Herausforderung. Gerade wenn es sich um zeitkritische Anwendungen handelt, darf die Portierbarkeit keinen großen Auswirkungen auf das Zeitverhalten des Systems haben. Die angewandte Strategie muss die Software portierbar machen und gleichzeitig möglichst wenig Einfluss auf die Performance des Systems haben. Ein Beispiel für die Umsetzung ist die Architektur der AUTOSAR-Entwicklungsgemeinschaft (Vgl. AUTOSAR Development Cooperation). AUTOSAR hat unter anderem eine standardisierte, geschichtete Software-Architektur (oder Layered Software Architecture) für die Steuergeräte im Auto entwickelt. Durch den Einsatz dieser ist die Trennung von anwendungs- und hardwarespezifischem Code möglich. Der Einsatz einer Middleware erleichtert beispielsweise die Wiederverwendbarkeit von Softwarekomponenten. Außerdem ermöglicht die hardwareunabhängige Anwendungssoftware einen Austausch oder die Erweiterung von Steuergeräten. Dadurch können Aufwand, Komplexität und Kosten erheblich reduziert werden.

4.1 Stack Architektur

Abbildung 4.1 zeigt die aktuelle Architektur des TTEthernet-Stacks sowie den Zugriff auf die Hardware. Wie zu erkennen ist, ist der Stack intern in mehrere Module unterteilt, wobei bis auf drei alle direkt auf die unterliegende Hardware zugreifen. Um die Ansprüche eines Echtzeitsystems zu erfüllen, wurde bei der Entwicklung des Stacks viel Wert auf die Performance gelegt. Dies hat zur Folge, dass der Zugriff auf die Hardware in den meisten Fällen direkt an der aktuell ausgeführten Stelle im Programmcode über die entsprechenden Zugriffsregister erfolgt. Durch den direkten Hardwarezugriff über die hardwarespezifischen Register ist der TTEthernet-Stack fest an den zur Entwicklung genutzten Microcontroller gekoppelt. Um die Portierbarkeit des

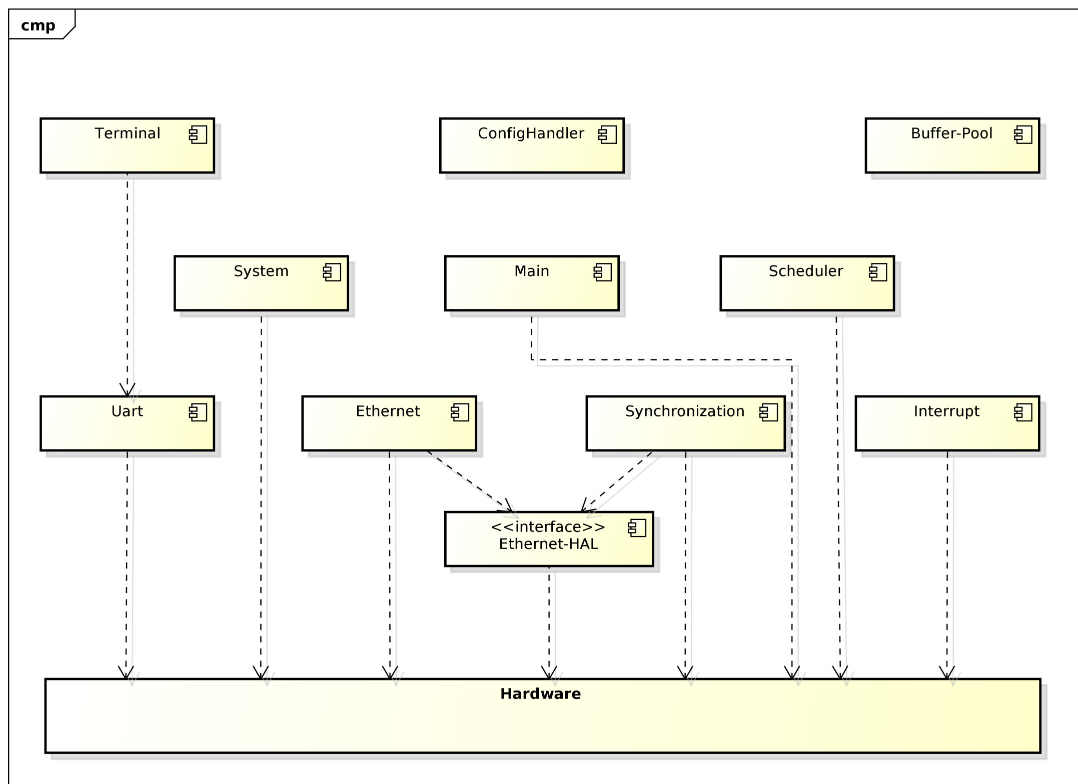

Abbildung 4.1: Architektur TTEthernet-Stack: Hardwarezugriffe der einzelnen Module (Vgl. Müller (2011))

Stacks zu ermöglichen, muss dieser von der Hardware entkoppelt werden.

4.2 Strategien Software Portierung

Für die Umsetzung von portierbarer Software haben sich unterschiedliche Techniken etabliert, welche die Abstraktion der Hardware auf unterschiedlichen Ebenen umsetzen. Einige typische Portierungstechniken werden im Folgenden diskutiert:

Eine Strategie zur Umsetzung von Portierbarkeit sind **Virtuelle Maschinen (VM)**. Dieses sind vollständig isolierte Softwarecontainer, die ihr eigenes Betriebssystem und ihre eigenen Anwendungen ausführen können. Dabei verhält sich die VM wie eine physische Maschine mit dem Unterschied, dass sie der Software die benötigten Hardwarekomponenten virtuell zur Verfügung

stellt (Vgl. VMware). Durch die Virtualisierung ist die VM unabhängig von der ihr zu Grunde liegenden Hardware. Eine Anwendung kann also innerhalb der Virtuellen Maschine unabhängig von der physischen Hardware ausgeführt werden, da sie über die auf die Anwendung angepasste virtuelle Hardware zugreift. Dabei kann es notwendig sein, dass von der Anwendung benötigte Hardware emuliert werden muss. Das bedeutet, dass diese komplett in Software nachgebildet werden. Die Verwaltung der VMs übernimmt der Hypervisor. Dieser bildet die Virtualisierungsschicht und wird entweder direkt auf der Hardware (Typ1) oder auf einem Host-Betriebssystem (Typ2) ausgeführt (Vgl. Abbildung 4.2).

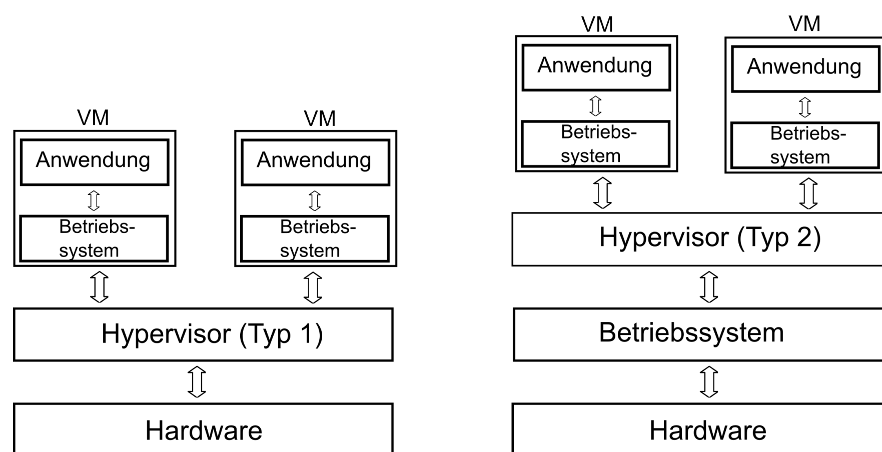

Abbildung 4.2: Architektur einer Virtuellen Maschine

Die Hardware-Abstraktion für die Anwendung findet in der VM auf Ebene des Betriebssystems statt. Durch den von der VM verursachten Verarbeitungs-und Speicher-Overhead sind Virtuelle Maschinen für Eingebettete Systeme mit beschränkter Hardware eher ungeeignet (Vgl. Marcondes u. a. (2006)). Da nicht direkt auf die Hardware zugegriffen wird, entstehen höhere Latenzzeiten, was gerade bei Echtzeit-Systemen zu Fehlfunktionen führen kann. Deshalb ist eine Virtuelle Maschine für die Umsetzung der Portierbarkeit ungeeignet.

Eine andere Technik sind **System Call Interfaces** wie sie zum Beispiel in Betriebssystemen genutzt werden (POSIX, WIN32). Diese bieten eine fest definierte Schnittstelle für Systemaufrufe des Betriebssystems und damit Zugriff auf die darunter liegende Hardware. Die Anwendungen nutzen die System Calls um Zugriff auf die Hardware zu erhalten (Vgl. Abbildung 4.3). Wenn eine Anwendung ein bestimmtes System Call Interface nutzt, kann diese auf jedem Betriebssystem ausgeführt werden, welches das Interface implementiert (Vgl. Marcondes u. a. (2006)).

Der Abstraktionslevel ist hier näher an der Hardware als bei der VM, da die Hardwareabstraktion zwischen Betriebssystem und Hardware stattfindet. Für die Umsetzung dieser Arbeit wird eine Portierungs-Strategie gesucht, welche möglichst nah an der Hardware ansetzt. Deshalb ist der Abstraktionslevel des System Call Interfaces zu hoch und somit nicht geeignet. Allerdings ist die Nutzung einer hardwareunabhängigen Schnittstelle ein Ansatz, welcher für die Umsetzung in Frage kommt.

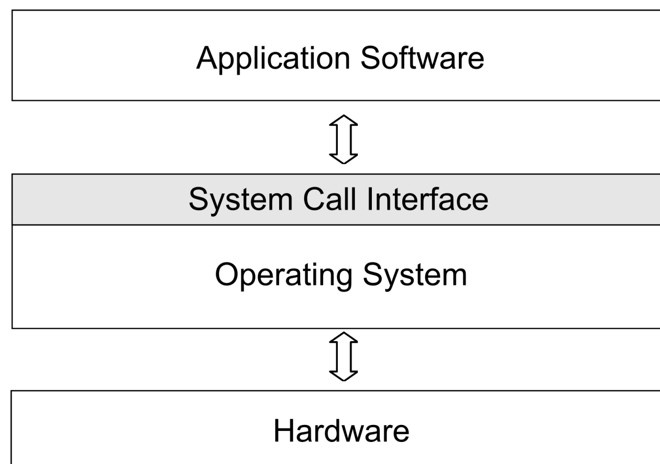

Abbildung 4.3: Architektur System Call Interface

Komponentenbasierte Entwicklung ist eine weitere Möglichkeit die Portierbarkeit von Software zu erreichen. Dafür wird das System in voneinander unabhängige Komponenten zerlegt. Jede Komponente bildet eine abgeschlossene Teilfunktion des Systems ab. Über definierte Schnittstellen findet die Kommunikation zwischen den Komponenten statt (Vgl. Abbildung 4.4). Das komponentenbasierte Design ermöglicht die Wiederverwendung und den Austausch einzelner Komponenten, was den Entwicklungsprozess erheblich vereinfacht. Durch eine geeignete Abstraktion kann eine Portabilität erreicht werden, so dass im besten Fall nur der low-level Code für jede Plattform neu geschrieben werden muss (Vgl. (Linder, 2008, S. 3)). Die Komponentenbasierte Entwicklung ist ein für Eingebettete Systeme gut geeignetes Vorgehen, um portierbare Software zu entwerfen. Der Zugriff auf die Hardware erfolgt direkt aus der Anwendung über die entsprechenden Komponenten. Es ist keine weitere Zwischenschicht wie bei den System Calls nötig. Die Komponentenbasierte Entwicklung ist ein Konzept, welches die grundlegende Struktur der Softwarearchitektur vorgibt und sollte von Beginn an zur Entwicklung des Systems genutzt werden. Wie in Abbildung 4.1 zu sehen, wurde das Komponentenbasierte Design bei der Entwicklung des Stacks bereits teilweise umgesetzt. Allerdings nicht mit dem Ziel, eine

Portierbarkeit der Software zu erreichen. Daher müsste, um eine Portabilität auf Basis des Komponentenbasierten Designs zu erreichen, die Architektur zum großen Teil angepasst werden. Es müsste eine neue Aufteilung der Module vorgenommen werden, um den hardwarespezifischen Code zusammenzufassen und die einzelnen Komponenten voneinander zu entkoppeln.

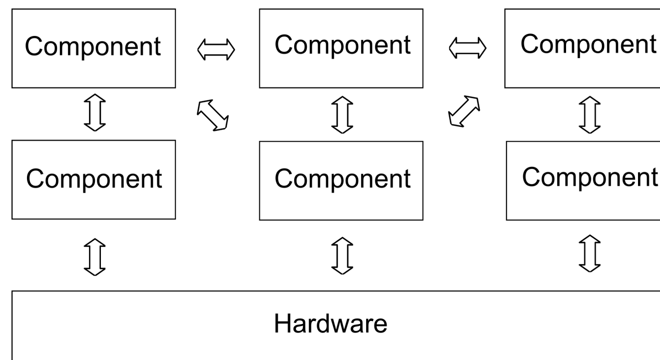

Abbildung 4.4: Architektur Komponentenbasierte Entwicklung

Eine weitere und häufig genutzte Technik ist der **Hardware Abstraction Layer(HAL)**. Vor allem Betriebssysteme wie z.B. Linux oder Windows nutzen ein HAL, um auf unterschiedlicher Hardware ausführbar zu sein. Der Hardware Abstraction Layer ist eine Schicht, welche zwischen der Hardware und der darüber liegenden Anwendungsschicht liegt und alle hardwarespezifischen Funktionen bündelt (Vgl. Abbildung 4.5). Der Zugriff auf die Hardware erfolgt über die vom HAL definierte Schnittstelle. Bei der Portierung der Anwendung auf eine anderen Hardwareplattform, muss nur der Hardware Abstraction Layer an die veränderte Hardware angepasst werden.
Der HAL ist eine effektive Methode für die Entwicklung portierbarer Software und ist durch den hardwarenahen Ansatz auch für Eingebettete Systeme gut geeignet. Für die Nutzung der Ethernet-Schnittstelle wird vom Stack bereits ein Ethernet-HAL genutzt, welcher übernommen werden kann (Vgl. Abbildung 4.1). Der Vorteil gegenüber der Komponentenbasierten Entwicklung ist vor allem die leichtere Umsetzbarkeit. Die Hardwarezugriffe werden aus den Modulen extrahiert und in den HAL ausgelagert. Die Architektur des Stack muss dafür nur geringfügig angepasst werden. Der HAL bietet eine einfache Möglichkeit, den Stack von der Hardware zu entkoppeln, ohne zu starken Einfluss auf die Verarbeitungszeit zu haben und wurde daher für die Umsetzung gewählt.

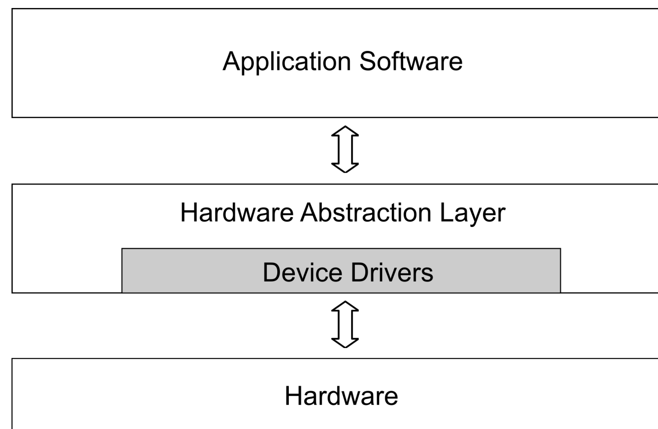

Abbildung 4.5: Architektur Hardware Abstraction Layer

4.3 Hardware Abstraction Layer

Der Hardware Abstraction Layer beinhaltet den Teil der Software, welcher direkten Zugriff auf die darunter liegende Hardware hat. Damit ist der HAL-interne Code direkt an die genutzte Hardwareplattform gekoppelt und eine Änderung der Hardware hat immer eine Anpassung des HAL zur Folge. Durch die strikte Trennung von hardware- und anwendungsspezifischen Code muss bei einer Portierung im besten Fall nur der Code des HAL an die neue Hardwarearchitektur angepasst werden. Der Anwendungscode bleibt unverändert, da diese nur über die definierte Schnittstelle des HAL auf die Hardware zugreift.

Ein wichtiger Bestandteil des HAL sind die Gerätetreiber, welche für die Konfiguration und den Zugriff komplexer Hardwarekomponenten genutzt werden. Die einzelnen Treiber dienen als Schnittstelle zwischen dem HAL und den Komponenten. Typische Hardwarekomponenten, die über Treiber angesprochen werden, sind zum Beispiel Serielle Schnittstellen, Timer, DMA Controller, Ethernetschnittstellen oder Flash Speicher. Auf andere, weniger komplexe Komponenten wie General Purpose IOs oder LEDs kann mittels einfacher Funktionen direkt über die entsprechenden Register zugegriffen werden. Diese müssen bei der Portierung entsprechend der Hardware angepasst werden. Die Funktionen für den Hardwarezugriff sind allerdings nicht die einzigen Dienste, die ein HAL üblicherweise zur Verfügung stellt. Weitere sind die Integration von benötigten Funktionsbibliotheken wie der Standard C-Library oder Funktionen zur Initialisierungen des Systems sowie der Peripherie, welche vor dem Start des eigentlichen Programms ausgeführt werden (Vgl. Abbildung 4.6). Der Großteil des HAL ist üblicherweise in C-Code geschrieben, sowie Teile in Assemblercode. Letzteres ist der prozessorspezifische Code, wie zum Beispiel Boot-Code oder Befehle zum Aktivieren und Deaktivieren der Interrupts (Vgl. (Ecker

u. a., 2009, S.74 f.)).
Die vom HAL angebotenen Dienste werden in einer Schnittstelle (HAL-API) zusammengefasst,

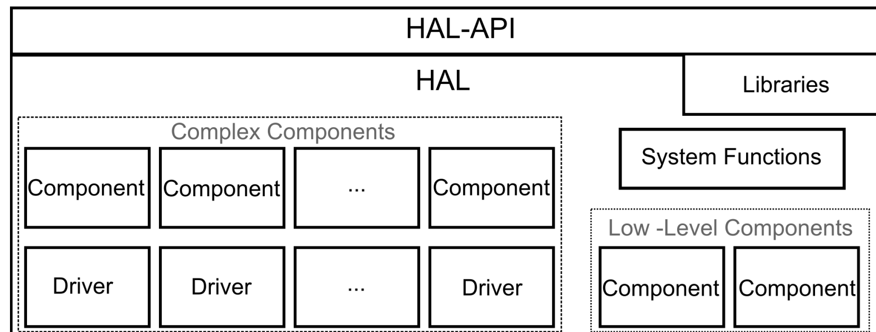

Abbildung 4.6: Aufbau Hardware Abstraction Layer

und den darüber liegenden Schichten wie Anwendungen oder Betriebssystem zur Verfügung gestellt. Die API ist eine Abstraktion des hardwarenahen HAL und bietet eine konsistente Schnittstelle die unabhängig vom hardwarespezifischen Code ist. Ohne die Implementierung der HAL-API für die Zielarchitektur bleibt die Anwendungsschicht hardwareunabhängig, was die Portierung der Software auf unterschiedliche Hardwarearchitekturen ermöglicht. Erst durch die Integration des hardwarespezifischen Codes wird die Anwendung an die genutzte Hardwareplattform gebunden. Die HAL-API ist üblicherweise in Kategorien unterteilt, welche jede ihre eigene API bereitstellt. Diese Kategorien fassen meist bestimmte Funktionen oder Typen von Geräten zusammen. Durch eine entsprechende Unterteilung ist es zum Beispiel möglich, Komponenten optional zu unterstützen, sodass nicht generell die komplette API unterstützt werden muss.

4.4 HAL TTEthernet-Stack

Für die Einführung des HAL in den TTEthernet-Stack muss die Vermischung von hardwarespezifischen und anwendungsspezifischen Code aufgehoben werden. Durch das Extrahieren des hardwarespezifischen Codes und die Einführung einer hardwareunabhängigen Schnittstelle (HAL-API) wird der Stack unabhängig von der genutzten Hardwareplattform. Nach der Einführung des HAL besteht der TTEthernet-Stack aus dem funktionalen Code, welcher die Stack Funktionalität und den hardwarenahen Code für den Zugriff auf die Hardware Ressourcen beinhaltet. Der funktionale Teil des Stacks greift nur über die in der HAL-API definierten Funktionen auf die darunterliegende Hardware zu.
Wie in Abbildung 4.1 zu sehen, besteht der Stack intern aus mehreren Modulen, welche für unterschiedliche Aufgaben zuständig sind. Einige dieser Module benötigen Zugriff auf die Hardware,

welcher wie oben beschrieben mit direktem Zugriff über die zuständigen Registern realisiert wurde. Damit die Architektur des Stacks erhalten bleibt, wurde bei der Umsetzung versucht, die Aufteilung der Module beizubehalten und Hardwarezugriffe in den Modulen durch Funktionsaufrufe des HAL zu ersetzen. Im folgenden Abschnitt wird die Strategie für die Umsetzung dargestellt.

Strategie

- Extrahieren aller Hardwarezugriffe.
- Erstellen der API in der alle Funktionen des HAL definiert sind.
  - Einteilung der API nach Hardware-Komponenten. Alle Funktionen für ein Gerät bzw. für eine Geräteklasse werden in einer Teil-API zusammengefasst.
  - Die Funktionen der API sind architekturunabhängig, um eine Vielzahl von Hardwareplattformen zu unterstützen.
  - Die Funktionen sollen die typische Nutzung der jeweiligen Geräteklasse so gut wie möglich abbilden. Das Ziel ist, mit der definierten Schnittstelle jede Hardware der jeweiligen Klasse nutzen zu können.
  - Optimierung der Funktionen für die Stack spezifische Nutzung der Hardware, um hohe Latenzen durch zu viele Funktionsaufrufe zu vermeiden.(z.B. zusammenhängende Hardwarezugriffe zu Funktionen zusammenfassen)
  - Unterstützung von Redundanz (paralleles Senden und Empfangen über zwei Ethernetports). Entsprechende Funktionen werden in Abhängigkeit vom jeweiligen Ethernet-Port ausgeführt.
  - Die Timer-API muss eine Funktion zur Ermittlung der aktuellen Systemzeit zur Verfügung stellen, damit auch Hardware unterstützt werden kann, welche keine Hardware-Stempeleinheit besitzt.
- Erstellen einer Doxygen-Dokumentation der HAL-API um die Portierung des Stacks und die Nutzung der API zu erleichtern.

Aufbau der HAL-API

In Abbildung 4.7 ist die vorgenommene Aufteilung der HAL-API dargestellt. Im Folgenden werden der Aufbau und Funktionen HAL-API grob umrissen. Eine ausführliche Übersicht und

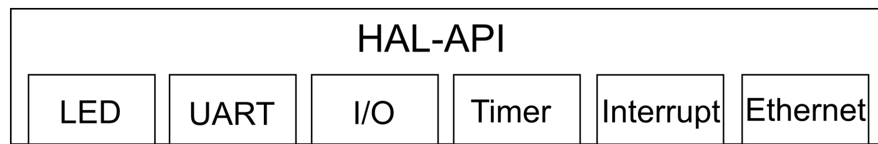

Abbildung 4.7: Struktur HAL API

Beschreibung der API Funktionen ist in der API-Dokumentation zu finden, welche im Projektverzeichnis der API-Implementierung abgelegt ist.

LED Die LED-API stellt Funktionen zur Initialisierung und zum Aktualisieren der LEDs zur Verfügung. Dies beinhaltet das Ein- und Ausschalten von einzelnen LEDs, sowie die Aktualisierung von Status- und Fehler-LEDs.

UART Über die UART-API ist es möglich, eine Serielle Schnittstelle zu nutzen. Die API definiert eine Funktion zur Initialisierung der UART-Schnittstelle und Funktionen zur Ausgabe von Datenblöcken. Funktionen zum Empfangen wurden nicht mit aufgenommen, da die Serielle Schnittstelle nur zur Ausgabe von Status- und Debugging- Informationen dient.

I/O Die I/O-API definiert Funktionen zur Nutzung der GPIOs. Eine zur Initialisierung und Konfiguration der GPIO-Pins und Funktionen zum Lesen- und Schreiben der GPIOs.

Timer In der Timer-API werden alle Timer-spezifischen Funktionen zusammengefasst. Die API definiert zum Beispiel Funktionen zur Initialisierung, Start, Stop und Reset eines Timers. Neben den Funktionen für die einfachen Timer sind auch Funktionen für den System-Timer enthalten. Der System-Timer dient als zentraler Zeitgeber des Systems und bietet spezielle Funktionen zur Anpassung der Schrittweite des Timers. Des Weiteren ist eine Zeitstempel Funktion zur Ermittlung der aktuellen Systemzeit enthalten.

Interrupt Die Interrupt-API definiert die Funktionen zur Interruptverwaltung. Die API stellt Funktionen zur Initialisierung und Konfiguration des Interrupt Controllers zur Verfügung. Das sind zum Beispiel die Funktionen zum Zurücksetzen der Interrupts oder zum setzen der Interrupt Service Routine. Des Weiteren sind mehrere Konstanten definiert, welche die Prioritäten für die Ausführung der Interrupt gesteuerten Tasks festlegen.

Ethernet Die Ethernet-API stellt die Funktionen für die Nutzung der Ethernet-Schnittstelle zur Verfügung. Unter anderem Funktionen zur Konfiguration von PHY und MAC oder Funktionen zum Senden und Empfangen von Frames. Diese sind so definiert, dass eine

Nutzung mit zwei Ethernetports möglich ist. Des Weiteren ist ein Datentyp definiert, welcher den internen Aufbau der Ethernet Frames festlegt, sowie Konstanten zum Setzen der MAC-Adresse oder zur Definition der Buffergröße.

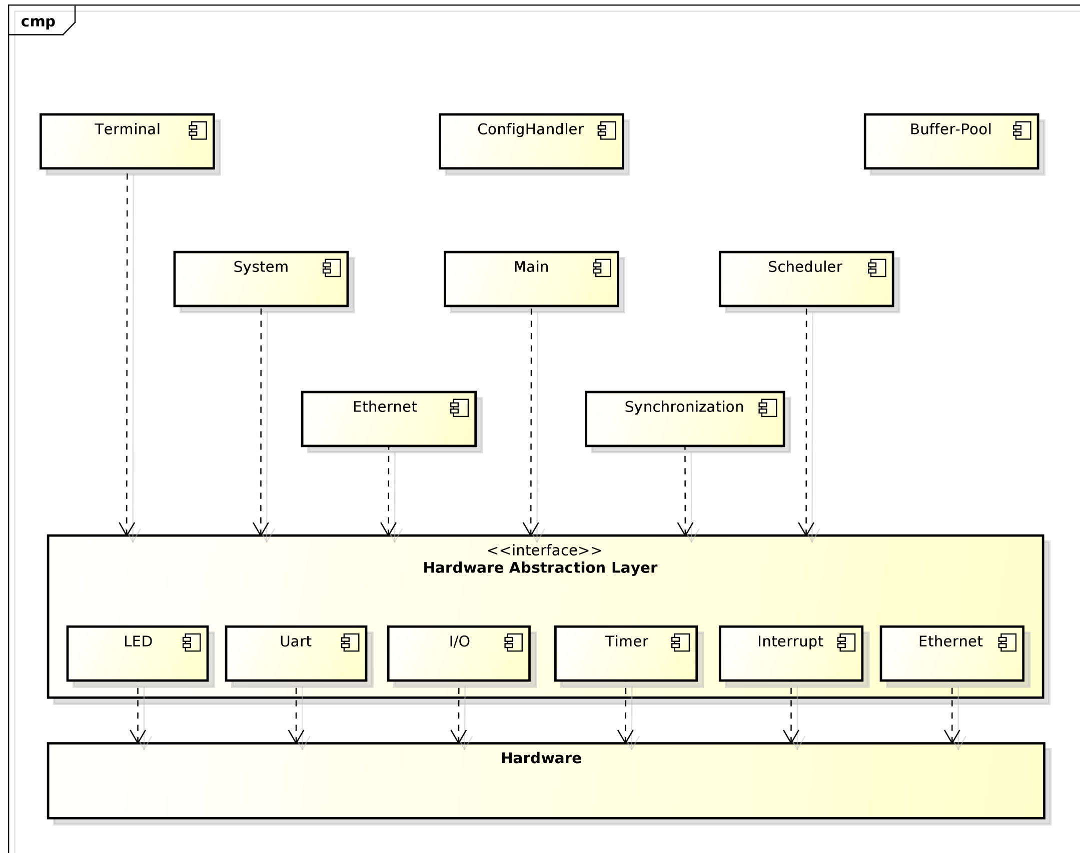

Abbildung 4.8: Architektur TTEthernet-Stack: Hardwarezugriffe nach Einführung des HAL

In Abbildung 4.8 ist die angepasste Architektur des Stacks nach der Einführung des HALs dargestellt. Der Hardwarezugriff findet nicht mehr direkt in den Stack-Modulen statt (Vgl. Abbildung 4.1) sondern über die Funktionen des HALs. Diese bündelt alle Hardwarezugriffe und liegt als hardwareabstrahierende Schicht zwischen den Stack-Modulen und der Hardware.

5 Umsetzung

In diesem Kapitel wird die Realisierung des Hardware Abstraction Layers erläutert. Unter anderem wird auf die Details einzelner HAL Komponenten eingegangen und daran die Portierung des Stacks auf das LPCXpressoBoard diskutiert.

5.1 Realisierung

Der Hardware Abstraction Layer sowie die HAL-API wurden, wie der bestehende TTEthernet-Stack in der Programmiersprache C implementiert. Die Nutzung von C bringt den Vorteil einer guten Quellcode-Portabilität mit sich. Da für die meisten Hardwareplattformen ein C-Compiler existiert, erleichtert C die Portierung des Stacks. So kann zum Beispiel, mit Hilfe der GNU Compiler Collection (GCC) der Stack auf eine große Auswahl an Hardwareplattformen portiert werden.

API

Die Struktur der HAL-API wurde nach der im letzten Kapitel dargestellten Aufteilung realisiert (Vgl. Abbildung 4.6). Den Kern der API bildet eine Header-Datei, über welche die weiteren API-Komponenten inkludiert werden. Jede API-Komponente hat seine eigene Header-Datei, in welcher von der API angebotenen Funktionen definiert sind. Dies ist mit Hilfe von Methoden realisiert, welche mit dem Schlüsselwort *extern* versehen sind. Das bedeutet, dass die Methoden in den Header-Dateien nur deklariert werden und die vollständige Implementierung in einer anderen Datei zu finden ist. Die Prototypen der Funktionen definieren lediglich den Namen der Methode, sowie Anzahl und Datentyp der Parameter. Des Weiteren werden mit den in C üblichen Strukturen eigene Datentypen definiert, welche von API-Funktionen und dem TTEthernet-Stack genutzt werden. Einige Funktionen des Stacks benötigen hardwareabhängige Konstanten, welche in Form von Symbolischen Konstanten in der API deklariert werden.

HAL

Parallel zur API-Struktur wurde für jede Komponente ein C-Modul erstellt, in welchen die in der API zuvor deklarierten Funktionen implementiert werden. Der Inhalt der Funktionen ist der hardwarespezifische Code, welcher bei Portierung des Stacks angepasst werden muss.
Durch die Einführung des HAL ist die Anzahl von Funktionsaufrufen erheblich angestiegen. Gerade bei oft ausgeführten oder zeitkritischen Funktionen kann dies zu höheren Latenzen des Systems führen. Um dem entgegen zu wirken, wurden entsprechende Funktionen mit dem Attribut *always_inline* deklariert. Mit dem Schlüsselwort *inline* ist es möglich, die Ausführungszeit des Codes durch den Compiler zu optimieren. Der Code der inline-Funktion wird vom Compiler an der Stelle eingefügt, wo der Aufruf der Funktion stattfindet. Wurden Funktionen mit inline deklariert, entscheidet der Compiler, ob eine Ersetzung des Codes sinnvoll ist oder nicht. Durch das Attribut *always_inline* wird das Einfügen des Codes durch den Compiler erzwungen. Der Vorteil der inline-Funktionen ist, dass der Code direkt ausgeführt wird und der Sprung in das Unterprogramm und der dadurch entstehende Overhead entfällt.

5.2 Timer-Funktionalität des TTEthernet-Stacks

Die Timer sind ein wichtiger Bestandteil des Systems, da der Stack die Timer nicht nur als Zeitgeber, sondern auch als Interrupt-Quelle zur Aktivierung der auszuführenden Tasks nutzt. Letzteres wurde mit Hilfe eines Standard-Timers umgesetzt. Für die Systemzeit und die Synchronisation des Systems nutzt der Stack einen speziellen Timer, welcher einige erweiterte Funktionen bietet. Diese werden im Folgenden kurz erläutert und deren Auswirkungen auf den HAL dargelegt sowie deren Umsetzung auf dem LPCXpressoBoard diskutiert.

Zeitstempeleinheit

Für die Funktion des TTEthernet-Stacks ist es notwendig, dass alle eingehenden Frames mit einem Zeitstempel mit einer Genauigkeit von unter 1µs versehen werden. Der Zeitstempel ist nötig, um festzustellen, ob eine TT-Nachricht rechtzeitig angekommen ist oder um die Zeit zwischen Ankunft und Verarbeitung eines PC-Frames zu bestimmen und damit die korrekte Zeitsynchronisation zu ermöglichen (Vgl. Abschnitt 2.2). Um eine möglichst hohe Genauigkeit zu erhalten, ist eine in Hardware realisierte Stempeleinheit, welche direkt in der Empfangslogik sitzt, die beste Lösung. Da diese den Frame unmittelbar nach dem Eintreffen stempelt, ist der Delay zwischen realer Empfangszeit und Zeitstempel konstant. Da nicht jede Hardware eine

Zeitstempeleinheit zur Verfügung stellt, muss das Stempeln der Frames alternativ in Software realisiert werden. Durch die Softwarelösung erhöht sich die Differenz zwischen realer Empfangszeit und Zeitstempel. Entscheidend ist allerdings nicht die Größe des Delays, sondern, dass dieser möglichst konstant ist. Besitzt die Zeitstempelfunktion einen konstanten Delay, kann der Zeitstempel durch die Subtraktion des Delays korrigiert werden.
Bei der Entwicklung des bestehenden TTEthernet-Stacks wurde die vom NXHX500 bereitgestellte Stempeleinheit genutzt, welche mit einer Auflösung von 10ns ankommende Frames stempelt. Um die Nutzung von Hardware ohne integrierter Stempeleinheit wie dem LPCXpresso Board zu ermöglichen, wurde in der Timer-API eine Zeitstempel-Funktion deklariert und der Stack soweit angepasst, dass optional ein Softwarezeitstempel genutzt werden kann. Die Funktion kann über eine Präprozessor Direktive aktiviert werden, welche in der Timer-API definiert ist. Damit Frames, welche in kurzen Abständen hintereinander eintreffen, korrekt gestempelt werden, muss der Delay zwischen Empfang und Stempelvorgang möglichst gering sein. Deshalb wird der Zeitstempel direkt nach dem Eintritt in die Interrupt Service Routine(ISR) der Ethernetschnittstelle zwischengespeichert und anschließend bei der Verarbeitung in das entsprechende Feld des Frames geschrieben. Dabei wird der Zeitstempel mit einem konstanten Delay verrechnet, welcher in Abhängigkeit von der Hardware bestimmt und angepasst werden muss. Der Interrupt, welcher die ISR auslöst, tritt auf sobald ein neuer Frame empfangen und in den Buffer geschrieben wurde. Durch den prioritätengetriebenen Ansatz haben die Interrupts, welche für das Starten der Tasks zuständig sind, unterschiedliche Prioritäten. Der Interrupt der Ethernetschnitstelle besitzt die zweit höchste Priorität im System und kann dementsprechend nur von einem höherpriorisierten Interrupt unterbrochen werden. Sollten beide Interrupts zeitgleich auftreten, kann sich die Ausführungszeit der Empfangs-ISR verschieben und Schwankungen des Delays verursachen. In den meisten Fällen sollte allerdings eine zeitnahe Verarbeitung des Interrupts und damit ein geringer Delay gewährleistet sein. Eine Untersuchung des Delays für den Software-Zeitstempels am Beispiel des LPCXpresso Boards ist in Kapitel 6 zu finden.

Synchronisation der lokalen Uhr

Wie in Abschnitt 2.2 beschrieben ist es für die Realisierung eines TTEthernet-Netzwerks unumgänglich, dass sich die Netzwerkteilnehmer untereinander auf eine netzwerkweite Zeit synchronisieren. Für notwendige Anpassung der lokalen Uhr nutzt der TTEthernet-Stack zwei unterschiedliche Verfahren. Die Systemzeit kann direkt gesetzt (Offset-Correction) oder indirekt beeinflusst werden, indem die Schrittweite der lokalen Uhr angepasst wird (Rate-Correction) (Vgl. (Müller, 2011, S.50)). Durch die Rate-Correction wird die Geschwindigkeit der lokalen Uhr verändert um damit eine stetige Anpassung der Uhr zu erreichen. Der Vorteil dieser Methode ist, dass fehlerhafte

Synchronisationsframes nicht sofort zu einer asynchronen Systemzeit führen. Der Nachteil bei diesem Verfahren ist die Zeitverzögerung die entsteht, bis die Synchronisation der Uhr erreicht ist. Hier liegt der Vorteil der Offset-Correction, da die Synchronisation sofort erreicht wird. Der Stack wendet eine Kombination der beiden Verfahren an und kann so die Vorteile beider Methoden nutzen.
Die Anpassung der Schrittweite ist eine spezielle Funktion, welche nicht jeder Timer beherrscht. Daher wurde der Stack soweit angepasst, dass die Anpassung der Schrittweite optional zur Synchronisation der lokalen Zeit genutzt werden kann. Da die Schrittweiten-Einstellung eine Funktion der Timer-Hardware ist, wurden die entsprechenden Funktionen der HAL-API angepasst und als optional deklariert, um auch Hardware zu unterstützen, welche die spezielle Timer-Funktion nicht bereitstellt. Über eine Präprozessor Direktive wird angegeben ob die genutzte Hardware die Funktion unterstützt. Ist dies der Fall, muss die entsprechende Funktion auch in der HAL implementiert werden. Abhängig von der Präprozessor Direktive wird außerdem die Synchronisationsroutine des Stacks angepasst. Bei der Ausführung der Synchronisations-Methoden auf einer Hardwareplattform, welche keine Rate-Correction unterstützt, wird die lokale Zeit dann mittels Offset-Correction angepasst.

Umsetzung auf dem LPCXpressoBoard

Wie in Kapitel 3.2 bereits besprochen, besitzt der LPC1769 keine Stempeleinheit welche ankommende Frames mit einem Zeitstempel versieht. Deshalb wurde für die Portierung auf das LPCXpressoBoard ein Softwarezeitstempel genutzt. Dieser wurde mit Hilfe eines 32 Bit Repetitive Interrupt Timers realisiert, welcher gleichzeitig als System Timer dient. Der Timer wird mit der Frequenz der CPU betrieben und bietet bei CCLK=100MHz eine Auflösung von 10ns. Das Stempeln der Frames wurde wie in Abschnitt 5.2 beschrieben mit Hilfe der entsprechenden Zeitstempel-Funktion des HAL realisiert.
Da der Timer als System Timer fungiert, muss dieser mit der netzwerkweiten Zeit synchronisiert werden. Der Timer besitzt keine Rate-Correction und wird daher mit der angepassten Synchronisationsroutine mittels Offset-Correction direkt auf die Netzwerkzeit gesetzt.

5.3 Interruptbehandlung des TTEthernet-Stacks

Ein zentrales Konzept des Stacks ist der Prioritäten getriebenen Ansatz. Dabei werden den zu verarbeitenden Aufgaben (Tasks) unterschiedliche Prioritäten für die Ausführung zugewiesen. Dies wurde bei der bestehenden Stack-Umsetzung mit Hilfe von Interrupts realisiert.

Vectored Interrupt Controller NXHX500

Der Vectored Interrupt Controller (VIC) des NETX500 besitzt eine Vektortabelle in welcher die Zuordnung von Interrupt-Quelle und Interrupt Service Routine (ISR) eingetragen werden. Durch die Position in der Tabelle wird die Priorität des Interrupts bestimmt. Dabei ist es möglich, eine Interrupt-Quelle mehrfach mit jeweils unterschiedlichen ISRs einzutragen und diese zu- oder abzuschalten

Tabelle 5.1: Vektortabelle

IRQ	Interrupt-Source	ISR
1	Ethernet0	ethernet_receive0
2	Ethernet1	ethernet_receive1
3	Timer0	scheduler_syncextern
4	Timer0	scheduler_syncactive
5	Timer2	synchronization_clientCall
6	Timer0	scheduler_sendTT
7	Timer0	scheduler_sendBG
8	Timer0	scheduler_task
9	Timer0	callback_task
10	Timer0	backround_task

Das Scheduling der Tasks auf TTEthernet-Stack ist stark an dieses Konzept angelehnt. Jede Task bekommt eine Interrupt-Leitung zugeordnet, welche gleichzeitig die Priorität der Task darstellt (Vgl. Tabelle 5.1). Dem Gerät, welches als Interrupt-Quelle dient, wird dann die Interrupt-Leitung zugewiesen. Die jeweilige ISR sorgt bei der Aktivierung des Interrupts für die Ausführung der Task. Die Möglichkeit einer Interrupt-Quelle mehrere ISRs zuzuordnen, wird vom Stack genutzt um mit Hilfe einer Interrupt-Quelle Tasks mit unterschiedlichen Prioritäten auszuführen. Die Prioritäten der einzelnen Tasks werden in der Initialisierungsphase des Systems zugewiesen, können allerdings vor dem Start geändert werden um diese an die jeweilige Anwendung anzupassen (Vgl. (Müller, 2011, S.36 f)).

Da es sich dabei um ein zentrales Architektur-Konzept des Stacks handelt, wurde bei der Umsetzung des HAL die Schnittstelle so gewählt, dass das bestehende Konzept der Interruptverarbeitung weiterhin genutzt werden kann. Das bedeutet, dass beim Eintragen der ISR die Interrupt-Quelle sowie die Priorität angegeben wird. Weiterhin können einer Interrupt-Quelle mehrere ISRs zugewiesen werden und einzeln zu- oder abgeschaltet werden. So bleibt das Grundkonzept des Interrupt Controllers erhalten.

Vectored Interrupt Controller LPCXpressoBoard

Der VIC des LPC1769 nutzt ein anderes Konzept zur Priorisierung der Interrupts. Im Unterschied zum NETX500 können die Prioritäten nicht frei für jede Interrupt-Leitung festgelegt werden, sondern jede Interrupt-Quelle bekommt eine Priorität direkt zugewiesen. Des Weiteren enthält die zugehörige Vektortabelle für jede Interrupt-Quelle einen Eintrag, in welchem die Sprungadresse der jeweiligen ISR eingetragen ist. Somit kann - anders als beim NETX500 - für jede Interrupt-Quelle nur eine ISR ausgeführt und damit auch nur eine Task pro Interrupt-Quelle gestartet werden.
Um dem Stack trotzdem die benötigte Funktionalität zur Verfügung zu stellen, wurden in der HAL Implementierung des LPCXpressoBoards zwei Datenstrukturen zur Verwaltung der ISRs angelegt, welche die Vektortabelle des NETX500 nachbilden. In diesen können für jede Interrupt-Quelle mehrere ISRs eingetragen werden. Die Priorität der einzelnen ISR ist über die Position in der Datenstruktur definiert. Des Weiteren können über die Datenstruktur die ISRs aktiviert/deaktiviert werden. Tritt ein Interrupt auf, wird eine Start-ISR angesprungen, in welcher mit Hilfe der in den Datenstrukturen gespeicherten Funktionspointer die weiteren ISRs ausgeführt werden.

5.4 Ethernet-Schnittstelle des TTEthernet-Stacks

Die Ethernet-Schnittstelle ist ein zentraler Bestandteil der Stack Architektur, da diese die Verbindung zum Netzwerk realisiert. In der ursprünglichen Version des Stacks wird für den Zugriff auf die Ethernet-Schnittstelle bereits eine vom Hersteller des NXHX500 bereitgestellte Ethernet-HAL genutzt. Daher wurde in den ersten Entwürfen des HAL eine Nutzung dieser Schnittstelle erwägt. Allerdings stellte sich im Laufe der Konzeptionierung heraus, dass die Schnittstellen-Definition die Anforderungen des Konzepts nicht vollständig erfüllt. Da nicht alle vom TTEthernet-Stack benötigten Funktionen angeboten werden, greift dieser in der bestehenden Implementierung teilweise auf Funktionen der Ethernet-Schnittstelle zu, ohne den Ethernet-HAL zu nutzen. Aus diesem Grund wurde die vorhandene Schnittstelle als Grundlage für eine eigene Definition genutzt und um die benötigten Funktionen erweitert. Dabei wurde sich vorrangig an der vom Stack benötigten Funktionalität orientiert und die Schnittstelle so angepasst, dass eine hardwareunabhängige Schnittstelle entsteht. Übernommen wurde unter anderem der im Ethernet-HAL definierte Datentyp, welcher die Stack interne Struktur der Ethernetframes beschreibt. Der Frame wird für die Verarbeitung im Stack um Felder für den Zeitstempel erweitert. Durch die Übernahme des Datentyps ist keine Anpassung der Stack internen Verarbeitung notwendig.
Alle Funktionen des neu eingeführten Ethernet-HAL sind so definiert, dass diese in Abhängigkeit

vom angegebenen Ethernet Port ausgeführt werden. Damit ist es möglich, Ausfallsicherheit durch Redundanz zu unterstützen, bei der die komplette Kommunikation parallel über zwei Ports erfolgt.

Ethernet NXHX500

Zum Speichern der Frames wird ein Speicherbereich reserviert und in der Framegröße entsprechende Blöcke unterteilt. Die Frames werden in zwei FIFO-Datenstrukturen verwaltet, wobei eine für ungenutzte und eine für genutzte Frames vorgesehen ist. Soll eine Nachricht versendet werden, wird über eine HAL-Funktion ein leerer Frame angefordert (Vgl. Abbildung 5.1). Damit erhält man exklusiv Zugriff auf den für den Frame allozierten Speicher. Wenn der Frame dann mit Inhalt gefüllt wurde, wird der Ethernet-Schnittstelle über eine weitere Funktion mitgeteilt, dass der Frame verschickt werden kann. Wenn der Sendevorgang abgeschlossen ist, wird der Frame wieder als frei markiert und kann erneut angefordert werden.

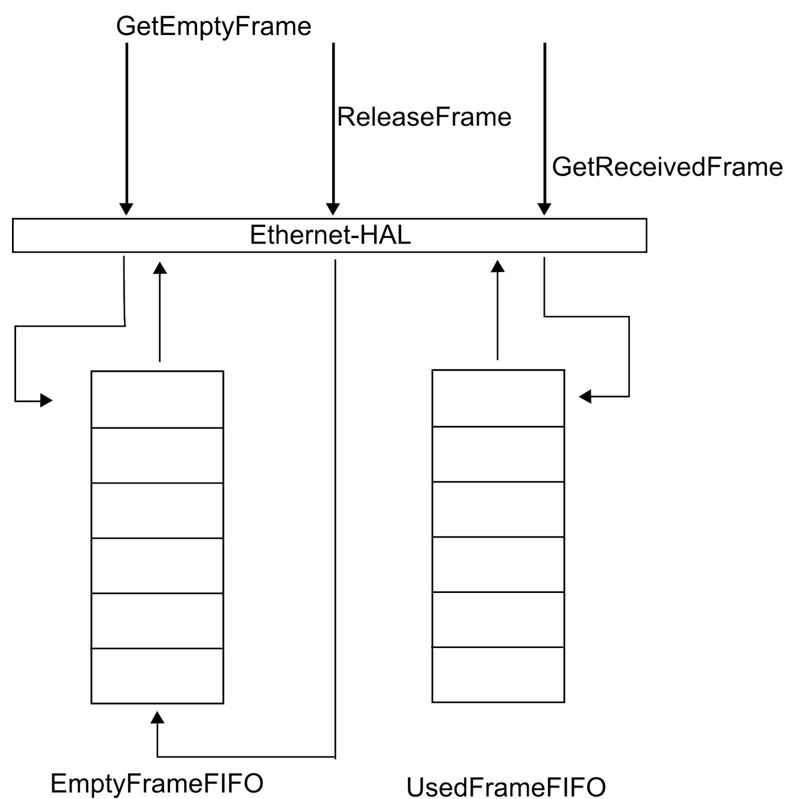

Abbildung 5.1: Frame Verwaltung NXHX500

Der Empfang eines Frames wird über den entsprechenden Interrupt propagiert. Anschließend erhält man über eine Funktion des HAL Zugriff auf den empfangenen Frame. Der Frame bzw. der allozierte Speicher bleibt so lange belegt, bis man den Frame über eine weitere HAL-Funktion wieder freigibt.

Ethernet LPCXpressoBoard

Die Verwaltung der Frames beim LPC1769 folgt einem anderen Konzept, welches ein andere Vorgehensweise für die Nutzung der Ethernet-Schnittstelle zur Folge hat. Für das Empfangen und Senden von Frames wird jeweils ein Buffer angelegt, welcher wiederum mit Hilfe von jeweils zwei Strukturen verwaltet wird. Jedem Frame wird dafür ein Descriptor zugewiesen, welcher sich aus einem Zeiger auf Speicherposition des Frames und einem Kontroll-Feld zusammensetzt. Die Descriptoren werden in einem Array verwaltet. Parallel zu dem Descriptor-Array existiert ein Status-Array, in welchem für jeden Descriptor ein Status-Feld angelegt ist und Flags über den Empfangs- bzw. Sende-Status enthält (Vgl. Abbildung 5.2). Für die beiden Descriptor-Arrays sind jeweils zwei Register (ProduceIndex/ConsumeIndex) definiert, welche den Zugriff auf die Descriptoren verwalten. Diese bestimmen welcher Descriptor als nächstes von der Hardware bzw. von der Software genutzt wird und sind als einfache Zähler realisiert (Vgl. (NXP Semiconductors, 2010, S. 172 f.)). Bei der Initialisierung muss die Software die Descriptor-Arrays, die Status-Arrays sowie die Daten-Buffer initialisieren.

Soll ein Frame verschickt werden, wird zunächst der aktuelle TxProduceIndex des TxDescriptor-Arrays gelesen. Dieser enthält die Position des Descriptors, welcher als nächstes von der Software genutzt werden kann um einen Frame zu verschicken. Damit kann anschließend der entsprechende Descriptor über das Kontroll-Feld konfiguriert werden und die Daten an die vom Descriptor verweisende Stelle geschrieben werden. Durch das Inkrementieren des TxProduceIndex wird der Frame an die Hardware zum Verschicken übergeben. Nach dem Verschicken inkrementiert die Hardware den TxConsumeIndex und übergibt den Descriptor damit wieder zur Nutzung an die Software.
Beim Empfang eines Frames wird der aktuelle RxConsumeIndex des RxDescriptor-Arrays gelesen. So erhält die Software die Descriptor-Position, welche als nächstes Verarbeitet werden muss. Der so erhaltene Descriptor verweist auf den ältesten, noch nicht gelesenen Frame im Daten-Buffer. Nachdem der Frame verarbeitet wurde, muss die Software den RxConsumeIndex inkrementieren, um den aktuellen Frame wieder an die Hardware zu übergeben und um Zugriff auf den nächsten

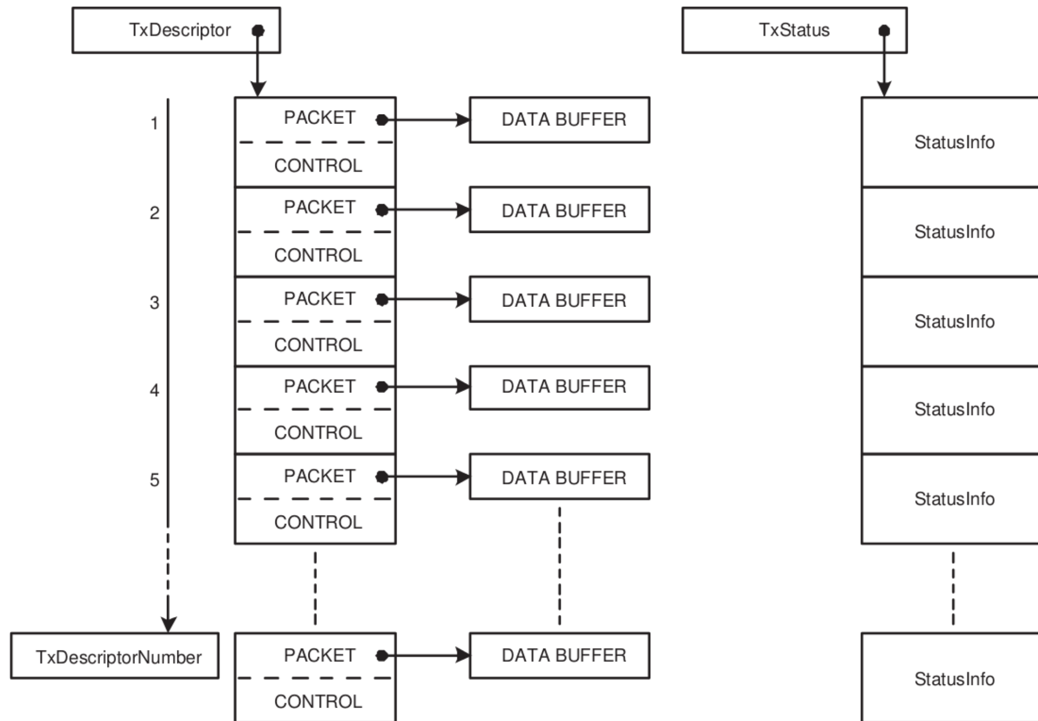

Abbildung 5.2: Transmit Descriptor Speicher Layout (Quelle: NXP Semiconductors (2010))

empfangenen Frame zu erhalten.

Für die Nutzung des TTEthernet-Stacks ist es nötig, dass die Software zeitgleich mehrere Frames reservieren und diese zu einem beliebigen Zeitpunkt wieder freigeben kann. Bei der Umsetzung auf dem NXHX500 ist dies über zwei Funktionen des Ethernet-HAL möglich. Das Konzept des LPCXpresso gibt durch die Verwaltung als Ring-Buffer vor, dass die Freigabe der Frames in der bei der Initialisierung festgelegten Reihenfolge geschieht. Hat die Software mehrere Frames reserviert, kann ein Frame erst freigeben werden, wenn sein Vorgänger freigegeben ist. Da die Frames unterschiedlich lang vom Stack reserviert werden, kann es dadurch zu einem Bufferüberlauf kommen. Eine Umsetzung der HAL-Funktionen war durch die Verwaltung der Frames als Ring-Buffer Struktur so nicht möglich und wurde deshalb, im von der Hardware gegebenen Rahmen, angepasst Aus diesem Grund wurde für die Implementierung der HAL-Funktionen die feste Zuordnung von Discriptor und Bufferelement aufgelöst. Die Zuordnung wird bei der Initialisierung - wie in Abbildung 5.2 zu sehen - festgelegt, kann sich aber während der Ausführung ändern und eine komplett neue Zuordnung entstehen (Vgl. Abbildung 5.3).

Wird über die HAL-Funktion ein Frame freigegeben, holt sich diese über den ConsumeIndex den aktuell von der Software genutzten Descriptor und schreibt die Adresse des freizugebenen Frames in das Adressfeld des Descriptors. Danach wird wieder der ConsumeIndex inkrementiert um den Frame an die Hardware zu übergeben. Das Senden der Frames funktioniert nach dem gleichen Prinzip, nur dass hier der TxProduceIndex den zu nutzenden Descriptor liefert. Die Verwaltung der Daten-Buffer wird über eine neu angelegte Struktur vorgenommen. Diese liefert über eine Funktion den nächsten freien Frame, welcher dann mit dem zuvor erhaltenen Descriptor verknüpft wird.
Nach der Anpassung ist es wie beim NXHX500 möglich, dass der Stack die Frames beliebig lang reserviert bis er sie über die entsprechende Funktion wieder freigibt. Damit können alle Funktionen wie von der HAL-API vorgegeben auf dem LPCXpressoBoard genutzt werden.

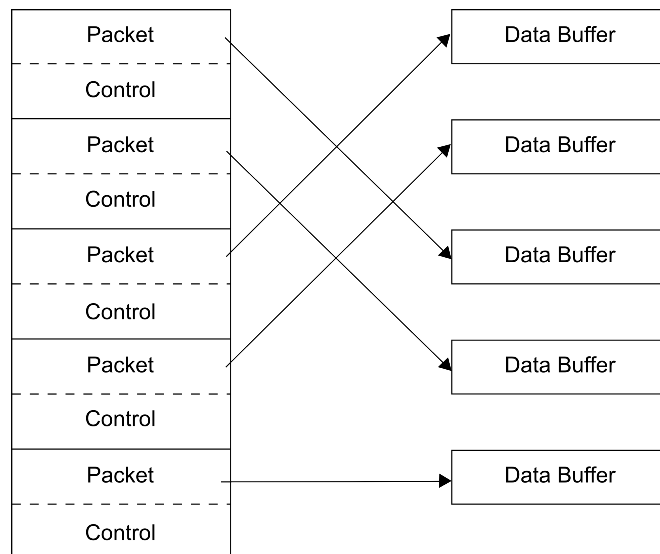

Abbildung 5.3: Mögliche Zuordnung Descriptor und Daten Buffer

5.5 Dokumentation

Um die Nutzung des HAL zu erleichtern, wurde mit Hilfe von Doxygen aus dem Quellcode der API eine Dokumentation generiert. Diese enthält umfangreiche Beschreibungen der einzelnen Funktionen, Erläuterungen zu den jeweiligen HAL-Komponenten sowie zu den definierten Datenstrukturen und Konstanten.

6 Ergebnisse

In diesem Kapitel geht es um die am Stack durchgeführten Tests und deren Auswertung. Mit Hilfe geeigneter Messverfahren werden die Eigenschaften des Systems nach Einführung des HAL in Bezug auf Funktion und Zeitverhalten überprüft und die Ergebnisse diskutiert. Des Weiteren werden die Ergebnissen der beiden Hardware-Plattformen gegenüber gestellt.

6.1 Zeitverhalten

Das Zeitverhalten eines Echtzeit Systems ist ein zentrales Kriterium für die Bewertung des Systems. Deshalb wurde mit Hilfe der unten erläuterten Messverfahren mögliche Auswirkungen des HAL auf das Zeitverhalten des Stacks untersucht. Des Weiteren werden die Ausführungszeiten der Implementierung auf dem LPCXpressoBoard ermittelt und mit den Ergebnisse des NXHX500 verglichen, um das Zeitverhalten der Implementierung einzuordnen.

Messaufbau

Für die Messung wurde ein minimales Netzwerk bestehend aus zwei Netzwerkknoten aufgebaut (Vgl. Abbildung 6.1). Des Weiteren wird ein Ethernet-fähiges Oszilloskop und eine passive Netzwerk-Tap verwendet. Die beiden Netzwerkknoten sind über den Netzwerk-Tap miteinander verbunden. Bei dem Netzwerk-Tap handelt es sich um eine Eigenbau-Tap, welche mit Hilfe eines Patchfeldes realisiert wurde. Mit dem Tastkopf (Probe) des Oszilloskops wird das Signal direkt an den Kontakten des Patchfeldes abgegriffen. So erhält man passiven Zugriff auf den Netzwerkverkehr, was mit Hilfe des Oszilloskops die Analyse und Darstellung einzelner Frames ermöglicht. Ein weiteres genutztes Hilfsmittel sind General Purpose IOs (GPIO). Diese ermöglichen es, über einen Pegelwechsel die aktuell ausgeführte Stelle im Code nach außen sichtbar zu machen. Dafür wird an der entsprechenden Stelle im Code der Pegel des GPIO-Pins geändert. Der Pegelwechsel kann dann mit Hilfe des Oszilloskops abgetastet und dargestellt werden.

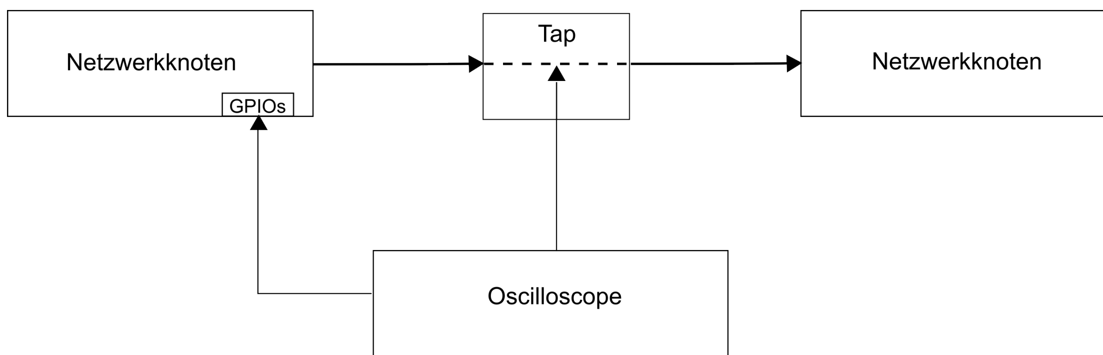

Abbildung 6.1: Messaufbau Zeitverhalten

Systemspezifische Delays

Die systemspezifischen Delays sind statische Verzögerungszeiten des Systems, welche bei der Konfiguration des Stacks angegeben werden müssen. Bei der Portierung des Stacks muss überprüft werden, ob eine Anpassung der Werte nötig ist, da die Nutzung einer anderen Hardwareplattform die Ausführungszeiten des Stacks verändern kann.
Damit Nachrichten und Tasks zu bestimmten Zeitpunkten verschickt bzw. ausgeführt werden können, nutzt der TTEthernet-Stack einen Scheduler, welcher die Ausführung veranlasst. Zwischen der Aktivierung durch den Scheduler und der tatsächlichen Ausführung vergeht eine bestimmte Zeit. Um eine möglichst schnelle Ausführungszeit zu gewährleisten, wurden Teile des HALs in den Tightly Coupled Memory (TCM) des NETX500 verlagert. Ein Vorteil des TCM ist, dass auf diesen im Gegensatz zum restlichen Speicher ohne Wait-States zugegriffen wird, was Jitter in den Ausführungszeiten verursachen kann. In der Tabelle 6.1 sind die gemessenen Verzögerungszeiten in Abhängigkeit der genutzten Hardware dargestellt. Um den möglichen Einfluss des HAL auf die Verzögerungszeiten zu bestimmen, wurden die Messungen für den NXHX500 jeweils ohne und mit implementierten HAL durchgeführt.

Für den Delay "Execute Task" wurde die Zeit bestimmt, welche zwischen erreichen des Zeitpunktes im Schedule und der eigentlichen Ausführung der Time-Triggered Task vergeht. Damit die Task genau zum geplanten Zeitpunkt ausgeführt wird, wird die Verzögerungszeit beim Eintragen in den Schedule verrechnet und damit die Ausführung der Task um den Delay nach vorn verschoben.
Der Delay "Send TT-Message" bezeichnet die Zeit, welche zwischen dem Auslösen des Sendevorgangs durch den Scheduler und dem Verlassen des Frames über den Ethernet-Port vergeht. Durch die Verrechnung des Delays kann die Sendezeit so verschoben werden, dass die TT-Nachricht

genau zum geplanten Zeitpunkt verschickt wird. Dies ist für das Senden von TT-Nachrichten essentiell, da diese bei einer verspäteten Ankunft das konfigurierte Empfangsfenster verpassen und verworfen werden.
Beim Delay "Send Sync-Frame" handelt es sich um die Verzögerungszeit zwischen dem vom Scheduler gestarteten Versenden eines PC-Frames und dem Verlassen des Frames über den Ethernet-Port. Der ermittelte Wert dient dazu, die im PC-Frame enthaltene Zeitbasis anzupassen, damit diese der tatsächlichen Systemzeit zum Sendezeitpunkt entspricht und eine möglichst genaue Synchronisation der restlichen Netzwerkteilnehmer ermöglicht.

Tabelle 6.1: Systemspezifische Delays

Hardwareplattform	Execute Task	Send TT-Message	Send Sync-Frame(PCF)
NXHX500	8,2µs	11,1µs	14,9µs
NXHX500 + HAL	8,1µs	11,3µs	14,3µs
LPCXpressoBoard	7,8µs	10,4µs	12,3µs

Auswertung

Die ermittelten Werte zeigen, dass der HAL nur minimalen Einfluss auf die Verzögerungszeiten des Systems hat. Der Delay "Execute Task" und der Delay "Sync-Frame" sind mit Nutzung des HAL 0,1µs, bzw. 0,6µs kürzer. Der Delay "Send TT-Message" ist mit eingeführten HAL 0,2µs länger. Diese Unterschiede sind für die Nutzung in einem 100 Mbit/s Netzwerk vernachlässigbar gering. Dass die Einführung des HAL trotz der erhöhten Anzahl von Funktionsaufrufen teilweise sogar positiven Einfluss auf die Ausführungszeiten hatte, könnte an der Auslagerung der HAL-Module in den TCM liegen. Dadurch können die Zugriffe auf die Hardware schnellstmöglich ausgeführt werden. Bei der ursprünglichen Version des Stacks konnten aufgrund der begrenzten Größe nur einige Module des Stacks in den TCM ausgelagert werden. Die neue Aufteilung des Speichers, insbesondere die Auslagerung der Hardwarezugriffe in den TCM, scheint sich somit positiv auf die Ausführungszeiten aus zu wirken. Eine genauere Untersuchung zu den Ausführungszeiten auf dem NXHX500 wurde nicht vorgenommen.
Die bei der Ausführung des Stacks auf dem LPCXpressoBoard gemessenen Werte zeigen, dass die Verzögerungszeiten in Abhängigkeit von der Hardware angepasst werden müssen. Die Delays für das LPCXpressoBoard liegen zwischen 0,9µs und 2,6µs unter den Vergleichswerten, welche auf dem NXHX500 mit HAL Implementierung gemessen wurden. Ein möglicher Grund für die schnellere Ausführung trotz der schwächeren Hardware des LPCXpressoBoards (Vgl. Abschnitt 3.2) könnte die performantere Interruptverarbeitung des Cortex-M3 Prozessors sein.

Beim Auftreten eines Interrupts muss zum einen der Inhalt einiger Register auf dem Stack gesichert werden zum anderen muss die Startadresse der ISR aus der Vektortabelle ausgelesen werden. Der Vorteil des Cortex-M3 ist, dass die Sicherung der Register auf dem Stack und das holen der ISR-Startadresse über zwei separate Bus-Schnittstellen geschieht und deshalb zur gleichen Zeit ausgeführt werden kann. Dies ermöglicht eine besonders schnelle Verarbeitung der Interrupts (Vgl. (Yiu, 2010, S.145 f.)). Eine genauere Untersuchung zu den Ausführungszeiten auf dem LPCXpressoBoard wurde nicht vorgenommen.

Software-Zeitstempel LPCXpressoBoard

Damit der Software-Zeitstempel genaue Ergebnisse liefert, muss wie in Abschnitt 5.2 besprochen ein konstanter Delay zur Korrektur des Zeitstempels bestimmt werden. Zur Bestimmung des Delays muss die Differenz zwischen Ankunft des Frames an der Schnittstelle und speichern des Zeitstempels möglichst genau erfasst werden.
Für die Messung ist der eine Knoten so konfiguriert, dass dieser konstant alle 5ms eine Nachricht verschickt. Mit Hilfe des zweiten Knotens wird die Messung durchgeführt. Mit dem Oszilloskop wird der Zeitpunkt für die Ankunft des Frames bestimmt (Vgl. Abbildung 6.2, links Messpunkt b). Um den Zeitpunkt des Zeitstempels nach außen zu führen, wurde ein GPIO genutzt. Über den Pegelwechsel des GPIOs wird der Zeitpunkt des Zeitstempels signalisiert und kann vom Oszilloskop erfasst werden (Vgl. Abbildung 6.2, rechts Messpunkt a). Die Messung wurde unter optimalen Bedingungen durchgeführt. Das bedeutet der Stack wurde so konfiguriert, dass neben dem Empfang der Frames keine weiteren Tasks ausgeführt werden, welche die Verarbeitungsgeschwindigkeit beeinflussen könnten. Die Messung ergab einen konstanten Delay von 14,75µs. Wie in Abschnitt 5.2 bereits erwähnt, kann es durch Unterbrechungen der Empfangs-ISR zu Schwankungen des Delays kommen. Um den maximalen Jitter zu bestimmen wurde die Ausführungszeit der unterbrechenden ISR gemessen. Die ermittelte Ausführungszeit beträgt 4,4µs. Damit entspricht der Delay 14,75 + 4,4 µs.

6.2 Funktionstest

Um zu überprüfen, ob nach der Einführung des HAL die Funktion des TTEthernet-Stacks erhalten geblieben ist sowie zur Verifikation der HAL Funktionalität, wurden verschiedene Tests durchgeführt, welche im Folgenden erläutert werden.

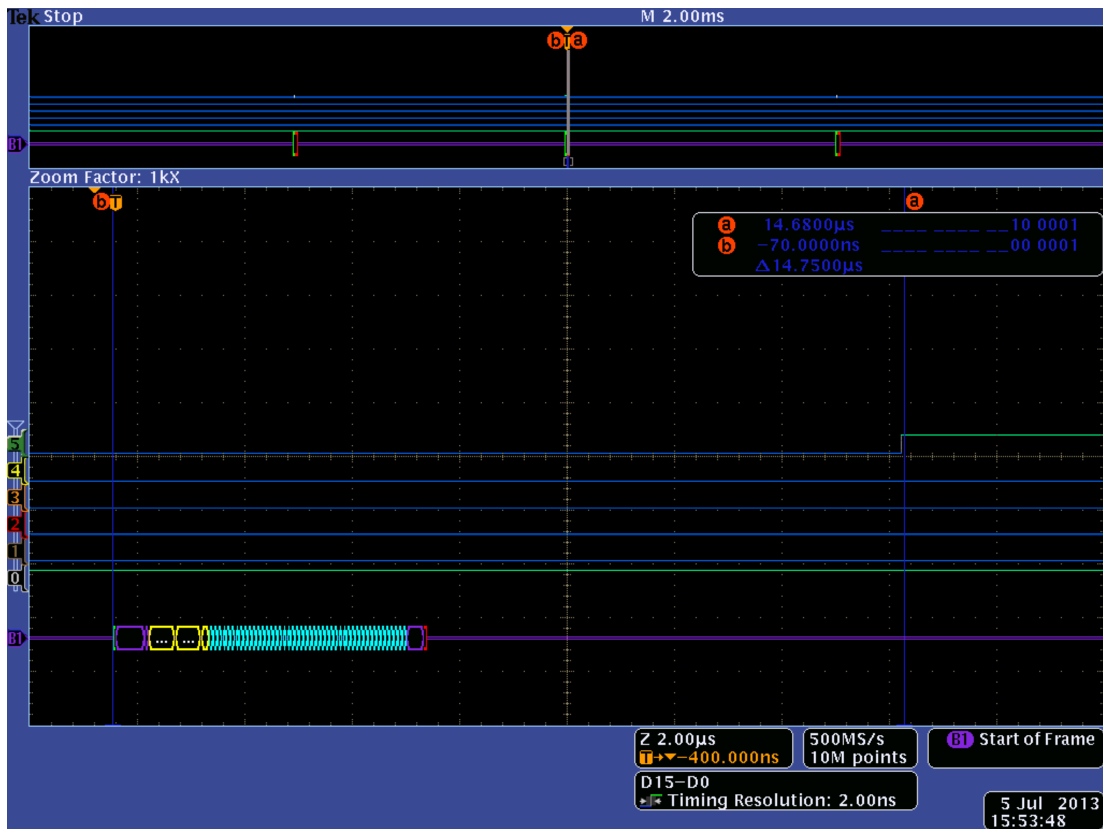

Abbildung 6.2: Delay Software-Zeitstempel LPCXpressoBoard

Komponenten

Um während der Umsetzung des HALs die Grundfunktion der einzelnen Komponenten sicher zu stellen, wurden diese entsprechend getestet. Die Komponenten LED, GPIO und die serielle Schnittstelle wurden mit Hilfe von einfachen Funktionstests überprüft.

Nach der Integration der Interrupt- sowie der Timer-Komponente wurde auch deren Funktion entsprechend überprüft. Da das Scheduling von Tasks auf der Timer- und Interrupt-Komponente aufbaut, wurde zur Überprüfung der Komponenten eine Task im Schedule definiert, welche zyklisch ausgeführt wird. Um die Ausführung der Task nach außen sichtbar zu machen, wird von der Task ein GPIO-Pin gesetzt, welcher mit dem Oszilloskop abgegriffen werden kann. Wird die Task zyklisch zu den im Scheduler angegebenen Zeiten ausgeführt, funktionieren Timer- und Interrupt-Modul korrekt.

Für den Test der Ethernet-Komponente wurde ein Messaufbau ähnlich dem in Abschnitt 6.1 beschriebenen genutzt. Zur Überprüfung wurde der Stack so konfiguriert, dass dieser zyklisch

Nachrichten verschickt und anschließend mit dem Oszilloskop das erfolgreiche Senden der Pakete überprüft. Das Empfangen von Nachrichten wurde mit einem GPIO, welcher in der Empfangs-Routine gesetzt wird, überprüft.
Diese Tests wurden nach der Umsetzung des HALs auf dem NXHX500 durchgeführt, sowie für die Implementierung auf dem LPCXpressoBoard.

Anwendungstest NXHX500

Im Rahmen des CoRE-Projekts ist ein Demonstrator entwickelt worden, welcher ein Prototyp für ein TTEthernet basierendes Backbone-Netzwerk im Auto ist (Vgl. CoRE (a)). Der Demonstrator zeigt einige Anwendungen im Auto, welche bereits mit Hilfe von TTEthernet realisiert werden können. Neben der Steer-by-Wire-Anwendung und einem Infotainment System ist auch eine Steuerung von Scheinwerfern und Blinkern möglich. Das Kernstück jeder Anwendung bildet der auf dem NXHX500 ausgeführte TTEthernet-Stack. Dieser ermöglicht es, die einzelnen Komponenten über TTEthernet zu steuern. Um die Funktion des kompletten Stacks nach der Einführung des HALs zu überprüfen, wurde dieser für den Test mit der Scheinwerfer-Anwendung in das Netzwerk integriert.
Für die Steuerung des Scheinwerfers ist der NXHX500 mit der Steuereinheit des Scheinwerfers verbunden. Die Anwendung kontrolliert den Scheinwerfer mit Hilfe von GPIOs und der Pulsweitenmodulation (PWM), wobei der Zugriff auf die Hardware direkt im Code über die entsprechenden Registerzugriffe realisiert wurde. Für die Integration musste die Scheinwerfer-Anwendung so angepasst werden, dass diese über den HAL auf die Hardware zugreift. Für die Erzeugung des PWM-Signals musste der HAL um eine Funktion für die PWM erweitert werden.
Für den Test wurde nur ein Teil des Demonstrator-Netzwerks aufgebaut, da die weiteren Komponenten nicht benötigt werden. Das Netzwerk besteht aus einem Scheinwerfer, einem Controller und dem Infotainment-System. Verbunden sind die Komponenten über zwei konfigurierbare TTEthernet-Switches. Der Aufbau des Netzwerks ist in Abbildung 6.3 zu sehen, wobei nicht genutzte Netzwerkkomponenten ausgegraut wurden. Der Scheinwerfer wird mit Hilfe der für den HAL angepassten Anwendung betrieben. Bei den anderen Komponenten handelt es sich um die für den Demonstrator genutzte Hardware. Eine Anpassung der Netzwerk- und Gerätekonfiguration war nicht notwendig, da die bestehende Konfiguration des Demonstrator-Netzwerks genutzt wird.
Der Scheinwerfer wird über das Bedienfeld des Infotainment-Systems gesteuert und unterstützt mehrere Funktionen wie Blinker und eine Regulierung der Helligkeit. Für den Test wurden alle verfügbaren Funktionen des Scheinwerfers überprüft. Dabei wurden alle Funktionen mit dem er-

warteten Ergebnis ausgeführt, womit die korrekte Funktion des TTEthernet-Stacks nachgewiesen ist.

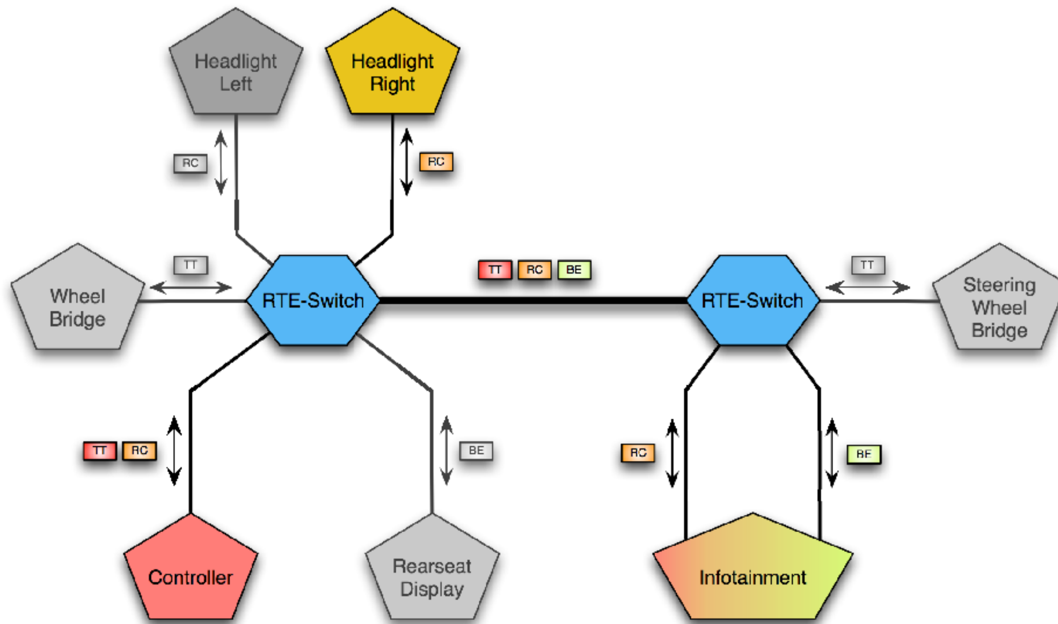

Abbildung 6.3: Demonstrator Netzwerk (Quelle: Florian Bartols)

Anwendungstest LPCXpressoBoard

Um die erfolgreiche Übertragung des Stacks auf das LPCXpressoBoard zu überprüfen, wurde auch die Scheinwerfer-Anwendung auf den Microcontroller portiert. Da die Anwendung für den Test des NXHX500 bereits an die Nutzung des HALs angepasst wurde, mussten lediglich die fehlende PWM-Funktion für das LPCXpressoBoard implementiert und die GPIO-Pins zugewiesen werden.
Der Versuchsaufbau und Konfiguration sind identisch mit den beim Anwendungstest für den NX-HX500 (Vgl. Abbildung 6.3). Der einzige Unterschied ist, dass die Anwendung für die Steuerung der Scheinwerfer auf dem LPCXpressoBoard ausgeführt wird.
Für den Test wurden alle verfügbaren Funktionen des Scheinwerfers überprüft. Alle Funktionen wurden mit dem erwarteten Ergebnis ausgeführt. Somit kann davon ausgegangen werden, dass die Portierung des TTEthernet-Stacks erfolgreich war und der Stack korrekt auf dem LPCXpressoBoard ausgeführt wird.

Synchronisation NXHX500

Wie in Abschnitt 2.2 beschrieben, müssen sich die Teilnehmer in einem TTEthernet-Netzwerk auf eine gemeinsame Zeitbasis synchronisieren. Die Synchronisation des Stacks muss möglichst genau sein, daher werden beim Synchronisationsprozess einige statische Delays verrechnet. Dies ermöglicht, es die hardwarespezifischen Verarbeitungszeiten mit einzubeziehen. Zum Beispiel muss die Verzögerungszeit zwischen Ankunft des PC-Frames und der tatsächlichen Verarbeitung ausgeglichen werden. Da die Einführung des HALs Einfluss auf die Verarbeitungszeit des Stacks hat, wurde überprüft ob dies auch Auswirkungen auf den Synchronisationsmechanismus hat.
Für die Synchronisation des Netzwerks muss ein Netzwerkteilnehmer die Rolle des Synchronisations-Masters übernehmen, welcher die Zeitbasis für die restlichen Netzwerkteilnehmer vorgibt. Für den Test wurde der gleiche Aufbau wie für den Anwendungstest genutzt (Vgl. Abbildung 6.3). In dem Aufbau übernimmt der Controller die Rolle des Synchronisations-Masters. Die Scheinwerfer-Anwendungen dient als Testkomponente und ist als Synchronisations-Client konfiguriert. Der Stack besitzt eine Funktion, welche die Genauigkeit der Synchronisation bestimmt und diese über ein LED-Feld ausgibt. Dabei wird die Genauigkeit in vier Stufen zwischen 1µs und 8µs unterteilt. So kann für den Test mit Hilfe der LEDs die Güte der Synchronisation festgestellt werden. In den durchgeführten Testläufen gelang es dem Synchronisations-Client immer sich auf die gemeinsame Zeitbasis zu synchronisieren. Die Genauigkeit der Synchronisation lag dabei zwischen 1µs und 2µs. Somit scheint die Einführung des HAL keine signifikanten Auswirkungen auf den Synchronisationsprozess zu haben.

Synchronisation LPCXpressoBoard

Für korrekte Funktion des TTEthernet-Stacks ist es notwendig, dass der Synchronisationsmechanismus funktioniert. Daher wurde nach der Portierung des Stacks auf das LPCXpressoBoard überprüft ob die Synchronisation korrekt durchgeführt wird. Dafür wurde der gleiche Versuchsaufbau wie für die vorherigen Tests genutzt (Vgl. Abbildung 6.3).
Wie in Abschnitt 5.2 besprochen, besitzt der Timer des LPCXpressoBoards keine Rate-Correction und muss deshalb via Offset-Correction auf die Netzwerkzeit gesetzt werden. Bei der Überprüfung wurde festgestellt, dass die bisher vom Stack genutzte Funktion zur Offset-Correction die zu setzende Zeit falsch berechnet hat und daher angepasst werden muss. Nach einer Anpassung der Funktion wird die Synchronisation korrekt durchgeführt. Die durchschnittliche Genauigkeit lag dabei im einstelligen Mikrosekundenbereich.
Allerdings funktioniert die Funktion momentan nur mit der Einschränkung, dass die Synchronisation am Anfang des Zyklus durchgeführt werden muss. Um die Synchronisation auch zu anderen

Zeitpunkten durchführen zu können, muss ein anderer Algorithmus für die Offset-Correction entwickelt werden.

7 Zusammenfassung und Ausblick

Ziel dieser Arbeit war es, den TTEthernet-Stack durch die Einführung eines Hardware Abstraction Layers portierbar zu machen. In diesem letzten Kapitel werden abschließend die Ergebnisse der Arbeit zusammengefasst und es wird ein Ausblick auf mögliche weiterführende Arbeiten gegeben.

7.1 Zusammenfassung

Nach der Analyse der Stack-Funktionalität und der Einarbeitung in den Quellcode wurde zunächst ein Konzept entwickelt, um die Portabilität des Stacks umzusetzen. Da es sich bei dem TTEthernet-Stack um ein echtzeitfähiges System handelt, wurde bei der Konzeptionierung darauf geachtet, eine performante Lösung zu wählen, welche möglichst wenig Einfluss auf das Zeitverhalten hat. Wie die Ergebnisse der Tests zeigen, sind die Auswirkungen des HAL auf die Ausführungszeiten des Stacks minimal. So ist trotz der erhöhten Anzahl von Funktionsaufrufen kein negativer Einfluss auf das Zeitverhalten festzustellen. Teilweise konnten sogar kürzere Ausführungszeiten gemessen werden. Um die Funktion des Stacks zu überprüfen wurde außerdem eine der Stack-Anwendungen für die Nutzung des modifizierten Stack angepasst. Bei mehreren Testläufen konnten alle Funktionen der Anwendung ohne Fehler ausgeführt werden.
Anschließend wurde der Stack auf das LPCXpressoBoard übertragen, um die entstandene Portierbarkeit zu verifizieren. Durch die unterschiedlichen Architektur-Konzepte der beiden Microcontroller mussten für die Implementierung der HAL-Funktionen einige Anpassungen an der architekturspezifischen Nutzung der Hardware vorgenommen werden. Letztendlich konnten aber alle in der HAL-API definierten Funktionen korrekt implementiert werden. Auch für das LPCXpressoBoard wurde ein Funktionstest durchgeführt und dafür die ausgewählte Anwendung auf den neuen Microcontroller übertragen. Alle Funktionen konnten fehlerfrei ausgeführt werden und führten zu den erwarteten Ergebnissen.
Abschließend kann somit festgehalten werden, dass das Ziel der Hardware-Abstraktion bzw. die Portierbarkeit des Stacks auf Grundlage des in Kapitel 4 erarbeiteten Konzepts erfolgreich umgesetzt werden konnte.

7.2 Ausblick

Für die Nutzung des Stacks im Demonstrator wurde bisher nur eine der Anwendungen aus Testzwecken für den Hardwarezugriff über den HAL angepasst. Um die anderen Anwendungen mit dem modifizierten Stack nutzen zu können, müssen auch diese angepasst werden.
Die Portierung auf das LPCXpressoBoard konnte mit Hilfe der entwickelten HAL-API gut umgesetzt werden. Bei der Übertragung auf weitere Microcontroller kann es evtl. nötig sein, die Schnittstelle anzupassen. Durch die modulare Umsetzung des HAL ist dies ohne Probleme möglich. Auch eine spätere Erweiterung ist somit einfach umzusetzen.
In Abschnitt 2.4 wurde einige mögliche Lizenzen für die Veröffentlichung des TTEthernet-Stacks als Open Source Software vorgestellt. Das Ziel der Veröffentlichung als OSS, ist es anderen die Nutzung zu ermöglichen, aber auch selber von einer möglichen Weiterentwicklung des Stacks zu profitieren. Die Zielgruppe für den praktischen Einsatz des Stacks ist hauptsächlich die (Automobil)-Industrie. Daher sollte der TTEthernet-Stack unter einer Lizenz veröffentlicht werden, welche die Möglichkeit bietet, den Stack mit proprietärer Software zu verbinden. Eine mögliche Lizenz, welche diese Ansprüche erfüllt, ist die Lesser General Public License (LGPL) (Vgl. Abschnitt 2.4). Diese ermöglicht die Verbindung mit proprietärer Software, stellt aber dabei sicher, dass Änderungen am Stack selber veröffentlicht werden müssen.
Des Weiteren ist es nötig, neben dem Quellcode auch eine umfangreiche Dokumentation zur Verfügung zu stellen. Für den HAL wurde im Laufe dieser Arbeit eine Dokumentation erstellt. Diese muss vor der Veröffentlichung des Stacks um die Dokumentation der Stack-Funktionalität erweitert werden, um anderen die Nutzung des Stacks zu ermöglichen.

Literaturverzeichnis

[AUTOSAR Development Cooperation] AUTOSAR DEVELOPMENT COOPERATION: *AUTomotive Open System ARchitecture.* – URL http://www.autosar.org

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

[Canalys 2013] CANALYS: *Android powered a third of all mobile phones shipped in Q4 2012.* 2013. – http://www.canalys.com/newsroom/android-powered-third-all-mobile-phones-shipped-q4-2012

[CoRE a] CoRE: *Demonstrator.* – https://core.informatik.haw-hamburg.de/en/projects/demonstrator.html

[CoRE b] CoRE: *RECBAR.* – http://core.informatik.haw-hamburg.de/en/projects/recbar

[Ecker u. a. 2009] ECKER, Wolfgang ; MÜLLER, Wolfgang ; DÖMER, Rainer: *Hardware-dependent Software: Principles and Practice.* Springer, 2009

[Hilscher a] HILSCHER: *netX 100/500 – networX on chip.* – http://www.hilscher.com/files_datasheets/D_4700eb29d0e39_uk.pdf

[Hilscher b] HILSCHER: *NXHX 500-ETM.* – http://www.hilscher.com/products_details_hardware.html?p_id=P_461ff2053bad1&bs=20

[Hilscher: Gesellschaft für Systemautomation mbH 2008] Hilscher: Gesellschaft für Systemautomation mbH (Veranst.): *Technical Data Reference Guide netX 500/100.* 2008

[IEEE] IEEE: *IEEE 802.3:ETHERNET.* – http://standards.ieee.org/about/get/802/802.3.html

[Keßler 2013] KEßLER, Steffen: *Anpassung von Open-Source-Software in Anwenderunternehmen.* Springer, 2013

[Linder 2008] Linder, Timm: *Komponentenbasierte Eingebettete Systeme.* 2008

[Marcondes u. a. 2006] Marcondes, H. ; Hoeller, A.S. ; Wanner, L.F. ; Frohlich, A.A.M.: *Operating Systems Portability: 8 bits and beyond.* Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference. 2006

[Müller 2011] Müller, Kai: *Time-Triggered Ethernet für eingebettete Systeme: Design, Umsetzung und Validierung einer echtzeitfähigen Netzwerkstack-Architektur*, HAW Hamburg, Bachelorarbeit, 2011

[NXP] NXP: *LPCXpresso.* – `http://www.nxp.com/techzones/microcontrollers-techzone/tools-ecosystem/lpcxpresso.html`

[NXP Semiconductors 2010] NXP Semiconductors (Veranst.): *LPC17xx User manual.* 2010

[OSI] OSI: *Open Source Initiative.* – `http://opensource.org/`

[Plate 2013] Plate, Prof. J.: *Grundlagen Computernetze: Ethernet.* 2013. – `http://www.netzmafia.de/skripten/netze/netz4.html`

[Renner u. a. 2005] Renner, Thomas ; Vetter, Michael ; Rex, Sascha ; Kett, Holger: Open Source Software: Einsatzpotenziale und Wirtschaftlichkeit / Fraunhofer-Gesellschaft. 2005. – Forschungsbericht

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

[TTTech Computertechnik AG] TTTech Computertechnik AG: *TTEthernet: Deterministic Ethernet Network.* – `http://www.tttech.com/technologies/ttethernet/`

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

[VDIWissensforum/AudiAG] VDIWissensforum/AudiAG: *Bordnetz Audi A8.* – `http://www.pressebox.de/attachment/464371/Bordnetz_Quelle_VDI_Wissensforum_AUDI_AG_300_dpi.jpg`

[VMware] VMware: *Virtualization-Basics.* – `http://www.vmware.com/de/virtualization/virtualization-basics/virtual-machine.html`

[VW 2013] VW, Volkswagen AG: *Car-to-X: Ein Kommunikationssystem für viele Anwendungen.* 2013. – `http://www.volkswagenag.com/content/vwcorp/content/de/innovation/communication_and_networking/connected_world/car_to_x.html`

[Yiu 2010] Yiu, Yoseph: *The defnitive guide to the ARM Cortex-M3.* Newnes, 2010

Tabellenverzeichnis

3.1 LPCXPresso1769: Vergleich mit Anforderungen aus Abschnitt 3.1 15

5.1 Vektortabelle 30

6.1 Systemspezifische Delays 38

Tables

Abbildungsverzeichnis

1.1 Bordnetz Audi A8 (Quelle: VDIWissensforum/AudiAG) 2
1.2 Einordnung des HAL in Stack Architektur 3

2.1 Aufbau eines Ethernet Frame nach IEEE 802.3 (Vgl. IEEE) 6
2.2 Aufbau eines TTEthernet Frame 7
2.3 Softwarearchitektur des TTEthernet Stacks 8

3.1 Blockdiagramm NXHX500 (Quelle: Hilscher (a)) 13
3.2 Ausschnitt Blockdiagramm LPC1769 (Quelle: NXP Semiconductors (2010) 14

4.1 Architektur TTEthernet-Stack: Hardwarezugriffe der einzelnen Module (Vgl. Müller (2011)) 17
4.2 Architektur einer Virtuellen Maschine 18
4.3 Architektur System Call Interface 19
4.4 Architektur Komponentenbasierte Entwicklung 20
4.5 Architektur Hardware Abstraction Layer 21
4.6 Aufbau Hardware Abstraction Layer 22
4.7 Struktur HAL API 24
4.8 Architektur TTEthernet-Stack: Hardwarezugriffe nach Einführung des HAL . . 25

5.1 Frame Verwaltung NXHX500 32
5.2 Transmit Descriptor Speicher Layout (Quelle: NXP Semiconductors (2010)) . . . 34
5.3 Mögliche Zuordnung Descriptor und Daten Buffer 35

6.1 Messaufbau Zeitverhalten 37
6.2 Delay Software-Zeitstempel LPCXpressoBoard 40
6.3 Demonstrator Netzwerk (Quelle: Florian Bartols) 42

Figures

Glossar

CT Critical Traffic

Delay(Latenz) Verzögerung oder Verzögerungszeit

Jitter Schwankung

Nested Interrupts Die Interrupts haben unterschiedliche Prioritäten. Ein Interrupt mit höherer Priorität kann einen Interrupt mit niedriger Priorität unterbrechen

Netzwerk Tap Ein Gerät welches den passiven Zugriff auf den Datenverkehr in einem Netzwerk ermöglicht. Dadurch ist die Analyse des Datenverkehrs möglich, ohne diesen aktiv zu beeinflussen

PHY Physikalische Schnittstelle für den digitalen Zugriff auf den Übertragungskanal. Ist Teil der Ethernet-Schnittstelle.

PWM Pulsweitenmodulation: Modulationsart bei der eine technische Größe (z.B Strom) zwischen zwei Werten wechselt (an/aus). Das Verhältnis zwischen Ein- und Ausschaltzeit nennt man Tastverhältnis und bestimmt das erzeugte Signal (mittlere Ausgangsspannung)

Scheduler Ein Modul welches die zeitabhängige Ausführung von Tasks regelt

Task Aufgabe die vom TTEthernet-Stack ausgeführt wird

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

Hamburg, 15. August 2013 Flemming Bunzel