Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Jan Kamieth

Entwurf einer Mikrocontroller basierten Bridge zur Kopplung von CAN Bussen über Time Triggered Realtime Ethernet

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Jan Kamieth

Entwurf einer Mikrocontroller basierten Bridge zur Kopplung von CAN Bussen über Time Triggered Realtime Ethernet

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Franz Korf
Zweitgutachter: Prof. Dr. Andreas Meisel

Abgegeben am 24. August 2011

Jan Kamieth

Thema der Bachelorarbeit
Entwurf einer Mikrocontroller basierten Bridge zur Kopplung von CAN Bussen über Time Triggered Realtime Ethernet

Stichwörter
Mikrocontroller, Bridge, Brücke, CAN, Controller Area Network, Time Triggered, Realtime, Ethernet, TTEthernet

Kurzzusammenfassung
Diese Arbeit beschäftigt sich mit der Entwicklung und Implementation einer Bridge, mit der zwei oder mehrere CAN-Busse über ein Time Triggered Realtime Ethernet, welches auf der Spezifikation von TTTech basiert, verbunden werden können. Konzipiert wird die Bridge für einen Einsatz im Automobilbereich, um den Anforderungen moderner Fahrzeugtechniken, wie den X-by-Wire Systemen und Multimediaanwendungen, gerecht zu werden. Das Hauptaugenmerk beim Design wird auf die effektive Nutzung der Bandbreite und Echtzeitfähigkeit gelegt. Um diese Ziele zu erreichen, werden unterschiedliche Verfahren zu den Themen Arbitrierung, Routing, Nachrichtenklassen und Nutzung des Ethernet Payloads diskutiert. Das Ende der Arbeit befasst sich mit der Implementation und Verifikation der Bridge.

Title of the paper
Design of a microcontroller based bridge to connect CAN-busses over a Time Triggered Realtime Ethernet

Keywords
microcontroller, bridge, CAN, Controller Area Network, time triggered, real-time, ethernet, ttethernet

Abstract
This thesis deals with the design and implementation of a bridge, which connects two or more CAN-busses by a time triggered realtime ethernet, which bases on the specification from TTTech. The bridge is planned to be installed in an automotive field to match modern requirements of automotive engineerings like X-by-Wire systems and multimedia applications. Essential points of the design are a effective usage of the bandwidth and real-time capability. To accomplish these goals different methods will be discussed in the topics allocation, routing, message classes and useage of the ethernet payload. The end of this paper deals with the implementation and verification of the bridge.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Zielsetzung und Abgrenzung 2

1.3 Inhaltlicher Aufbau der Arbeit 2

1.4 Verwandte Arbeiten 3

1.4.1 Time-Triggered Ethernet für eingebettete Systeme: Design, Umsetzung und Validierung einer echtzeitfähigen Netzwerkstack-Architektur 3

1.4.2 Mikrocontroller und CAN-basierte verteilte Regelung einer Steer-by-Wire Lenkung mit harten Echtzeitanforderungen 3

2 Grundlagen **4**

2.1 Time-Triggered Ethernet 4

2.1.1 Nachrichtenklassen 5

2.1.2 Konfiguration und Scheduling 6

2.1.3 Synchronisation 7

2.2 Controller Area Network 8

2.2.1 Nachrichten-Aufbau 8

2.2.2 Arbitrierung, Priorität 9

2.2.3 Zeitverhalten 10

2.3 Bridge Design 12

2.3.1 CAN basierte Lösung 12

2.3.2 Ethernet basierte Lösung 13

3 Anforderungen **15**

4 Konzeption Bridge **16**

4.1 Arbitrierung 16

4.2 Datenkapselung 18

4.3 Routing 18

4.3.1 Verfahren 1: Fluten 19

4.3.2 Verfahren 2: Selektives Fluten 21

4.3.3 Verfahren 3: Routing Tabellen . . . 22
4.3.4 Verfahren 4: Vereinfachte Routing Tabellen . . . 23
4.4 Wahl der Nachrichtenklassen . . . 24
4.4.1 RC- und TT-Nachrichten . . . 24
4.4.2 BE-Nachrichten . . . 25
4.5 Message Stuffing . . . 25

5 Umsetzung des Bridgedesigns und Implementierung 27
5.1 Mikrocontroller . . . 27
5.1.1 Übersicht . . . 27
5.1.2 Ethernet Modul . . . 29
5.1.3 CAN Modul . . . 30
5.1.4 Virtueller CAN-Stack . . . 32
5.2 Implementiertes Bridgedesign . . . 32
5.2.1 Arbitrierung . . . 33
5.2.2 Datenkapselung . . . 33
5.2.3 Routing . . . 33
5.2.4 Nachrichtenklassen . . . 34
5.2.5 Message Stuffing . . . 34
5.3 Implementation . . . 34
5.3.1 Bridge Modulbeschreibung . . . 34
5.3.2 Virtueller CAN-Stack . . . 38

6 Integration 40
6.1 Aufbau des Demonstrators . . . 40
6.2 Konfiguration . . . 41

7 Verifikation und Zeitmessung 43
7.1 Verifikation . . . 43
7.2 Zeitmessung . . . 46

8 Fazit 49

Literaturverzeichnis 51

Tabellenverzeichnis

2.1 CAN Nettodatenraten bei 1 Mbit/s 11

4.1 Benötigte Bandbreite bei bitweiser Arbitrierung 17
4.2 Routing Tabelle für das 3. Verfahren 22
4.3 Routing Tabelle für das 4. Verfahren 23
4.4 Verwendung des Ethernet Payloads bei Message Stuffing 26

5.1 CAN Modul: FIFO Register 30

Abbildungsverzeichnis

2.1 Aufbau des TTE Frames 6
2.2 Aufbau des CAN Frames 8
2.3 CAN-Arbitrierung 10
2.4 Reine CAN Architektur 12
2.5 CAN Ethernet Architektur 14

4.1 Topologie des diskutierten Systems 19
4.2 Nachrichtenfluss beim Fluten 20
4.3 Nachrichtenfluss beim selektiven Fluten 22

5.1 Architektur und Schnittstellen des NXHX 500 28
5.2 Aufbau des Ethernet Payload bei gekapselten CAN-Daten 33
5.3 Modulaufbau 36
5.4 Programmablauf 38

6.1 Aufbau des Demonstrators 40
6.2 Virtuelle Links im Demonstrator-System 41

7.1 Aufbau des Testsystems 43
7.2 Ausschnitt CANalyzer 44
7.3 Ausschnitt Wireshark 45
7.4 Ausschnitt Wireshark 46
7.5 Ausschnitt CANalyzer 46
7.6 Zeitmessung der CAN Routine 47
7.7 Zeitmessung der TT-Ethernet Routine 48

Kapitel 1

Einleitung

1.1 Motivation

In den letzten Jahrzehnten hat der Einsatz von elektronischen Komponenten im Automobilbereich einen starken Wandel erfahren. In modernen Fahrzeugen können die Kosten für elektronische Bauteile über 20% der gesamten Herstellungskosten betragen und Experten gehen davon aus, dass in Zukunft 80% der kommenden Innovationen im Automobil auf der Weiterentwicklung von elektronischen Systemen beruhen (vgl. Leen und Heffernan (2002)). Dies sind heutzutage komplexe verteilte Systeme (vgl. Tanenbaum (2003b)) im Fahrzeug, die verschiedene Anforderungen an die Vernetzung stellen. Das Kommunikationsprotokoll muss offen sein, damit die elektronischen Subsysteme der unterschiedlichen Zulieferer miteinander interagieren können. Subsysteme, die herkömmliche und bewährte mechanische Systeme ersetzen, wie zum Beispiel Steer-by-Wire oder Brake-by-Wire (vgl. Seidel (2009)), müssen mindestens so zuverlässig sein, wie die Systeme, die sie ersetzen.
Zusammen erreichen diese Subsysteme eine Summe von bis zu 70 ECUs (Electronic Control Units). Diese Steuergeräte tauschen untereinander im Betrieb mehr als 2500 verschiedene Variablen und Signale aus. Die Automobilindustrie hat deswegen in der Vergangenheit versucht, in Zusammenarbeit Lösungen für diese steigenden Anforderungen an die interne Kommunikationsstruktur zu finden (vgl. Thomas Noltet and Hans Hanssont and Lucia Lo Bello (2006)).
Das in den Anfängen verwendete Verfahren, sämtliche Komponenten direkt miteinander zu verdrahten, hat die Konstrukteure sehr bald an Grenzen stoßen lassen. Nicht zuletzt ist das steigende Gewicht und der Platzbedarf ein großes Problem bei dieser Vernetzungstechnik. 1955 hatte ein Automobil eine Verkabelung von 45 Metern, die heute auf bis zu 4 Kilometer angewachsen ist. Dabei führen 50 kg an zusätzlichen Kabelbäumen im Fahrzeug zu einem erhöhten Kraftstoffverbrauch von ca. 0,2 Litern auf 100 Kilometern (vgl. Leen und Heffernan (2002)).
Um diesem steigendem Bedarf an Punkt zu Punkt Verkabelung entgegenzuwirken, werden

seit den frühen 80er Jahre Kommunikationsnetzwerke eingesetzt, die auf seriellen Protokollen aufbauen. Ein Beispiel für den Erfolg dieser Netzwerke ist eine Pressemitteilung von Motorola aus dem Jahre 1998, in der bekannt gegeben wurde, dass durch den Einsatz eines lokalen Netzwerks in den 4 Türen eines BMW Kraftfahrzeugs 15 Kilogramm an Verkabelung eingespart wurde.
Eines dieser Kommunikationsprotokolle ist das Controller Area Network Protokoll (vgl. Robert Bosch GmbH), dass Mitte der 80er Jahre von der Robert Bosch GmbH entwickelt wurde und heutzutage zu den meist verwendeten Protokollen im Automobilbereich gehört.
Seit der Entwicklung von CAN hat sich der Bedarf an elektronischen Systemen stark weiterentwickelt. Hinzugekommene Anwendungsgebiete sind zum Beispiel X-by-Wire, Multimedia und Infotainment, die weitere Ansprüche an das eingesetzte Kommunikationsprotokoll stellen und vom CAN Protokoll nicht mehr erfüllt werden können. Dazu gehören vor allen die steigende Bandbreite für Multimediasysteme oder Rückfahrkameras und ein Bedarf an leistungs- und echtzeitfähigen Protokollen für die X-by-Wire Technologien.
Eine Lösung für diese Systeme ist das Time-Triggered Protocol (TTP), dass seit den 90er Jahren von TTTech (vgl. TTTech Computertechnik AG) entwickelt wird. In diesem Rahmen wurde auch das Time-Triggered Ethernet (TTE) Protokoll (vgl. GE Fanuc Intelligent Platforms (2009)) entwickelt, dass den Ansprüchen an hohe Bandbreite und Echtzeitfähigkeit genügt.

1.2 Zielsetzung und Abgrenzung

Das Ziel dieser Arbeit ist es, dass Time-Triggered Ethernet Protokoll in bestehende CAN-Netzwerkstrukturen zu integrieren. Es wird eine Bridge auf der Basis eines Mikrocontrollers entwickelt, die es ermöglicht CAN-Nachrichten über das TT-Ethernet zu versenden und so mehrere CAN-Busse miteinander zu koppeln.
Um eine optimale Umsetzung zu gewährleisten werden die Aspekte des Bridgedesigns diskutiert und mehrere Verfahren zu den Themen Arbitrierung, Routing, Nachrichtenklassen und Scheduling vorgestellt.
Die gewählten Methoden und letztendliche Implementierung zielt besonders auf einen Einsatz im Steer-by-Wire Demonstrator ab, der von der CoRE Projektgruppe (Communication over Real-time Ethernet (CoRE)) entwickelt wird.

1.3 Inhaltlicher Aufbau der Arbeit

Im zweiten Kapitel werden die Grundlagen zum Controller Area Network und zum Time-Triggered Ethernet vermittelt, die benötigt werden, um eine Bridge zwischen diesen beiden Systemen zu realisieren.
Das dritte Kapitel fasst die Anforderungen an diese Bridge zusammen. Es folgt die Diskus-

sion zu diesen Anforderungen im vierten Kapitel. Es werden unterschiedliche Konzepte zu Themen wie Arbitrierung, Routing, Nachrichtenklassen und Message Stuffing vorgestellt, mit denen sich diese Anforderungen erfolgreich erfüllen lassen.
Im fünften Kapitel folgt die Implementierung der Bridge. Es wird der eingesetzte Mikrocontroller vorgestellt und eine Einführung in die verwendeten Software-Module gegeben, die für die Implementierung benötigt werden. Darauf folgt ein Abschnitt über das letztendlich implementierte Bridgekonzept und eine Begründung der getroffenen Designentscheidungen. Im darauf folgenden Abschnitt werden die implementierten Module beschrieben.
Das sechste Kapitel befasst sich kurz mit der Integration der Bridge in den Steer-by-Wire Demonstrator und gibt ein Beispiel für den Einsatz der Bridge.
Das vorletzte Kapitel befasst sich mit der Validierung der Bridge und gibt einen Überblick über die vorgenommenen Tests und Messungen.
Abgeschlossen wird diese Arbeit mit einem Fazit und einer Zusammenfassung über das erreichte und inwiefern die Weiterentwicklung der Bridge aussehen könnte.

1.4 Verwandte Arbeiten

Im Kontext dieser Arbeit werden noch mehrere Arbeiten entwickelt, die sich mit der Umsetzung des Steer-by-Wire Demonstrators befassen. Dazu gehört die Time-Triggered Ethernet Implementierung und eine ausführliche Beschreibung des Demonstrators.

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

Kai Müller beschäftigt sich in dieser Arbeit mit der Implementierung des Time-Triggered Ethernet Protokolls nach der Vorlage von TTTech. Dazu gehört die Synchronisation des Systems, das Scheduling und eine Umsetzung der verschiedenen Nachrichtenklassen. Seine Entwicklung stellt die Basis bereit, um die Bridge auf dem verwendeten Mikrocontroller umzusetzen (vgl. Kai Müller (2011)).

1.4.2 Mikrocontroller und CAN-basierte verteilte Regelung einer Steer-by-Wire Lenkung mit harten Echtzeitanforderungen

In dieser Arbeit beschreibt Vitalij Stepanov den Steer-by-Wire Demonstrator im Detail. Dazu gehört eine Übersicht über die Komponenten des Systems und die Nachrichten, die von der jeweiligen Hardware versendet werden. Es wird der entwickelte Regelkreis beschrieben und das dafür benötigte Scheduling der Nachrichten (vgl. Vitalij Stepanov (2011)).

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen vermittelt, die nötig sind um die Anforderungen und möglichen Umsetzungen einer Bridge zwischen CAN-Bussen über Time-Triggerd Realtime Ethernet zu diskutieren. Dabei wird zuerst auf die Eigenschaften von Time-Triggered Ethernet und auf die des CAN-Bus eingegangen. Im Abschnitt Bridgedesign werden mögliche Architekturen eines gekoppelten Systems vorgestellt.

2.1 Time-Triggered Ethernet

Das Time-Triggered Ethernet (TT-Ethernet) Protokoll ist eine Erweiterung des ursprünglichen Ethernet Standards, der in der IEEE-Norm 802.3 (vgl. Institute of Electrical and Electronics Engineers (2005)) spezifiziert ist. Ethernet entspricht im ISO/OSI-Referenzmodell dem Layer 1 (physikalische Schicht) und dem Layer 2 (Datensicherungsschicht) und ist das derzeit meistverbreitete Kommunikationsprotokoll im LAN (Local Area Network) Bereich (vgl. Tanenbaum (2003a)). Das Ethernet-Netz wird in einer Stern-Topologie aufgebaut, dessen Kern ein Switch bildet. Man spricht dann von einem Switched Ethernet. Zusätzlich wird im Vollduplexmodus übertragen, so dass jeder Teilnehmer gleichzeitig zum Switch senden und empfangen kann, der hierfür jeweils einen Sende- und Empfangspuffer zur Verfügung stellt. Der Switch nimmt die Nachricht auf dem Eingangsport des Senders entgegen und reicht die Nachricht an einen Ausgangsport des Empfängers weiter. Üblich sind dabei heute Übertragungsraten von 100 Mbit/s und 1000 Mbit/s. Durch Switch und Vollduplexmodus können beim Übertragen keine Kollisionen mehr auftreten. Aber es kann zu einer Sendeverzögerung im Switch kommen, wenn der Ausgangsport bereits belegt ist, so dass ein sofortiges Zustellen der Nachricht nicht sichergestellt werden kann. Da der Switch jede Nachricht gleich behandelt, eine sogenannte Best-Efford Übertragung, kann die Übertragung theoretisch beliebig lang verzögert werden. Ein Einsatz von Switched Ethernet als Bussystem in sicherheitskritischen Gebieten, wie zum Beispiel einer Motorsteuerung, ist damit nicht denkbar. Doch durch den steigenden Bedarf an Bandbreite im Automobilbereich, wird der Einsatz von

Switched Ethernet sehr interessant. Das Switched Ethernet Protokoll wird dafür um eine Echtzeitfähigkeit erweitert. Eine Umsetzung dafür ist Time-Triggered Ethernet von TTTech, die zeit- und eventgesteuerte Übertragung auf dem gleichen physikalischem Medium ermöglicht. Um diese Echtzeitfähigkeit zu gewährleisten, wird das Switched Ethernet Protokoll um verschiedene Nachrichtenklassen, Synchronisation und ein komplexes Scheduling erweitert. Diese Komponenten werden in den folgenden Abschnitten vorgestellt.

2.1.1 Nachrichtenklassen

TT-Ethernet definiert 3 Nachrichtenklassen, die unterschiedlich im System priorisiert werden.

Time-Triggered-Traffic (TT) wird für zeitkritischen Netzwerkverkehr verwendet. Er erlaubt deterministisch vorhersagbare Übertragungen und damit ein Minimum an Latenz und Jitter. TT-Nachrichten haben die höchste Priorität im System und werden über einen Scheduler vorkonfiguriert und über diesen versendet.

Rate-Constrained-Traffic (RC) ist ein ereignisbasierter Nachrichtentyp, der auf dem AFDX (Avionics Full Duplex Switched (X) Ethernet) Standard basiert (vgl. Aeronautical Radio Incorporated (2009)). RC-Nachrichten unterliegen keinem Scheduling und haben eine niedrigere Priorität als TT-Traffic. Sie werden daher im Switch von TT-Nachrichten verdrängt.

Best-Efford-Traffic (BE) entspricht dem Standard Ethernet Netzwerkverkehr und hat die niedrigste Priorität im System. Die Nachrichten werden mit der noch zur Verfügung stehenden Bandbreite versendet.

Der TT-Ethernet Frame basiert auf dem Standard Ethernet Frame, wie er unter der IEEE-Norm 802.3 beschrieben ist. Um die verschiedenen Nachrichtenklassen im Switch, der sich um die Verteilung der Nachrichten kümmert, zu identifizieren, wird das 48 Bit große Zieladressfeld Ethernet Frame anders interpretiert. 32 Bit werden als Markierung für TT-Nachrichten verwendet und 12 Bit, um der Nachricht einen Identifier zu geben. Mit diesem ID-Feld ist es möglich 4096 verschiedene TT-Nachrichten im System zu erstellen. Das Zieladressfeld wird beim Empfang im Switch mit der Bitmaske 0xFFFFFFFF0000 maskiert. Wird dabei die Markierung einer TT-Nachricht erkannt, entscheidet der Switch anhand der TT-ID, ob es sich um eine TT-Nachricht oder um eine RC-Nachricht handelt. BE-Nachrichten werden komplett ohne Modifikation des Standard-Ethernet-Frames verwendet. Auch das Typ-Feld wird ohne Modifikation verwendet. Für zeitkritischen Verkehr enthält es aber eine besondere Kennung, nämlich das durch die IEEE spezifizierte Bitmuster 0x88D7.
Die folgende Grafik stellt den Aufbau eines TT-Ethernet Frames dar und zeigt unter anderem auch die veränderte Nutzung des Zieladressfeldes.

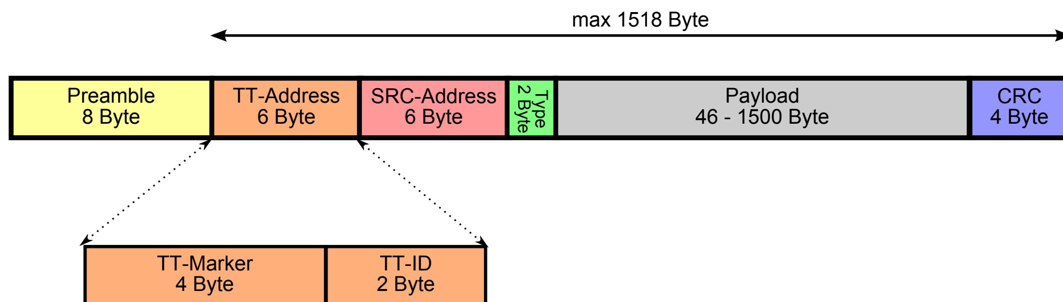

Abbildung 2.1: Aufbau des TTE Frames

2.1.2 Konfiguration und Scheduling

Das Versenden der TT- und RC-Nachrichten erfolgt über sogenannte virtuelle Links, die zur Designzeit konfiguriert werden. Jeder virtuelle Link steht dabei für eine logische Verbindung zwischen einem Sender und einem oder mehreren Empfängern. Über diese Links werden die Nachrichten versendet.
Das Scheduling wird zur Designzeit des Systems festgelegt und baut auf einem zeitlichem Zyklus auf. In diesem Zyklus muss sichergestellt werden, dass für die dort auftretenden Nachrichten genug Bandbreite zur Verfügung steht.

- Für TT-Nachrichten wird im Schedule ein fester Zeitpunkt definiert, an dem die Nachricht im Zyklus versendet wird. Dieser Zyklus muss so geplant werden, dass nie 2 Nachrichten zur gleichen Zeit an den selben Empfänger gesendet werden.

- RC-Nachrichten werden über Bandwidth-Allocation-Gaps (BAGs) konfiguriert. Die RC-Nachrichten sind nicht an den Zyklus gebunden, sondern erhalten einen BAG Wert, der festlegt, wie oft die Nachricht versendet werden darf, zum Beispiel alle 100 µs. So kann man festlegen, wieviel Bandbreite durch diese Nachricht verwendet werden darf. Die RC-Nachrichten werden dabei innerhalb ihrer Prioritätsklasse nach dem FIFO-Prinzip behandelt. Das heißt, die Nachricht, die zuerst vom Switch empfangen wird, wird auch zuerst wieder versendet. Beim Planen dieser Nachrichten muss sicher gestellt werden, dass im Schedule genügend BAGs vorhanden sind, damit die Nachrichten innerhalb des Zyklus gesendet werden.

- BE-Nachrichten unterliegen keinem Schedule und werden vom Switch wie normale Ethernet Nachrichten behandelt, die bei noch verfügbarer Bandbreite versendet werden. Die Laufzeiten sind dabei nicht deterministisch, so dass nicht sicher gestellt werden kann, in welchem Zeitrahmen die Nachrichten beim Empänger ankommen.

2.1.3 Synchronisation

Um ein systemweites Scheduling zu ermöglichen, müssen sämtliche Teilnehmer ihre Systemzeit angleichen, damit ein zeitlich korrekter Ablauf der Aufgaben möglich wird. Da eingebaute Taktgeber nie hundertprozentig synchron arbeiten, laufen die Systemzeiten nach einer gewissen Zeitspanne wieder auseinander. Eine fortlaufende Synchronisation ist damit unabdinglich.
TT-Ethernet sieht dafür eine fehlertolerante Zwei-Wege-Synchronisation vor, die durch eine Master-/Slave Architektur erreicht wird. Die Teilnehmer nehmen dafür unterschiedliche Rollen im Netzwerk an.

Synchronisation-Master (SM) initialisieren die Synchronisation. Dafür versenden sie einen Synchronisations-Frame, einen sogenannten Protocol-Control-Frame (PCF). Dieser enthält die aktuelle Systemzeit des Masters.

Compression-Master (CM) empfangen die gesendete Zeit von jedem SM und berechnen anhand dieser eine neue globale Systemzeit, die sie dann wieder an alle Clients senden.

Synchronisation-Client (SC) aktualisieren nach dem Erhalt der neuen globalen Systemzeit vom Compression-Master ihre lokalen Systemzeiten.

Für die Synchronisation senden zunächst alle Synchronisations-Master ihre lokale Systemzeit mittels eines PCF-Frames an den Compression-Master. Dieser berechnet anhand dieser Zeiten die neue globale Systemzeit und sendet diese an die Synchronisations-Clients. Im Netzwerk können dabei mehrere Synchronisations-Master definiert werden, damit auch bei einem Ausfall die Synchronisation noch möglich ist.

2.2 Controller Area Network

Das Controller Area Network (CAN) ist ein Bussystem, dass 1983 von Bosch entwickelt wurde (vgl. Robert Bosch GmbH) und zusammen mit Intel 1987 vorgestellt wurde. Es ist das am meisten verbreitete Protokoll im Automobilbereich, dessen Anwendungsbereich von der Steuerung der Spiegelverstellung bis hin zu der Kommuniktion zwischen Motor- und Getriebesteuerung reicht. CAN ist in der ISO 11898 (vgl. (International Organization for Standardization, 2003, Controller Area Network)) Familie international standardisiert und entspricht im ISO/OSI-Referenzmodell dem Layer 1 (physikalische Schicht) und dem Layer 2 (Datensicherungsschicht). Der CAN-Bus wird über zwei miteinander verdrillte Adern zu einer Linienstruktur aufgebaut, an dessen Enden sich jeweils ein Abschlusswiderstand befindet. Die maximale Übertragungsrate hängt von der Spezifikation ab. Beim 'Low Speed'-CAN (ISO 11898-3) erreicht der Bus eine Rate von 125 kbit/s bei einer Leitungslänge von 500 m. Der 'High Speed'-CAN (ISO 11898-2) erreicht maximal 1 Mbit/s bei 40 m Buslänge. Als Übertragungsverfahren kommt CSMA/CD + AMP (Carrier Sense Multiple Access / Collision Detection + Arbitration on Message Priority) zum Einsatz. Das bedeutet, dass der Bus ein Multimaster System ist, bei dem alle Teilnehmer senden können und die dabei auftretenden Kollisionen durch Bitarbitrierung verhindert werden. Durch dieses Verfahren sendet immer nur der Teilnehmer mit der höchsten Nachrichtienpriorität, während die restlichen Busteilnehmer die Nachricht lesen. Anhand dieser Prioritäten lässt sich auch das Zeitverhalten der Nachrichten in dem ereignisgesteuertem CAN-Bus beeinflussen.

2.2.1 Nachrichten-Aufbau

Die Kommunikation zwischen den Busteilnehmern erfolgt durch Nachrichten, so genannte CAN-Telegramme. Diese Telegramme sind in mehrere Felder eingeteilt, die wiederum eine bestimmte Anzahl an Bits enthalten. Der genaue Aufbau wird aus Abbildung 2.2 ersichtlich (vgl. (Lawrenz und Obermöller, 2011, Seite 19)).

Abbildung 2.2: Aufbau des CAN Frames

Das erste Bit des Telegramms ist das Startbit und dient zur Phasensynchronisation der frei laufenden Busteilnehmer. Das Arbitrations-Feld hat bei der Version CAN 2.0A eine Länge von 11 Bit, in denen die Priorität und damit gleichzeitig auch die logische Adresse (Identifier)

des Telegramms gespeichert ist. Durch diese 11 Bit sind 2^{11} (= 2048) verschiedene logische Adressen möglich. Dieser Adressraum wurde durch CAN 2.0B auf 29 Bits erweitert. Im folgende RTR-Bit (Remote-Transmission-Request-Bit) wird gespeichert, ob das Telegramm selber Daten enthält oder von einem anderen Knoten Daten anfordert. Das Kontroll-Feld enthält ein IDE-Bit (Identifier Extension), ein reserviertes Bit r0 und die Längeninformation (Data Length Control) über die nachfolgenden Daten. Sollte das IDE-Bit gesetzt sein, folgt diesem noch ein weiterer Adressbereich mit 18 Bits für die CAN 2.0B Erweiterung. Die Länge des Daten-Feldes wird über das Kontroll-Feld gesteuert und kann 0 - 8 Bytes enthalten, dem eigentlichen Payload. Das CRC-Feld (Cyclic Redundancy Check) besitzt 16 Bit, enthält den Fehlercode über alle vorangegangen Felder und dient nur zur Fehlererkennung. Dabei können bis zu sechs Einzelbitfehler erkannt werden. Vor dem abschließenden EOF-Feld (End of Frame), welches das Ende des Telegramms kennzeichnet, bzw. das Telegramm vom folgenden trennt und insgesamt 10 Bit enthält, befinden sich noch zwei Acknowledgebits, mit denen alle Busteilnehmer eine korrekte Übertragung quittieren. Zusätzlich können die Felder bis zum CRC-Feld mit zusätzlichen Stuff-Bits ergänzt werden. Dabei wird immer nach genau 5 gleichen Bits ein komplementäres Stuff-Bit eingefügt, um den lokalen Bittakt aller Empfänger nachzusynchronisieren. Dies können bis zu 19 Stuff-Bits bei CAN 2.0A und bis zu 23 Stuff-Bits bei CAN 2.0B sein. Daraus ergibt sich eine maximale Gesamtlänge von 130 bzw. 154 Bits des CAN-Telegramms. Bei späteren Überlegungen gehen wir immer von dieser maximalen Datenlänge aus.

2.2.2 Arbitrierung, Priorität

Der Zugriff auf den Bus wird beim CAN-Protokoll wird durch das CSMA/CD + AMP (Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority) Übertragungsverfahren geregelt. Dazu sind die beiden logischen Pegel *Null* und *Eins* auf dem Bus *dominant* bzw. *rezessiv*. Sollte also bei der Arbitrierung von einem Teilnehmer eine *Eins* gesendet werden, wird diese von einer *Null* eines anderen Teilnehmers überschrieben. So wird die Kollision zerstörungsfrei erkannt und der Gewinner kann seinen Sendevorgang fortsetzen, ohne das Telegramm von vorne senden zu müssen. Durch diese Vereinbarung wird auch die Priorisierung geregelt. Da der Pegel *Null* den Pegel *Eins* überschreibt und damit die Arbitrierung gewinnt, haben die Telegramme mit einem niedrigen Identifier eine höhere Priorität als Telegramme mit einem hohen Identifier. Das Telegramm mit dem Identifier *0x00* hat damit die höchste Prioriät im System. Im folgenden Beispiel soll die Arbitrierungsphase kurz zwischen zwei konkurrierden Teilnehmern erklärt werden (vgl. (Lawrenz und Obermöller, 2011, Seite 23)).

Das Beispiel enthält 2 Teilnehmer, die beide auf den Bus übertragen wollen. Nachdem beide erkannt haben, dass der Bus frei ist, senden sie ihr dominantes Startbit. Diese beiden Startbits überlagern sich auf dem Bus, so dass beide Teilnehmer auf ihrem Rückkanal den gesendeten dominanten Pegel erkennen. So lange ein Teilnehmer auf seinem Rückkanal

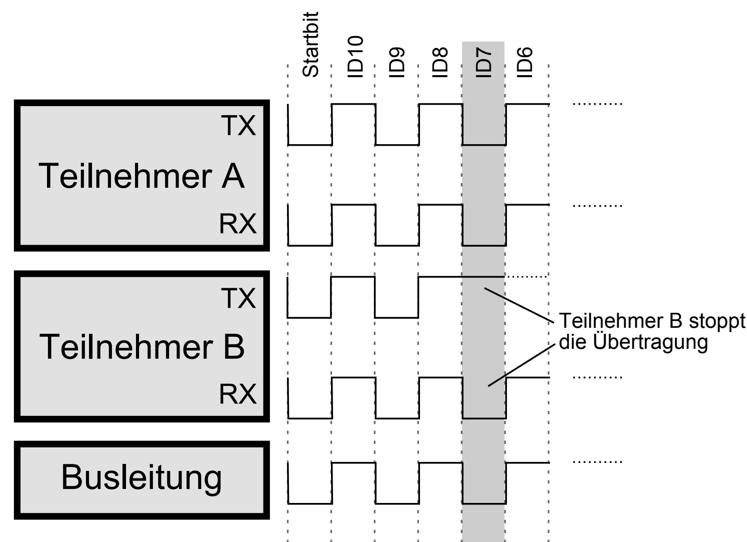

Abbildung 2.3: CAN-Arbitrierung

den gleichen Pegel erkennt, den er auch gesendet hat, fährt er mit seiner Übertragung fort. Nach dem Startbit wird der Identifier übertragen und zwar mit dem MSB zuerst. In unserem Beispiel sind die ersten 3 Bits der beiden Identifier gleich, so dass erst beim 4. Bit der dominate Pegel des Teilnehmers B durch den rezessiven Pegel des Teilnehmers A überschrieben wird. Teilnehmer B erkennt nun die verlorene Arbitrierung und bricht seine Übertragung ab. Teilnehmer A kann sein Telegramm erfolgreich senden.

2.2.3 Zeitverhalten

Wie beim Nachrichten-Aufbau beschrieben, kann die Länge des Telegramms stark variieren. Der erste Faktor ist das Format des Identifiers, nämlich ob es sich um einen Standard-Identifier handelt oder um einen Extended-Identifier. Der zweite, stärker gravierende Faktor ist der eigentliche Payload, der zwischen 0 und 8 Datenbytes enthalten kann. Je nach Art des Identifier und Länge des Datenfeldes können sich damit unterschiedliche Nettodatenraten ergeben. Folgende Tabelle zeigt diese Werte bei einer Übertragungsrate von 1 Mbit/s ohne die Berücksichtigung der Stuff-Bits.
Ein weiteres Zeitverhalten stellt die maximale Verzögerungszeit bis zum Senden eines Telegramms dar. Während für hochpriorisierte Telegramme noch eine Verzögerungszeit angegeben werden kann, kann diese für niedrigpriorisierte nur noch über stochastische Methoden

DATENLÄNGE	NETTODATENRATE BEI	
	STANDARD-IDENTIFIER	EXTENDED-IDENTIFIER
0	-	-
1	72,1 kbit/s	61,1 kbit/s
2	144,1 kbit/s	122,1 kbit/s
3	216,2 kbit/s	183,2 kbit/s
4	288,3 kbit/s	244,3 kbit/s
5	360,4 kbit/s	305,3 kbit/s
6	432,4 kbit/s	366,4 kbit/s
7	504,5 kbit/s	427,5 kbit/s
8	576,6 kbit/s	488,5 kbit/s

Tabelle 2.1: CAN Nettodatenraten bei 1 Mbit/s

ermittelt werden. Die Wartezeit für das höchstpriorisierte Telegramm ergibt sich aus der Anzahl der zu übertragenden Bits und der Übertragungsrate. Beim Standard-Identifier ergibt sich eine maximale Wartezeit von 130 µs bei 130 Bits und einer Rate von 1 Mbit/s für das Telegramm und entsprechend 154 µs beim Extended-Identifier (vgl. (Lawrenz und Obermöller, 2011, Seite 30)). Der letzte wichtige Aspekt im Zeitverhalten ist die maximale Frequenz, in der Telegramme versendet werden können. Diese liegt bei einer Übertragungsrate von 1 Mbit/s und einem Payload von 0 Datenbytes vor. Bei 100% Busauslastung kann bei einem Standard-Identifier alle 44 µs und beim Extended-Identifier alle 64 µs ein Telegramm gesendet werden.

2.3 Bridge Design

In diesem Abschnitt werden zwei verschiedene Architekturen vorgestellt, mit denen sich mehrere CAN-Busse über Bridges koppeln lassen (vgl. Scharbarg u. a. (2005)).

2.3.1 CAN basierte Lösung

Die erste Architektur ist eine rein CAN basierte Lösung. Jeder Bus hat 2 lokale Steuereinheiten und 2 Bridges, mit denen eine Verbindung zu anderen Bussen aufgebaut werden kann. Sollten mehr als 3 Busse zu einem System verbunden werden, bedeutet das, dass nicht jeder Bus mit jedem direkt verbunden ist. Folgende Abbildung verdeutlicht diesen Aufbau.

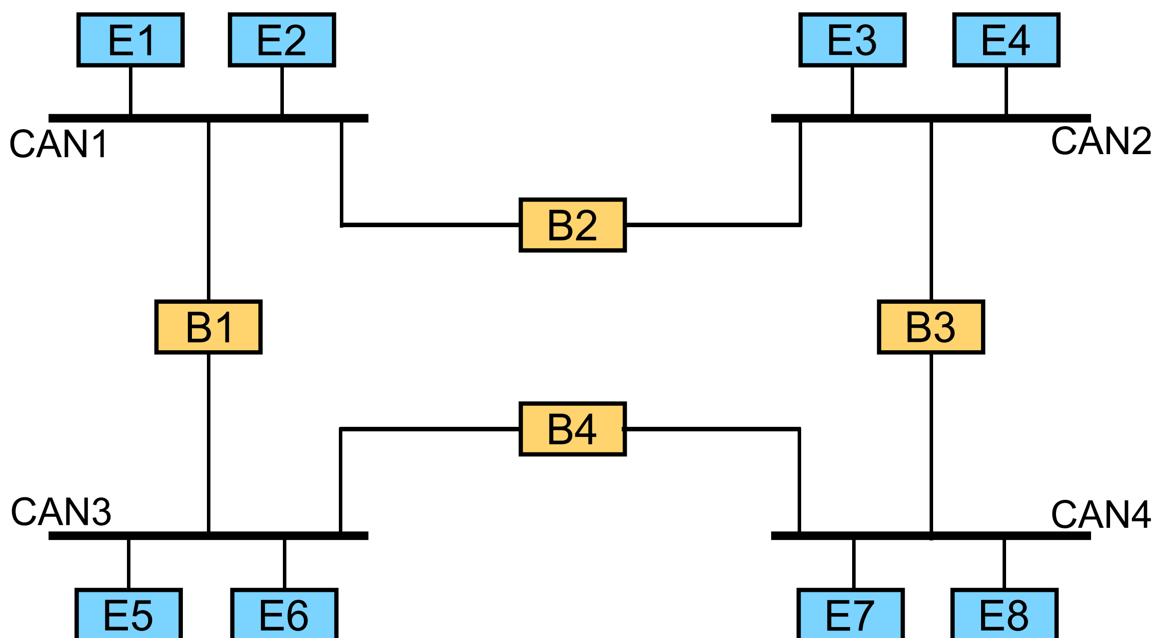

Abbildung 2.4: Reine CAN Architektur

In diesem System treten 3 verschiedene Nachrichtenwege auf.

1. Lokale Nachrichten in einem Bus. Zum Beispiel E1 nach E2.
2. Nachrichten von Bus CAN1 an Bus CAN2, wobei die beiden Busse direkt über eine Bridge verbunden sind. Eine Nachricht von E1 an E3 wird dann zunächst von der Bridge B2 empfangen und auf den Bus CAN2 geschrieben.
3. Nachrichten von z.B. Bus CAN1 an Bus CAN4. Die Busse sind nicht direkt miteinander verbunden. Eine Nachricht von E1 muss zunächst von einer Bridge empfangen

werden, die die Nachricht an einen Bus weitergeben kann, der mit CAN4 verbunden ist. So könnte zunächst B2 die Nachricht auf den Bus CAN2 schreiben und damit die Bridge B3 die Nachricht an CAN4 weitergeben, so dass die Nachricht letztendlich von E7 empfangen wird.

Die Laufzeiten, die die Nachrichten benötigen, um von einem Bus zu einem anderen gesendet zu werden, können sich daher stark voneinander unterscheiden. Nehmen wir für die Berechnung der Laufzeiten an, dass alle Busse die gleiche Übertragungsgeschwindigkeit von 1 Mbit/s haben und CAN-Nachrichten im Extended-Frame-Format mit maximalem Payload versendet werden, also eine Größe von 154 Bit haben. Damit ergibt sich eine Übertragungszeit von $T_t = 154\mu s$ auf einem CAN-Bus. Der zweite Faktor ist die Verzögerung, die beim Senden auftreten kann, wenn der Bus gerade blockiert ist. Diese liegt für hochpriore Nachrichten im Bereich $T_d = [0, 154]\mu s$. Der nächste Faktor, der zu beachten ist, ist die Zeit, die die Bridge benötigt, um die Nachricht zu verarbeiten. Wir nehmen an, dass diese bei $T_b = 100\mu s$ liegt. Für die Nachrichtenwege ergeben sich dann folgende Zeiten.

- $T_1 = T_t + T_d = [154, 308]\mu s$
- $T_2 = 2 * (T_t + T_d) + T_b = [408, 716]\mu s$
- $T_3 = 3 * (T_t + T_d) + 2 * T_b = [662, 1124]\mu s$

Es ist zu erkenne, dass die Zeiten bei dieser Architektur schnell anwachsen, je mehr Busse miteinander gekoppelt werden und über sämtliche Busse übertragen werden muss.

2.3.2 Ethernet basierte Lösung

Die zweite Architektur setzt auf eine Verbindung der CAN-Busse über ein gemeinsames Switched Ethernet Netzwerk. Jeder Bus hat neben seinen lokalen Steuereinheiten eine Bridge, mit der der Bus Nachrichten über das Ethernet an die anderen Busse senden kann. Dabei treten, wie auch anhand der folgenden Grafik zu erkennen ist, nur noch 2 verschiedene Nachrichtenwege auf.

1. Lokale Nachrichten in einem Bus. Zum Beispiel E1 nach E2.
2. Nachrichten von Bus A an Bus B. Zum Beispiel von E1 in CAN1 nach E3 in CAN2. Die Nachricht wird dabei zuerst von B1 empfangen und per Ethernet an B2 übertragen, welche die Nachricht an E3 sendet.

Für die Berechnung der Laufzeiten nehmen wir zusätzlich an, dass das Switched Ethernet mit 100 Mbit/s überträgt und im Switch keine Verzögerungen auftreten. Der Ethernet Frame hat eine Größe von 512 Bit und wird dann innerhalb von $T_e = 5,12\mu s$ übertragen. Für die beiden Nachrichtenwege ergeben sich dann folgende Laufzeiten.

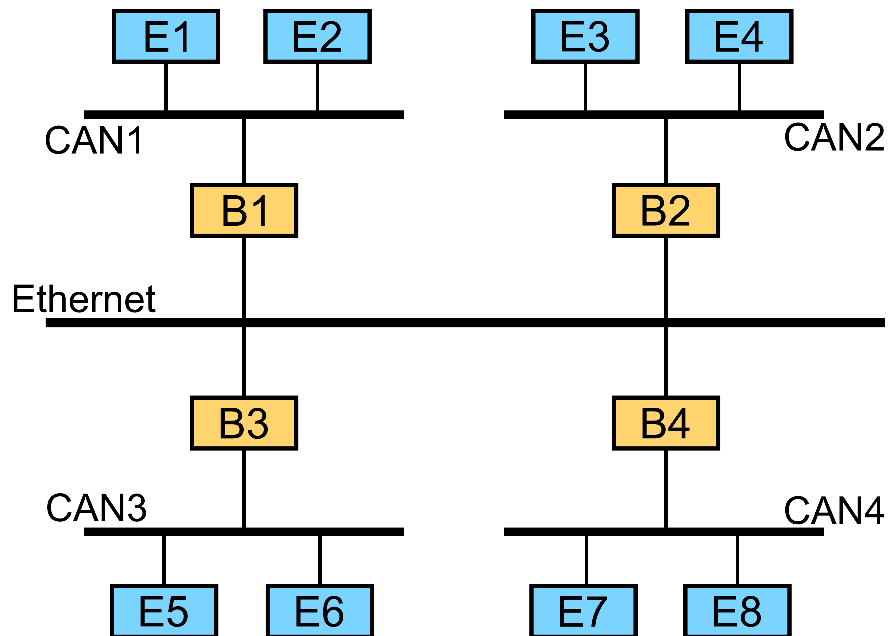

Abbildung 2.5: CAN Ethernet Architektur

- $T_1 = T_t + T_d = [154, 308]\mu s$
- $T_2 = 2 * (T_t + T_d) + 2 * T_b + T_e = [514, 822]\mu s$

Zu sehen ist, dass die Laufzeit für den zweiten Nachrichtenweg etwa 100 µs länger ist, als bei der reinen CAN Architektur, allerdings ist diese für beliebig viele gekoppelte CAN-Busse die selbe, solange der Verkehr nicht so hoch ist, dass innerhalb des Switches Kollisionen auftreten. Um diese Kollisionen zu vermeiden und die Laufzeiten deterministisch zu halten, bietet sich der Einsatz eines Time-Triggered Ethernets an.

Kapitel 3

Anforderungen

Die Hauptanforderung an eine Bridge ist die erfolgreiche Verbindung von zwei Segmenten in einem Computernetzwerk. Im Fall dieser Arbeit die Verbindung von zwei oder mehr physikalischen Teilbussen eines ursprünglich zusammenhängendem CAN-Busses über einen Time-Triggered Realtime Ethernet-Backbone. Dazu muss eine Bridge eine CAN-Nachricht von einem CAN-Bus empfangen, über den Backbone an eine andere Bridge weiterleiten und von dieser wieder auf einen CAN-Bus geschrieben werden.
Trotz dieser Überbrückung der Kommunikation darf diese in ihrer Funktionsweise nicht beeinträchtigt werden. Das Verhalten der einzelnen Steuereinheiten muss auf den getrennten physikalischen Teilbussen unverändert zu einem zusammenhängendem Bus sein. Dies betrifft auch das Arbitrierungsverfahren des CAN-Busses und damit die Einhaltung der Nachrichtenprioritäten. Zudem müssen die Nachrichten von der Bridge richtig geroutet werden. Sollte der CAN-Bus in mehr als zwei physikalische Teilbusse aufgeteilt worden sein, muss die Bridge eine empfangene Nachricht an den richtigen Teilbus weiterleiten.
Eine weitere Anforderung ist die möglichst effektive Nutzung der Bandbreite, sowohl auf der Seite des CAN-Busses, wie auch auf der Seite des TT-Ethernet Backbones. Auf CAN Seite wird Bandbreite verschwendet, wenn eine Nachricht fälschlicherweise an physikalische Teilbusse weitergeleitet wird, an die keine Steuereinheiten angeschlossen sind, die diese Nachricht empfangen müssen. Im Ethernet Backbone wird durch die Wahl des Arbitrierungsverfahrens, sowie durch das gewählte Routingverfahren Traffic erzeugt. Diese müssen entsprechend gewählt werden.
Auch das Zeitverhalten ist eine Anforderung. Die Verzögerungen beim Weiterleiten müssen möglichst gering gehalten werden. Wenn eine Bridge eine CAN-Nachricht empfängt, muss die Bridge diese möglichst zeitnah über die richtige Route an die andere Bridge senden. Dies hängt wieder hauptsächlich von der Wahl des Routingsverfahrens ab.
Im nächsten Kapitel werden unterschiedliche Aspekte des Bridgedesigns diskutiert und mehrere Möglichkeiten vorgestellt, mit denen diese Anforderungen umgesetzt werden können.

Kapitel 4

Konzeption Bridge

In diesem Kapitel werden die unterschiedlichen Aspekte diskutiert, die bei der Konzeption einer Bridge zur Kopplung von CAN Bussen über Real Time Ethernet auf Basis von Mikrokontrollern beachtet werden müsssen. Dazu werden Vor- und Nachteile zu jedem Bereich herausgearbeitet, anhand derer die Designentscheidungen gefällt werden.
Im ersten Abschnitt werden wir uns mit dem Thema der Arbitrierung beschäftigen und um eine mögliche Umsetzung beim Einsatz einer Bridge. Der darauf folgende Abschnitt behandelt das Routing, welches benötigt wird, um einen effektiven Einsatz der Bridge gewährleisten zu können. Der dritte Abschnitt setzt sich mit den Möglichkeiten auseinander, die durch die unterschiedlichen Nachrichtenklassen des Time-Triggered Ethernets gegeben sind und wie sie für das Routing eingesetzt werden können. Der letzte Abschnitt erklärt das Message Stuffing und wie mit dieser Methode die Arbeitsweise der Bridge angepasst werden kann.

4.1 Arbitrierung

Wie im Kapitel 2.2.2 über die Grundlagen der Arbitrierung beschrieben, erfolgt die Buszugriffssteuerung beim CAN bitweise. Hat man nun den Bus über eine Bridge gekoppelt und möchte diese Art der Arbitrierung beibehalten, treten unterschiedliche Probleme auf. Da jedes Bit einzeln über die Bridge gesendet werden muss, muss man zunächst betrachten, ob die Geschwindigkeit der Bridge hierfür ausreicht. Wir betrachten diesen Aspekt dabei unter der Annahme, dass der CAN Bus mit einer Geschwindigkeit von 1 Mbit/s sendet und nur 1 logischer Bus über die Bridge gekoppelt wird. Die Bridge muss dann pro Mikrosekunde 1 Bit übertragen, also pro Mikrosekunde einen Ethernet Frame senden. Auch wenn pro Frame effektiv nur 1 Bit an Nutzdaten übertragen wird, hat der Frame, bedingt durch die benötigte Mindestgröße, eine Länge von 64 Byte. Daraus lässt sich eine Bandbreite von 512 Mbit/s berechnen, die für die Übertragung im Ethernet benötigt wird. Die verbrauchte Bandbreite multipliziert sich mit dem entsprechendem Faktor, wenn mehre logische Busse über die

LOG. BUSSE	BANDBREITE IM	
	CAN	ETHERNET
1	1 Mbit/s	512 Mbit/s
1	500 kbit/s	256 Mbit/s
1	125 kbit/s	64 Mbit/s
2	1 Mbit/s	1024 Mbit/s
2	500 kbit/s	512 Mbit/s
2	125 kbit/s	128 Mbit/s
3	1 Mbit/s	1536 Mbit/s
3	500 kbit/s	768 Mbit/s
3	125 kbit/s	192 Mbit/s

Tabelle 4.1: Benötigte Bandbreite bei bitweiser Arbitrierung

Bridge gekoppelt werden. In der folgenden Tabelle ist eine kurze Übersicht zu diesen Faktoren zusammen gestellt.
Anhand der hohen Vorraussetungen an die Bandbreite im Ethernet-Bus erkennt man, dass eine Umsetzung der bitweisen Arbitrierung gerade für mehrere gekoppelte logische CAN-Busse wenig sinnvoll ist. Mit einem Gigabit-Switch ließen sich noch 2 logische CAN-Busse mit 1 Mbit/s koppeln. Darüber hinaus ist die Arbitrierung mit der vorhandenen Hardware nicht möglich.

Sollte 1 logischer Bus in 2 physikalische Busse aufgeteilt werden und keine gemeinsame Arbitrierung durch eine Bridge erfolgen, ergibt sich das Problem, dass die beiden physikalischen Busse unabhängig voneinander arbeiten. Das heißt, es ist möglich dass zur gleichen Zeit jeweils in jedem Teilbus eine Nachricht gesendet wird. Bei gemeinsamer Arbitrierung würde dies seriell geschehen. Diese Nachrichten werden nun zeitversetzt im jeweils anderen Bus empfangen. Ein besonderes Problem, welches bei diesem Verhalten auftreten kann, ist jenes, dass eine niedrigpriore Nachricht in Teilbus A gesendet wird und parallel dazu damit begonnen wird, nacheinander mehrere hochpriore Nachrichten in Teilbus B zu versenden. Unter diesen Umständen kann die niedrigpriore Nachricht beliebig lange in der Bridge verzögert werden, bevor sie erfolgreich an den Teilbus B übertragen werden kann. Noch kritischer wird es, wenn in diesem Zeitraum noch weitere Nachrichten im Teilbus A gesendet werden. Selbst wenn diese eine höhere Priorität haben, werden sie durch die niedrigpriore Nachricht in der Bridge blockiert. Dieses Problem muss zur Designzeit des Systems ausgeschlossen werden.

Eine weitere Vorraussetzung für die bitweise Arbitrierung ist, dass man den entsprechnenden Zugriff auf den CAN Bus hat. Das heißt, man muss bei Verwendung eines Mikro-

controllers direkt auf die Hardware zugreifen und kann keinen bereitgestellten Hardware Abstraction Layer benutzen, da diese nur Funktionen zum Senden und Empfangen von kompletten CAN Frames zur Verfügung stellen und nicht den direkten Zugriff auf einzelne empfangene Bits ermöglichen.

4.2 Datenkapselung

Wenn man die Frames eines Netzwerkprotokolls innerhalb eines Frames eines anderen Protokolls versenden möchte, stellt sich die Frage, wie diese in dem anderen Frame gekapselt werden sollen. In diesem Fall also, wie der CAN-Frame in den Ethernet-Frame eingebettet werden soll. Dieses Problem kann man grundlegend auf 2 verschiedene Wege lösen.

1. Man versendet den CAN-Frame komplett innerhalb des Ethernet Frames
2. Nur die Nutzdaten des CAN-Frames werden in den Payload des Ethernet Frames übernommen

Der komplette CAN-Frame beinhaltet sämtliche Bits, um die Nachricht direkt wieder auf einen CAN-Bus schreiben zu können. Die Nutzdaten umfassen nur das Arbitrierungsfeld (hauptsächlich den Identifier), Kontrollfeld und Datenfeld. Der CAN-Frame hat im Extended-Frame-Format bei maximalem Payload eine Länge von 154 Bit. Arbitrierungsfeld, Kontrollfeld und Datenfeld umfassen dabei 121 Bit. Der Unterschied zwischen den beiden Varianten wäre also nur, dass bei der ersten 33 Bit weniger übertragen werden. Wenn man im Ethernet von einer Übertragungsgeschwindigkeit von 100 Mbit/s ausgeht, fällt dieser Overhead kaum ins Gewicht, so dass man den kompletten CAN-Frame über das Ethernet übertragen kann. Dies erspart einem unter anderem auch die Neuberechnung der CRC-Prüfsumme beim erneuten Senden der CAN-Nachricht auf den Bus.
Letztendlich ist bei dieser Designentscheidung aber auch zu beachten, welchen Zugriff man auf den empfangenen CAN-Frame bekommt. Wenn man durch den Hardware-Abstraction-Layer des Mikrocontrollers nur Zugriff auf die Nutzdaten bekommt, können auch nur diese über das Ethernet gesendet werden.
In den folgenden Betrachtungen dieses Kapitels gehen wir der Einfachheit halber davon aus, dass immer der komplette Frame übertragen wird.

4.3 Routing

In diesem Abschnitt werden mehrere Möglichkeiten vorgestellt, wie man das Routing der Nachrichten umsetzen kann. Dabei wird mit einem einfachen Verfahren begonnen, dessen Komplextät dann gesteigert wird. Interessant ist dabei auch, wie das Scheduling konfiguriert

werden muss und wieviele verschiedene Nachrichten verwendet werden müssen.
Für die Diskussion nehmen wir an, dass die CAN-Busse mit einer Geschwindigkeit von 1 Mbit/s senden und CAN 2.0B, also das Extended-Frame Format verwenden. Des weiteren nehmen wir an, dass die Nachrichten den maximalen Payload von 8 Byte besitzen und damit eine Gesamtlänge von 154 Bit besitzen und komplett von der Bridge übertragen wird. Der TT-Ethernet Backbone arbeite mit einer Geschwindigkeit von 100 Mbit/s und transportiere pro Ethernet Nachricht 1 CAN Nachricht. Die Länge der Ethernet Nachricht beträgt dabei 64 Byte. In der folgenden Grafik 4.1 wird zudem eine Übersicht über die Topologie gegeben, anhand der wir das Routing diskutieren.

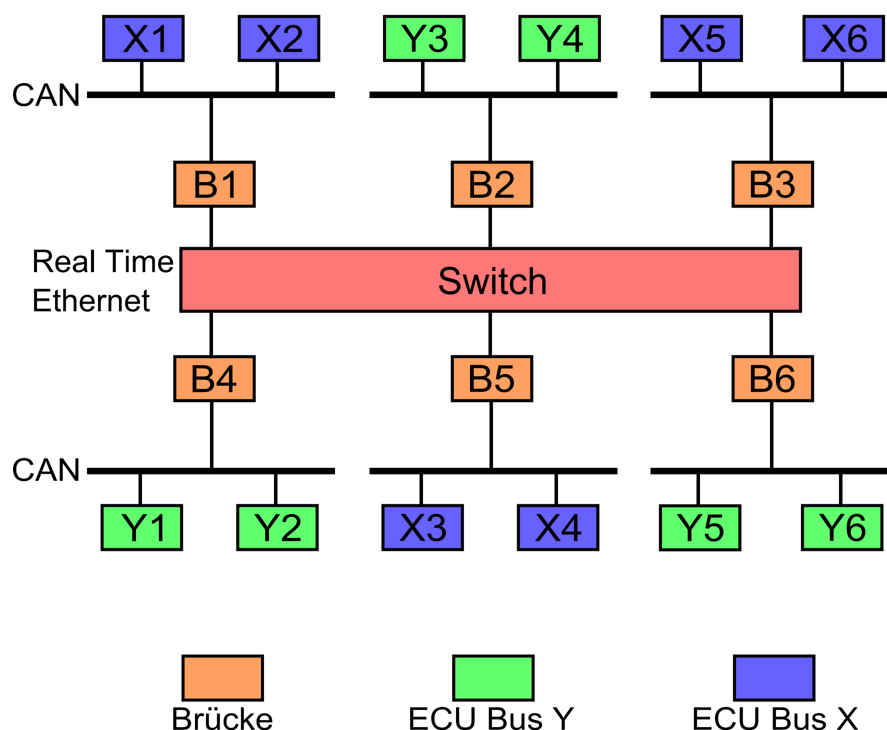

Abbildung 4.1: Topologie des diskutierten Systems

In dem dargestellten System gibt es 2 logische CAN-Busse, die in 6 physikalische Busse aufgeteilt wurden. Jeder physikalische Bus hat 2 Electronic Control Units und ist über eine Bridge an den TT-Ethernet Backbone angeschlossen.

4.3.1 Verfahren 1: Fluten

Das einfachste Routingverfahren ist das sogenannte Fluten (vgl. (Tanenbaum, 2003a, Seite 393)). Bei diesem statischen Routingverfahren wird eine Nachricht an alle Empfänger im

System gesendet. Betrachten wir den Fall, dass X1 eine Nachricht sendet. X2 befindet sich auf dem gleichen physikalischem Bus und muss nicht näher behandelt werden. B1 empfängt die Nachricht und flutet sie an die Bridges B2 bis B6, die wiederum auf ihren physikalischen Bus schreiben. Damit haben alle Empfänger im System die Nachricht von X1 empfangen. Dieses Verhalten wird realisiert, indem man B1 mit einem entsprechenden virtuellen Link konfiguriert. Es wird 1 TT-Nachricht von B1 gesendet, die von den anderen Bridges empfangen wird. Die folgende Abbildung veranschaulicht den Nachrichtenfluss, wenn X1 eine Nachricht sendet. Die Nachrichten treten dabei zeitlich in der Reihenfolge blau, grün und orange auf.

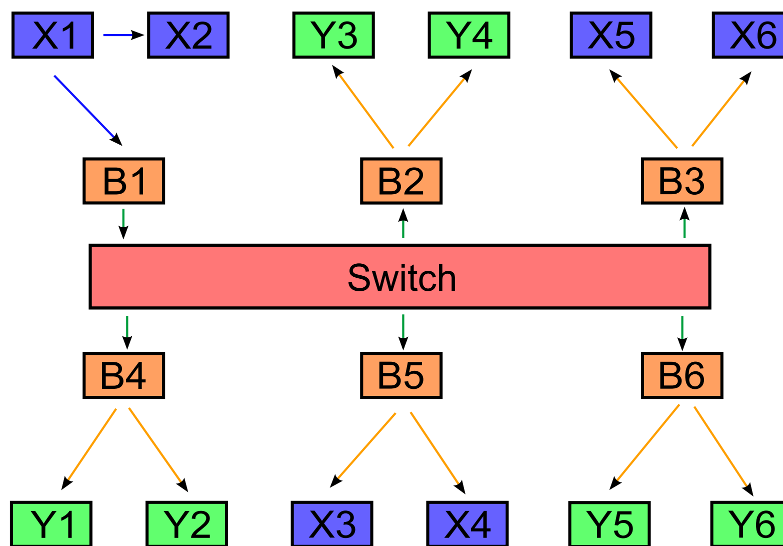

Abbildung 4.2: Nachrichtenfluss beim Fluten

Für das Scheduling muss bedacht werden, mit welcher Häufigkeit CAN-Nachrichten gesendet werden können. Sollte X1 permanent senden, muss die Bridge B1 alle 154 µs ($\frac{154Bit}{1Mbit/s}$) eine CAN-Nachricht verarbeiten, das heißt, dass das Scheduling die TT-Nachrichten mit einem Zyklus von 154 µs versendet. Jede TT-Nachricht hat eine Länge von 64 Byte, die bei einer Datenrate von 100 Mbit/s in 5,12 µs ($\frac{512Bit}{100Mbit/s}$) versendet wird. Damit könnten theoretisch von einer Bridge pro Zyklus 29 (($\frac{154\mu s}{5{,}12\mu s}$) TT-Nachrichten empfangen werden. Diese Nachrichten sollten im Scheduling jeweils versetzt konfiguriert werden, so dass im Switch keine Kollisionen auftreten können.

An der Anzahl der versendeten Nachrichten wird das Problem an diesem Verfahren deutlich. Wenn mehrere logische Busse über Bridges gekoppelt werden, die ein hohes Nachrichtenaufkommen haben, werden die einzelnen Busse mit Nachrichten überflutet, da die Bridges nicht entscheiden können, an welchen physikalischen Teilbus die CAN-Nachricht weiter-

geleitet werden muss. Senden in unserem Beispiel die ECUs X1 und Y1 in jedem Zyklus, also alle 154 µs eine CAN-Nachricht, empfängt zum Beispiel die Bridge B3 pro Zyklus 2 TT-Nachrichten und müsste demnach 2 CAN-Nachrichten auf den Bus schreiben. Da die Bridge aber pro Zyklus nur 1 CAN-Nachricht schreiben kann, stauen sich die Nachrichten auf und können lange verzögert werden. Zudem wird durch das Fluten Bandbreite im TT-Ethernet Backbone verschwendet.
Ein weiteres Problem betrifft die Identifier der CAN-Nachrichten. Es kann der Fall auftreten, dass in 2 verschiedenen logischen Bussen der gleiche Identifier vergeben ist. So können zum Beispiel X1 und Y1 CAN-Nachrichten mit dem gleichen Identifier versenden. Wenn X3 diese Nachrichten empfängt, kann die ECU nicht erkennen, welche der Nachrichten die richtige ist.

Der Vorteil an diesem Verfahren ist, dass man das Routing nicht konfigurieren braucht. Ein Einsatz kommt aber nur in Frage, wenn zur Designzeit des System sicher gestellt werden kann, dass die oben genannten Problem nicht auftreten können.

4.3.2 Verfahren 2: Selektives Fluten

Das Hauptproblem am ersten Verfahren war, dass jede CAN-Nachricht im gesamten System propagiert wurde. Wenn bekannt ist, dass die verschiedenen logischen CAN-Busse unabhängig voneinander sind, dass heißt, das aus dem Bus X keine Nachrichten an den Bus Y gesendet werden müssen und umgekehrt, kann das Fluten eingeschränkt werden. Eingeschränkt in der Hinsicht, dass jede Bridge die CAN-Nachricht nur noch an die physikalischen Teilbusse flutet, die zum entsprechenden logischen Bus gehören. Bridge B1 flutet eine CAN-Nachricht also nur noch an die Bridges B3 und B5. Der Virtuelle Link aus dem erste Verfahren wird dementsprechend konfiguriert.
Die folgende Abbildung 4.3 verdeutlicht dieses Verfahren. Die Nachrichten treten dabei wieder in der zeitlichen Reihenfolge blau, grün und orange auf. Die Konfiguration des Schedulers muss nicht verändert werden.
Die Menge an erzeugten Nachrichten wird durch dieses Verfahren stark eingeschränkt und weniger Bandbreite im Ethernet verschwendet. Dennoch kann es wie im ersten Verfahren noch zu einem Nachrichtenstau kommen, sollten 2 physikalische Teilbusse parallel damit beginnen, viele CAN-Nachrichten zu senden. Dies muss weiterhin beim Design des Systems ausgeschlossen werden. Allerdings wird das Problem mit den doppelten CAN Identifiern auf diese Art umgangen.
Der Nachteil an dem selektiven Fluten ist, dass keine Nachrichten mehr zwischen den logischen Bussen ausgetauscht werden können, sollte dies nötig sein.

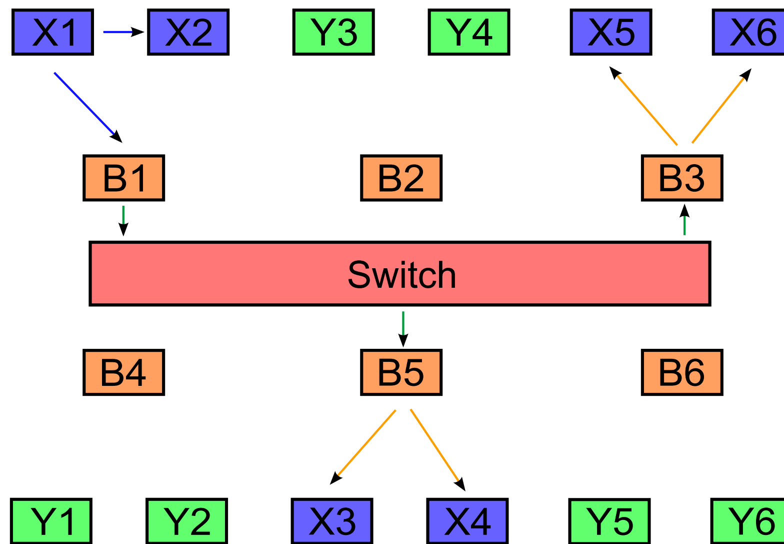

Abbildung 4.3: Nachrichtenfluss beim selektiven Fluten

4.3.3 Verfahren 3: Routing Tabellen

Das dritte Verfahren verfolgt einen komplett anderen Ansatz. Obwohl es sich immer noch um ein statisches Routing handelt, wird für jede CAN-Nachricht ein eigenes Routing-Verhalten konfiguriert. Für jede Bridge wird eine Routingtabelle erstellt, in der festgelegt wird, welche CAN-Nachricht an welche Bridge weitergeleitet werden muss. Dafür muss beim Design des Systems bekannt sein, welche CAN-Nachricht an welchen physikalischen Teilbus gesendet werden muss. Ist dies bekannt, kann für jeden CAN Identifier ein virtueller Link eingerichtet werden. Jede CAN-ID wird auf eine entsprechende TT-ID abgebildet, mit welcher sie übertragen wird. Nehmen wir an unsere ECUs X1 und X2 senden 6 unterschiedliche CAN-Nachrichten. Die Routing Tabelle für B1 könnte dann folgendermaßen aussehen.

CAN-ID	TT-ID
0x20	0x201
0x30	0x301
0x35	0x304
0x50	0xAA
0x60	0xBB

Tabelle 4.2: Routing Tabelle für das 3. Verfahren

Senden X1 oder X2 eine CAN-Nachricht aus dieser Tabelle, sendet die Bridge die entsprechende TT-Nachricht an die Zielbridge. Sollte eine CAN-Nachricht gesendet werden, für die

keine Routinginformation konfiguriert wurde, wird diese CAN-Nachricht von der Bridge nicht weiter im System propagiert. Dieses Verhalten hat den Vorteil, dass keine unnötigen Nachrichten im System verteilt werden und jede CAN-Nachricht, sollte sie auf einem anderen physikalischem Bus benötigt werden, direkt nur an diesen gesendet werden kann. Zudem ist es so möglich, Nachrichten auch gezielt zwischen 2 logischen CAN Bussen auszutauschen. Ein Problem ist, dass für die TT-ID nur 12 Bit zur Verfügung stehen. Damit können nur $2^{12} = 4096$ verschiedene TT-Nachrichten definiert werden. Wenn für den CAN Bus das Extended Frame Format verwendet wird, kann dieses Routing an seine Grenzen stoßen, da damit bis zu 2^{29} verschiedene CAN-Nachrichten im System vorkommen können und so nicht genügend TT-IDs zur Verfügung stehen. Zusätzlich muss jede TT-Nachricht im Scheduling konfiguriert werden, was dieses Verfahren komplexer als die anderen macht. Es muss zur Designzeit festgelegt werden, zu welchem Zeitpunkt im Zyklus eine CAN-Nachricht erwartet wird. Der Vorteil an dieser komplexen Konfiguration ist, dass sichergestellt werden kann, dass jede CAN-Nachricht ohne Verzögerung versendet wird. Die zugehörige TT-Nachricht muss dann entsprechend im Scheduling eingetragen werden. Der Zyklus kann dabei im Gegensatz zu den vorherigen Verfahren je nach Anwendung beliebig groß gewählt werden.

4.3.4 Verfahren 4: Vereinfachte Routing Tabellen

Für dieses Verfahren greifen wir Aspekte aus dem selektiven Fluten und den eben erklärten Routing Tabellen auf, die weiterhin eingesetzt werden. Es wird jedoch nicht mehr für jede CAN-ID eine Route eingerichtet, sondern nur noch für jede benötigte Verbindung zwischen 2 Bridges. Folgende Tabelle zeigt eine mögliche Konfiguration für die Bridge B1.

CAN-ID	TT-ID
0x20	0x201
0x30	0x301
0x35	0x301
0x50	0x201

Tabelle 4.3: Routing Tabelle für das 4. Verfahren

Die TT-ID 0x201 steht dabei für einen virtuellen Link zwischen der Bridge B1 und B3 und die ID 0x301 für einen Link zwischen B1 und B5. Die virtuellen Links sind ähnlich wie beim Fluten nicht mehr an bestimmte CAN-Nachrichten gekoppelt. Da nicht bekannt ist, welche CAN-Nachricht in welchem Zyklus gesendet wird, nutzt man für die virtuellen Links RC-Nachrichten. Damit kann Ereignisgesteuert der richtige virtuelle Link gewählt werden. Sendet zum Beispiel X1 im ersten Zyklus die CAN-ID 0x20, so wird die CAN-Nachricht über die RC-Nachricht 0x201 an die Bridge B3 gesendet. Sollte im darauffolgenden Zyklus weder von X1 noch von X2 eine Nachricht gesendet werden, wird auch im Ethernet keine Bandbreite

reserviert. Sollte dagegen die CAN-Nachricht 0x30 versendet werden, wird diese über die RC-Nachricht 0x301 an die Bridge B5 gesendet. Die Länge des Zyklus ist dabei unabhänig vom Routing. Dieses Verfahren vereinfacht die Konfiguration des Schedulers signifikant, da für die CAN-Nachrichten keine genauen Zeitpunkte mehr definiert werden müssen. Es muss für die virtuellen Links nur ein optimaler BAG Wert ermittelt werden, damit die Nachrichten mit möglichst wenig Verzögerung übermittelt werden. Dieser würde zum Beispiel 154 µs betragen, wenn die Nachrichtenlänge immer 154 Bits beträgt.

4.4 Wahl der Nachrichtenklassen

Da das eingesetzte Time-Triggered Ethernet Protokoll mehrere Nachrichtenklassen definiert, lassen sich mit der Wahl dieser Klassen weitere verschiedene Konzepte und Varianten der vorgestellten Routingverfahren umsetzen.

4.4.1 RC- und TT-Nachrichten

Anstatt das Fluten mit TT-Nachrichten zu konfigurieren, kann man das Verfahren auch mit RC-Nachrichten konfigurieren. Dabei löst man sich von dem Zyklus des Schedulers und definiert statt dessen die BAG Werte der virtuellen Links so, wie vorher der Zyklus festgelegt war. Der Vorteil ist, dass der Zyklus losgelöst vom Routing definierbar ist und unter Umständen weniger Bandbreite verbraucht wird, da TT-Nachrichten immer gesendet werden, auch wenn gar keine CAN-Nachrichten vorliegen.

Auch beim 3. Verfahren ist der Einsatz von RC-Nachrichten anstatt der TT-Nachrichten möglich, liefert aber keine Vorteile beim Bandbreitenverbrauch, da das Scheduling bereits so konfiguriert ist, dass hier ein Minimum erzielt wird. Ein Vorteil ergibt sich hier nur dadurch, dass der Schedule weniger komplex ausfällt, wodurch man aber auch in Kauf nimmt, dass das System weniger deterministisch wird, dass heißt, man kann nicht mehr davon ausgehen, dass die Nachrichten in jedem Zyklus zur gleichen Zeit eintreffen, da die RC-Nachrichten keinem zeitlichem Ablauf unterliegen.

Beim 4. Verfahren wird wiederum der Einsatz von RC-Nachrichten vorrausgesetzt, um ein Minimum an Bandbreite zu verwenden. Um hier mit TT-Nachrichten ein ähnliches Ergebnis zu erzielen, muss die Konfiguration ähnlich komplex wie beim 3. Verfahren erfolgen, das heißt, man muss das Scheduling so festlegen, dass durch die TT-Nachrichten möglichst wenig Bandbreite verwendet wird und auch die zeitliche Verzögerung gering ausfällt.

4.4.2 BE-Nachrichten

BE-Nachrichten weisen kein deterministischen Verhalten auf. Das heißt weder die Laufzeit der Übertragung noch die Sicherstellung der Übertragung ist gewährleistet. Damit kommt dieser Nachrichtentyp für keinen zeit- und sicherheitskritischen Verkehr in Frage. Allerdings ist zum Beispiel ein Einsatz im Telemetriebereich denkbar, da die gesendeten Daten nur für eine Auswertung nötig sind.
Bei den Routingstrategien mit einer Routingtabelle ist eine Umsetzung möglich, indem jede CAN-Nachricht zusätzlich mit einem Nachrichtentyp konfiguriert wird. So kann festgelegt werden, ob die CAN-Nachricht mit einer TT-Nachricht oder einer BE-Nachricht weitergeleitet wird. Dies ermöglicht eine flexible Konfiguration, ob die Nachrichten zeitkritisch sind oder nicht.

4.5 Message Stuffing

Unter dem Begriff Message Stuffing versteht man die Methode, mehre CAN-Frames über 1 Ethernet-Frame zu versenden. Bisher wurde davon ausgegangen, dass jeder CAN-Frame in einem eigenen Ethernet-Frame versendet wird. Die Alternative ist, 2 oder mehr CAN-Frames in einem Ethernet-Frame zu versenden.

In den besprochenen Routingverfahren wurde immer davon ausgegangen, dass jeder CAN-Frame in einem eigenen Ethernet-Frame versendet wird. Dies vereinfacht die Implementation und ermöglicht minimale Verzögerungszeiten in der Bridge, da jede CAN-Nachricht sofort über den Ethernet Backbone versendet werden kann.
Der Nachteil dabei ist, dass Bandbreite verschwendet wird, da der minimale Payload für einen Ethernet-Frame 368 Bit beträgt. Die maximale Länge eines CAN-Frames beträgt aber im Extended-Frame-Format nur 154 Bit. Wenn 1 CAN-Frame mit einem Ethernet-Frame versendet wird, wird trotzdem ein Payload von 368 Bit versendet, damit der Ethernet Frame die Mindestlänge von 512 Bit erfüllt. Es wird mehr als die Hälfte der Bandbreite nicht effektiv genutzt. Die folgende Tabelle 4.4 zeigt, wie sich die Größe des Ethernet Frames zu der Anzahl der transportierten CAN-Frames verhält. Der Wert in der 4. Spalte spiegelt den Anteil der Bits in Prozent wieder, den die CAN-Daten im Bezug zur kompletten Länge des Ethernet-Frames einnehmen.
Es ist zu erkennen, dass die Ethernet Bandbreite effektiver genutzt wird, wenn mehrere CAN-Frames mit einem Ethernet Frame versendet werden. Bis hin zu 3 Nachrichten steigt der Anteil an versendeten Nutzdaten stark an, so dass es sinnvoll ist, gegebenenfalls 3 oder mehr CAN-Nachrichten über einen Ethernet-Frame zu versenden.

Implementieren lässt sich das Message-Stuffing auf mehrere Arten. Eine Designent-

ANZAHL CAN-NACHRICHTEN	CAN DATEN	LÄNGE DES ETHERNET FRAMES	ANTEIL CAN/ETHERNET
1	154 Bit	512 Bit	30%
2	308 Bit	512 Bit	60%
3	462 Bit	606 Bit	76%
4	616 Bit	760 Bit	81%
5	770 Bit	914 Bit	84%

Tabelle 4.4: Verwendung des Ethernet Payloads bei Message Stuffing

scheidung ist, ob eine lokale oder globale Konfiguration vorgenommen wird. Man kann für alle CAN-Nachrichten im System das gleiche Message-Stuffing vorgeben oder für jeden CAN-Identifier einzeln konfigurieren. Dies könnte zum Beispiel durch eine Erweiterung der Routing-Tabelle erfolgen.
Weiterhin ist zu entscheiden, wie lange CAN-Nachrichten gesammelt werden sollen. Konfiguriert man das Stuffing zum Beispiel so, dass 5 CAN-Nachrichten gesammelt werden sollen, muss man warten, bis diese 5 CAN-Nachrichten vorliegen. Dadurch werden die ersten Nachrichten gegebenenfalls stark verzögert oder sogar nie versendet, weil keine 5. Nachricht eintrifft. Um dieses Verhalten zu entschärfen, kann man das Stuffing zeitlich limitieren. Anstatt ewig zu warten, bis alle Nachrichten zum versenden bereit liegen, wartet man nur eine vorgegebene Zeitspanne und versendet dann die vorhandenen Nachrichten.

Kapitel 5

Umsetzung des Bridgedesigns und Implementierung

Nachdem im vorherigen Kapitel unterschiedliche Strategien für ein Bridgedesign diskutiert wurden, liefert dieses Kapitel eine Übersicht über die gefällten Designentscheidungen und eine kurze Einführung in den eingesetzten Mikrocontroller. Dabei wird zuerst näher auf die Hardware eingegangen, da dessen Eigenschaften Einfluss auf bestimmte Designentscheidungen hat. Zum Schluss wird dann näher auf die eigentliche Implementierung der Bridge eingegangen.

5.1 Mikrocontroller

Der Mikrocontroller NXHX-500 ist ein Entwicklungsboard, das von der Hilscher Gesellschaft für Systemautomation mbH entwickelt und vertrieben wird. Das Board basiert auf dem netX 500 Netzwerk-Controller und bietet hohe Rechenleistung, umfangreiche Peripheriefunktionen und ein integriertes Debug Interface (vgl. Lipfert (2008a)).

5.1.1 Übersicht

Der netX 500 ist ein System-On-Chip Design mit einem ARM 926EJ-S. Dies ist eine 32 Bit CPU mit einer Taktfrequenz von 200 Mhz und Jazelle und DSP Support. Zudem ist der ARM mit einer Memory Management Unit ausgestattet. Die Peripheriefunktionalität umfasst USB, UART, Timer, Interrupt Controller, Watchdog und GPIOs (general purpose IOs). Darüber hinaus besitzt der netX 4 Kommunikationskanäle, die über jeweils 3 frei programmierbare ALUs verfügen, die sich die Kommunikationsfunktionen untereinander aufteilen.
Jeder Kanal hat damit eine 16 Bit Receive Processing Uni (RPU), eine 16 Bit Transmit Processing Unit (TPU) und einen 32 Bit Protocol Execution Controller (xPEC). Alle 3 ALUs arbeiten dabei mit einer Taktrate von 100 Mhz. Der RPU und TPU Controller werden logisch

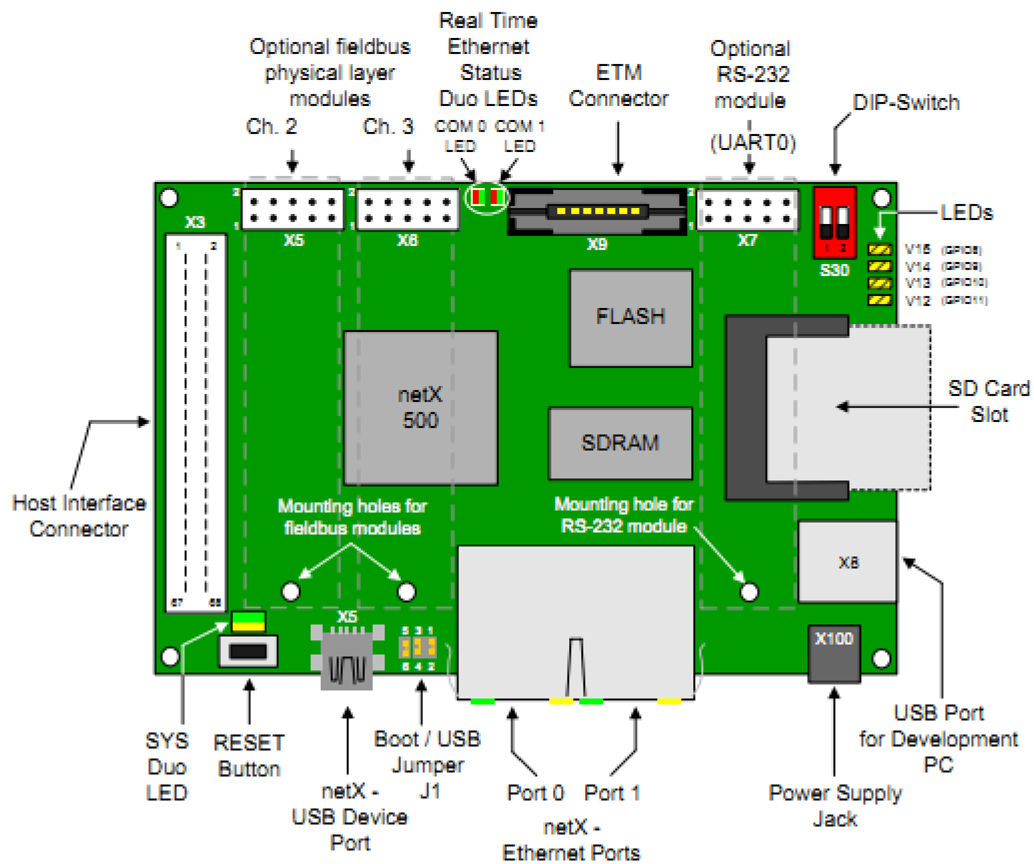

Abbildung 5.1: Architektur und Schnittstellen des NXHX 500

zu einem xMAC (Medium Access Controller) zusammengefasst und sind für den Zugriff auf das physikalische Übertragungsmedium zuständig. Zusätzlich übernehmen sie die Codierung, Konvertierung und Überprüfung (CRC) der Daten. Der xPEC übernimmt die zentralen Aufgabe bei der Kommunikation, kontrolliert den Zugriff auf den restlichen Speicher des ARMs und kann verschiedene Interrupts auf dem ARM auslösen.
Die Kommunikationskanäle 0 und 1 sind für die beiden Ethernet Ports reserviert, während die Kanäle 2 und 3 für die Kommunikation mit Feldbussen vorgesehen sind. Unterstützte Bussysteme sind dabei AS-Interface, CANopen, CC-Link, DeviceNet und PROFIBUS.

Folgend wird kurz die Funktionsweise der beiden benutzten Protocoll Stacks erklärt und die Datenstrukturen vorgestellt, die den Ethernet- und CAN-Frame intern repräsentieren, da diese Einfluss auf die Implementierung nehmen.

5.1.2 Ethernet Modul

Da die Bridge die CAN-Nachrichten über Time-Triggerd Ethernet (TTE) bridgen soll, kann die Bridge nicht mit Hilfe des durch Hilscher zur Verfügung gestellten Ethernet Stacks implementiert werden, da dieser nicht echtzeitfähig ist. Statt dessen wird die von Kai Müller entwickelte TTE-API genutzt, die den vorhandenen Ethernet Stack um die gewünschte Echtzeitfähigkeit erweitert. Nähere Informationen und eine ausführliche Beschreibung können dieser Arbeit entnommen werden (vgl. Kai Müller (2011)).
Der Kern diess Moduls basiert auf einem Scheduler, der den zeitlichen Ablauf kontrolliert und Input- bzw. Output-Buffers, über die das Empfangen und Senden realisiert wird. Die API stellt für die Konfiguration und den Zugriff auf diese Buffer entsprechende Funktionen bereit. Werden TT-Ethernet-Frames in den Output-Buffer geschrieben, wird durch den Scheduler ein Task ausgeführt, der diese Frames zum konfigurierten Zeitpunkt versendet. Liegen empfangende Frames im Input-Buffer bereit, wird durch den Scheduler zum konfiguriertem Zeitpunkt eine zuvor definierte Callback-Funktion ausgeführt, in der der Frame weiter verarbeitet werden kann. Der TT-Ethernet-Frame wird dabei intern durch folgende Struktur repräsentiert.

Listing 5.1: C-Struktur des Ethernet Frames

```
1  typedef struct {
2      /** Length of the payload in bytes. */
3      uint16_t length;
4
5      /** Ethernet header in network byte order. */
6      tte_eth_hdr_t eth_hdr;
7
8      /** Address of the payload. */
9      uint8_t *data;
10
11     /** Critical traffic ID (for TT messages). */
12     uint16_t ct_id;
13
14     /** Ethernet channel/port number (for background messages). */
15     uint8_t channel;
16 } tte_frame_t;
```

Der folgende Pseudocodeausschnitt demonstriert die Handhabung beim Senden und Empfangen eines Frames mit Hilfe der TTE-API (vgl. TTTech Computertechnik AG (2008)).

Listing 5.2: Senden und Empfangen mit der TTE-API

```
1  /* Senden */
2  tte_get_ct_output_buf(ct_id, output_buf);
3  tte_open_output_buf(output_buf, frame);
4
```

```
5                    frame.ct_id = ct_id;
6                    frame.data = "Hello World";
7
8                    tte_close_output_buf(output_buf);
9
10
11                   /* Empfangen */
12                   tte_open_input_buf(input_buffer, frame);
13                   //read frame data
14                   tte_close_input_buf(input_buffer);
```

Der Scheduler wird über Schedulingtabellen konfiguriert, in denen sich die Zeitpunkte für die Ausführung von Tasks und für das Senden und Empfangen von Nachrichten festlegen lassen. Zudem wird jeder Eintrag mit einer Deadline konfiguriert. Sollte diese verpasst werden, wird die entsprechende Nachricht verworfen.

5.1.3 CAN Modul

Für die Kommunikation mit dem CAN-Bus wird der von Hilscher zur Verfügung gestellte Hardware Abstraction Layer und Protocoll Stack benutzt. Das Modul stellt eine umfangreiche API zur Verfügung, die sich in Kontroll-Funktionen, Status-Funktionen, Sende-Funktionen und Empfangs-Funktionen aufteilen lassen. CAN-Nachrichten werden durch das Modul in 2 Nachrichtenklassen aufgeteilt, hochpriore und niedrigpriore. Der Zugriff auf die Hardware erfolgt durch die API über 5 FIFO Register.

REGISTER	TIEFE
Indication High	25 CAN-Frames
Indication Low	25 CAN-Frames
Confirmation	40 Acknowledgements
Request High	20 CAN-Frames
Request Low	20 CAN-Frames

Tabelle 5.1: CAN Modul: FIFO Register

In den *Indication High* und *Indication Low* Registern werden die empfangenen Nachrichten gespeichert. Standardmäßig werden sie in das niedrigpriore Register gespeichert. Die API stellt Funktionen zur Verfügung, um CAN-Identifier festzulegen, die in das hochpriore Register gespeichert werden sollen. Beim Empfang einer Nachricht, wird ein Interrupt ausgelöst und die Register können innerhalb der Interrupt Service Routine (ISR) ausgelesen werden. Die *Request High* und *Request Low* Register speichern die Nachrichten zwischen, die gesendet werden sollen. In welches Register gespeichert werden soll, wird dem API-Aufruf

zum Senden als Parameter übergeben. Das *Confirmation* Register speichert die Bestätigungen, die erfolgen, wenn ein CAN-Frame erfolgreich über den Bus übertragen wurde. Diese Bestätigungsnachrichten kann man über die API aktivieren und deaktivieren. Nachfolgend werden die wichtigsten API-Funktionen vorgestellt(vgl. (Lipfert, 2008b, Seite 110)).

Can_Init initialisiert das Modul mit dem gewünschtem Kommunikationskanal des Boards und der gewünschten Baudrate des CAN-Busses.

Can_EnableInterrupt wählt die Busereignisse, die einen Interrupt auslösen sollen.

Can_Start startet das Modul nach der Initialisierung und ist nun bereit Nachrichten zu empfangen und zu senden.

Can_SendFrame speichert eine Nachricht in die Request-FIFOs und gibt sie so für die TPU zum senden frei.

Can_GetInterruptCom liefert die Art des ausgelösten Interrupts.

Can_GetIndicationLoFillLevel liefert die Anzahl der Nachrichten aus dem *Indication Low* Register.

Can_ReceiveLoFrame empfängt eine Nachricht aus dem Register für niedrigpriore Nachrichten.

Can_GetConfirmationFillLevel liefert die Anzahl der *Acknowledgements*

Can_GetSendConfirmation holt eine Nachricht aus dem Confirmation-FIFO

Der Identifier und der Payload werden in Form folgender internen Struktur in die Register geschrieben, bzw. aus den Registern gelesen.

Listing 5.3: C-Struktur des CAN Frames

```
1  typedef struct CAN_FRAME_Ttag {
2      /**< unique packet identifier (not send to wire) */
3      unsigned long ulUniqueId;
4
5      /**< Frame information */
6      unsigned long ulFrameInfo;
7
8      /**< Frame identifier */
9      unsigned long ulIdentifier;
10
11     /**< Frame data (length is defined in frame information) */
12     unsigned char abData[CAN_MAX_BYTE_PER_FRAME];
13 } CAN_FRAME_T;
```

Folgender Pseudocode demonstriert die Handhabung der CAN-API.

Listing 5.4: Senden und Empfangen mit den CAN-API

```
/* Senden */
frame.ulIdentifier = 0x100;
frame.abData = "Hello World";
Can_SendFrame(frame);


/* Empfangen in der ISR */
Can_GetInterruptCom(ulIrq);

if(ulIrq == RECEIVE_LOW_MSG) {
        Can_GetIndicationLoFillLevel(ulFill);
        while(ulFill-- > 0) {
        Can_ReceiveLoFrame(frame);
        //read frame data
    }
}
```

5.1.4 Virtueller CAN-Stack

Um die Bridge flexibler zu gestalten, wurde zusätzlich ein virtueller CAN-Stack implementiert. Dieser Stack wurde mit dem gleichen Interface implementiert wie dem des vorhandenen CAN-Moduls. Damit ist es möglich ohne größeren Implementationsaufwand zwischen realem und virtuellem CAN-Bus zu wechseln. Virtuell bedeutet, dass die CAN-Nachrichten nicht auf den CAN-Bus geschrieben werden, sondern von der Bridge direkt über den TT-Ethernet Backbone weitergeleitet werden. So ist es möglich, Applikationen, die zuvor an einen CAN-Bus angeschlossen waren, direkt an den Ethernet-Switch anzuschließen. Nähere Informationen zur Umsetzung dieses Moduls finden sie im Abschnitt Implementation (5.3).

5.2 Implementiertes Bridgedesign

In diesem Abschnitt werden die umgesetzten Designentscheidungen dargelegt und begründed, anhand derer die letztendliche Implementierung der Bridge durchgeführt wurde. Dabei gleicht der Aufbau dem vorherigen Kapitel über die Diskussion zu den Konzepten einer Bridge.

5.2.1 Arbitrierung

Auf eine gemeinsame Arbitrierung über die Bridge hinweg wird verzichtet. Das heißt, dass die Teile eines logischen Busses unabhängig voneinander sind. Die Vorteile, die eine gemeinsame Arbitrierung haben würde, stehen in keiner sinnvollen Relation zu der verbrauchten Bandbreite. Auch müsste für eine gemeinsame Arbitrierung der bereitgestellte CAN HAL und Protocoll Stack angepasst werden, damit ein bitweiser Zugriff auf den Bus möglich wird. Das eine systemweite Arbitrierung nicht nötig ist, wird durch das Design unseres Zielsystems sicher gestellt. Der Nachrichtenschedule ist so festgelegt, dass in einem logischen Bus keine Nachrichten paralell gesendet werden, bzw. sich gegenseitig blockieren.

5.2.2 Datenkapselung

Die Entscheidung über die Art der Datenkapselung wird wieder in Abhängigkeit zum bereitgestellten HAL und Protocoll Stack getroffen. Da dieser in einer C-Struktur den Identifier, die Länge des Payloads und den eigentlichen Payload zur Verfügung stellt, werden diese Daten in den Ethernet-Frame gekapselt und versendet. Der Payload des Ethernet-Frames hat dabei folgenden Aufbau.

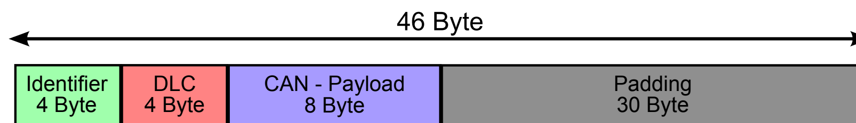

Abbildung 5.2: Aufbau des Ethernet Payload bei gekapselten CAN-Daten

Eine Kapselung des kompletten CAN-Frames wäre nur mit erheblich erhöhtem Implementationsaufwand zu verwirklichen und würde keine signifikanten Vorteile bringen.

5.2.3 Routing

Das Routing wurde als statisches Routing nach Verfahren 3 (Kapitel 4.3.3) mit einer Routingtabelle umgesetzt. Jede CAN-Nachricht im System hat einen zugehörigen virtuellen Link und wird über diesen übertragen. Unser Zielsystem unterliegt einem festen Nachrichtenschedule und ermöglicht eine problemlose Konfiguration der Routingtabelle und des Schedulers. Diese Methode gewährleistet zudem eine optimale Verwendung der Bandbreite und minimale Verzögerungszeiten.
Ein anderes Routingverfahren lässt sich mit überschaubarem Aufwand implementieren, sollten sich die Anforderungen an die Bridge ändern.

5.2.4 Nachrichtenklassen

Für die virtuellen Links wird bevorzugt die TT-Nachrichenklasse gewählt. Der Schedule sämtlicher Nachrichten im System ist bekannt und kann zur Designzeit konfiguriert werden. Die Verwendung von RC-Nachrichten, also eventbasiertem Verkehr, ist jedoch ohne Änderungen an der Implementierung möglich, da die Entscheidung welche der beiden Nachrichtenklassen verwendet wird, nur von der Konfiguration abhängig ist.
Die Integration von BE-Nachrichten in den Kommunikationsfluss ist nicht implementiert und wird für unser Zielsystem nicht benötigt. Bei wechselnden Anforderungen ist eine Implementation aber möglich. Innerhalb der Routingtabelle könnte dann noch zusätzlich konfiguriert werden, ob die Nachricht mittels einer Best-Efford-Nachricht übertragen werden soll.

5.2.5 Message Stuffing

Message Stuffing wird nicht implementiert. Diese Funktionalität wird für unser Zielsystem nicht benötigt, da zu keinem Zeitpunkt eine genügende Anzahl von Nachrichten über einen virtuellen Link übertragen werden muss, so dass sich aus einem Stuffing keine Vorteile erzielen ließen. Durch die Verzögerungen, die durch das Warten auf mehrere Nachrichten entstehen würden, würde es zudem zu Problemen in unserem Zielsystem führen, da die Nachrichten und die damit zusammenhängenden Tasks einem engen Schedule unterliegen.
Der wichtigste Grund für einen Einsatz von Message Stuffing, die effektivere Nutzung des TT-Ethernet Payloads und damit die Einsparung von Bandbreite, hat in unserem Zielsystem auch keine Relevanz, da das Nachrichtenaufkommen gering ist.
Eine nachträgliche Implementation von Message Stuffing ist möglich, sollte dies zu einem späteren Zeitpunkt benötigt werden. Auch in diesem Fall ist eine Konfiguration über die Routingtabelle denkbar, indem angegeben wird, wieviele Nachrichten zusammen übertragen werden sollen und die maximale Wartezeit festgelegt wird.

5.3 Implementation

Die Bridge wurde als eigenständiges C-Modul implementiert, welches in diesem Kapitel näher vorgestellt wird. Dazu wird zunächst die Konfiguration der Bridge vorgestellt, der Aufbau des Moduls und die Funktionen beschrieben und zum Schluss ein Überblick über den Programmablauf gegeben.

5.3.1 Bridge Modulbeschreibung

Das Modul besteht aus einer Sammlung von Funktionen, die die Aufgaben der Bridge nachbilden und 2 Header-Dateien, mit denen die Bridge konfiguriert wird.

Konfiguration

Die Konfiguration des Moduls erfolgt über 2 Konstanten und einer Struktur, die die Routinginformationen beinhaltet. Über die Konstante CAN_CHANNEL_NUMBER wird der Kommunikationskanal des Boards gewählt, mit dem der CAN-Bus verbunden werden soll. Die Konstante CAN_VIRTUAL definiert, ob das Board direkt mit dem CAN-Bus verbunden ist oder ob die CAN-Nachrichten über den Ethernet-Port versenden und empfangen werden sollen. Dieses verhalten wird im Kapitel Virtueller CAN-Stack (5.1.4) näher beschrieben. Diese Konfiguration wird in der *bridge_can.h* vorgenommen.
Das Routingverhalten der Bridge wird über folgende Struktur festgelegt und befindet sich in der Datei *bridge_routing.h*.

Listing 5.5: Bridge-Routing-Table

```
1  typedef struct {
2      /** CAN−Frame Identifier */
3      unsigned long can_id;
4
5      /** TTE−Msg Identifier */
6      uint16_t eth_id;
7
8      /** TTE−Msg Buffer */
9      tte_buffer_t buf;
10 } bridge_rtable_entry;
```

Mit dieser Struktur werden zwei Tabellen definiert. Für eingehnenden und ausgehenden Verkehr. Über die *can_id* und *eth_id* Einträge wird eine Beziehung zwischen einer CAN-Nachricht und einer TT-Ethernet-Nachricht hergestellt. Die CAN-Nachricht mit der ID X wird also immer über die TT-Ethernet-Nachricht mit der ID Y versendet. Der *buf* Eintrag ist ein Pointer auf den zugehörigen Buffer, der für Nachrichten mit dieser *eth_id* verwendet wird.

Die erste Tabelle ist die *bridge_rtable_tx*, in der gespeichert wird, auf welchen virtuellen Link die empfangenen CAN-Nachrichten abgebildet werden sollen. Das heißt auch, dass empfangene CAN-Nachrichten, die keinen Eintrag in dieser Tabelle besitzen, verworfen werden.

Die *bridge_rtable_rx* Tabelle ist das entsprechende Gegenstück für empfangene TT-Ethernet-Nachrichten. Der *can_id* Eintrag hat in dieser Tabelle aber keinen Einfluss auf das Verhalten der Bridge. Da sich der CAN-Identifier mit im Payload der TTE-Nachricht befindet, muss beim Empfang keine Abbildung mehr von TT-ID auf CAN-ID stattfinden. Die CAN-Nachricht kann sofort auf den Bus geschrieben werden. Die RX-Tabelle speichert nur, welche TTE-Nachrichten verarbeitet werden müssen und aus welchem Buffer diese zu lesen sind.

Funktionen

bridge_can
CAN_CHANNEL_NUMBER CAN_VIRTUAL BRIDGE_SUCCESS BRIDGE_FAILURE bridge_rtable_tx[]: bridge_rtable_entry bridge_rtable_rx[]: bridge_rtable_entry *bridge_virtual_can_isr(void): void
bridge_can_init(void): char bridge_route(unsigned long ulIdentifier): bridge_rtable_entry* bridge_can2eth(CAN_FRAME_T *tFrame): void bridge_reth_receive_callback(int pos): void bridge_can_send(CAN_FRAME_T *tFrame): void bridge_virtual_can_send(CAN_FRAME_T *tFrame): void bridge_can_isr_recv_low(void): void

Abbildung 5.3: Modulaufbau

bridge_can_init
Das Modul wird durch diese Funktion je nach Konfiguration in 2 verschiedenen Modi initialisiert. Der erste Modus initialisert den normalen Betrieb mit Anschluss an einen physischen CAN-Bus. Der zweite Modus verwendet den virtuellen CAN-Stack.
Bei Verwendung des normalen CAN-Moduls wird dieses auf den verwendeten Kommunikatinskanal und die Baudrat des Busses eingestellt. Es werden die Busereignisse gewählt, die einen Interrupt auslösen sollen und eine Interrupt-Service-Routine (ISR) für diese Interrupts festgelegt.
Diese Initialisierung des CAN-Moduls findet bei Verwendung des virtuellen CAN-Stack nicht statt. Stattdessen wird aus dem Vector-Interrupt-Controller (VIC) des Boards die Adresse der ISR ausgelesen, die durch ein anderes Modul des Boards festgelegt wurde. Mit dieser Adresse wird die korrekte Funktionsweise des virtuellen CAN-Stacks sichergestellt.
Im letzten Schritt werden die Input-Buffer des Ethernet-Moduls anhand der *bridge_rtable_rx* Tabelle so konfiguriert, dass beim Empfang eines TT-Ethernet-Frames die Callbackfuntion der Bridge für empfangene Ethernet-Frames ausgeführt wird.
bridge_route
Diese Funktion erhält als Parameter einen CAN-Identifier und durchsucht anhand diesem die Routingtabelle, ob für diesen Identifier ein Eintrag vorhanden ist. Bei Erfolg wird der Eintrag zurück gegeben.

bridge_can2eth
Mit dieser Funktion wird eine CAN-Nachricht über den TT-Ethernet Backbone gesendet. Als Parameter wird eine *CAN_FRAME_T* Struktur übergeben, die in den Datenbereich einer *tte_frame_t* Struktur kopiert wird. Diese Struktur, die einen TT-Ethernet Frame repräsentiert, wird in den durch die Routing-Tabelle bestimmten Output-Buffer geschrieben und durch das TT-Ethernet Modul versendet.

bridge_reth_receive_callback
Dies ist die Callbackfunktion, die vom TT-Ethernet Modul ausgeführt wird, wenn eine TTE-Nachricht empfangen wurde. Als Parameter wird die entsprechende Position aus der Routingtabelle übergeben, mit die der TT-Ethernet Frame in Form einer *tte_frame_t* Struktur aus dem Input-Buffer gelesen werden kann. Die CAN-Daten werden aus dieser Struktur in eine *CAN_FRAME_T* Struktur kopiert und mittels des CAN-Moduls auf den Bus geschrieben.

bridge_can_send
Diese Funktion nutzt die API des CAN-Moduls, um eine *CAN_FRAME_T* Struktur zu versenden. Mit dieser Funktion versendet die Bridge sämtliche CAN-Nachrichten.

bridge_virtual_can_send
Sollte die Bridge mit dem virtuellen CAN-Stack konfiguriert sein, werden die CAN-Nachrichten mit dieser Funktion über das virtuelle CAN-Modul versendet.

bridge_can_isr_recv_low
Dies ist die ISR, die vom CAN-Modul ausgelöst wird, sobald eine CAN-Nachricht empfangen wurde. Die Funktion liest die *CAN_FRAME_T* Struktur aus dem *Indication Low* Register und übergibt sie an die *bridge_can2eth* Funktion.

Programmablauf

In der Abbildung 5.4 wird eine Übersicht über den Programmablauf beim Senden und Empfangen gegeben und deutlich gemacht, in welchen Zusammenhang die implementierten Funktionen der Bridge stehen.

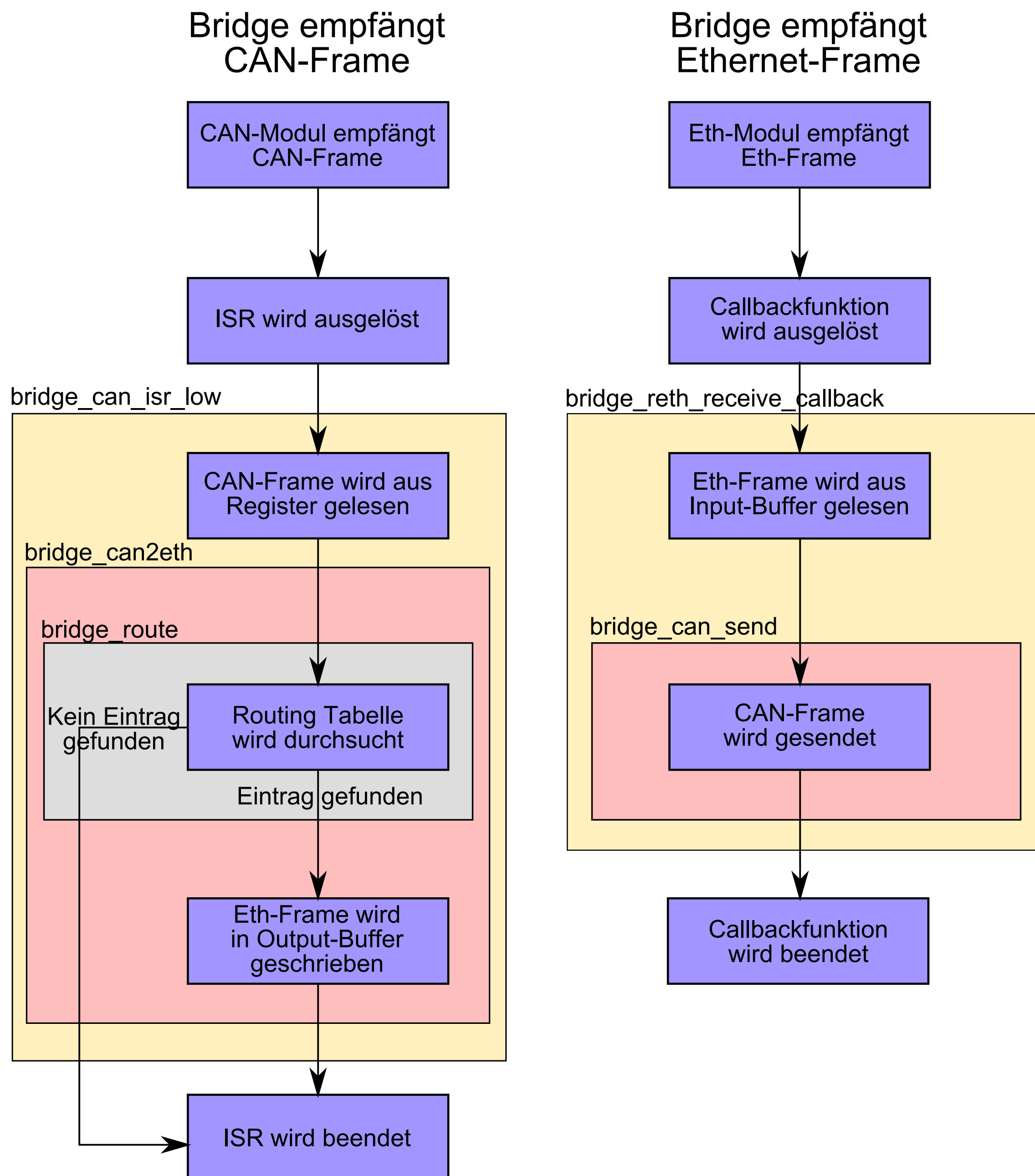

Abbildung 5.4: Programmablauf

5.3.2 Virtueller CAN-Stack

Der virtuelle CAN-Stack wird als eigenständiges Modul implementiert und nutzt das gleiche Interface, wie das von Hilscher vorgegebene CAN-Interface. Aktiviert wird das Modul, indem

man bei der Konfiguration der Bridge die Konstante CAN_VIRTUAL definiert.
Die Funktionen die zu implementieren sind, sind durch das CAN-Modul vorgegeben. Für den virtuellen Stack wird aber eine geringere Funktionalität benötigt. So müssen nur die Funktionen *Can_SendFrame*, *Can_GetInterruptCom*, *Can_GetIndicationLoFillLevel* und *Can_ReceiveLoFrame*, nachgebildet werden. *Can_SendFrame* nutzt intern die *bridge_can2eth* Funktion, um die CAN-Nachricht direkt übers Ethernet zu versenden. Die anderen Funktionen greifen auf simulierte Indication-FIFO-Register, die Input-Buffer des CAN-Moduls, zu. Auf diese Art wird der Zugriff auf empfangene CAN-Nachrichten nachgebildet. Diese simulierten Register sind als Ringbuffer implementiert, deren Größe über die Konstante CAN_FRAME_BUFFER_SIZE konfiguriert werden. Die anderen Funktionen des CAN-Moduls, die hier nicht aufgeführt wurden, werden durch leere Funktionskörper ersetzt. Die Handhabung der API bleibt unverändert.

Kapitel 6

Integration

Die Bridge wird in den Steer-by-Wire Demonstrator integriert, der von der CoRE (Communication over Real-time Ethernet) Projektgruppe entwickelt wird. Dieser Demonstrator besteht aus 4 Komponenten. Einem Lenkradmotor, einem Motor für das Lenkgetriebe, einem Kraftaufnehmer an der Lenkstange und einem Controller, der diese Komponenten zu einem Regelkreis verbindet. Alle Teilnehmer in diesem System sind CAN-basiert und werden mit Bridges über das TT-Ethernet zu einem Netzwerk verbunden. Dieses Kapitel verschafft dabei nur einen groben Überblick über den Demonstrator und zeigt, wie die Bridge im Kontext zu den anderen Komponenten eingesetzt wird. Eine ausführliche Beschreibung findet sich in der der Arbeit von Vitalij Stepanov (vgl. Vitalij Stepanov (2011)).

6.1 Aufbau des Demonstrators

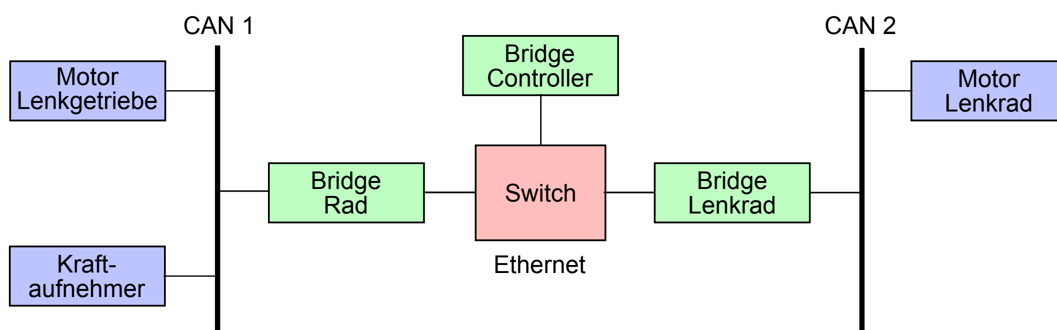

Abbildung 6.1: Aufbau des Demonstrators

Das System besteht nach der Integration des TT-Ethernet Backbones aus 3 Teilsystemen, die über einen 100 Mbit/s TT-Switch verbunden sind. Die CAN-Busse sind mit einer Baudrate von 1 Mbit/s konfiguriert. Das erste Teilsystem besteht aus dem CAN-Bus 1, an dem der

Motor und der Kraftaufnehmer für die Lenkstange hängen. Das zweite Teilsystem ist der CAN-Bus 2, über den der Motor für das Lenkrad angeschlossen ist. Beide Teilsysteme sind über eine Bridge mit dem Switch verbunden. Das dritte Teilsystem besteht nur aus einer Bridge, auf der auch der Controller läuft. Dieser nutzt das virtuelle CAN-Interface der Bridge um die CAN-Nachrichten direkt über das TT-Ethernet zu senden, bzw. zu empfangen.

Für jeden CAN-Identifier wird ein virtueller Link eingerichtet, über den die Nachrichten über das TT-Ethernet übertragen werden. Diese virtuellen Links werden mit der ID konfiguriert, die auch die zugehörige CAN-Nachricht hat. Dies erleichtert die Konfiguration. Die CAN-Nachricht mit der ID 0x200 wird demnach über den virtuellen Link mit der ID 0x200 versendet. Nochfolgende Abbildung gibt einen Überblick über die virtuellen Links. Die TT-Nachrichten sind in roter Schriftfarbe und die RC-Nachrichten in blauer Farbe abgebildet.

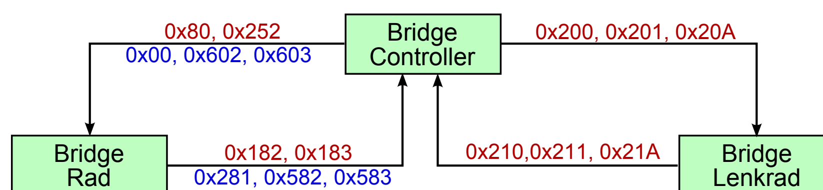

Abbildung 6.2: Virtuelle Links im Demonstrator-System

6.2 Konfiguration

Nachdem ein Überblick über Aufbau des Demonstrators und die verwendeten virtuellen Links gegeben wurde, beschreibt dieser Abschnitt wie die Bridges konfiguriert werden. Dabei wird hier nur die Konfiguration der Bridge des Lenkrades als Beispiel aufgeführt. Die komplette Konfiguration befindet sich auf der beiliegenden CD.

config_demonstrator_main.h
In dieser Datei wird über Konstanten festgelegt, mit welchen Konfigurationsdateien die Bridge konfiguriert wird. Die Konstante BRIDGE_STEER setzt die Zugehörigkeit der Bridge zum Lenkrad und über die Konstante CAN_VIRTUAL wird das gewünschte CAN-Modul gewählt.
config_steer.h
In der config_steer.h befinden sich die Tabellen zur Konfiguration der Nachrichten und des Schedulers. Nähere Informationen zur Umsetzung dieser Konfiguration kann den Arbeiten von Kai Müller und Vitalij Stepanov entnommen werden.
bridge_routing.h
Innderhalb dieser Datei befinden sich die Routingtabellen *bridge_rtable_tx* und

bridge_rtable_rx für die Bridge. Der folgende Codeausschnitt zeigt diese Tabellen für den CAN-Bus, an den das Lenkrad angeschlossen ist.

Listing 6.1: Routingtabelle für die Bridge am CAN-Bus 2

```
static bridge_rtable_entry bridge_rtable_tx[] = {
        {0x210, 0x210}, //{CAN-ID, TT-ID}
        {0x211, 0x211},
        {0x21A, 0x21A}
};

static bridge_rtable_entry bridge_rtable_rx[] = {
        {0x200, 0x200},
        {0x201, 0x201},
        {0x20A, 0x20A}
};
```

In der ersten Spalte der TX-Tabelle stehen die CAN-IDs, die über den virtuellen Link mit der ID aus der zweiten Spalte versendet wird. Die Einträge sind jeweils gleich, weil die IDs für die virtuellen Links so gewählt wurden, um die Konfiguration der Routingtabellen zu erleichtern. Die RX-Tabelle ist entsprechend gegengesetzt zu interpretieren.

Kapitel 7

Verifikation und Zeitmessung

Um die Funktionsfähigkeit der Bridge zu gewährleisten, wurden mehrere Testläufe durchgeführt und die Ergebnisse mit geeigneten Diagnosewerkzeugen überprüft. In diesem Kapitel wird der Versuchsaufbau beschrieben und die verwendeten Werkzeuge vorgestellt. Abschließend wurde noch eine Zeitmessung durchgeführt, mit der gemessen wurde, wie lange die Bridge intern für die Verarbeitung der Nachrichten braucht.

7.1 Verifikation

Für den Test der Bridge wurde ein einfacher Versuchsaufbau gewählt, der die Anzahl der Komponenten im Testsystem klein hält und so mögliche Fehlerquellen ausschließt. So wurde zum Beispiel auf die Verwendung des TT-Ethernet Switches verzichtet, um Fehler in der Konfiguration auszuschließen.

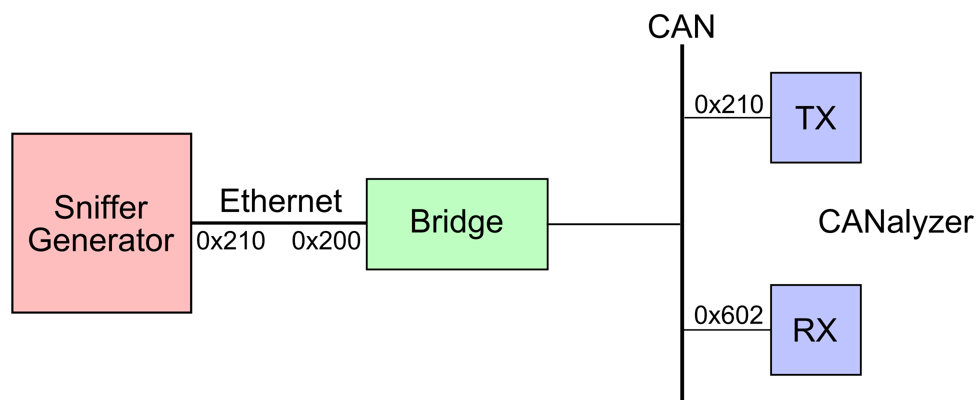

Abbildung 7.1: Aufbau des Testsystems

Der Kern des Testsystems bildet die Bridge. An den CAN-Bus wird ein CANalyzer angeschlossen. Dabei handelt es sich um ein Software-Analysewerkzeug für serielle Bussyste-

me, die in diesem Testsystem auf einem Notebook installiert ist. Neben dem reinen Bus-Monitoring bietet diese Software auch Möglichkeiten selber frei konfigurierbare Nachrichten zu senden. Um parallel Nachrichten empfangen und senden zu können, verfügt der CANalyzer über zwei Anschlüsse an den CAN-Bus, einen RX- und einen TX-Kanal. Dies ermöglicht ein durchgängiges Überwachen des Busses (vgl. Vector Informatik GmbH (2011)).
Über den TT-Ethernet Port der Bridge wird direkt ein PC angeschlossen, auf dem ein Packet-Sniffer und ein Packet-Generator installiert ist. Als Sniffer kommt Wireshark zum Einsatz. Wireshark erfasst den Netzwerkverkehr auf Paketbasis und stellt die Paketdaten detailiert dar (vgl. Wireshark Foundation (2011)). Als Generator wird der Colasoft Packet Builder verwendet, mit dem Ethernet-Packets erstellt und versendet werden können. Dabei kann nicht nur der Payload gewählt werden, sondern auch die Source und Destination Adresse des Pakets (vgl. Colasoft Co., Ltd. (2011)).
Wie aus der Abbildung zum Aufbau des Testsystems zu erkennen ist, werden folgende virtuelle Links und Routingtabellen konfiguriert. Mit dem CANalyzer werden CAN-Nachrichten mit der ID 0x210 versendet, die über die Bridge mit dem virtuellen Link 0x210 an den Packet-Sniffer gesendet werden. Mit dem Packet-Builder werden CAN-Nachrichten mit der ID 0x602 über den virtuellen Link mit der ID 0x200 gesendet.

Listing 7.1: Routingtabelle des Testsystems

```
1 static bridge_rtable_entry bridge_rtable_tx[] = {
2         {0x210, 0x210} //{CAN−ID, ETH−ID}
3 };
4 
5 static bridge_rtable_entry bridge_rtable_rx[] = {
6         {0x602, 0x200} //{CAN−ID, ETH−ID}
7 };
```

Der Scheduler wird dementsprechend für das Senden und Empfangen der virtuellen Links 0x210 und 0x200 konfiguriert. Die Deadlines für empfangene Nachrichten werden deaktiviert, da es sich bei unserem Testsystem um kein synchronisiertes System handelt. Synchronisation wird für den Funktionsnachweis der Bridge nicht benötigt, da das eigentliche Weiterleiten der Nachrichten unabhängig vom Scheduling ist.
Im ersten Versuch versendet der CANalyzer pro 1 ms eine CAN-Nachricht. Die Periode des Schedulers wird passend auf 1 ms eingestellt.

Time	Chn	ID	Dir	DLC	Data	Counter	Diff time
14.920289478	1	210	Tx	8	aa bb cc dd ee ff 11 22	14920	0.000260244

Abbildung 7.2: Ausschnitt CANalyzer

Die Abbildung 7.2 zeigt einen Ausschnitt aus der CANalyzer Oberfläche, auf dem man erkennt, dass 14920 CAN-Nahrichten mit der ID 0x210 und dem Payload 0xAABBCCDDE-

EFF1122 versendet wurden. In der nächsten Grafik 7.3 ist der Ausschnitt aus dem Wireshark Packet-Sniffer zu sehen, der den empfangenen Verkehr aufgezeichnet hat. Zu sehen ist die Source-Adresse 00::10:20:30:40:50 der Bridge und die Destination-Adresse, an die das Paket gerichtet ist. Diese besteht aus dem TT-Marker 03:04:05:06, mit der das Paket als TT-Nachricht markiert ist und der TT-ID 02:10, die der ID des virtuellen Links 0x210 entspricht. Das Type Feld enthält den Wert 0x88D7, der für ein Time-Triggered-Ethernet Paket steht. Im Datenfeld befinded sich die CAN-Nachricht. Man erkennt die CAN-ID 0x210, den DLC Wert 0x08 und den Payload 0xAABBCCDDEEFF1122. Die restlichen Daten sind Padding-Bytes, um die TT-Ethernet Nachricht auf die Mindestgröße von 64 Bytes zu bekommen.

No.	Time	Source	Destination	Protocol	Info
14777	14.924065	00:10:20:30:40:50	03:04:05:06:02:10	TTE 0x88d7	Bogus TTEthernet Frame
14778	14.925065	00:10:20:30:40:50	03:04:05:06:02:10	TTE 0x88d7	Bogus TTEthernet Frame

```
⊞ Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
⊟ TTEthernet
  ⊞ Destination: 03:04:05:06:02:10 (03:04:05:06:02:10)
    Source: 00:10:20:30:40:50 (00:10:20:30:40:50)
    Type: Unknown (0x88d7)
⊟ Data (46 bytes)
    Data: 1002000008000000aabbccddeeff1122e09e55d09f300306...
    [Length: 46]
```

CAN-ID DLC Payload

Abbildung 7.3: Ausschnitt Wireshark

Auffällig ist, dass weniger Pakete empfangen, als gesendet wurden. Es fehlen 142 Pakete. Dies lässt sich damit erklären, dass die Zeitgeber im CANalyzer und in der Bridge nicht synchron laufen, dass heißt, 1 ms ist im CANalyzer kürzer als in der Bridge. Dadurch kann es vorkommen, dass der CANalyzer 2 Nachrichten innerhalb einer Periode der Bridge sendet. Die Bridge verwirft dann die zweite CAN-Nachricht. Bei einem Test mit einer Periode von 900 µs in der Bridge hat sich gezeigt, dass alle Pakete vom Sniffer erfasst werden.

Um die Bridge auf einen Empfang einer TT-Ethernet Nachricht zu testen, wurde mit einem Packet-Builder ein eine TTE-Nachricht erzeugt und periodisch versendet. Die folgenden Abbildungen 7.4 und 7.5 demonstrieren den erfolgreichen Test.

No.	Time	Source	Destination	Protocol	Info
3283	55.771172	11:22:33:44:55:66	03:04:05:06:02:00	TTE 0x88d7	Bogus TTEthernet Frame
3284	55.786772	11:22:33:44:55:66	03:04:05:06:02:00	TTE 0x88d7	Bogus TTEthernet Frame

```
⊞ Frame 1: 27 bytes on wire (216 bits), 27 bytes captured (216 bits)
⊟ TTEthernet
  ⊞ Destination: 03:04:05:06:02:00 (03:04:05:06:02:00)
    Source: 11:22:33:44:55:66 (11:22:33:44:55:66)
    Type: Unknown (0x88d7)
⊟ Data (13 bytes)
    Data: 02060000080000002b4060000f
    [Length: 13]
```

Abbildung 7.4: Ausschnitt Wireshark

Der Auszug aus Wireshark zeigt, dass 3284 TTE-Packets über den virtuellen Link 0x200 versendet wurden. Im Payload stehen die Daten für eine CAN-Nachricht mit der ID 0x602 und den Nutzdaten 0x2B4060000F000000. Die mit dem CANalyzer empfangenen Daten bestätigen das diese TTE-Packets von der Bridge erfolgreich weitergeleitet wurden.

Time	Chn	ID	Dir	DLC	Data	Counter	Diff time
82.546085933	1	602	Rx	8	2b 40 60 00 0f 00 00 00	3284	0.015600638

Abbildung 7.5: Ausschnitt CANalyzer

7.2 Zeitmessung

Während, wie im vorherigen Kapitel beschrieben, die Synchronisation und das Scheduling für die eigentliche Funktion der Bridge nicht wichtig ist, wird für die Vorhersagbarkeit des Gesamtsystems trotzdem ein korrektes Scheduling benötigt. Um dieses erstellen zu können, müssen unter anderem die Laufzeiten der Vorgänge innerhalb der Bridge bekannt sein, um sie mit in die Planung einbeziehen zu können. In diesem Abschnitt werden die dazu vorgenommenen Zeitmessungen beschrieben und die Ergebnisse vorgestellt.
Als Versuchsaufbau wurde der Steer-by-Wire Demonstrator gewählt. Die Messungen wurden an der Bridge durchgeführt, über die das Lenkrad am TT-Ethernet angeschlossen ist. An dieser Bridge werden pro Periode 6 Vorgänge durchgeführt. Es müssen 3 CAN-Nachrichten über das TT-Ethernet weitergeleitet und 3 TT-Ethernet Nachrichten empfangen werden. Um die Messung durchführen zu können, wurde der GPIO Pin 13 des Mikrocontrollers verwendet. Dieser wurde so konfiguriert, dass er standardmäßig den Pegel *High* führt. Beim Eintritt in die gemessene Routine wird dieser Pegel auf *Low* gesetzt, um ihn am Ende wieder auf

High zu legen. Gemessen wird dieser Pegel mit einem Oszilloskop. Die folgenden Abbildungen 7.6 und 7.7 zeigen Ausschnitte aus den Messergebnissen.

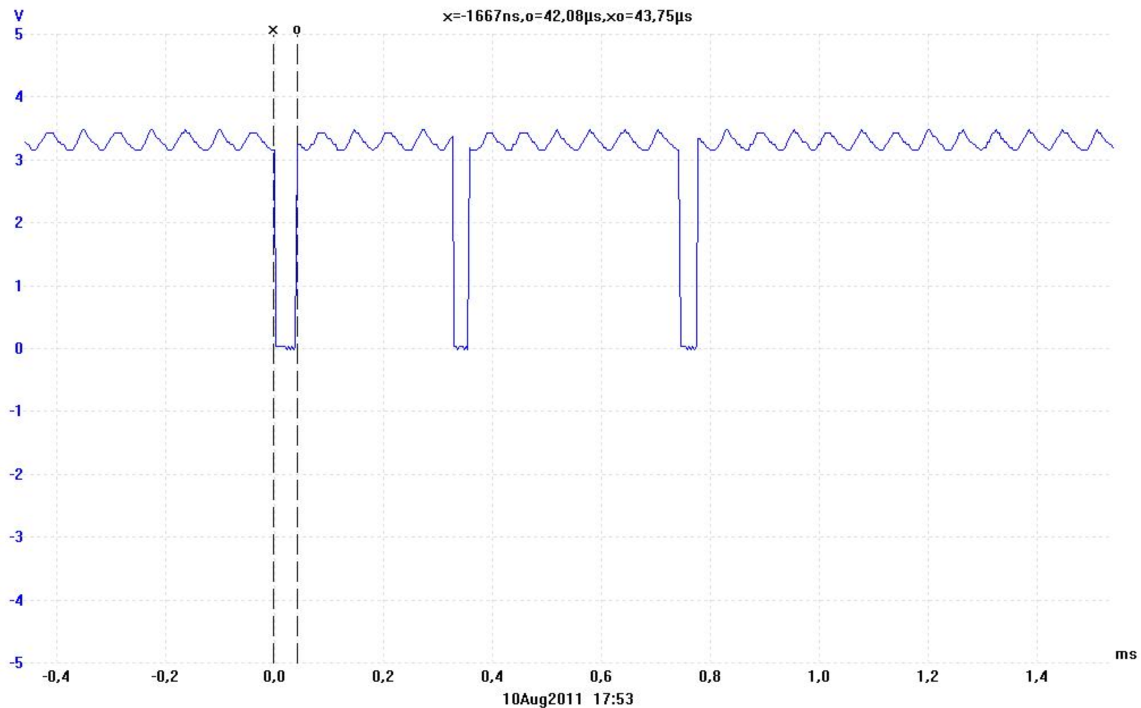

Abbildung 7.6: Zeitmessung der CAN Routine

Nach dem Empfang einer CAN-Nachricht benötigt die Bridge 35 bis 45 µs, bis die Daten In den TT-Ethernet Outputbuffer geschrieben sind. Umgekehrt vergehen nach dem Empfang einer TT-Ethner Nachricht 25 bis 35 µs, bis die Daten auf den CAN-Bus geschrieben werden. Abschließend wird die Zeit berechnet, die benötigt wird, um eine CAN-Nachricht von einem Bus A über TT-Ethernet an einen Bus B zu senden. Die CAN-Nachricht habe eine Länge von 154 Bit und die CAN-Busse haben eine Baudrate von 1 Mbit/s. Die TT-Ethernet Nachricht hat eine Länge von 512 Bit und wird mit 100 Mbit/s übertragen. Die Gesamtzeit berechnet sich somit aus den folgenden Teilzeiten.

$$T_{gesamt} = T_C + T_{BCE} + T_E + T_S + T_E + T_{BEC} + T_C \tag{7.1}$$

T_C ist die Zeit, die benötigt wird, bis die von einem Steuergerät gesendete CAN-Nachricht über den CAN-Bus übertragen wurde. T_{BCE} ist die Zeit, die die erste Bridge benötigt, um die empfangene CAN-Nachricht als TT-Ethnernet Nachricht zu versenden. T_E bezeichnet die Zeitspanne, die gebraucht wird, um diese Nachricht über das TT-Ethernet zu senden, T_S ist die Verzögerungszeit innerhalb des Switches und wird mit $T_S = 20\mu s$ angenommen. T_{BEC} ist die Zeit die vergeht, bis die zweite Bridge die Daten wieder auf den CAN-Bus schreibt.

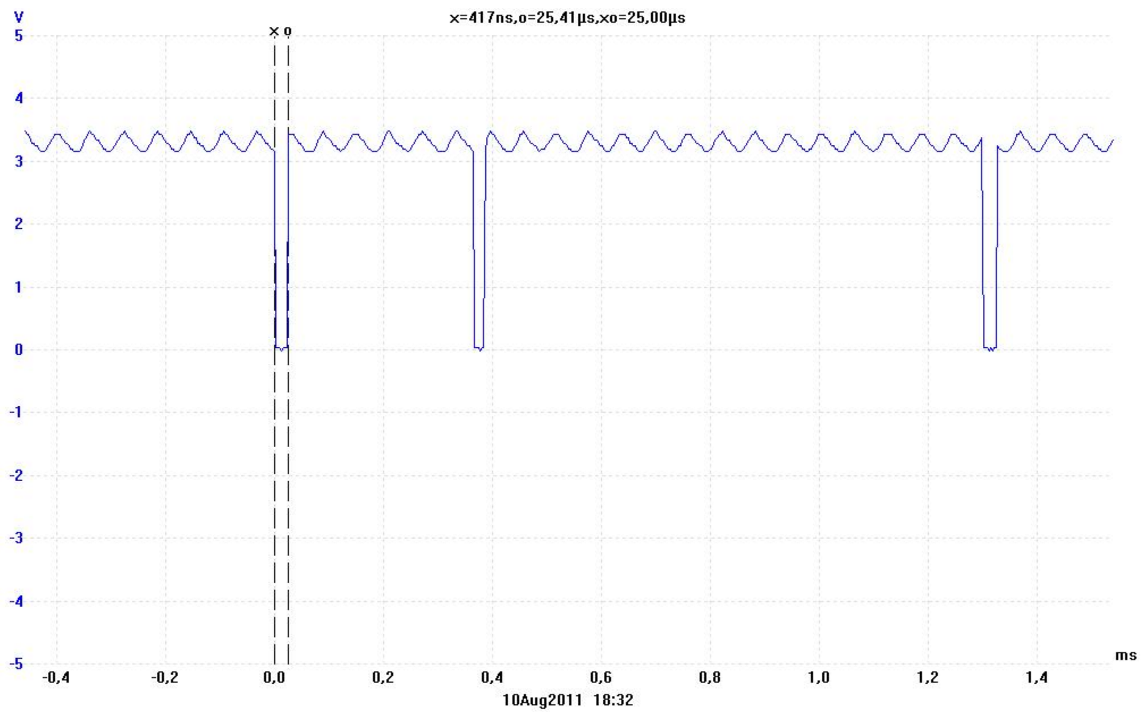

Abbildung 7.7: Zeitmessung der TT-Ethernet Routine

In diesem Bus wird dann wieder die Zeit T_C benötigt, bis die Steuergeräte die Nachricht komplett empfangen haben. Aus diesen Zeiten lässt sich nun die Gesamtzeit berechnen.

$$\begin{aligned} T_{gesamt} &= \frac{154Bit}{1Mbit/s} + 45\mu s + \frac{512Bit}{100Mbit/s} + 20\mu s \\ &+ \frac{512Bit}{100Mbit/s} + 35\mu s + \frac{154Bit}{1Mbit/s} \\ T_{gesamt} &= 154\mu s + 45\mu s + 5,12\mu s + 20\mu s + 5,12\mu s + 35\mu s + 154\mu s \\ T_{gesamt} &= 418,24\mu s \end{aligned} \tag{7.2}$$

Kapitel 8

Fazit

Im Rahmen dieser Bachelor-Thesis sollte eine Möglichkeit erarbeitet werden, um CAN-Busse über ein Time-Triggered Realtime Ethernet Netzwerk zu koppeln. Ein besonderes Augenmerk sollte dabei auf Bandbreiteneffizienz und Echtzeitfähigkeit gelegt werden, um den Ansprüchen an ein modernes Kommunikationsnetzwerk im Automobilbereich gerecht zu werden.

Um diese Anforderungen erfolgreich umsetzen zu können, wurden zunächst die erforderlichen Grundlagen zusammengetragen. Dazu gehörten die Themenbereiche Time-Triggered Ethernet, Controller Area Network und Bridgedesign.
Mit diesen Kenntnissen wurden dann die Designentscheidungen diskutiert, die bei der Implementation einer solchen Bridge zu beachten sind. In diesem Zusammenhang wurden zu den Aspekten Arbitrierung, Routing, Nachrichtenklassen und Message Stuffing mehrere Möglichkeiten vorgestellt und Vor- und Nachteile zusammengetragen.
Es wurde der Schluss gezogen, dass die gewünschten Ziele am besten ohne eine gemeinsame Arbitrierung zu erreichen sind und das ein statisches Routing mit Routingtabellen am meisten Effizienz im Bereich der Bandbreite garantiert. Durch das Routing wird auch die Vorhersagbarkeit und damit die Echtzeitfähigkeit des Systems gewährleistet, da durch das festzulegende Scheduling der Nachrichten sämtlicher Netzwerkverkehr zur Entwurfszeit festgelegt wird. Zu den Punkten Nachrichtenklassen und Message Stuffing wurden einfache Lösungen gewählt, die eine einfache Implentation ermöglichen und trotzdem zu keinen Nachteilen führen. Für die Bridge werden nur Time-Triggered Nachrichten verwendet, die jeweils nur eine CAN-Nachricht in ihren Payload aufnehmen. Für einen flexibleren Einsatz der Bridge wurde zusätzlich ein virtuelles CAN-Interface entworfen, welches es auf der Bridge laufenden Apllikationen ermöglicht, ohne Änderungen in der Implementation entweder auf den angeschlossenen CAN-Bus zu senden oder die CAN-Nachricht direkt über das TT-Ethernet an einen entfernten CAN-Bus zu übertragen.
Mit diesem Design wurde die Bridge schließlich umgesetzt und die durchgeführte Imple-

mentation beschrieben. Dabei wurde erklärt, wie die Bridge in ein bestehendes und komplett CAN-basiertes System integriert wird und wie die Konfiguration vorzunehmen ist. Zusätzlich wurde der Steer-by-Wire Demonstrator vorgestellt und wie die Integration in diesem System vorgenommen wurde.
Abschließend wurde durch einen einfachen Testaufbau sicher gestellt, dass die implementierte Bridge fehlerfrei arbeitet und die konfigurierten Routingtabellen im Betrieb korrekt umgesetzt werden. Um das Scheduling der Tasks und Nachrichten so präzise wie möglich zu erstellen, wurde noch eine Zeitmessung der Bridge durchgeführt. Zudem wurde die maximale Zeit berechnet, die eine CAN-Nachricht im System benötigt um über das Time-Triggered Ethernet von CAN-Bus zu CAN-Bus übertragen zu werden.

Mit der Integration der Bridge in den Steer-by-Wire Demonstrator wurde erfolgreich demonstriert, dass ein Einsatz im Automobilbereich möglich und durch die erhöhte Bandbreite und einfachere Verkabelung sinnvoll ist. Die derzeitige Implementation bietet alles, um bestehende Systeme zu erweitern und neuere Konzepte wie die X-by-Wire Systeme umzusetzen.
Dennoch hat die Umsetzung noch Bereiche für Verbesserungen und bietet Ansätze für Erweiterungen. Als Beispiel sei hier der Einsatz von Best-Efford Nachrichten im TT-Ethernet genannt, um CAN-Daten zu übertragen, die nicht zeitkritisch sind. In diesen Bereich fallen zum Beispiel Daten zur Analyse und Auswertung von Steuergeräten, die zur Erfassung von telemetrischen Daten nötig sind.

Literaturverzeichnis

[Aeronautical Radio Incorporated 2009] AERONAUTICAL RADIO INCORPORATED: Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network / ARINC. 2009 (ARINC Report 664P7-1). – Standard

[Colasoft Co., Ltd. 2011] COLASOFT CO., LTD.: *Colasoft Packet Builder*. 2011. – URL http://www.colasoft.com/packet_builder/. – Zugriffsdatum: 2011-08-20

[Communication over Real-time Ethernet (CoRE)] COMMUNICATION OVER REAL-TIME ETHERNET (CORE): *Communication over Real-time Ethernet*. – URL http://www.informatik.haw-hamburg.de/core.html

[GE Fanuc Intelligent Platforms 2009] GE FANUC INTELLIGENT PLATFORMS: *TTEthernet - A Powerful Network Solution for Advanced Integrated Systems*. August 2009. – URL http://www.ge-ip.com/library/detail/12014. – Zugriffsdatum: 2011-01-18. – GFT-751

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

[International Organization for Standardization 2003] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION: Controller Area Network (CAN) / ISO. Genf, 2003 (11898). – ISO

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

[Lawrenz und Obermöller 2011] LAWRENZ, Wolfhard ; OBERMÖLLER, Nils: *Controller Area Network: Grundlagen, Design, Anwendungen, Testtechnik*. Bd. 5. VDE Verlag GmbH, 2011

[Leen und Heffernan 2002] LEEN, Gabriel ; HEFFERNAN, Donal: Expanding Automotive Electronic Systems. In: *Computer* 35 (2002), S. 88 – 93

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

[Lipfert 2008b] LIPFERT, Jan: *The Insider's Guide to netX*. Hilscher GmbH. Dezember 2008. – URL http://www.hilscher.com

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

[Scharbarg u. a. 2005] SCHARBARG, Jean-Luc ; BOYER, Marc ; FRABOUL, Christian: CAN-Ethernet Architectures for Real-Time Applications / IRIT ENSEEIHT. 2005. – Forschungsbericht

[Seidel 2009] SEIDEL, Fred: X-by-Wire / Chemnitz University of Technology. 2009. – Forschungsbericht

[Tanenbaum 2003a] TANENBAUM, Andrew S.: *Computernetzwerke*. Pearson Studium, 2003. – ISBN 978-3-8273-7046-4

[Tanenbaum 2003b] TANENBAUM, Andrew S.: *Verteilte Systeme Grundlagen und Paradigmen*. Pearson Studium, Oktober 2003. – ISBN 978-3-8273-7057-4

[Thomas Noltet and Hans Hanssont and Lucia Lo Bello 2006] THOMAS NOLTET AND HANS HANSSONT AND LUCIA LO BELLO: Automotive Communications - Past, Current and Future / Malardalen Univ., Vasteras. 2006. – Forschungsbericht

[TTTech Computertechnik AG]

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

[Vector Informatik GmbH 2011] VECTOR INFORMATIK GMBH: *Steuergeräte-Analyse mit CANalyzer*. 2011. – URL http://www.vector.com/vi_canalyzer_de.html. – Zugriffsdatum: 2011-08-20

[Vitalij Stepanov 2011] VITALIJ STEPANOV: Mikrocontroller und CAN-basierte verteilte Regelung einer Steer-by-Wire Lenkung mit harten Echtzeitanforderungen / . 2011. – Forschungsbericht

[Wireshark Foundation 2011] WIRESHARK FOUNDATION: *Wireshark*. 2011. – URL http://www.wireshark.org/. – Zugriffsdatum: 2011-08-20

Inhalt der beiliegenden CD

/**bridge**/ Das Bridge-Modul

/**pdf**/ Die Arbeit als PDF-Dokument

/**item[/steer_by_wire_demonstrator**/ Das komplette Steer-by-Wire Projekt

Versicherung über Selbstständigkeit

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

Hamburg, 22. August 2011

Ort, Datum	Unterschrift