Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Jan-Erik Lange

Entwicklung eines modularen Analysators für proprietäre, Ethernet-basierte Kommunikationsprotokolle

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Jan-Erik Lange

Entwicklung eines modularen Analysators für proprietäre, Ethernet-basierte Kommunikationsprotokolle

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Thomas Lehmann
Zweitgutachter : Prof. Dr. rer. nat. Jochen Schneider

Abgegeben am 28. Februar 2011

Jan-Erik Lange

Thema der Bachelorthesis

Entwicklung eines modularen Analysators für proprietäre, Ethernet-basierte Kommunikationsprotokolle

Stichworte

Ethernet, Xilinx, Cypress, Wireshark, Protokoll, Analyse

Kurzzusammenfassung

Bei der Firma Airbus Operations wurde ein proprietäres Kommunikationsprotokoll für ein Kabinenmanagement-System entwickelt. Für die Entwicklung von Geräten, die mit diesem Protokoll arbeiten, ist es nötig, eine Möglichkeit zu schaffen, den Datenverkehr mitzulesen und zu analysieren.

Es wurde ein Konzept erarbeitet, in dem dargestellt wird, welche Hardware- und Software-Komponenten für die Entwicklung des Analyse-Systems zum Einsatz kommen können. Infolge dessen wurde mit Hilfe des Xilinx Spartan 3A Starterkit und des Cypress FX2 Mikrocontroller ein Hardwaresystem für das Einlesen eines Datenstroms aus dem proprietären Kommunikationssystem entwickelt. Die Analyse der Daten erfolgt mit Hilfe des Programms Wireshark. Die individuelle Protokoll-Analyse wurde durch die Entwicklung von Wireshark-Dissectors realisiert. Um die proprietären Kommunikationsprotokolle mit Wireshark analysieren zu können, wurde ein Konvertierungsprogramm entwickelt, das die Roh-Daten in das Wireshark-konforme pcap-Dateiformat konvertiert.

Jan-Erik Lange

Title of the paper

Development of a modular analyzer for proprietary, Ethernet based communication protocols

Keywords

Ethernet, Xilinx, Cypress, Wireshark, protocol, analysis

Abstract

A proprietary communication protocol of a cabin management has been developed at the company Airbus Operations. For the development of devices using this protocol, a way needs to be created to read and to analyse the data traffic in the cabin management system.

A concept has been worked out, which hardware and software components can be used for the development of the data analysis system. As a result a hardware system for reading the datastream from the proprietary communication system has been developed. This hardware system consists of the Xilinx Spartan 3A starter kit and the Cypress FX2 microcontroller. The analysis of the data has been realized by using the program Wireshark. The individual protocol analysis has been realized by the development of Wireshark dissectors. To analyse the proprietary communication protocols with Wireshark, a conversion program is neccessary, which also has been developed in this thesis.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Vorstellung des Unternehmens

Airbus begann 1985 mit der Entwicklung eines integrierten Kabinenmanagement-Systems, dem Cabin Intercommunication Data System (CIDS). Dieser Bereich wuchs sehr schnell, sodass im Jahre 1991 daraus der Standort in Buxtehude entstand. 1995 wurde KID-Systeme als ein Profitcenter mit Sitz im Standort Buxtehude gegründet.

Am Standort entwickelte Produkte, die an andere Flugzeughersteller wie z.B. Boeing verkauft werden, tragen die Marke KID-Systeme. 1999 wurde KID-Systeme als 100%ige Tochter von Airbus eine eigene Firma mit Sitz im Standort Buxtehude. In 2005 wurde das Werk in Buxtehude reintegriert. Die KID Systeme GmbH blieb in verkleinerter Größe parallel am Standort bestehen.

1.2. Cabin Intercommunication Data System

Das CIDS ist der Kern der Kabinenelektronik aller Airbus-Flugzeuge. Es betreibt und überwacht die verschiedenen Passagier- und Crew-Funktionen. Zu diesen Funktionen gehören z.B. ein Kommunikationssystem für das Kabinenpersonal, die Kabinenbeleuchtung, Füllstandsanzeige der Wasser-/Abwassertanks, die Regulierung der Kabinentemperatur und weitere Überwachungsfunktionen.

1.2.1. Komponenten des Cabin Intercommunication Data System

Das CIDS ist ein zentral gesteuertes System mit mehreren redundanten Servern (Director). Das CIDS besteht je nach Größe des Flugzeugs aus einer bestimmten Anzahl von Directors, die das Herzstück des Systems sind. Sie verarbeiten die Signale der gesamten Peripherie. Zu der Peripherie zählen die verschiedenen Sensoren und Komponenten, die durch das CIDS gesteuert werden. Zwischen der Peripherie und des Directors gibt es Netzknoten, die Decoder Encoder Units (DEU), die die Signale, die zum Director führen oder vom Director ausgehen, umsetzen. Als einfachstes Beispiel wird in den DEUs eine A/D- beziehungsweise D/A-Umsetzung der Signale durchgeführt.

Ein Flight Attendant Panel (FAP) dient dabei der Statusanzeige diverser Funktionen des CIDS. Außerdem lassen sich dort Einstellungen wie Boarding Music[1] und Beleuchtungseinstellungen vornehmen. Die Abbildung 1.1 zeigt einige der zum CIDS gehörenden Komponenten.

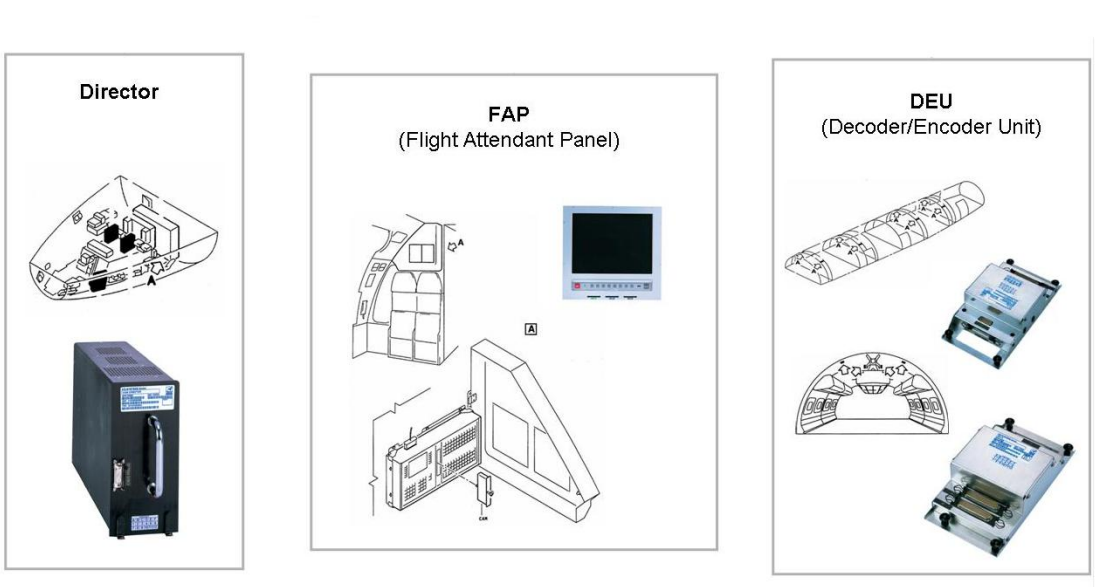

Abbildung 1.1.: Die Komponenten des CIDS

[1] Boarding Music, engl.: Musik, welche abgespielt wird, während die Passagiere einsteigen oder das Flugzeug verlassen.

1.2.2. Die Datenbusse Middle- und Top-Line

Der Director kommuniziert über zwei Typen von Datenbussen mit der Elektronik-Peripherie, die Middle- und Top-Line genannt werden (siehe Abbildung 1.2). Im Middle-Line-Datenbus werden Informationen übertragen, die die Funktionen des Kabinenpersonals betreffen. Eine DEU, die an die Middle-Line angeschlossen ist, wird auch DEU-B genannt. Im Top-Line-Datenbus werden Informationen übertragen, die die passagierbezogenen Funktionen betreffen. Eine DEU, die an die Top-Line angeschlossen ist, wird auch DEU-A genannt.

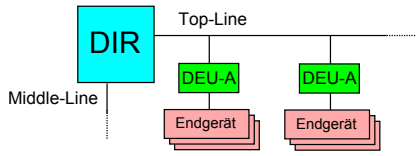

Abbildung 1.2.: Aufbau des CIDS

1.3. CIDS Protokoll

Das Kommunikationsprotokoll, mit dem in der Middle- und Top-Line Informationen ausgetauscht werden, ist ein proprietäres Echtzeit-Protokoll. Auf der physikalischen Schicht wird eine Ethernet Variante mit einer Geschwindigkeit von 10 Mbit/s benutzt. Das Zeitverhalten dieses Protokolls orientiert sich an der Samplingrate der Audio-Daten, die über den Datenbus geschickt werden. Die Samplingrate beträgt 32 kS/s.

1.4. Hybrides Protokoll

Um den Datenbus des CIDS leistungsfähiger zu machen, wird als physikalische Schicht nun Ethernet 100BASE-TX verwendet, das eine Datenrate von 100 Mbit/s besitzt. Das Kommunikationsprotokoll wird um einen Nicht-Echtzeit-Teil erweitert, da in der selben Zeit mehr Daten übertragen werden können.

Das hybride Protokoll orientiert sich nach wie vor an der Sampling-Rate der Audio-Daten und übertragt die Frames des alten Protokolls mit derselben Frequenz von 32 kHz. Diese Frames werden in diesem Zusammenhang als Echtzeit-Frames bezeichnet. Neben den Echtzeit-Frames werden Daten getunnelt, die nicht die Anforderungen haben, in Echtzeit übertragen zu werden.

Das Tunneln der Frames wird mit einem Multiplexer realisiert, der abwechselnd die Echtzeit-Frames und die Nicht-Echtzeit-Frames auf den Datenbus schaltet. Das Verfahren, mit dem der Multiplexer arbeitet, ist das synchrone Zeitmultiplexverfahren Time Division Multiple Access (TDMA). Beim synchronen TDMA wird dem Sender der Echtzeit-Frames und dem Sender der Nicht-Echtzeit-Frames ein jeweils fester Zeitabschnitt zur Übertragung zur Verfügung gestellt.

Mit der Änderung des physikalischen Übertragungsmediums hat sich auch die Bus Topologie des CIDS verändert. Die DEUs sind nun über ein Daisy-Chain-Netzwerk verbunden, da Ethernet 100BASE-TX eine Punkt-zu-Punkt Verbindung vorsieht. Aus historischen Gründen wird im Folgenden aber weiter von einem Bussystem gesprochen. Der Aufbau des Netzwerks ist in Abbildung 1.3 zu erkennen.

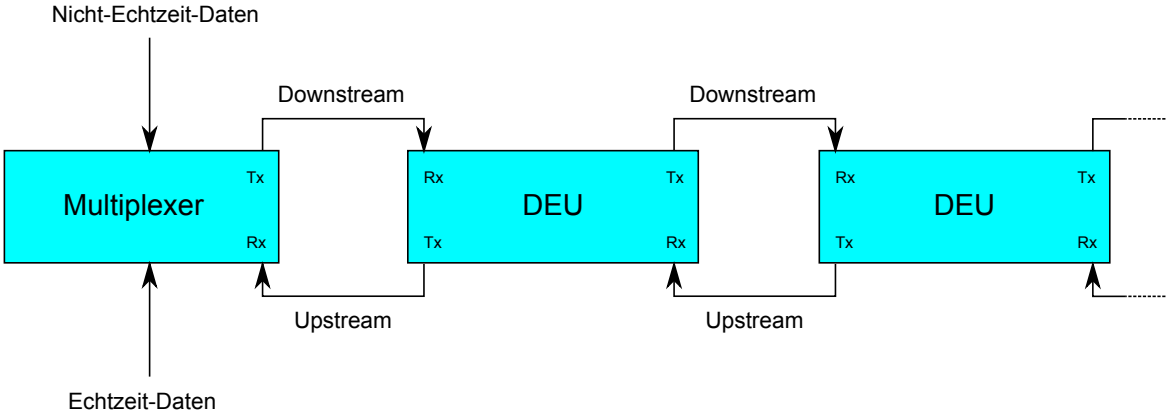

Abbildung 1.3.: Aufbau des hybriden Datenbus

1.5. Darstellung des Themas

Für die Fehlersuche bei der Entwicklung von Geräten, die das hybride Protokoll verwenden, ist es hilfreich, die auf dem Netzwerk gesendeten Daten mitlesen und analysieren zu können. Ein Analyse-System soll für einen bestimmten Zeitraum Daten auf der physikalischen Schicht von Ethernet 100BASE-TX mitlesen und diese auf einem PC-System zur weiteren Analyse bereitstellen. Auf dem PC-System sollen die Daten interpretiert und übersichtlich dargestellt werden können. Das Analyse-System soll modular aufgebaut werden, um es leicht an unterschiedliche Protokolle anpassen zu können.

1.6. Grundlagen zu Ethernet 100BASE-TX

Im Folgenden wird über die Grundlagen von Ethernet 100BASE-TX berichtet, da das hybride Protokoll diese Übertragungstechnik auf der physikalischen Ebene verwendet. Die Sachverhalte lassen sich anschaulich mit Hilfe des Open-Systems-Interconnection-Schichtenmodells (OSI-Schichtenmodell) erklären. Das OSI-Schichtenmodell beschreibt die verschiedenen Abstraktionsebenen der Kommunikation zwischen Netzwerkteilnehmern. Für Ethernet 100BASE-TX werden die Schichten von einer Projektarbeitsgruppe mit dem Namen „Institute of Electrical and Electronics Engineers 802.3“ (IEEE 802.3) definiert.

Die für das Verständnis dieser Arbeit wichtigen Sachverhalte begrenzen sich auf die ersten beiden Schichten dieses Modells. Sie lauten PHYSICAL Layer (physikalische Schicht) und DATA LINK Layer (Sicherungsschicht) und werden im Folgenden beschrieben.

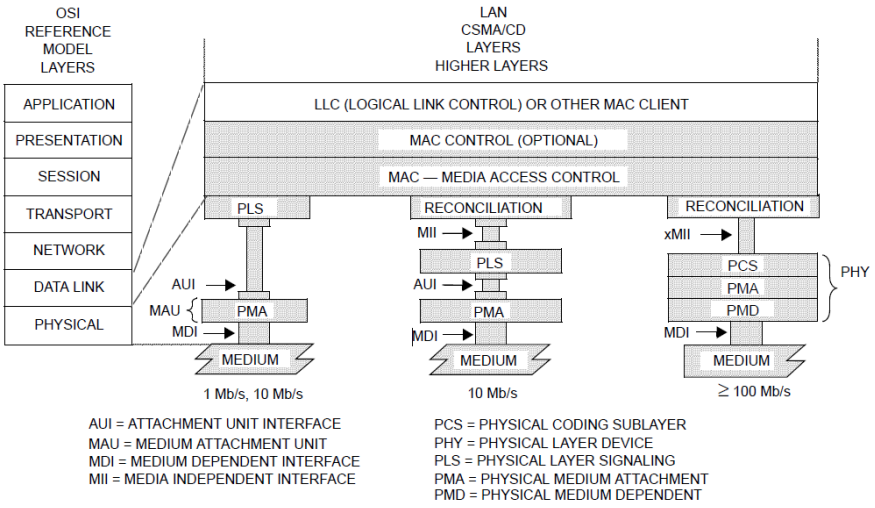

Abbildung 1.4.: OSI Schichtenmodell (vgl. [IEEE 802.3], S. 77)

1.6.1. Übertragungsmedium

Noch unter der physikalischen Schicht wird das Medium definiert, auf dem die Übertragung stattfindet. Im Fall von 100BASE-TX werden geschirmte oder ungeschirmte Twisted-Pair-Kabel[2] (STP oder UTP) der Kategorie 5 verwendet. Falls das gesamte Kabel noch einmal geschirmt ist, lautet die Bezeichnung S/STP und S/UTP-Kabel. Das S steht für „screened“[3] und beudeutet, dass das gesamte Kabel zusätzlich von einem Drahtgeflecht umhüllt ist. Die Grenzfrequenz dieser Leitungen beträgt 100 MHz.

Der Wellenwiderstand dieser Leitungen beträgt 100 Ω. Die Leitungen müssen also sender- und empfängerseitig mit 100 Ω abgeschlossen werden, um Reflexionen auf der Leitung zu vermeiden und Leistungsanpassung zu erreichen.

Die Kabel enthalten 4 Adernpaare, von denen bei 100BASE-TX je ein Pärchen für das Empfangen und ein Pärchen für das Versenden verwendet wird. Die weiteren Adernpaare werden bei 100BASE-TX nicht genutzt. Über ein Adernpaar werden die Signale differentiell übertragen. Diese Art der Übertragung macht die Übertragung des Signals störungstolerant gegenüber Einwirkungen von elektromagnetischen Feldern (vgl. [Erich Stein (2008)]).

1.6.2. Physikalische Schicht

In den drei Schichten Physical Coding Sublayer (PCS), Physical Medium Attachement (PMA) und Physical Medium Dependent (PMD), die in der Abbildung des OSI-Schichtenmodells (siehe Abbildung 1.4) zu dem Begriff PHY zusammengefasst wurden, wird die Übertragung des Digitalsignals im Basisband definiert. Unter anderem werden die verwendeten Leitungscodierungen definiert, die nun beschrieben werden.

Um bei der Taktregeneration im Empfänger lange Eins- und Null-Folgen zu vermeiden, wird die 4B/5B-Codierung verwendet. Bei dem Codiervorgang werden vier Nutzdatenbits auf fünf Codebits abgebildet. Durch das Einfügen eines weiteren Bit, erhöht sich die Datenrate um den Faktor 5/4 gegenüber dem uncodierten Signal. Dies bedeutet auch, dass sich das Signalspektrum zu hohen Frequenzen verschiebt. Die Taktrate beträgt auf der Leitung durch die Erhöhung der Datenrate um den Faktor 5/4 daher 125 MHz.

Eine Analyse des Spektrums des Signals ergibt bei 125 MHz die größten Spektralanteile des Signals. Wie bereits erwähnt, hat das Übertragungsmedium aber eine Grenzfrequenz von 100 MHz. Das Signal würde deshalb ohne weitere Maßnahmen verzerrt werden. Um dem entgegenzuwirken, wird zusätzlich die MLT-3 Leitungscodierung verwendet. Dies ist ein dreistufiger Code mit den Pegeln -1 V, 0 V und +1 V.

[2] twisted pairs, engl.: verdrillte Paare
[3] screened - engl.: abgeschirmt

Beim Codiervorgang entspricht eine logische Eins einem Pegelwechsel. Bei der Übertragung einer Null wird kein Pegelwechsel durchgeführt. Ein direkter Übergang zwischen -1 V und +1 V und umgekehrt ist nicht erlaubt. Wenn zum Beispiel vier Einsen hintereinander gesendet werden, ändert sich der Spannungspegel auf der Leitung nach dem Schema +1 V, 0 V, -1 V, 0 V. Dies ist der Extremfall, in dem sich das Signal am schnellsten ändert. Dieser Zyklus dauert vier Bit-Längen. Aufgrund dieser Tatsache reduziert sich das Signalspektrum auf der Frequenzachse auf 25% des 4B/5B-Signalspektrums. Nun befinden sich die größten Spektralanteile bei 31,25 MHz und das Signal wird durch das Übertragungsmedium nicht mehr verzerrt (vgl. [Klaus Dembowski (2007)] S. 137).

An der xMII Schnittstelle werden der nächstgelegenen Sicherungsschicht die übertragenen Nutz-Daten und Steuersignale bereitgestellt. Im Fall 100BASE-TX gibt es das MII und RMII. Beim MII wird ein 4-Bit breiter Datenstrom bei einer Taktfrequenz von 25 MHz zur Verfügung gestellt. Beim RMII beträgt die Breite des Datenstromes 2-Bit und die Taktfrequenz 50 MHz.

1.6.3. Sicherungsschicht

Die Sicherungsschicht ist in zwei Unterschichten aufgeteilt. Diese lauten Media Access Controll (MAC) und Logical Link Control (LLC). In diesen beiden Schichten wird im Wesentlichen der Zugriff auf das Übertragungsmedium geregelt und eine Daten-Kapselung durchgeführt.

Zugriffssteuerung auf das Übertragungsmedium

Laut IEEE 802.3 werden bei Ethernet 100BASE-TX zwei Betriebsmodi zur Verfügung gestellt: Halb-Duplex und Voll-Duplex. Im Halb-Duplex Modus sendet nur ein Teilnehmer zur Zeit. Dieser Modus wird bei Ethernet 100BASE-TX nicht mehr häufig verwendet, da es getrennte Sende- und Empfangsleitungen gibt, sodass zwei Teilnehmer gleichzeitig senden können. Hingegen wird bei 10BASE2 zur Übertragung nur ein Koaxialkabel verwendet, auf dem nur ein Teilnehmer zur Zeit senden kann.

Sende- und Empfangsstation müssen beim Halb-Dublex die Erlaubnis, auf der Leitung zu senden, deshalb untereinander aushandeln. Dies wird durch den CSMA/CD-Algorithmus realisiert. Von einer näheren Betrachtung dieses Algorithmus wird abgesehen, da er für Ethernet 100BASE-TX nicht mehr von wesentlicher Bedeutung ist.

Im Voll-Duplex Modus können die Sende- und Empfangsstation gleichzeitig Senden und Emfangen. Laut IEEE 802.3 müssen drei Vorraussetzungen erfüllt sein, damit die Sende- und Empfangsstation in diesem Modus arbeiten darf:

1. Auf dem physikalischen Medium ist es möglich, gleichzeitig senden und empfangen zu können.
2. Es gibt maximal zwei Stationen im Netzwerk. So wird eine Punkt-zu-Punkt Verbindung zwischen den Stationen erreicht. Ein Algorithmus, wie CSMA/CD ist hier deshalb unnötig.
3. Beide Stationen sind fähig, im Voll-Duplex-Modus zu arbeiten.

Daten-Kapselung:

Eine weitere Aufgabe der Sicherungs-Schicht ist die Daten-Kapselung. Der Standard definiert ein MAC-Paket, das in Abbildung 1.5 zu erkennen ist. Die Felder dieses Pakets werden im Folgenden erklärt.

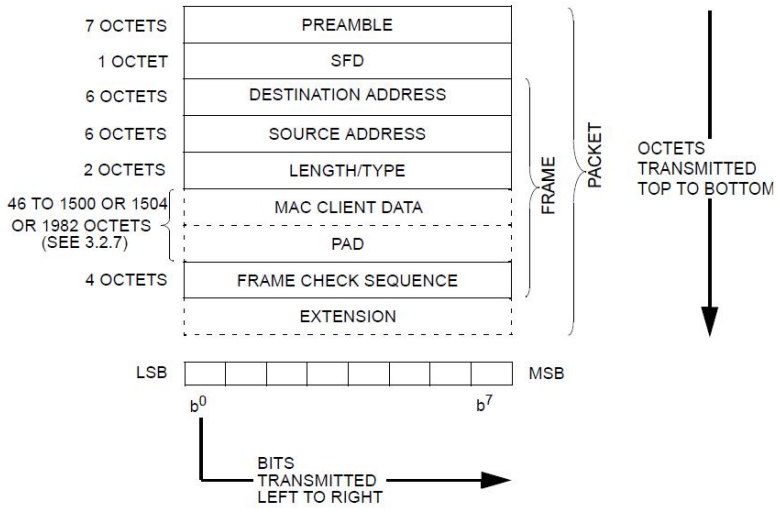

Abbildung 1.5.: Aufbau eines Ethernet konformen Packets (vgl. [IEEE 802.3], S. 123)

- PREAMBLE: Die Präambel dient der Taktrückgewinnung in der Empfängerschaltung. Die Sequenz hat die Form `101010 101010`. Diese Sequenz wird sieben Mal hintereinander zu Beginn jedes Frames gesendet.
- SFD: Nach der Preambel wird die Sequenz `101010 101011` geschickt. Nach diesem Feld beginnt definitionsmäßig der Frame.
- DESTINATION ADDRESS, SOURCE ADDRESS: Dieses Feld gibt an, an welchen Teilnehmer der Frame adressiert ist und von wem der Frame kommt. Diese Adresse, die verwendet wird, wird als MAC Adresse bezeichnet. Dies ist eine Hardware-Adresse, die jeder Ethernet-Controller besitzt.

- LENGTH/TYPE: Dieses Feld kann zwei Bedeutungen haben. Sie sind abhängig von dem Wert, der in dem Feld steht.
  1. Wenn der Wert gleich oder größer ist als 1536 beziehungsweise hexadezimal x0600, wird dieses Feld als Typen-Feld behandelt.
  2. Ist der Wert kleiner als 1536, wird das Feld als Längen-Feld interpretiert. Es gibt an, wie viele Byte das folgende Datenfeld enthält.
- MAC CLIENT DATA: Dieses Feld enthält das Datenfeld. Es gibt drei verschiedene Datenfeldtypen:
  1. Einfache Frames, maximale Länge: 1500 Bytes
  2. Q-tagged frames, maximale Länge: 1504 Bytes
  3. Envelope Frame, maximale Länge: 1982 Bytes
- PAD: Dieses Feld dient zum Auffüllen eines Ethernet-Frame auf eine minimale Framelänge. Die minimale Framelänge definiert sich durch den im Standard definierten Parameter „minFrameSize = 512 Bits“. Dies ist für die korrekte Funktion der Buszugriffssteuerung CSMA/CD nötig.
- FCS: Dieses Feld enthält eine 4 Byte lange CRC32-Prüfsumme, die von der Sendestation des Frame berechnet wurde. Diese Prüfsumme wird über den gesamten Framen, abzüglich Preambel- und SFD-Feld, berechnet. Eine exakte Berechnungsvorschrift zur Berechnung dieser CRC32-Prüfsumme ist in der Quelle [IEEE 802.3] auf Seite 125 zu finden.
- EXTENSION: Dieses Feld dient ebenfalls zum Auffüllen des Frame. Anders als beim PAD-Feld wird das Paket hier bis zu dem Parameter „slotTime“ aufgefüllt. Bei Ethernet-Technologien spielt dieses Feld aber keine Rolle, da die Parameter „slotTime“ und „minFrameSize“ hier gleich sind. Bei 1 Gb-Ethernet beträgt der Parameter „minFrameSize“ 512 Bits und der Parameter „slotTime“ 4096 Bits. Hier würde das Paket entsprechend aufgefüllt werden.

2. Anforderungen

Der Inhalt dieses Kapitels stellt die Anforderungen an den Protokollanalysator dar. Das Kapitel ist in zwei Teile aufgeteilt. Im ersten Teil werden die Anforderungen an die Hardware behandelt. Im zweiten Kapitel werden die Anforderungen an die Analyse-Software dargestellt.

2.1. Anforderungen an die Hardware

Um den Datenstrom mit einem Computer analysieren zu können, muss zunächst eine Busankopplung an den Datenbus geschaffen werden. Im Folgenden werden die Anforderungen diese Ankopplung dargestellt.

2.1.1. Anforderungen an die Busankopplung

Vermeidung zeitlicher Belastung: Der hybride Datenbus darf maximal einer zeitlichen Verzögerung von $t_v < 1\mu s$ pro DEU ausgesetzt sein. Wäre die Verzögerung länger, dann käme es zu Fehlfunktionen im Bussystem. Mathematisch lässt sich das Verhalten des Ankopplungssystems mit der Gleichung

$$g(t) = s(t - t_v) \tag{2.1}$$

ausdrücken. Dabei ist $s(t)$ das Eingangssignal und $g(t)$ das Ausgangssignal.

Vermeidung von Reflektionen: Durch das Einfügen des Ankopplungsglieds in die Leitung darf der Wellenwiderstand der Leitung nicht verändert werden. Die Abschlusswiderstände an den Leitungsenden sind dann nicht gleich dem Wellenwiderstand, was dazu führt, dass es zu Reflexionen an den Leitungsenden kommt. Außerdem besteht dann keine Leistungsanpassung mehr, was sich beim Empfänger ebenfalls durch eine schlechte Signalqualität bemerkbar macht.

Ankopplung beider Übertragungsrichtungen: Die Ankopplung muss so realisiert werden, dass sich das Übertragungsmedium beider Übertragungsrichtungen ankoppeln lässt. Es ist aber nicht zwingend notwendig, dass beide Richtungen gleichzeitig ankoppelbar sind.

Off-the-shelf-Komponenten: Es sollten, wenn möglich, nur Off-the-shelf-Komponenten verwendet werden, um die Kosten gering zu halten.

2.1.2. Anforderungen an das Einlesemodul

Das Einlesemodul muss so beschaffen sein, dass es den Datenstrom aus dem Ankopplungszweig empfängt und zum Computer überträgt. Es muss auf der physikalischen Schicht von Ethernet 100BASE-TX arbeiten. Der gesamte Datenstrom muss ohne Einschränkungen analysierbar sein. Deshalb darf die Einlese-Hardware oder das Betriebssystem im Computer keine Teile des Datenstroms heraus filtern.

2.2. Anforderungen an die Analyse-Software

Modularität: Das Analyse-System muss modular aufgebaut sein, um es leicht an unterschiedliche Protokolle anpassen zu können.

Echtzeitanforderungen: Es ist nicht vorgesehen, dass die Analyse in Echtzeit durchführbar ist, weil auf dem Datenbus ein ständiger Informationsaustausch zwischen den DEUs und dem Director herrscht. Die Analyse dieser Datenmenge wäre in Echtzeit nicht durchführbar. Stattdessen sollen die empfangenen Daten zunächst in einer Datei zwischengespeichert werden.

Visualisierung der Daten: Durch eine grafische Benutzeroberfläche soll der Datenstrom im Detail analysierbar sein. Eine zusätzliche Darstellung in binär-Format ist erwünscht, damit der Benutzer einzelne Flags des Protokolls untersuchen kann, ohne zwischen Hex- und Binärformat umrechnen zu müssen.

Analysewerkzeuge zur Untersuchung des Datenstroms: Um einen bestimmten Vorgang bei einer Übertragung zu untersuchen, sind nur bestimmte Informationen von Interesse. Dazu wird eine Filterfunktion benötigt, mit der man sich durch Benutzervorgaben nur bestimmte Frames anzeigen lassen kann. Eine Sortierfunktion ist auch hilfreich.

Zeitinformationen zu den Frames: Eine Anforderung für die Analyse der Daten ist, dass Zeitinformationen zu den Frames zur Verfügung stehen, um das Zeitverhalten untersuchen zu können. Die Genauigkeit sollte mindestens 1 μs betragen.

Protokollerkennung: Die Auswahl, ob es sich bei den Rohdaten um einen Datenstrom handelt, der aus der Top-Line oder Middle-Line kommt und ob er auf dem Downstream- oder Upstream-Pfad mitgelesen wurde, soll durch den Benutzer konfigurierbar sein. Die Software dient der Analyse und der Fehlersuche. Daher ist eine Automatik bei eventuell fehlerhaften Frames nicht sinnvoll.

Im Datenbus des CIDS herrscht ein Broadcast von dem Director zu den DEUs. Alle 31,25 μs wird ein Echtzeit-Frame vom Direktor verschickt. Innerhalb von 1 s werden daher 32000 Echtzeit-Frames verschickt. Zwischen den Echtzeit-Frames werden im hybriden Datenbus zusätzlich Nicht-Echzeit-Frames verschickt. Bei der Analyse der Daten wäre es für den Benutzer des Analysesystems zu viel Aufwand manuell für jeden Frame festzulegen, ob er als Echtzeit- oder Nicht-Echtzeit-Frame interpretiert werden soll.

Innerhalb des empfangenen Datenstroms muss das Analysewerkzeug deshalb selbständig erkennen können, ob es sich um einen Nicht-Echtzeit-Frame oder einen Echtzeit-Frame handelt. Eine Automatik ist in diesem Fall nicht zu vermeiden. Eine automatische Erkennung der Frame-Typen setzt voraus, dass im Frame ein Indikator vorhanden ist, durch den die Automatik die Erkennung durchführt. Wenn der Indikator fehlerhaft ist, wird die Automatik nicht korrekt funktionieren.

Wie bereits erwähnt, dient die Software der Analyse und Fehlersuche eines Protokolls. Eine automatische Fehler-Detektion der empfangenen Frames des hybriden Protokoll ist daher nicht vorgesehen. Durch geeignete Tests muss deshalb sichergestellt werden, dass das Analyse-System den Datenstrom möglichst fehlerfrei einliest. Durch den modularen Aufbau des Systems soll es aber möglich sein, bei Bedarf die Software in der Art zu erweitern, dass eine Fehlererkennung durchführbar ist.

2.3. Recherche zu bestehenden Analyse-Systemen

Es gibt verschiedene Möglichkeiten zur Analyse von Kommunikationsnetzen. Für die Entwicklung dieses Analyse-Systems wurde deshalb eine Recherche zu bestehenden Analyse-Systemen durchgeführt, um feststellen zu können welche Art von Analyse-System die genannten Anforderungen erfüllt. Um einen Überblick über die Analyse-Arten zu erhalten, lässt sich das OSI-Schichtenmodell benutzen, mit dem die verschiedenen Arten logisch voneinander getrennt werden können.

2.3.1. Analyse eines Netzwerks auf der physikalischen Schicht

Zur Analyse eines Netzwerks auf physikalischer Schicht gibt es Testgeräte, die eine Bitfehlerratenmessung am Übertragungskanal durchführen können. Mit dem AXS-200/850 der Firma EXFO lassen sich zum Beispiel Bitfehlerraten- und Jittermessungen des Digitalsignals durchführen (vgl. [EXFO]). Mit dem gleichen Gerät lassen sich auch Messungen durchführen, ob die Sendestationen die Frame Check Sequence (FCS) korrekt berechnen. Diese Messungen entsprechen Analysen eines Netzwerks auf der Sicherungsschicht und der physikalischen Schicht.

2.3.2. Analyse eines Netzwerks ab der Vermittlungsschicht

Es gibt verschiedene Anwendungsfälle für die Analyse von Netzwerken ab der Vermittlungsschicht. Mit Hilfe von Last-Profilen lässt sich zum Beispiel anzeigen, welche Datenmenge je Zeiteinheit über die Datenleitung übertragen wird. Diese Analysemöglichkeit wird zum Beispiel zur Planung von Firmennetzwerken angewendet.

Zur Analyse von Sicherheitslücken in einem Computernetzwerk werden LAN-Analysen durchgeführt (vgl. [synapse]). Bei LAN-Analysen werden Sniffer[1]-Programme verwendet, um den Datenstrom grafisch darzustellen. Ein Beispiel für ein Sniffer-Programm ist Wireshark.

Wireshark stellt den Datenverkehr, den das Betriebssystem zur Verfügung stellt, grafisch dar. Den Datenverkehr erhält Wireshark dabei über Programmbibliotheken. Unter Windows lautet diese Bibliothek winpcap und unter Linux libpcap. Diese Bibliotheken werden auch capture-engines[2] genannt und ermöglichen Zugriff auf den Netzwerk-Datenverkehr. Die beiden Bibliotheken erhalten die Ethernet-Frames von dem jeweiligen Betriebssystem. Das Betriebssystem kommuniziert über einen Treiber mit der Netzwerkkarte. Das bedeutet, dass es von dieser Komponenten-Zusammensetzung abhängt, welche Daten sich mit dem Analyse-Programm anzeigen lassen.

Zusammenfassung: Die zu entwickelnde Analyse-Software ist vergleichbar mit einem Sniffer-Programm, mit dem die Analyse des Datenstroms möglich ist. Das zu entwickelnde Analyse-System soll Daten auf der physikalischen Schicht von Ethernet 100BASE-TX analysieren. Es ist zu prüfen, welche Komponenten-Zusammensetzung nötig ist, um den Datenstrom einzulesen.

[1] sniffer - engl.: Schnüffler
[2] to capture - engl.: empfangen

3. Konzept

Der Systemaufbau für den Protokollanalysator ist in Abbildung 3.1 zu sehen. In diesem Kapitel wird geprüft, welche Technologien und Bauteile für die jeweiligen Systemkomponenten in Frage kommen. Es werden Alternativen vorgestellt, die bewertet werden.

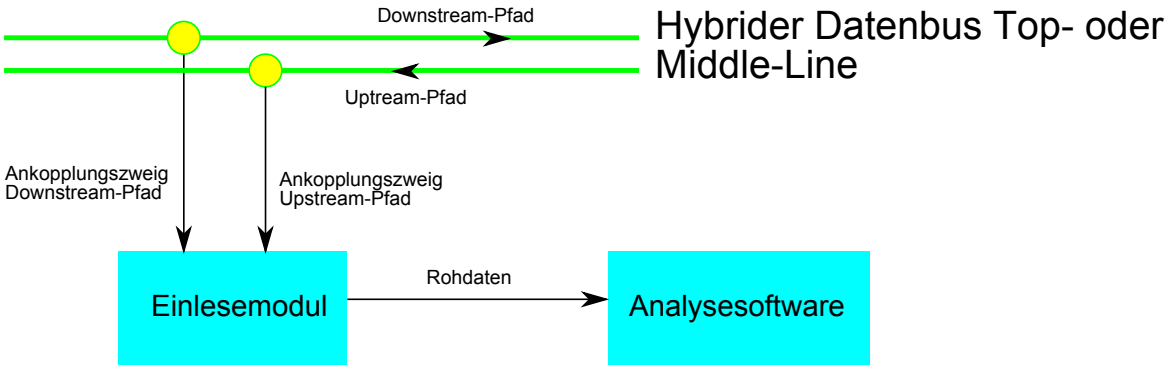

Abbildung 3.1.: Aufbau des Gesamtsystems

3.1. Busankopplung

3.1.1. Passive Ankopplung

Um die Busankopplung an das Bussystem zu realisieren, gibt es verschiedene Möglichkeiten. Die ideale Ankopplungsart ist eine rein passive Ankopplung, da dadurch keinerlei Zeitverzögerung entsteht. Bei der passiven Ankopplung wird für die Busankopplung ein Y-Kabel verwendet. Dabei wird ein weiteres Leitungsstück an die Leitung gelötet und an das Analyse-System angeschlossen.

Auswirkungen der Ankopplung mit einem Y-Kabel

Der Wellenwiderstand einer Cat5 UTP- oder STP-Leitung beträgt 100 Ω. Durch das Parallelschalten der Leitungen sieht die Leitung mit der Signalquelle am Ankopplungspunkt den halben Wellenwiderstand mit einem Wert von 50 Ω (Abbildung 3.2). Dadurch ist zu erwarten, dass Reflexionen auf der Leitung entstehen. Durch Reflexionen entsteht eine schlechte Signalqualität, was eventuell zur Folge hat, dass die Empfangsschaltung die Signale nicht mehr korrekt detektieren kann. In einer Messung muss die Signalqualität überprüft werden.

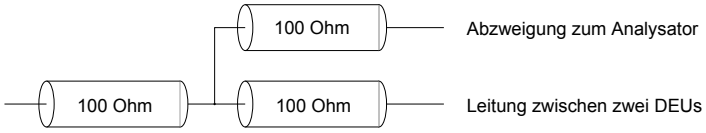

Abbildung 3.2.: Parallelschaltung von zwei Leitungen

Messungen am Y-Kabel

Um bei digitalen Signalen die Signalqualität messtechnisch erfassen zu können, verwendet man das Augendiagramm. Das Augendiagramm ist die Summe aller übereinander gezeichneten Ausschnitte eines Digitalsignals, deren Dauer jeweils gleich der Taktperiode ist. Um alle möglichen Übergänge zu erfassen, muss das Digitalsignal näherungsweise ein Zufallssignal sein.

Es wurde ein Augendiagramm für den Fall mit Abschlusswiderstand an der Abzweigeleitung (Abbildung 3.4) und für den Fall ohne Abschlusswiderstand (Abbildung 3.3) an der Abzweigung aufgenommen. Da kein differentieller Tastkopf zur Verfügung stand, wurde nur an einer der zwei Leitungen eines Adernpaares gemessen. Der Masse-Anschluss des Tastkopfes wurde mit der Masse der DEU verbunden.

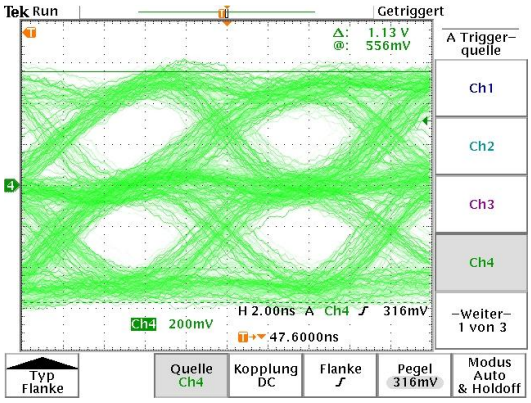

Abbildung 3.3.: Augendiagramm ohne Abschlusswiderstand an der Abzweigeleitung

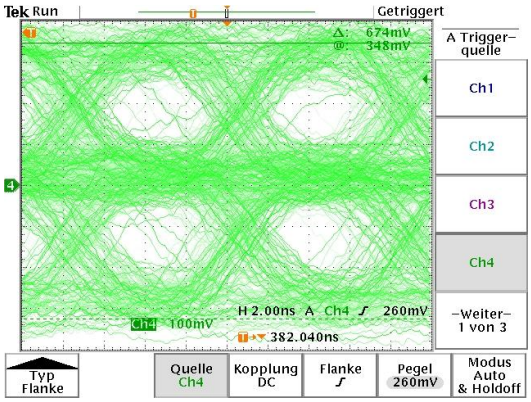

Abbildung 3.4.: Augendiagramm mit Abschlusswiderstand an der Abzweigeleitung

Messergebnis

In Abbildung 3.4 ist zu erkennen, dass das Auge aufgrund der schlechten Signalqualität weiter geschlossen ist als in Abbildung 3.3. Außerdem ist der Signalpegel mit dem Abschlusswiderstand kleiner. Das Hinzuschalten des Abschlusswiderstandes wirkt sich also negativ auf das zu analysierende Signal aus.

Überträgt man mittels der Personal-Announcement[1]-Funktion des CIDS, Audio-Daten, während die Ankopplungsleitung mit dem Abschlusswiderstand abgeschlossen ist, macht sich die schlechte Signalqualität insofern bei der Übertragung von Sprache bemerkbar, als ständig ein Knacken und Knistern zu hören ist. Diese Ankopplung verfälscht das zu analysierende Signal also so stark, dass es von Empfänger nicht mehr korrekt detektiert werden kann.

Kompensierung der Auswirkungen

Verhinderung des Pegeleinbruchs: Ein Grund für den Einbruch des Signalpegels könnte sein, dass der Innenwiderstand der Spannungsquelle nun nicht mehr mit dem Gesamtwiderstand der parallel geschalteten Abschlusswiderstände übereinstimmt. Das Parallelschalten bewirkt, dass der Gesamtwiderstand sinkt, die Spannungsquelle zu stark belastet ist und der Signalpegel deshalb einbricht.

Der Abschlusswiderstand der Abzweigeleitung müsste also um mindestens den Faktor 10 hochohmiger, als die übrigen Abschlusswiderstände sein, um die Signalquelle nicht so stark zu belasten. Dies wäre mit einem Impedanzwandler realisierbar, der im Frequenzbereich bis ungefähr 100 MHz ein nahezu lineares Verhalten aufweist.

[1] Personal-Announcement - engl.: Flugbegleiter-Durchsage

Vermindern der Reflexionen: Um die Reflexionen zu vermindern, muss eine Anpassung des Wellenwiderstandes durchgeführt werden. Da das Spektrum des Signals sehr breitbandig ist, lässt sich keine Stichleitung und kein einfacher $\lambda/4$-Transformator verwenden. Eine Möglichkeit wäre, einen Übertragerbaustein mit einem bestimmten Wicklungsverhältnis zu verwenden, um eine Impedanzanpassung zu erreichen (siehe Abbildung 3.5).

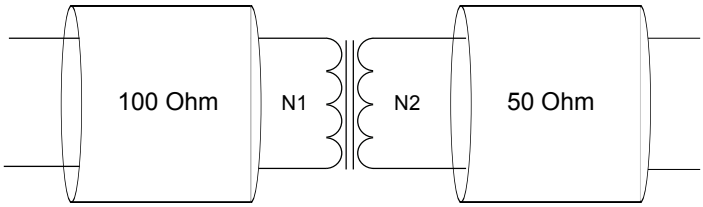

Abbildung 3.5.: Anpassung des Wellenwiderstandes mit Hilfe eines Übertragers

Um eine Widerstandsanpassung zu erreichen, müsste man ein Windungsverhältnis von

$$\frac{N_1}{N_2} = \sqrt{2} : 1 = 1,41 : 1 \tag{3.1}$$

verwenden. Dann sieht man am Abzweigepunkt eine Impedanz von

$$\frac{R_1}{R_2} = \left(\frac{N_1}{N_2}\right)^2 \rightarrow R_1 = R_2 \cdot \left(\frac{N_1}{N_2}\right)^2 = 50\Omega \cdot \left(\frac{1.41}{1}\right)^2 = 100\Omega. \tag{3.2}$$

Ein Nachteil ist, dass durch diese Anpassung die Spannung auf der Sekundärseite wiederum heruntertransformiert wird:

$$\frac{N_1}{N_2} = \frac{U_1}{U_2} \rightarrow U_2 = U_1 \cdot \frac{N_2}{N_1} = U_1 \cdot \frac{1}{1.41} = 0,709 \cdot U_1 \tag{3.3}$$

Dieses Verhalten birgt wiederum die Gefahr, dass die Empfänger das Signal nicht korrekt detektieren können. Die weitere Verfolgung dieses Ansatzes der Busankopplung wird nicht weiter verfolgt, da dies nicht den Schwerpunkt dieser Arbeit darstellt. Diese Ankopplungsart ist aufgrund der Verschlechterung der Signalqualität des Ethernet-Signals nicht geeignet.

3.1.2. Daisy-Chain Ankopplung

Eine Möglichkeit zur Ankopplung an den CIDS Bus ist, die Eigenschaft der Netzwerkarchitektur auszunutzen: Jedes Kettenglied reicht alle ankommenden Daten an das nächste Kettenglied weiter (Abbildung 3.6).

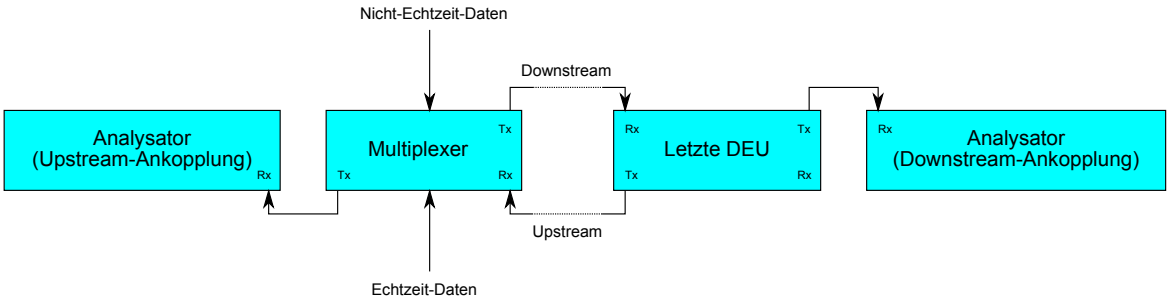

Abbildung 3.6.: Busankopplung mit der Daisy-Chain Methode

Die Ankopplung an den Downstream-Pfad ist über die letzte DEU in der Kette möglich, indem man den Ankopplungszweig an ihren Downstream-Tx-Port anschließt. Möchte man sich an den Upstream-Pfad ankoppeln, müsste der Datenstrom-Multiplexer das gleiche Verhalten aufweisen, wie eine DEU. Er müsste sich also wie ein Kettenglied im Daisy-Chain Netzwerk verhalten und die ankommenden Informationen an ein weiteres Kettenglied weiterleiten. Dieses Verhalten ist bis zum jetzigen Zeitpunkt jedoch nicht implementiert, sodass eine Ankopplung an den Upstream Pfad auf diese Weise nicht möglich ist.

3.1.3. Verwendung eines Ethernet 100BASE-TX Hub

Ein Ethernet Hub arbeitet auf der physikalischen Schicht. Er ist jedoch nicht Voll-Duplex-fähig. Würde man ihn in das Netzwerk hineinschalten, würden Kollisionen entstehen. Die Abbildung 3.7 beschreibt die Situation, in der eine Kollision entsteht.

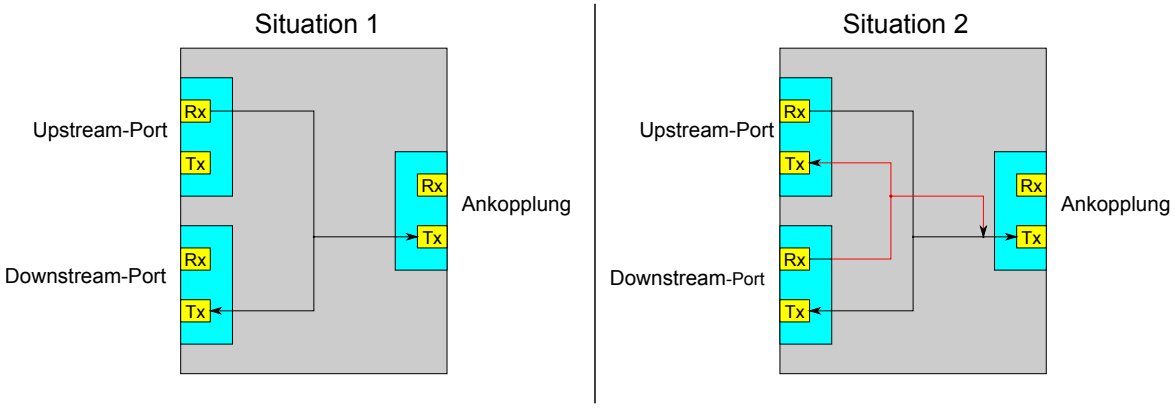

Abbildung 3.7.: Situation für eine Kollision im Hub

In Situation 1 werden an dem Port Daten empfangen, der an den Upstream-Pfad angeschlossen ist. Die Signale, die an dem Rx-Port anliegen, werden an alle weiteren Tx-Ports weitergeleitet. In Situation 2 werden an dem Downstream-Port ebenfalls Daten empfangen. Auch in diesem Fall werden die Signale an alle Tx-Ports weitergeleitet. Nun entsteht an dem Tx-Port des Abzweig-Ports eine Kollision der beiden Datenströme. Dieses Verhalten macht einen Hub für die Busankopplung unbrauchbar.

Anmerkung: Die Verwendung eines Ethernet-Switch schließt sich aus dem Grund aus, als er auf OSI-2 arbeitet und nicht-Ethernet-konforme Frames verwirft.

3.1.4. T-Stück mit Ethernet-Transceivern

Eine Möglichkeit, sich in das Netzwerk an zu koppeln, ist, zwei Ethernet-Transceiver zu benutzen, um die Daten mit dem einen Transceiver zunächst zu empfangen und anschließend mit dem anderen Transceiver weiter zu verschicken. Zusätzlich ist eine Verarbeitungseinheit nötig, die die Daten von Transceiver zu Transceiver überträgt und intern zwei Ankopplungszweige besitzt, um die Daten mitzulesen.

Die Abbildung 3.8 zeigt dieses Prinzip als Blockschaltbild. Dieses Element entspricht einem Kettenglied im Daisy-Chain-Netzwerk. Es entsteht eine Zeitverzögerung, die identisch mit der eines Daisy-Chain-Elements ist, wenn diese Verarbeitungseinheit genauso funktioniert, wie die einer DEU.

Die Anzahl der maximal möglichen DEUs in einer Kette ist auch abhängig von ihrer internen Verzögerung, daher verringert sich diese Zahl um 1 bei dieser Art der Ankoppplung. Die maximale Anzahl an DEUs im hybriden Bussytem beträgt 16, was bedeutet, dass maximal 15 DEUs in das Bussystem angeschlossen sein dürfen.

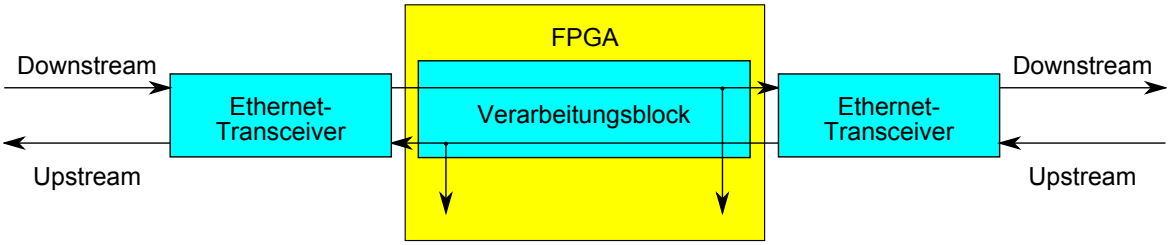

Abbildung 3.8.: Blockschaltbild zur Busankopplung mir zwei Transceivern und Verarbeitungsblock

3.1.5. Verwendung eines Ethernet-Tap

Ein Ethernet-Tap ermöglicht die Überwachung eines Datenstroms in einem Netzwerk. Er eignet sich für das Mitlesen des Datenstroms, weil er auf OSI-Schicht 1 arbeitet. Im Gegensatz zu der Lösung unter 3.1.4 erzeugt ein Ethernet-Tap keinerlei zusätzliche zeitliche Belastung für das Netzwerk, in das er geschaltet wird, sodass während der Analyse die maximal mögliche Anzahl an DEUs an den Bus angeschlossen sein kann. Zudem ist er im Gegensatz zu einem Ethernet-Hub Voll-Duplex-fähig (vgl. [HORUS-NET]).

Die Abbildung 3.9 zeigt die Wirkungsweise eines Taps. Es gibt zwei Ports, die dazu dienen, sich zwischen die Ethernet-Leitung zu schalten. Nebenan gibt es zwei weitere Ports, die das Auskoppeln des Rx- oder Tx- Datenstroms ermöglichen. Diese Eigenschaften machen einen Ethernet Tap zu der optimalen Lösung für die Busankopplung.

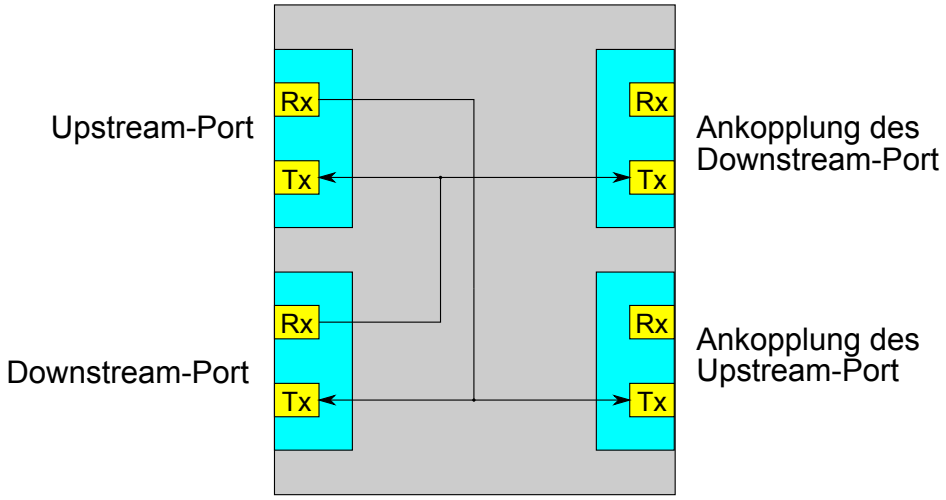

Abbildung 3.9.: Prinzip eines Ethernet TAP

3.2. Einlesemodul

Das Einlesemodul hat die Aufgabe, den Datenstrom aus dem Ankopplungszweig zu empfangen, vorzuverarbeiten und zum Computer zu übertragen (siehe Abbildung 3.1).

3.2.1. Einsatz eines Ethernet-Controllers als Einlesemodul

Um beurteilen zu können, ob man als Einlese-Modul einen Ethernet-Controller für den Empfang des Datenstroms aus dem hybriden Bussystem verwenden kann, muss man sich über den Aufbau von Ethernet-Controllern im Klaren sein. In Abbildung 1.6 ist der Aufbau des Ethernet-Controller Yukon FE+ 88E8040 der Firma Marvel gezeigt. Wie zu erkennen ist, befinden sich in diesem Controller die Komponenten PHY und MAC. Dies bedeutet, dass in diesem Controller die Schichten 1 und 2 des OSI-Schichtenmodells fest eingebaut sind.

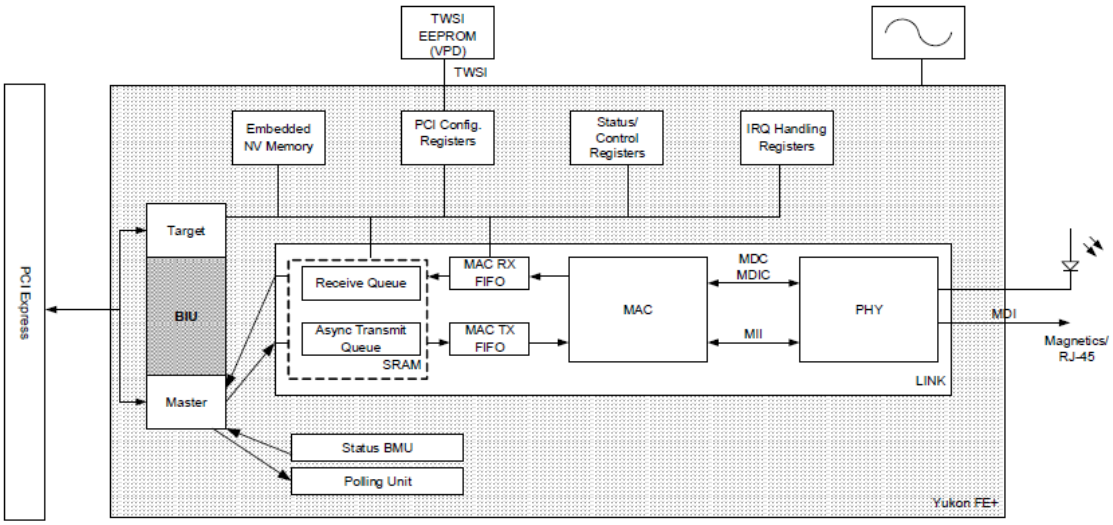

Abbildung 3.10.: Aufbau des Ethernet-Controllers Yukon FE+ 88E8040 der Firma Marvell (vgl. [Marvel])

Um zu prüfen, ob unter diesen Voraussetzungen die Frames des hybriden Protokolls empfangen werden können, wird Bezug auf die Arbeitsgruppe IEEE 802.3 genommen (vgl. [IEEE 802.3]). Die Arbeitsgruppe hat das Aussehen eines Ethernet-konformen Frames definiert.

Definition ungültiger Frames: Der Abschnitt „3.4 Invalid MAC frame“ sagt aus, dass ein Frame ungültig ist, wenn

- die auf dem empfangenen Frame berechnete Prüfsumme nicht mit der sich im FCS-Feld befindenden Prüfsumme übereinstimmt (vgl. 4.2.4.1.2 Frame check sqeuence validation).
- die Framelänge nicht mit dem Wert übereinstimmt, der im Längen/Typ-Feld des Frame steht. Es kann auch sein, dass in diesem Feld ein Typ-Wert steht. Wenn dieser Wert ein gültiger Typ-Wert ist, gilt der Frame als gültig.
- wenn die Anzahl der Bits im Frame kein Vielfaches von 8 ist.

Minimale Framelänge:

Der Abschnitt „4.2.4.2.2 Collision filtering“ sagt aus, dass Ethernet 100BASE-TX konforme Frames eine minimale Länge von 512 Bits haben müssen. Wenn ein Frame kürzer ist, wird davon ausgegangen, dass er das Ergebnis einer Kollision auf dem Bus ist und wird verworfen.

Diskrepanz zu Frames des hybriden Protokoll:

Mit Bezug auf die genannten Punkte stehen die Frames des hybriden Protokolls zu den Ethernet konformen Frames in den folgenden Punkten in Diskrepanz:

- Die Frames des Protokolls des Upstream-Pfads der Topline haben eine Länge von 25 Bytes. Die Frames sind also kürzer als die Mindestlänge von Ethernet Frames und werden deshalb verworfen.
- Die Frames gelten als ungültig, da das Längenfeld nicht mit der wirklichen Framelänge übereinstimmt.
- Außerdem wird die berechnete CRC Prüfsumme immer falsch sein, egal wie lang der Frame ist.

Behandlung ungültiger Frames:

Zur Behandlung dieser ungültigen Frames sagt der Standard aus: „(...) ein ungültiger MAC-Frame sollte nicht an die LLC- oder MAC-Unterschichten weitergegeben werden. Er kann ignoriert oder verworfen werden. Die Nutzung solcher Frames entzieht sich dem Geltungsbereich dieses Standards“ [IEEE 802.3]. Das bedeutet, dass Ethernet-Controller, die strikt dem Standard folgen, die Frames des hybriden Protokolls nicht weiterleiten.

Zusammenfassung:

Der erarbeitete Analyse ergab, dass Ethernet-Controller, welche dem Standard folgen, hybride Datenframes nicht weiterleiten. Einige Hersteller, die die MAC- und PHY-Komponenten entwickeln, weichen eventuell von dem Standard ab. Dies ist für die weitere Planung des Einlesemoduls aber eine zu riskante Voraussetzung, da der gesamte Datenstrom ohne Einschränkungen zur Verfügung stehen muss. Im Folgenden wird deshalb ein alternatives Gesamtsystem erstellt.

3.2.2. Alternative zum Ethernet-Controller

Eine Hardware, die die Aufgaben der physikalischen Schicht von Ethernet übernimmt, nennt sich Ethernet-PHY oder -Transceiver. In diesem Punkt gibt es keine Alternativen. Bei Recherchen nach geeigneter Hardware wurde kein Microcontroller-Board gefunden, das einen Mikrocontroller zusammen mit einem Ethernet-Transceiver auf einem Entwicklungsboard vereint. Es wurden nur Ethernet-Controller gefunden, die die ersten beiden OSI-Schichten implementiert haben. Aus den bereits genannten Gründen ist die Verwendung dieser Ethernet-Controller aber nicht möglich.

Bei FPGA Boards gibt es Boards, die Ethernet-Transceiver statt Ethernet-Controller enthalten. Eine Möglichkeit ist das Spartan 3A Starter Kit von Xilinx. Auf diesem Board ist ein Spartan 3A FPGA vom Typ „XC3S700A“ integriert. Die Bezeichnung des auf dem Board integrierten Ethernet-Transceiver lautet „Standard Microsystems LAN8700 10/100 Ethernet physical layer (PHY) interface“. Das Board hat einen Preis 146 Euro und ist damit nicht zu teuer.

3.2.3. Übertragung zum Computer

Es muss eine Übertragungstechnik gefunden werden, die die Daten aus dem FPGA zuverlässig zum Computer schickt. Die Bewertungskriterien lauten:

- Wie hoch ist die erreichbare Nutzdatenrate?
- Wie kontinuierlich ist der Daten-Transfer
- Wie groß ist der Aufwand für die Realisierung?
- Wie hoch sind die Kosten für die Realisierung?

Ethernet 100BASE-TX: Wenn man Ethernet 100BASE-TX für das Senden des Datenstroms vom FPGA zum PC in Betracht zieht, muss man sich im Klaren darüber sein, dass die beim hybriden Datenbus eingesetzte Übertragungstechnik ebenfalls 100BASE-TX lautet. Während bei den Ethernet-konformen Frames zusätzliche Informationen zu den Nutzdaten versendet werden (vgl. Kapitel 1.6), müssen mit dem Protokollanalysator sämtliche Informationen des hybriden Protokolls übertragbar sein. Aus diesem Grund ist diese Übertragungstechnik zu langsam für die Daten-Übertragung zum PC.

Gigabit-Ethernet: Die Übertragungstechnik Gigabit-Ethernet wäre aus dem Gesichtspunkt der Übertragungsgeschwindigkeit geeignet als Übertragungtechnik, da die Datenrate 1 Gbit/s beträgt. Die Frames des hybriden Protokolls könnte man im Nutzdatenfeld des Ethernet-Frame übertragen. Ein Vorteil der Verwendung von Gigabit-Ethernet ist die Tatsache, dass man computerseitig keine Treiber programmieren muss, um die Daten zu empfangen. Es genügt ein 1 Gbit fähiger Ethernet-Controller.

Ein Nachteil ist, dass man den MAC Layer zur Kommunikation mit dem Computer im FPGA implementieren müsste. Die IP Cores von Xilinx, die man hierfür benutzen könnte, wären durch ihre Lizenz nur zeitlich begrenzt benutzbar. Die Lizenz müsste also zusätzlich gekauft werden. Ein weiterer Grund für Entscheidung gegen Gigabit-Ethernet ist der Preis, der bei der Recherche gefundenen Boards. Ein für diese Applikation ideales Board wäre das XUPV2P Virtex II Pro gewesen. Es hat einen Gigabit-Ethernet Transceiver sowie einen Ethernet 100 BASE-TX Transceiver integriert. Mit 1217,79 Euro ist der Preis für dieses Board aber zu hoch.

Firewire / IEEE-1394a/b: Die Übertragungsgeschwindigkeit, mit der bei Firewire gesendet werden kann, beträgt laut der Arbeitsgruppe IEEE-1394a 100, 200 oder 400 Mbit/s und nach Standard IEEE-1394b 800Mbit/s. Um die Daten auf den Computer zu übertragen, würde die Übertragungsrate also ausreichen.

Bei Recherchen zu verwendbarer Hardware, wurde das Entwicklungsmodul UC1394A-1 der Firma Orsys gefunden. Die folgenden Punkte sprechen aber gegen den Einsatz dieses Moduls:

- Es gibt kein Entwicklungsboard und durch die kleine Größe ist das Modul schlecht lötbar.
- Die Qualität der Treiber und des Supports sind nicht bekannt.
- Der Preis ist mit 3500 bis 3900 Euro hoch.

Ein anderes Entwicklungsboard wurde nicht gefunden. Diese Punkte sprechen gegen eine Verwendung für Firewire als Übertragungstechnik.

Universal Serial Bus 2.0

Der Universal Serial Bus besitzt in seiner High-Speed Variante eine theoretische Datenübertragungrate von 480 Mbit/s. Die Nutzdatenrate beträgt 30 bis 35 MByte/s und ist somit höher als die maximal mögliche Datenrate des hybriden Bussystems. Es gibt vier verschiedene Transferarten, die für bestimmte Applikationen ausgelegt sind. Mit Interrupt-Transfers lassen sich kontinuierliche Datenraten erzielen. Dieser Transfertyp führt außerdem eine Fehlerkorrektur durch, sodass man die Gewissheit hat, dass die übertragenen Daten korrekt sind.

Bei Recherchen für mögliche Hardware wurde der Cypress FX2 Mikrocontroller gefunden. Dieser Controller lässt sich als Slave FIFO konfigurieren und kann auf diese Weise von einem externen Gerät gesteuert werden und Daten zum Host-PC senden. Laut Cypress ist die volle Datenrate, die mit USB 2.0 möglich ist, mit dem FX2 realisierbar. Hier kommt es darauf an, wie leistungsfähig die Software ist, die mit dem FX2 kommuniziert.

Für die Kommunikation mit dem FX2 gibt es zwei Alternativen. Es gibt die Funktionsbibliothek libusb für Linux beziehungsweise libusb-win32 für Windows, durch die man mit USB-Geräten kommunizerien kann. Außerdem gibt es das „WinDriver^{TM} driver development kit“ von der Firma Jungo. Diese beiden Alternativen werden in Kapitel 3.2.4 näher vorgestellt.

Mit 66 Euro pro Stück ist der Preis für ein Erweiterungsboard, auf dem dieser Controller integriert ist, günstig genug. Diese Argumente machen USB 2.0 zu der optimalen Übertragungstechnik für diese Anwendung.

3.2.4. Einlesesoftware

Für die Kommunikation zwischen dem Schaltwerk im FPGA und dem Computer wird der Cypress FX2 verwendet, der mit der Übertragungstechnik USB 2.0 arbeitet. Die Einlesesoftware, welche auf dem PC läuft, ist die Software, die die Kommunikation zwischen dem FX2 und dem Computer realisiert. Zur USB-Kommunikation werden nun zwei Alternativen vorgestellt.

WinDriver^{TM}

WinDriver^{TM} ist ein Entwicklungswerkzeug der Firma Jungo zur Erstellung von Gerätetreibern für zum Beispiel USB, PCI und PCMCIA. Die Treiberentwicklung erfolgt dabei über eine grafische Benutzeroberfläche. In dieser Benutzeroberfläche kann der Benutzer Treiber an die eigenen Bedürfnisse anpassen. Durch eine automatische Code-Generierung wird als Resultat der komplette Quelltext des Treibers erzeugt. Allerdings ist die Nutzung von WinDriver^{TM} kostenpflichtig.

WinDriver^{TM} bietet folgende Vorteile (vgl. [Jungo]):

- Die Entwicklungszeit reduziert sich nach Einarbeitung erheblich.
- Die Treiber laufen mit individueller Kompilierung auf allen unterstützten Betriebssystemen, wie Windows und Linux.
- Für Cypress gibt es unter anderem eine erweiterte Unterstützung für die Entwicklung von Treibern speziell für den FX2.

libusb-win32

libusb-win32 ist eine Funktionsbibliothek für Windows Betriebssysteme, die eine Kommunikation mit USB-Endgeräten erlaubt. Libusb-win32 ist ein Opensource[2]-Projekt. Zur Realisierung der Kommunikation muss keinerlei Treiberentwicklung betrieben werden. Die Kommunikation erfolgt über Funktionen der Bibliothek. Die Funktionsbibliothek kommuniziert ihrerseits über den Treiber libusb0.sys mit den USB-Treibern der niederen Systemebene (vgl. [Johannes Erdfelt]).

[2] opensource - engl.: quelloffen

Es wird eine synchrone und ein asynchrone Application-Interface[3] (API) zur Verfügung gestellt. Die synchrone API bietet die Möglichkeit, auf einfache Art und Weise eine Kommunikation mit einem USB-Gerät zu realisieren. Die asynchrone API bietet laut der Entwicklungsgemeinde die fortgeschritteneren Techniken zur Kommunikation mit USB Geräten.

Zusammenfassung

Für die Kommunikation zwischen dem Cypress FX2 und dem Computer wird die libusb-win32-Funktionsbibliothek eingesetzt, da die Nutzung von WinDriver^{TM} kostenpflichtig ist.

[3] Application-Interface - engl.: Anwendungs-Schnittstelle

3.3. Analysesoftware

Nun werden Optionen vorgestellt, die die Aufgabe der Visualisierung der Daten übernehmen können. Die Bewertungskriterien für die verschiedenen Alternativen werden aus den Anforderungen entnommen. Zusammenfassend werden sie noch einmal genannt:

1. Können die Daten visualisiert werden?
2. Sind Analysewerkzeuge zur Untersuchung der Daten realisierbar?
3. Ist es möglich die Definition der Protokolle modular zu gestalten?

3.3.1. Entwicklung eigener Software

Eine Option ist die Entwicklung eigener Software. Für diese Software gibt es verschiedene Alternativen, die jetzt kurz vorgestellt werden. Die Alternativen werden nach dem Muster aus Kapitel 3.3 bewertet.

Datenbank als Speicherort

Die Realisierung könnte durch ein Programm erfolgen, das die Daten parst[4] und die Informationen anschließend in einer Datenbank speichert.

1. Für die Realisierung der Software kann jede Programmiersprache verwendet werden, die eine Schnittstelle zu einem Datenbanksystem wie mySQL bereitstellt. Zur Visualisierung muss eine neue Software erstellt werden.
2. Sortier- und Filterfunktionen sind in jedem Datenbanksystem realisierbar. Sie können mit Hilfe von Datenbankabfragen realisiert werden.
3. Für das Einfügen der Daten in die Datenbank muss ein Programm entwickelt werden. Dieses Programm parst den gespeicherten Datenstrom und fügt die Informationen in die Datenbank ein. Dieses Programm muss so konzipiert sein, dass sich die Protokoll-Definition leicht verändern lasst.

Bewertung: Der Vorteil dieser Lösung ist, dass sämtliche Verwaltungsfunktionen verfügbar sind, die es für große Datenmengen in Datenbank-Verwaltungsystemen gibt. Ein weiterer Vorteil ist, dass die Daten nur einmal geparst werden müssen. Anschließend stehen sie in einer Datenbank zur Verfügung. Der Nachteil ist, dass ein Programm entwickelt werden

[4] to parse - engl.: aufgliedern

muss, das die Daten in die Datenbank einfügt und ein Programm, das sie anschließend visualisiert.

Baumstruktur mit XML

Die Realisierung könnte mit Hilfe eines XML-Baumes erfolgen, in den die Informationen aus den Frames des hybriden Protokolls eingefügt werden.

1. Zur Verwaltung eines XML Baumes gibt es spezielle Funktionsbibliotheken. Für C++ gibt es zum Beispiel die Qt-Klasse QDomDocument, um einen solchen Baum zu erstellen und ihn zu verwalten. Zur Visualisierung könnten die Frames als Baumstruktur in einem Browser oder in einer selbstgeschriebenen ausführbaren Anwendung angezeigt werden.
2. Mit der selben Klasse könnte der XML-Baum geparst werden. Beim Parsen könnten mit Hilfe von Bedingungen nur bestimme Knoten anzeigt werden und so zum Beispiel Filter realisiert werden. Im Gegensatz zu einem Datenbanksystem stehen aber keine optimierten Algorithmen zur Verfügung.
3. Zum Erzeugen der XML-Datei ist ein zu entwickelndes Programm nötig, das den gespeicherten Datenstrom analysiert und in einen XML-Baum einfügt. Dieses Programm muss wie in der Datenbank-Alternative leicht anpassbar sein, um Änderungen an der Protokoll-Definition vornehmen zu können.

Bewertung: Ein Vorteil dieser Lösung ist, dass man nicht zwingend ein Programm schreiben muss, um die Daten zu visualisieren. Jeder Browser kann wohlgeformte XML-Dokumente als Baumstruktur anzeigen lassen. Wenn man aber Filter- und Sortierfunktionen hinzufügen möchte, muss man zusätzliche Software entwickeln.

Ein Nachteil ist, dass man ein Programm entwickeln muss, das die Daten parst und in den XML-Baum einfügt und ein Programm, das die Daten anschließend visualisiert und Filter- und Sortierfunktionen bereitstellt. Ein weiterer Nachteil ist, dass bei großen Datenmengen die XML-Datei sehr groß wird, da neben den Nutzdaten viele andere Informationen zur Beschreibung der XML Datei gespeichert werden müssen.

3.3.2. Erweiterung für Wireshark

Eine weitere Option ist die Verwendung eines existierenden Programms, welches alle oder Teile der Anforderungen bereits erfüllt. Hier bietet es sich an, das Programm Wireshark mit Plugins zu erweitern. Diese Plugins werden Dissector-Plugins[5] genannt. Im Folgenden wird auf die in Kapitel 3.3 genannten Bewertungskriterien eingegangen.

1. Bei Wireshark könnte man zur Visualisierung die bewährte Anzeige der Daten verwenden. Das Fenster wird dabei in drei Teile aufgeteilt. Im oberen Teil werden die Frames aufgelistet. Hier könnte man die Echtzeit-Frames und die Nichtechtzeit-Frames differenziert anzeigen lassen. Im mittleren Teil werden die Details eines Frames angezeigt. Im unteren Teil wird der Frame in Roh-Form wahlweise in Hex- oder Binär-Format angezeigt.
2. Zur Analyse der Daten gibt es in Wireshark Werkzeuge, wie zum Beispiel Sortier- und Filterfunktionen. Außerdem lassen sich mit „IO Graphs“ Statistiken über die Daten erstellen.
3. Mit Hilfe von Dissector-Plugins lassen sich neue Protokolle in Wireshark integrieren. Diese Plugins werden nach einem Konzept entwickelt, das in der Dokumentation von Wireshark beschrieben ist. Die Modularität wäre also gegeben.

Bewertung Der Vorteil, den man durch die Verwendung von Wireshark hat, ist, dass man auf jahrelange Entwicklungsarbeit zurückgreift. Als Opensource-Programm wird es durch viele Entwickler betreut und weiterentwickelt. Durch Plugins kann man Protokoll-Module in Wireshark integrieren und von diesen Vorteilen profitieren. Zudem werden Benutzer, die schon einmal mit Wireshark gearbeitet haben, wenig Zeit brauchen, sich in die Analyse-Software einzuarbeiten.

Der Nachteil ist, dass man eine Entwicklungsumgebung einrichten muss. In dieser Software muss der gesamte Quellcode von Wireshark kompilierbar sein. Außerdem muss man sich in die Entwicklung der Dissector-Plugins einarbeiten.

3.3.3. Fazit für die Analysesoftware

Für die Analyse der Daten wurde Wireshark verwendet. Die verschiedenen Protokoll-Definitionen werden mit Hilfe von Dissector-Plugins realisiert. Der Ausschlag für die Wahl für die Verwendung von Wireshark ist, sich dessen Vorteile als langjährig betreutes Opensource-Programm zu Nutze zu machen.

[5] to dissect - engl.: zerteilen

3.4. Zusammenfassung

Zusammenfassend werden nun die Komponenten genannt, die für den Protokollanalysator zum Einsatz kommen. Für die Busankopplung wird ein Ethernet-Tap verwendet, da dieses Gerät die genannten Anforderungen erfüllt. Für das Einlesen der Busdaten wird ein Ethernet-Transceiver in Verbindung mit einem FPGA verwendet. Es wird das Xilinx Spartan 3A Starter Kit benutzt. Um die Daten zum Computer zu übertragen, wird der Cypress FX2 Mikrocontroller in der Art vom FPGA gesteuert, dass dieser den Datenstrom über USB zum Computer überträgt.

Computerseitig werden die Daten über die Funktionsbibliothek libusb-win32 eingelesen, da für die Nutzung dieser Bibliothek keine Kosten anfallen. Zur Analyse der Daten wird Wireshark verwendet, da die Vorteile, die dieses Programm bietet, in einer selbst geschriebenen Anwendung nur mit viel Aufwand verwirklicht werden können.

4. Design des Einlesemoduls

Die Aufgabe des Einlesemoduls ist, den Datenstrom aus dem Ankopplungszweig zu empfangen und zum Computer zu senden. In diesem Kapitel wird das Design des Einlesemoduls dokumentiert.

4.1. Erstellung eines Blockdiagramm

In Abbildung 4.1 ist ein Blockschaltbild zu sehen, das die Dokumentation des Entwurfs des Einlesemoduls unterstützt. Sämtliche Komponenten, die in diesem Kapitel erklärt werden, finden sich in diesem Blockschaltbild wieder.

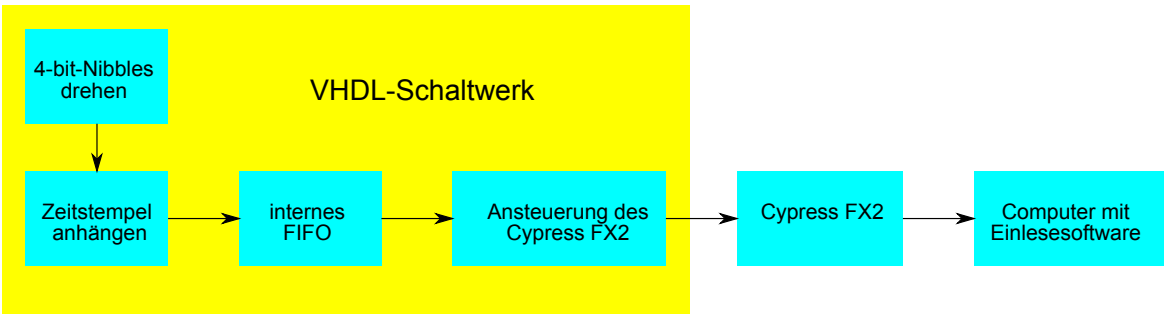

Abbildung 4.1.: Systemblöcke des Einlesemoduls

4.2. Entwurf des VHDL-Schaltwerks

Da kein Ethernet-Controller sondern ein Ethernet-Transceiver in Verbindung mit einem FPGA und dem FX2-Mikrocontroller verwendet wird, musste ein Schaltwerk entworfen werden, das den Datenstrom in geeigneter Art dem FX2-Mikrocontroller zur Verfügung stellt, der diesen über USB zum Computer überträgt. Das Design dieses Schaltwerks wird im Folgenden beschrieben.

4.2.1. Das interne FIFO

Aufgabe der Systemfunktion: Das interne FIFO hat die Aufgabe, die Frames des hybriden Protokolls zu puffern, falls sie nicht gleich zum Computer geschickt werden können. Außerdem lassen sich mit einem FIFO zwei Taktdomänen koppeln, sodass man es mit einem schnelleren Takt auslesen kann, als mit dem 25 MHz schnellen Takt des MII.

Anordnung der Funktion in dem System: Das interne FIFO ist der zentrale Systemblock des Einlesemoduls. Die Anordnung der folgenden Funktionen orientiert sich an diesem Block.

4.2.2. Anhängen der Zeitstempel

Aufgabe der Systemfunktion: Jeder Frame des hybriden Protokolls soll mit einem Zeitstempel versehen werden, der anzeigt, zu welchem Zeitpunkt er empfangen wurde. Die zeitliche Auflösung der Zeitstempel ist durch Wireshark auf 1 μs begrenzt. Der Stempel setzt sich aus einem Sekunden- und einem Mikrosekundenteil zusammen. Wenn der Mikrokunden-Teil den Wert 999999 μs erreicht, muss der Wert des Sekunden-Teils um 1 inkrementiert werden. Umgerechnet in das Hex-Zahlenformat beträgt der maximale Wert für den Mikrosekunden-Teil 0x0F423F μs. Die Länge des Zeitstempels wurde deshalb auf 3 Bytes spezifiziert.

Anordnung der Funktion in dem System: Bevor ein Frame in den Datenpuffer geschrieben wird, muss er mit dem Zeitstempel versehen werden. Der Grund dafür ist, dass sich die Daten im Datenpuffer sammeln, wenn sie nicht sofort zum Computer verschickt werden können. Nachdem sie dann aus dem FIFO gelesen werden, lässt sich nicht mehr individuell zu jedem Frame die Empfangszeit bestimmen. Das ist der Grund dafür, dass der Systemblock vor den Datenpuffer geschaltet ist.

4.2.3. Tauschen der 4-Bit-Nibbles

Aufgabe der Systemfunktion: Durch den 4B/5B-Codierungsvorgang werden die 4-Bit Nibbles im empfangenen Datenstrom vertauscht. Das Umtauschen in die korrekte Reihenfolge wird nicht von dem Transceiver erledigt. Nachdem diese 4-Bit-Gruppen empfangen wurden, müssen sie deshalb wieder umgetauscht werden. Dieser Block hat die Aufgabe, diesen Tausch durchzuführen. Die Abbildung 4.2 zeigt, auf welche Weise der Datenstrom verarbeitet werden muss.

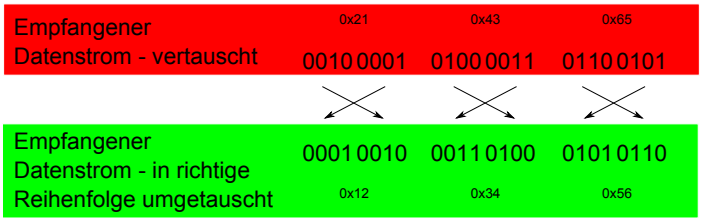

Abbildung 4.2.: Vertauschen der Nibbles durch Verarbeitsungsblock

Anordnung der Funktion in dem System: Das Tauschen der 4-Bit-Nibbles kann technisch gesehen sowohl innerhalb des FPGA als auch durch die Analysesoftware erfolgen. Es ist sinnvoller, den Tauschvorgang im FPGA durchzuführen. Die Begründung hierfür liegt darin, dass im Systemmodell nach dem Transceiver der Systemblock folgt, der die Zeitstempel an die Frames anhängt. Wenn man die Nibbles durch die Analysesoftware tauschen würde, müsste die Software bei jedem Byte des Frame prüfen, ob es sich gerade um den Zeitstempel handelt, weil der Zeitstempel nicht gedreht werden darf. Damit diese Überprüfung nicht gemacht werden muss, wird die Vertauschung vor dem Anhängen der Zeitstempel vollzogen.

4.2.4. Ansteuerung des Slave FIFO Interface des FX2

Aufgabe der Systemfunktionen: Aus dem internen FIFO muss der Datenstrom zum Slave-FIFO des FX2 übertragen werden, damit er von dort aus über USB zum Computer geschickt wird. Die Aufgabe dieses Blocks ist, die Datenflusskontrolle zwischen dem internen FIFO und dem FX2 Slave FIFO zu übernehmen. Der Datenfluss ist unidirektional.

Anordnung der Funktion in dem System: Diese Funktion wird im Systemmodell zwischen das interne FIFO und das FX2 Slave FIFO geschaltet.

4.3. Cypress FX2 Firmware

Um den FX2 im Slave FIFO Modus zu betreiben, ist eine Firmware erforderlich, die den FX2 in diesem Modus arbeiten lässt. In diesem Kapitel werden die dazu nötigen Einstellungen und Eigenschaften genannt. Aufgrund der Tatsache, dass der FX2 die Übertragungstechnik USB zur Kommunikation mit dem Computer verwendet, werden die nötigen Begriffe und Sachverhalte im Folgenden erklärt. Die Folgenden Informationen stammen aus der Quelle [Jan Axelon (2007)].

4.3.1. Einführung zum Universal Serial Bus

Begriffe

Der Universal Serial Bus ist ein durch einen Host gesteuertes Bussystem. Auch die Übertragungsrichtungen werden aus Sicht des Hosts definiert: Transfers zum Host werden als IN-Transfers und Transfers zu den Endgeräten werden als OUT-Transfers bezeichnet. Sollen Daten innerhalb des Bussystems übertragen werden, kann nur der Host die Transfers initiieren. Dies geschieht mit Hilfe von Sendeaufforderungen.

USB-Deskriptoren sind Datenstrukturen, die Informationen über angeschlossene Endgeräte enthalten. Über den „device-Deskriptor“ kann zum Beispiel ein Gerät durch seine Vendor- und Product-ID identifiziert werden.

Endpunkte sind Puffer, in die ein USB-Controller Daten hineinschreiben kann oder herauslesen kann. Endpunkte, in die der Controller Daten hineinschreibt, werden OUT-Endpunkte genannt und Endpunkte, aus denen Daten herausgelesen werden, werden IN-Endpunkte genannt. Die verfügbare Datenmenge, die die Endpunkte fassen können, lässt sich durch die USB-Deskriptoren ermitteln.

Transferarten

Es gibt 4 Transferarten, mit denen der Datenfluss im USB realisiert werden kann. Bulk-Transfers finden nur statt, wenn auf dem Bus Zeit zum Übertragen verfügbar ist. Sie werden deshalb in der Regel für zeitunkritische Datenaufkommen verwendet. Wenn aber nur ein Endgerät an den Bus angeschlossen ist, kann mit diesem Transfertyp die höchste Datenrate aller Transfertypen erreicht werden. Die theoretisch maximal erreichbare Datenrate beträgt 384 MBit/s.

Interrupt-Transfers besitzen Interrupt-Endpunkte ein Abfrage-Intervall, das sicherstellt, das sie vom Host regelmäßig abgefragt werden. Sie werden daher verwendet, wenn Daten innerhalb eines bestimmten Zeitraums übertragen werden müssen. Die theoretisch maximal erreichbare Datenrate beträgt 192 MBit/s.

Wenn man Daten mit einer konstanten Datenrate übertragen möchte, sind Isochrone Transfers nützlich. Isochone Transfers werden zum Beispiel bei der Echtzeitübertragung von Sprache und Musik verwendet. Zum Austausch von Daten ist dieser Transfertyp nicht geeignet, da es hier keine Fehlerkorrektur gibt. Sie eignen sich also nur für Transfers, bei denen gelegentlich Fehler akzeptabel sind. Die theoretisch maximal erreichbare Datenrate beträgt 192 MBit/s.

Control Transfers dienen dazu, kurze Datenpakete sicher zu transferieren. Mit Control Transfers werden die USB-Endgeräte vom Host-Controller konfiguriert. Die theoretisch maximal erreichbare Datenrate beträgt 4 MBit/s.

4.3.2. Verständnis zum Cypress FX2 Mikrocontroller

Im Slave FIFO-Modus erlaubt der FX2 einer externen Schaltung, über Steuer-, Status- und Datensignale Daten über USB zum Host-Computer zu übertragen. Der Unterschied zu einem normalen FIFO ist, dass die Daten, die in das Slave FIFO hineingeschrieben werden, über eine integrierte Schaltung im FX2 in die USB IN-Endpunkte eingegeben werden. Von dort aus werden sie automatisch zum Computer übertragen. Dieser Vorgang passiert ohne Einwirkung der CPU des FX2. Das hat den Grund, dass bei einer hohen Belastung der CPU die Datenrate nicht einbricht (vgl. [Cypress TRM], S. 163ff).

Die Schnittstelle zu diesem Slave FIFO ist vergleichbar mit der Schnittstelle eines normalen FIFOs. Es gibt ein FULL- und EMPTY-Signal sowie das Signal SLWR, mit dem dem gesteuert wird, ob die Daten, die am Datenbuseingang FD<15:0> anliegen, in das FIFO geschrieben werden sollen. Die Abbildung 4.3 zeigt die Signale.

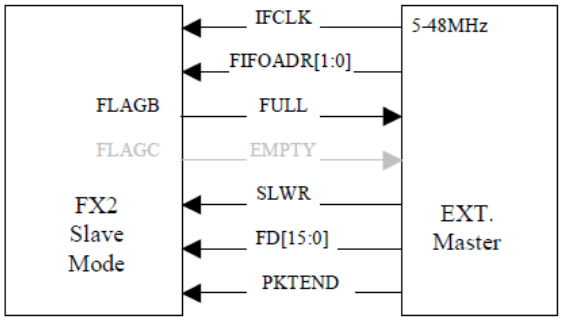

Abbildung 4.3.: Signale zur Ansteuerung des Slave-FIFO (vgl. [Cypress TRM], S. 172)

4.3.3. Festlegen des Transfertyps

Anforderungen an die Übertragung: Zum Festlegen des Transfertyps sind zwingende Anforderungen an die Übertragung der Daten zu beachten. Zusammengefasst lauten die Anforderungen:

- Die Datenübertragung muss fehlerfrei sein.
- Es muss eine garantierte Datenübertragungsrate geben.
- Die Datenrate muss mindestens 100 MBit/s betragen.

Die Eigenschaften der verschiedenen Transferarten, die bei USB verwendet werden können, wurden bereits in Kapitel 4.3.1 genannt. Stellt man die Transfertypen gegenüber, erhält man folgende Übersicht:

	Control	Isochron	Interrupt	Bulk
Fehlerkorrektur?	Ja	Nein	Ja	Ja
Garantierte Bandbreite?	max. 10%	max. 90%	max. 90%	nein[1]
Datenrate von $100MBit/s$?	$4MBit/s$	$192MBit/s$	$192MBit/s$	$384MBit/s$

Tabelle 4.1.: Übersicht über die Transferarten (vgl. [Udo Erhardt (2001)], S. 63)

.

Wahl des Transfertyps: Das Ergebnis ist, dass der Interrupt-Transfer für den Transfer der Daten geeignet ist. Außerdem ist der Bulk-Transfer geeignet, wenn das Analysegerät das einzige am Bus angeschlossene Gerät ist. Der Interrupt-Transfer wird für diese Übertragung der Bus-Daten verwendet, da die genannten Anforderungen bei diesem Transfer-Typ ohne Einschränkungen erfüllt werden.

[1] „Auf einem ansonsten freien Bus stellen Bulk-Transfers den schnellsten Transfer-Typ dar.“ [Jan Axelon (2007)], S. 84

4.3.4. Synchroner oder asynchroner Betrieb

Das Slave-FIFO lässt sich im synchronen oder asynchronen Modus betreiben. Bei asynchroner Ansteuerung werden die Daten, die am Daten-Bus FD<15:0> anliegen, bei jedem Übergang von High nach Low des Signals SLWR in das FIFO geschrieben. Um die korrekte Funktion des FX2 bei diesem Modus zu gewährleisten, müssen bestimmte Zeitparameter eingehalten werden, die im Datenblatt des FX2 zu finden sind (vgl. [Cypress Datasheet], S. 43).

Im synchronen Modus werden die Daten, die am FD<15:0>-Datenbus anliegen, bei jeder steigenden Flanke des Taktsignals IFCLK in das FIFO hineingeschrieben, wenn SLWR gesetzt ist. Der Takt kann zwischen 5 MHz und 48 MHz liegen. Auch hier sind Zeitparameter einzuhalten, die im Datenblatt zu finden sind [Cypress Datasheet], S. 42. Für diese Applikation wird der synchrone Modus verwendet, weil in diesem Modus die Werte für Zeitparameter leichter einzuhalten sind.

4.3.5. Auto-In/-Out Modus

Der Auto-In/-Out Modus erlaubt einer externen Schaltung ohne Einwirkung der Firmware, per USB Daten zum Host zu übertragen. Im Auto-Modus muss die Firmware weder Daten verpacken, noch eine Flusskontrolle mit dem Host bewerkstelligen. In dieser Applikation wird der Auto-In Moduls verwendet.

4.4. Einlesesoftware

Mit der Einlesesoftware wird der Datenstrom über USB in den Computer eingelesen. Um das Design der Software zu beschreiben, wird in diesem Kapitel die synchrone und asynchrone API von libusb-win32 vorgestellt, um erklären zu können, wie die Einlesesoftware realisiert wurde. Um vergleichen zu können, ob die asynchrone oder synchrone API zum Einlesen der Daten verwendet werden sollte, wird mit einem Zeitstrahl das zeitliche Verhalten der APIs beschrieben.

Synchrone API

Wenn die Funktion `usb_bulk_read()`, die zur synchronen API gehört, aufgerufen wird (1), tritt die Funktion solange in den blocked-Zustand ein, bis Daten eingegangen sind. Die Zeit, in der das Programm im blocked-Zustand verharrt, bleibt dabei ungenutzt. Wenn Daten eingegangen sind (2), wird der Inhalt des Empfangspuffers (der Parameter „bytes“ in der Parameterliste der Funktion `usb_bulk_read()`) von der Einlesesoftware in einer Schleife in den Gesamtpuffer geschrieben, der der Puffer des gesamten Datenstromes ist. In dem Zeitraum, in dem die Daten durch die Einlesesoftware in den Gesamtpuffer geschrieben werden, bleibt ungenutzt.

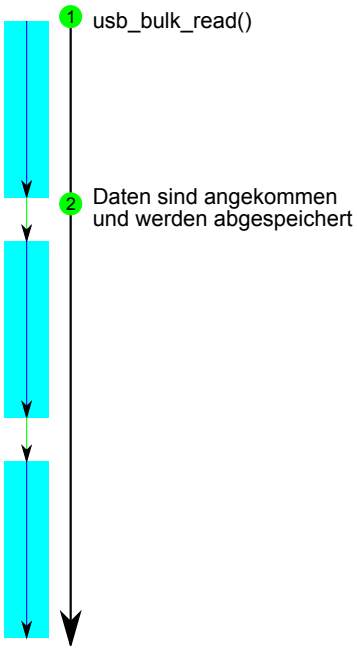

Abbildung 4.4.: Zeitverhalten der synchronen API

Asynchrone API

Im Gegensatz dazu verhält sich die asynchrone API flexibler. Hier wird zunächst mit Hilfe der Funktion `usb_submit_async()` eine Sendeaufforderung an den Treiber „libusb0.sys“ losgeschickt (3). Danach tritt die Funktion aber nicht in den blocked-Zustand ein, bis die Daten eingegangen sind. Es lassen sich deshalb danach noch andere Aufgaben erledigen. Mit der Funktion `usb_reap_async()` lässt sich die Sendeaufforderung abbauen und die Daten werden in den Gesamtpuffer geschrieben, sobald sie angekommen sind.

Sendeaufforderungsschlange

Um den Datenstrom kontinuierlich zu machen, wurde ein Verfahren erarbeitet, das mit der asynchronen API arbeitet. Zu Beginn eines Einlesevorgangs werden drei Sendeaufforderungen (3) an den Treiber geschickt. Anschließend fragt man die Sendeaufforderungen 1 ab, ob sie fertig ist. Dies geschieht mit Hilfe der Funktion `usb_reap_async()` (4). Wenn der Transfer abgeschlossen ist, werden die Daten gespeichert (5).

Da zu Beginn der Übertragung gleich drei Sendeaufforderungen losgeschickt wurden, beginnt nach dem ersten Transfer sofort der nächste Transfer (5). Der exakte Zeitpunkt, zu dem der zweite Transfer beginnt, konnte nicht ermittelt werden. Mit Hilfe dieses Verfahrens überlässt man die zeitliche Verwaltung der Transfers der Funktionsbibliothek, sodass die Pause zwischen den Transfers minimal klein wird. Die blau hinterlegte Fläche in der Abbildung 4.5 verdeutlicht, dass bei der Annahme, dass der Treiber „libusb0.sys“ keine Pausen zwischen Transfers macht, ein ständiger Datentransfer besteht.

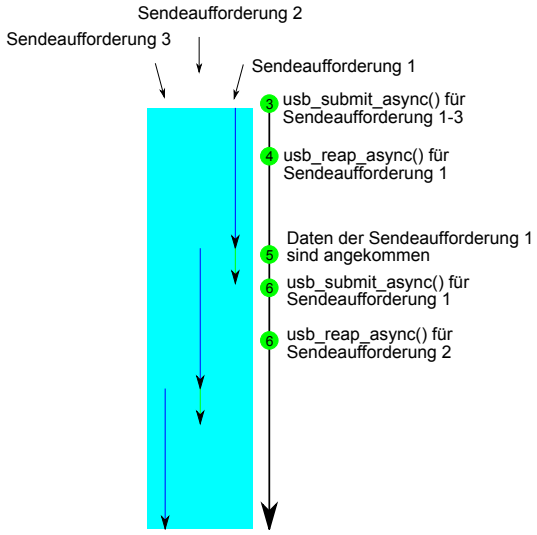

Abbildung 4.5.: Zeitverhalten der asynchronen API

5. Realisierung des Einlesemoduls

5.1. VHDL-Schaltwerk

Im Folgenden wird die Realisierung des VHDL-Schaltwerkes dokumentiert. In den Unterkapiteln werden die Teilblöcke des in der Abbildung 5.1 abgebildeten Systems beschrieben. Sämtliche Systemblöcke, außer dem MII_COUNT_GEN-Block, werden mit der ansteigenden Takt-Flanke abgefragt. Zu sämtlichen Systemblöcken wurden VHDL-Testbenches erstellt, um die korrekte Funktion zu überprüfen. Der MII_COUNT_GEN-Block wurde für Testzwecke erstellt. Wenn das Analyse-System Ethernet-Daten empfangen soll, nimmt das wirkliche MII des Transceivers den Platz dieser Komponente ein.

Die VHDL-Quelltexte und die Testbenches sind im elektronischen Anhang zu finden. Im Anhang A ist der Pfad zu den Dateien des elektronischen Anhangs beschrieben. Die Realisierung der Zustandsautomaten im Schaltwerk wurde nach dem Muster aus dem Lehrbuch [Jürgen Reichard (2009)] S. 235ff durchgeführt.

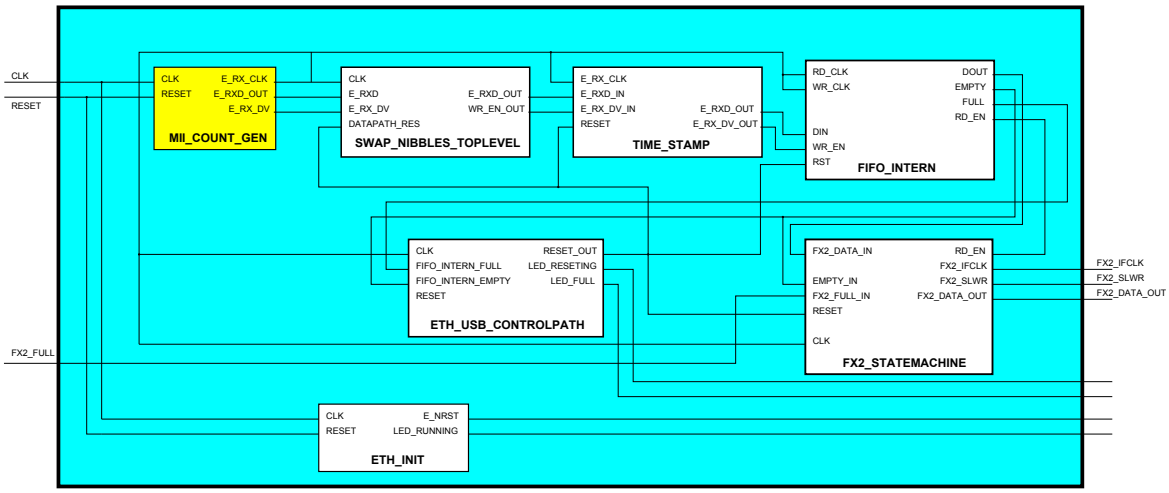

Abbildung 5.1.: Blockschaltbild des VHDL-Schaltwerkes ETH_USB

5.1.1. MII_COUNT_GEN-Einheit

Die Quelle des Ethernet-Datenstroms ist das MII des Ethernet-Transceivers. Das Zeitverhalten der für den Empfang von Daten wichtigen Signale ist in dem Impulsdiagramm in Abbildung 5.2 zu sehen. Das Taktsignal E_RX_CLK beträgt 25 MHz. Das Signal E_RXD<0:3> ist der Datenausgang des MII. Das Signal E_RX_DV wird gesetzt, wenn gültige Daten an dem Datenausgang anliegen. Die Signale werden bei der fallenden Flanke von E_RX_CLK generiert.

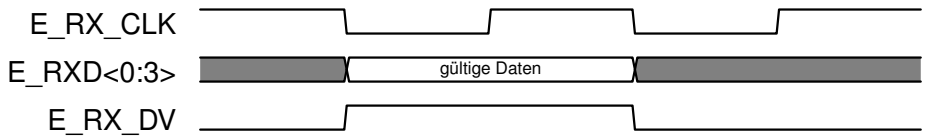

Abbildung 5.2.: Zeitverhalten des MII

Die Einheit MII_COUNT_GEN dient dazu, das Verhalten der MII des Transceivers zu simulieren. Das Simulationsergebnis der reinen Test-Einheit MII_COUNT_GEN ist in Abbildung 5.3 zu sehen. Die Ausgangsdaten werden dabei durch einen Zähler generiert. Es ist zu erkennen, dass die 4-bit-Nibbles in der Reihenfolge vertauscht sind. Dies entspricht dem wirklichen Verhalten des Transceivers.

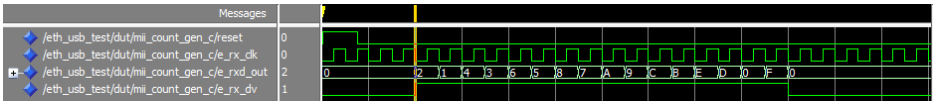

Abbildung 5.3.: Simulation der MII_COUNT_GEN-Einheit

5.1.2. SWAP_NIBBLES_TOPLEVEL-Einheit

In dieser Einheit werden die 4-bit-Nibbles umgetauscht. Dazu wurde das Modell einer Einheit entwickelt, dessen Zeitverhalten in der Abbildung 5.4 zu sehen ist. Um den Tauschvorgang zu realisieren, wird ein Schieberegister verwendet, durch das das Datensignal E_RXD geführt wird. Mit Hilfe eines Multiplexers wird abwechselnd das Signal E_RXD1 und E_RXD3 auf den Ausgangsdatenbus E_RXD_OUT geschaltet. Als Steuersignal für den Multiplexer wird das Signal E_RXD_EN verwendet, das durch einen Zähler generiert wird.

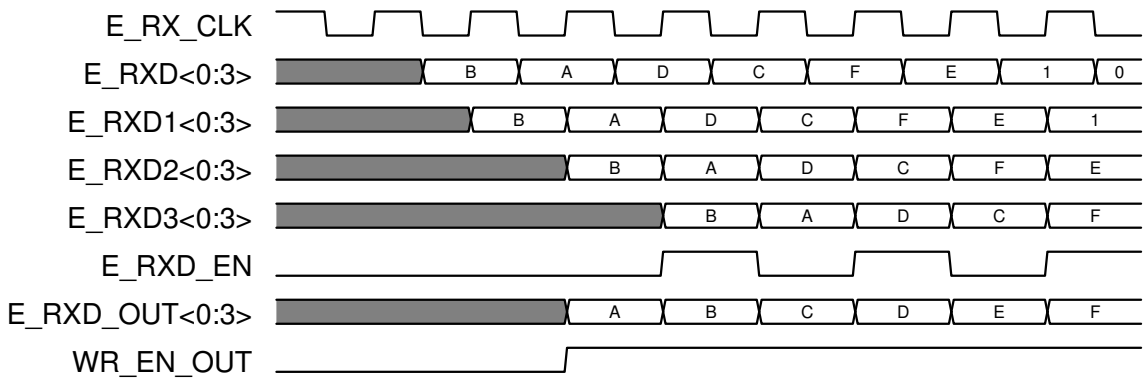

Abbildung 5.4.: Impulsdiagramm zur Verdeutlichung des Vertauschen der 4-bit Nibbles

Fehlerhaftes Verhalten: Auf Grundlage dieses Modells wurde die VHDL-Einheit SWAP_NIBBLES erstellt, in der dieses Verhalten realisiert ist. Mit diesem Schaltwerk kam es beim Empfangen von Daten zu Fehlern. Die Fehler machten sich darin bemerkbar, dass die Nibbles um eine Position versetzt vom Computer empfangen wurden. Für die Lösung zu diesem Problem, muss man sich im Klaren darüber sein, dass in der Datenverarbeitung, die kleinste ansprechbare Dateneinheit, ein Byte ist. Die über USB empfangenen Bits werden deshalb spätestens im Computer in Bytes eingeteilt. Sind die Nibbles um eine Position versetzt, werden die Daten folgendermaßen vom Computer eingelesen.

	A	B	C	D	E	F	0	ungewünschtes Verhalten
A	B	C	D	E	F	0	1	gewünschtes Verhalten

Tabelle 5.1.: Fehlerhafter Datenstrom aufgrund versetzter Nibbles

Analyse des Fehlers: In der Abbildung 5.5 ist der Ursprung dieses Fehlers zu erkennen. Bei der Markierung 1 wird das globale System-Reset-Signal auf 0 gesetzt. Zu diesem Zeitpunkt empfängt der Transceiver aber schon Daten. Da das E_RX_DV-Signal gesetzt ist, beginnt die Einheit damit, die Nibbles zu tauschen. Sie beginnt damit aber bei einer ungeraden Position der Nibbles (siehe Markierung 2). Dadurch wird die Reihenfolge der Nibbles nicht korrigiert, sondern weiter verfälscht (Markierungen mit weißem Pfeil).

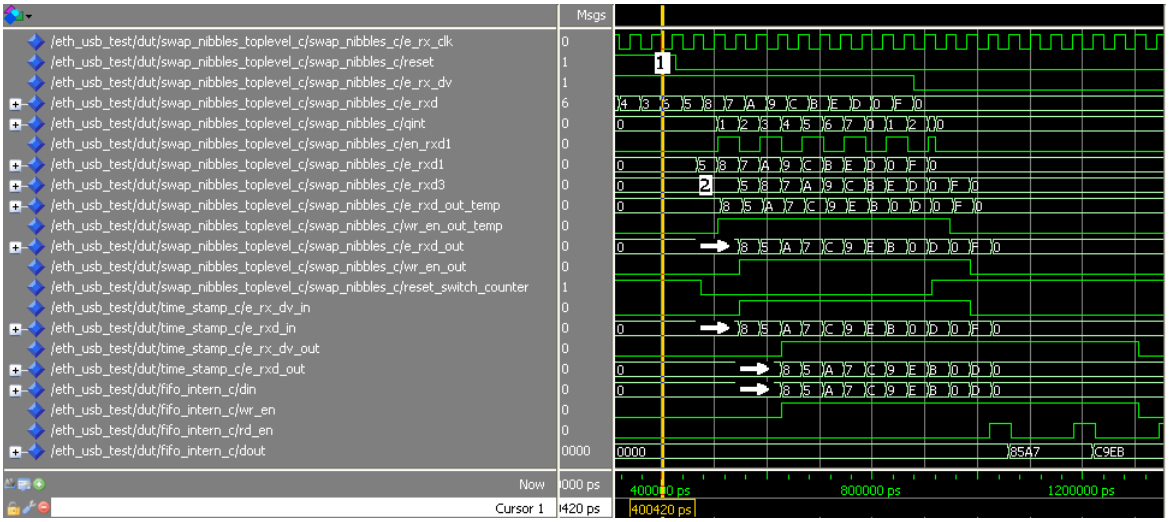

Abbildung 5.5.: Verworfenes Design der SWAP_NIBBLES-Einheit

Für die Analyse der Daten ist allein dieser Fehler aber nicht kritisch, da diese Daten verworfen werden können. Das sich ergebende Problem ist, dass wie im im obigen Fall gezeigt, eine ungerade Anzahl an Nibbles in das FIFO geschrieben wird. Sämtliche folgenden Daten werden auch in das FIFO geschrieben. Spätestens im Computer werden dann sämtliche Daten zu Bytes zusammengefasst und es ergibt sich das unerwünschte Verhalten, wie in Tabelle 5.1 gezeigt.

Korrektur des Fehlers: Um das erwähnte Problem zu lösen, wurde eine Steuerpfad-Komponente erstellt, die die SWAP_NIBBLES-Einheit kontrolliert. Das Verhalten der Steuerpfad-Einheit wurde mit einem Zustandsautomaten modelliert, dessen Zustandsdiagramm in Abbildung 5.7 zu sehen ist. Der Zustandsautomat bleibt, wenn nach einem System-Reset gerade ein Frame empfangen wird, zunächst im IDLE-Zustand (siehe markiertes Signal). Wenn der laufende Frame zu Ende ist, und deshalb das Signal E_RX_DV auf 0 gesetzt wird, geht der Automat in den Zustand SWAPPING über. Nun ist sichergestellt, dass die Nibbles wie gewünscht getauscht werden. Außerdem ist gewährleistet, dass zu Beginn jedes Frames die Anzahl der Nibbles im FIFO einen geraden Wert hat. In Abbildung 5.6 ist das Simulationsergebnis zu sehen.

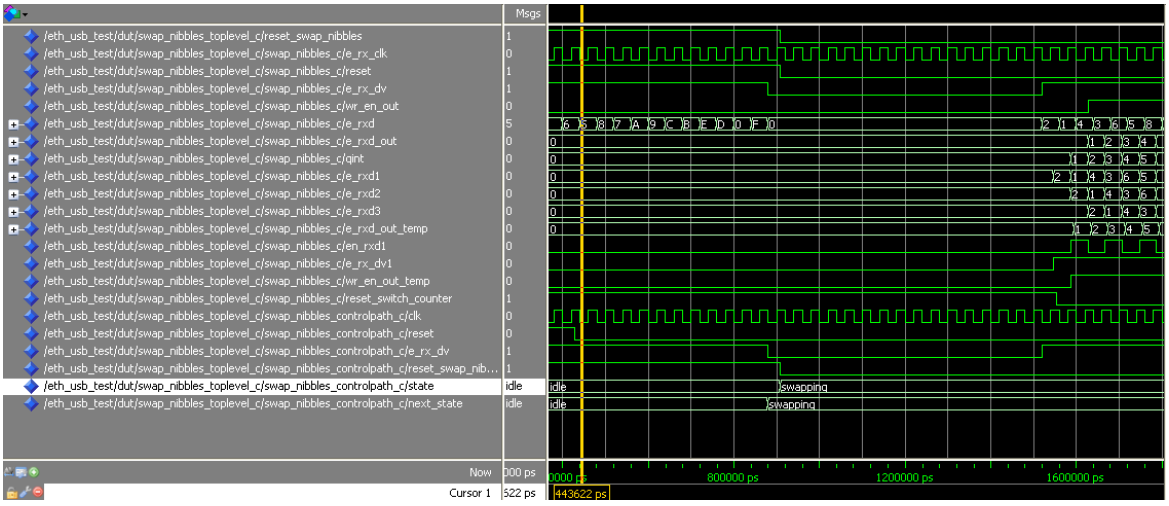

Abbildung 5.6.: System mit Kontrollpfad

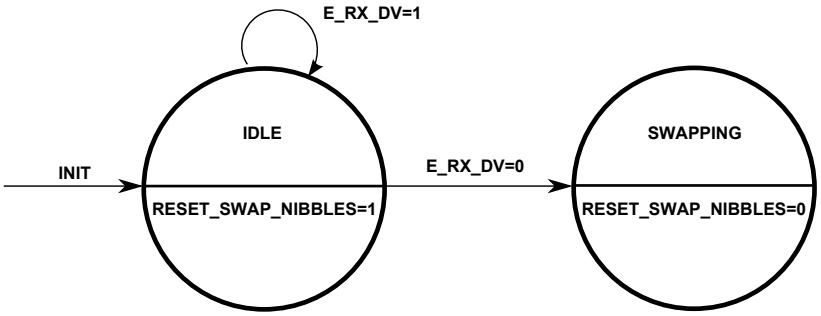

Abbildung 5.7.: Zustandsdiagramm der Steuerpfad-Komponente der SWAP_NIBBLES-Einheit

5.1.3. TIMESTAMP-Einheit

Die TIMESTAMP-Einheit hängt die Zeitstempel an die Frames. Der Aufbau der Einheit ist in Abbildung 5.8 zu sehen. Die Einheit USEC_COUNTER beinhaltet den 24-bit-Zähler, der die Mikrosekunden-Werte, die an die Frames angehängt werden, generiert. Die Einheit USEC_MUX beinhaltet den Multiplexer, der den 24-bit Zählerstand auf sechs 4-bit Gruppen aufteilt, da der FIFO-Eingangsdatenbus 4-bit breit ist. Die Einheit DATASTREAM_MUX beinhaltet den Multiplexer, der entweder den Datenstrom (E_RXD) oder den Zählerstand (USEC_COUNT_4BIT) auf den Ausgang schaltet. Diese Blöcke werden durch die Einheit CONTROL_PATH gesteuert.

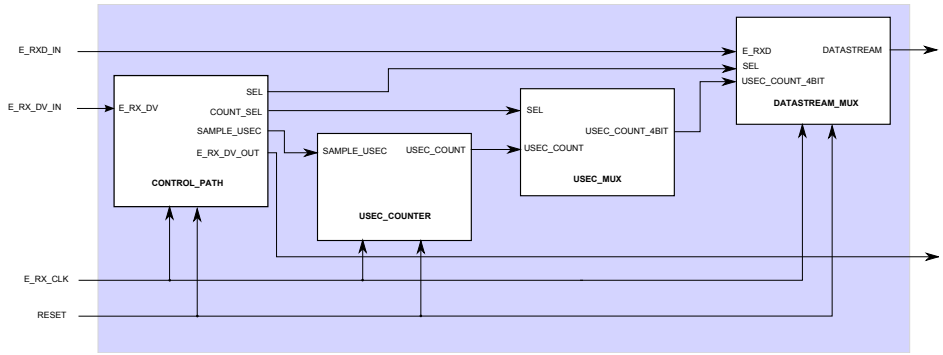

Abbildung 5.8.: Aufbau der TIMESTAMP-Einheit

Die Komponente CONTROL_PATH wurde durch einen Zustandsautomaten realisiert, dessen Zustandsdiagramm in Abbildung 5.9 zu sehen ist.

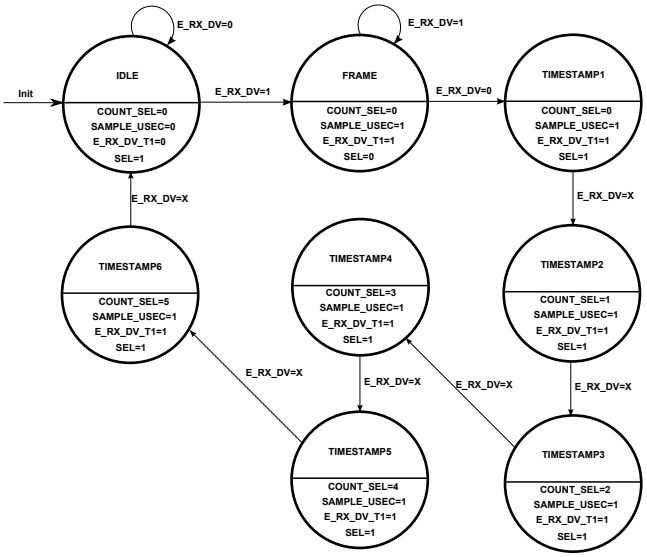

Abbildung 5.9.: Zustandsdiagramm des Kontroll-Pfads der TIME_STAMP-Einheit

Solange das Signal E_RX_DV nicht gesetzt ist, verharrt der Automat im IDLE-Zustand. Wenn Daten empfangen werden, wird das Signal durch den Transceiver gesetzt und es erfolgt ein Zustandswechsel in den Zustand FRAME. Solange E_RX_DV gesetzt ist, bleibt der Automat in diesem Zustand. In diesem Zustand wird das Signal SAMPLE_USEC gesetzt. Es bewirkt, dass der aktuelle Zählerstand des Mikro-Sekunden-Zählers (USEC_COUNT) festgehalten wird, damit der Wert des Zählerstandes an den Frame angehängt wird, der zu Beginn des Frames existierte. Außerdem wird das Signal E_RX_DV_T1-Signal gesetzt. Es bewirkt, dass alle folgenden Signale als gültig interpretiert werden.

Wenn das Signal E_RX_DV wieder zurückgesetzt wird, also keine Daten mehr empfangen werden, soll der Zeitstempel an den Frame gehängt werden. Deshalb erfolgt ein Übergang in den Zustand TIMESTAMP1. Von diesem Zustand aus, geht der Automat schrittweise in den Zustand TIMESTAMP6 über. Während dieser Zustände wird das Signal SEL auf 1 gesetzt, was bewirkt, das das Datensignal mit dem Zählerstand auf den Datenausgang geschaltet wird. Der Multiplexer hierfür ist in dem Block DATASTREAM_MUX enthalten (siehe Abbildung 5.8).

Während der Übergänge der Zustände TIMESTAMP1 bis TIMESTAMP6 wird außerdem das Steuersignal COUNT_SEL für den Multiplexer generiert, der den Datenausgang des 24-bit Zählers in sechs 4-bit Kanäle unterteilt. Das Signal COUNT_SEL wird, beginnend mit dem Wert 0, in jedem der sechs Zustände, bis zum Wert 5, inkrementiert. Das bewirkt, dass zum Ende eines Zyklus der komplette Zählerstand in den Datenausgang geschrieben wurde.

Das Simulationsergbnis aus Abbildung 5.10 entspricht dem gewünschten Verhalten, dass der Zählerstand an das Ende eines jeden Frame angehängt werden soll. Die Werte der letzten beiden Nibbles des Eingangsdatenstromes E_RXD_IN betragen 0xF und 0x0. An das Ende des Signals E_RXD_IN wird der aktuelle Zählerstand mit dem Wert von 0x123456 angehängt. Das resultierende Signal ist das Signal E_RXD_OUT. Das Signal E_RX_DV_OUT wird während der Ausgabe des Zählerstandes auch gesetzt, sodass diese Daten für das interne FIFO in dem nächsten Systemblock als gültig gelten.

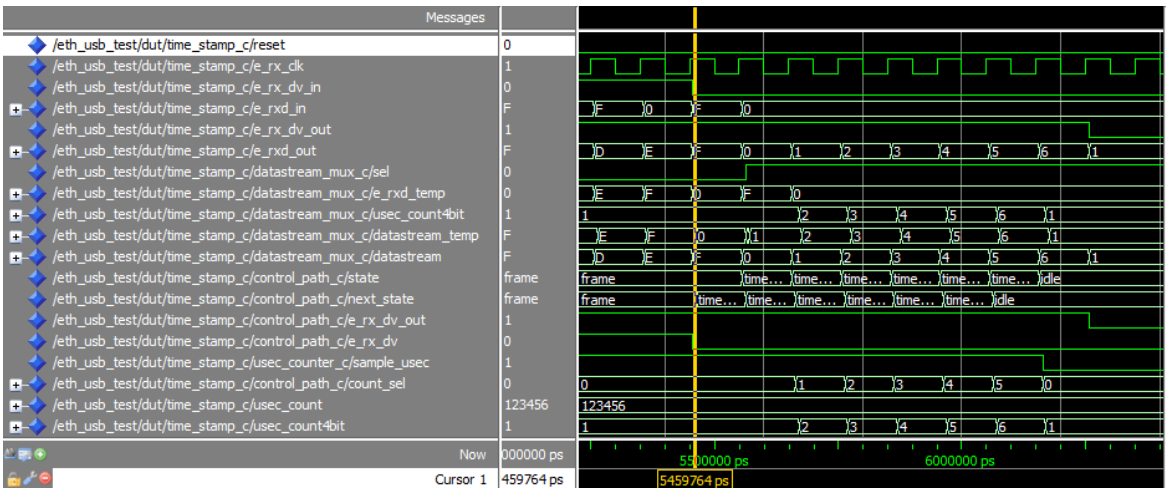

Abbildung 5.10.: Simulationsergebnis der TIMESTAMP-Einheit

5.1.4. FIFO_INTERN-Einheit

Das interne FIFO wurde mit dem „LogiCORE^{TM} IP FIFO Generator v6.2“ generiert. Das FIFO besitzt eine Eingangsdatenbusbreite von 4-bit entsprechend des Datensignals E_RXD<3:0> des MII des Ethernet Transceivers. Die Ausgangsdatenbusbreite beträgt 16-bit entsprechend des FD<15:0>-Signals des Slave-FIFO-Interface.

5.1.5. FX2_STATEMACHINE-Einheit

Implementierung: In Abbildung 5.11 ist das Zustandsdiagramm des Zustandsautomaten abgebildet. Im DECISION_STATE-Zustand wird geprüft, ob das FX2-Slave-FIFO nicht voll ist (FX2_FULL=0) und das interne FIFO nicht leer ist (EMPTY=0). Wenn beide Bedingungen wahr sind, erfolgt ein Zustandsübergang in den RD_EN_STATE, in dem das Signal RD_EN gesetzt wird, und darauf ein Übergang in den SLWR_ON_STATE-Zustand, in dem das Signal FX2_SLWR gesetzt wird. Durch das Setzen des RD_EN-Signals wird ein Datensatz aus dem internen FIFO gelesen und durch das Setzen des Signals FX2_SLWR wird dieser Wert in das FX2 Slave FIFO geschrieben.

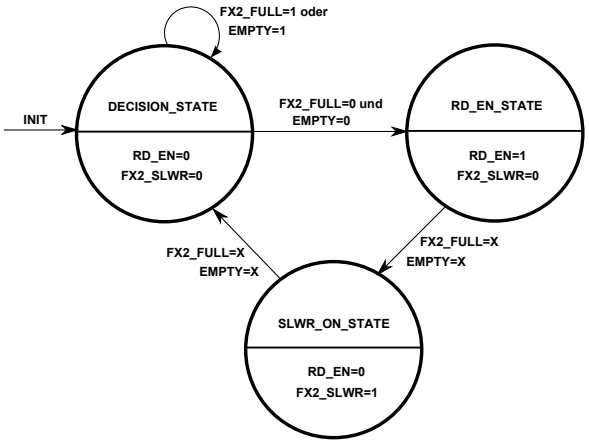

Abbildung 5.11.: Zustandsdiagramm der FX2_STATEMACHINE-Einheit

Simulation: Im Folgenden werden die Simulationsergebnisse des Zustandsautomaten gezeigt. Zunächst wird gezeigt, wie der Automat reagiert, wenn das interne FIFO leer ist. Anschließend wird die Reaktion des Automaten gezeigt, wenn durch den FX2 das FX2_FULL-Signal gesetzt wird.

Reaktion auf das Setzen des EMPTY-Signals: Die Abbildung 5.12 dient der Veranschaulichung der Reaktion auf das Setzen des EMPTY-Signals.

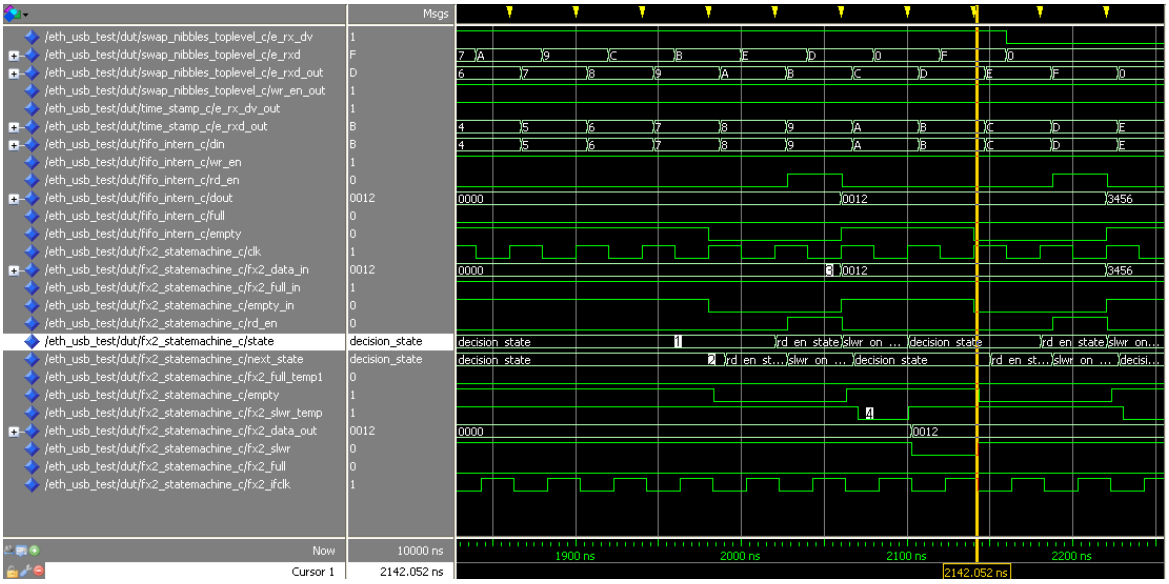

Abbildung 5.12.: Reaktion auf das Setzen des EMPTY-Signals

1. Der Automat ist im DECISION_STATE Zustand.
2. Der Wert des EMPTY-Signals des internen FIFOs wird 0. Durch die kombinatorische Logik im Übergangsschaltnetz des Automaten erfolgt nach einer symbolischen Verzögerungszeit eine Änderung des Signals NEXT_STATE in den symbolischen Zustandstyp RD_EN_STATE.
3. Bei der nächsten steigenden Flanke nimmt der Automat den Zustand aus dem Übergangschaltnetz ein und das Signal RD_EN wird auf 1 gesetzt. Dadurch wird ein Datensatz aus dem FIFO gelesen.
4. Es erfolgt ein Zustandsübergang in den FX2_SLWR-Zustand und das Signal wird gesetzt. Dieses Signal ist low-aktiv. Wenn dieses Signal auf 0 gesetzt ist, schreibt der FX2 den Wert am Datenbuseingang in das Slave-FIFO.

Reaktion auf das Setzen des FULL-Signals: Die Abbildung 5.13 dient der Veranschaulichung der Reaktion auf das Setzen des FULL-Signals.

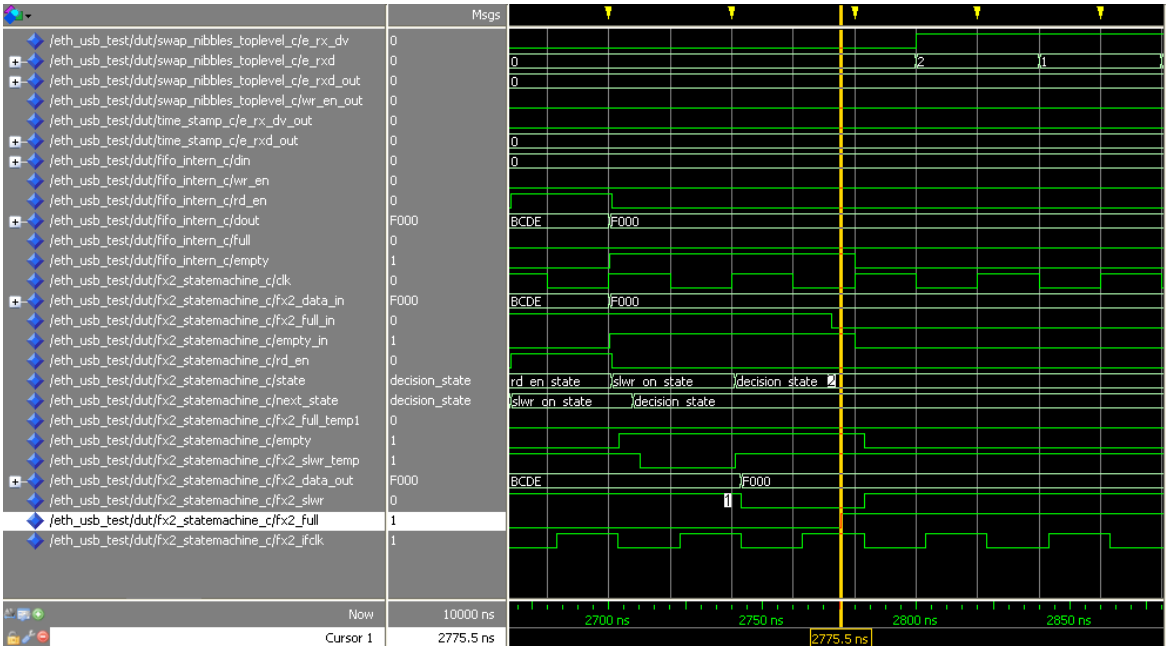

Abbildung 5.13.: Reaktion auf das FULL-Signal

1. SLWR wird auf 0 gesetzt und bei der nächsten steigenden Flanke von FX2_IFCLK werden die Werte des Signals FX2_DATA_OUT in das FIFO geschrieben. Nach einer Verzögerungszeit von $t_{XFLG} = 9,5ns$ wird das FULL-Signal des FX2 gesetzt. Der Wert der Verzögerungszeit ist im Datenblatt zu finden (vgl. [Cypress Datasheet]).
2. Wenn sich der Automat im DECICION_STATE-Zustand befindet, bleibt er so lange in diesem Zustand, bis das FX2_FULL-Signal wieder auf 0 gesetzt wird.

5.1.6. ETH_USB_CONTROLPATH-Einheit

Dieser Systemblock dient als Kontroll-Einheit für das gesamte Schaltwerk. Er verarbeitet die Statussignale FIFO_INTERN_EMPTY und FIFO_INTERN_FULL des internen FIFOs und generiert daraus das Steuersignal RESET_OUT, mit dem sämtliche Systemkomponenten, außer der Test-Einheit MII_COUNT_GEN bzw. der wirklichen MII, verbunden sind. Das Verhalten des Blocks wurde mit einem Zustandsautomaten realsiert. Das Zustandsdiagramm ist in Abbildung 5.14 zu sehen.

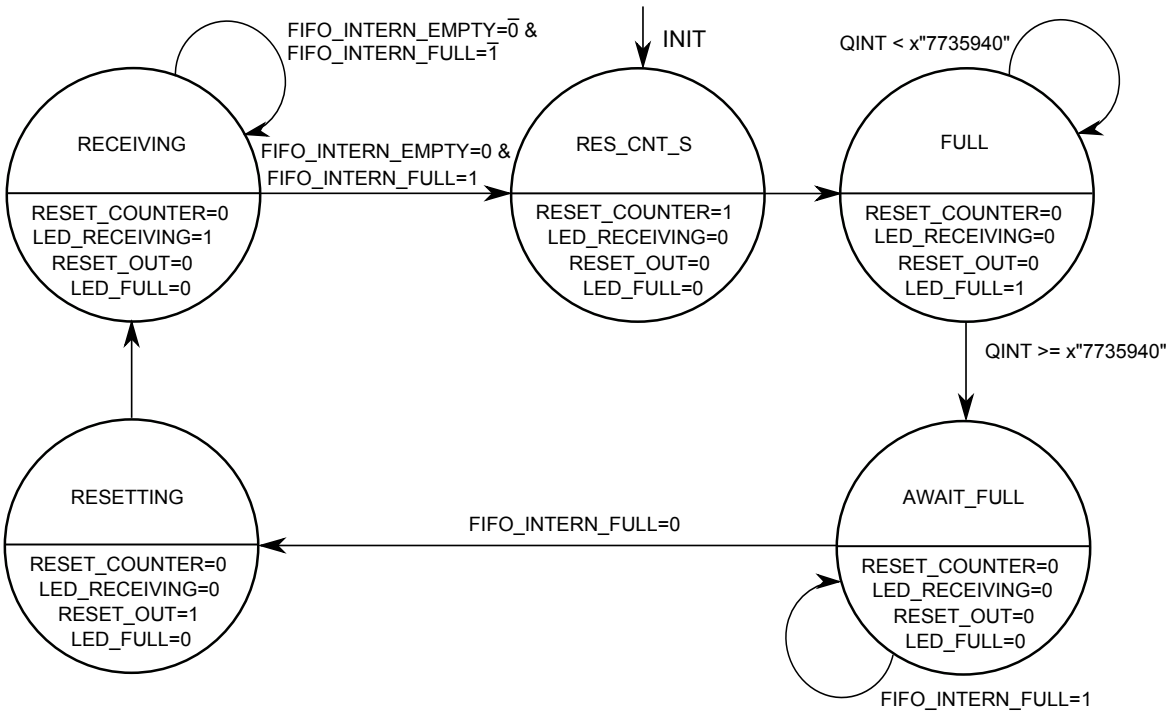

Abbildung 5.14.: Zustandsdiagramm der ETH_USB_CONTROLPATH-Einheit

Um die Funktion des Automaten anschaulich zu beschreiben, wird angenommen, dass der Automat den Zustand RECEIVING angenommen hat. In diesem Zustand ist das Signal LED_RECEIVING gesetzt, das bewirkt, dass eine entsprechende Status-LED auf dem Xilinx-Board leuchtet. Der Automat verharrt solange in dem Zustand, bis das FIFO voll ist. Das Status-Signal FULL besitzt dann den Wert 1 und das Signal EMPTY den Wert 0.

Es kann zwei Gründe geben, warum das FIFO voll wird. Ein Grund ist, dass ein Einlesevorgang beendet ist, was bedeutet, dass keine Daten mehr zum Computer übertragen werden. Der andere Grund ist, dass das interne FIFO aufgrund einer zu langsamen Übertragungsrate vom FX2 zum Computer übergelaufen ist. Für die eingelesenen Daten bedeutet das zwangsläufig, dass sie fehlerhaft sind, da die Daten, die nicht in das interne FIFO geschrieben wer-

den konnten, verloren gehen. Diese Fehlfunktion muss dem Benutzer des Analyse-Systems gemeldet werden. Dieser Meldevorgang wird folgendermaßen realisiert.

Ist das FIFO voll, erfolgt ein Zustandsübergang in den Zustand RES_CNT_S, in dem der Zähler, der das Signal QINT generiert, zurückgesetzt wird. Das Rücksetzen geschieht durch das Setzen des Signals RESET_COUNTER auf den Wert 1 in dem Zustand RES_CNT_S. Für einen Taktzyklus bleibt der Automat in diesem Zustand und geht dann in den FULL-Zustand über. In diesem Zustand bleibt der Automat, solange das Signal QINT kleiner oder gleich als der Wert 125000000 bzw. hexadezimal 0x7735940 ist. Bei einer Taktfrequenz von 25 MHz bleibt der Automat deshalb 5 Sekunden in diesem Zustand. Während dieses Zustandes, ist das Signal LED_FULL gesetzt, das bewirkt, dass eine Status-LED leuchtet. Fängt diese LED während eines Einlesevorgangs an zu leuchten, ist dies ein Indikator für den Benutzer, dass das interne FIFO voll ist und die Daten fehlerhaft sein werden.

Nachdem der Zähler den Wert 125000000 überstiegen hat, erfolgt ein Zustandsübergang in den Zustand AWAIT_FULL. In diesem Zustand wartet der Automat solange, bis das FULL-Signal des internen FIFOs zurückgesetzt ist. Sobald ein neuer Einlese-Vorgang begonnen hat und deshalb das FULL Signal zurückgesetzt wird, erfolgt ein Übergang in den Zustand RESETTING. In diesem Zustand wird das Signal RESET_OUT gesetzt, was bewirkt, dass sämtliche Schaltwerk-Einheiten, außer der MII_COUNT_GEN-Einheit, zurückgesetzt werden. Danach erfolgt der Zustandsübergang in den Zustand RECEIVING, in dem der Automat wieder solange verharrt, bis das FIFO voll wird.

5.2. Cypress FX2 Firmware

In diesem Kapitel wird die Realisierung der Firmware für den Cypress FX2 Mikrocontroller dokumentiert. Wie im Kapitel 4.3 beschrieben muss der FX2 im Slave-FIFO-Modus arbeitet. Außerdem sind eine Reihe weiterer Maßnahmen erforderlich, damit der Mikrocontroller wie gewünscht den Datenstrom zum Computer überträgt. Als Grundlage für die Firmware wurde eine Firmware angepasst, die im Benutzerhandbuch des FX2 zu finden ist (vgl. [Cypress TRM], S. 188). Die komplette Firmware ist im elektronischen Anhang (A) zu finden. Die Werte für die elementaren Register werden im Folgenden beschrieben.

Interface Configuration: `IFCONFIG = 0x03;`

In diesem Register wird festgelegt, dass der FX2

- als Slave-Fifo arbeitet,
- eine externe Taktquelle verwendet
- und im synchronen Modus arbeitet.

Endpoint 6/Slave FIFO Configuration: `EP6FIFOCFG = 0x0D;`

In diesem Register wird festgelegt, dass

- der AUTOIN-Modus verwendet wird
- und der Eingangsdatenbus des Slave-FIFO 16-bit breit ist.

Endpoint 6 Configuration: `EP6CFG = 0xF0;`

In diesem Register wird festgelegt, dass

- die Übertragungsrichtung IN lautet,
- der Endpunkt 6 aktiv ist,
- Interrupt-Transfer zur Übertragung verwendet wird,
- der Endpunkt Buffer 512 Byte groß ist
- und quad-buffering angewendet wird.

5.3. Einlesesoftware

5.3.1. Definition einer Klasse zum Einlesen

Zur Kommunikation mit dem FX2 wurde eine Klasse definiert, die alle Fähigkeiten enthält, die man für die Initialisierung der Funktionsbibliothek libusb-win32 und zur Kommunikation mit dem FX2 benötigt. Diese Klasse trägt den Namen USBHandler. Um mit dem FX2 zu kommunizieren, wird eine Methode benötigt, die das Gerät identifiziert. Diese Methode wurde `findDevice()` benannt. Eine weitere Methode ist nötig, die libusb initialisiert. Sie wurde `init()` benannt. Es bietet sich an, dass diese Methoden durch den Konstruktor der Klasse (`USBHandler()`) ausgeführt werden.

Um die Klasse in einer grafischen Benutzeroberfläche (GUI) zu verwenden, ist es nützlich, wenn es eine Methode gibt, die einen Einlesevorgang durchführt. Durch Klicken auf eine Schaltfläche, wird diese Methode dann aufgerufen. Durch ein Auswahlmenü wird die Datenmenge definiert, die eingelesen werden soll. Deshalb enthält die Parameterliste der Funktion `read()` den Parameter `rxMByte`, über den die Datenmenge mitgeteilt wird.

Mit der Funktion `usb_release_interface()` muss die Kontrolle des Gerätes wieder an das Betriebssystem abgegeben werden. Außerdem ist es vorgeschrieben, die Funktion `usb_close()` auszuführen, um die Kommunikation mit dem USB-Gerät zu schließen. Dies übernimmt der Destruktor der Klasse, der automatisch aufgerufen wird, wenn das Objekt zerstört wird. Ein Klassendiagramm der Klasse ist in Abbildung 5.15 zu sehen.

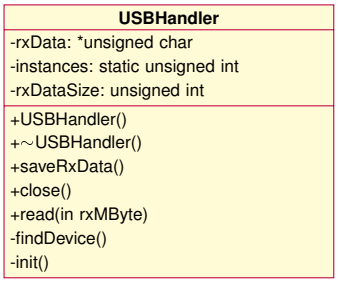

Abbildung 5.15.: Klassendiagramm der Klasse USBHandler

5.3.2. Realisierung der Einlesemethode

Nun wird beschrieben, wie die Einlesemethode `read()` der Klasse `USBHandler` realisiert wurde. Diese Methode liest den Datenstrom aus dem FX2 ein. Die Abbildung 5.16 zeigt das Flussdiagramm für den Einlesevorgang. Wie in Kapitel 4.4 ausgearbeitet, wird eine Anforderungsschlange für den Transfer verwendet.

Es werden zunächst drei Sendeanforderungen verschickt (submit request 1-3), in denen jeweils 65536 Byte eingelesen werden sollen. In einer while-Schleife wird jede Anforderung abgebaut (reap) und in den buffer geschrieben (buffer). Anschließend wird sie wieder zum FX2 geschickt (submit). Die while-Schleife prüft nach jedem Durchlauf, ob noch mehr Daten eingelesen werden müssen. Dies geschieht durch eine Zählvariable, die die Schleifendurchläufe zählt. Wenn keine Daten mehr eingelesen werden müssen, steigt das Programm aus der Schleife aus und die drei Anforderungen werden abgebaut (reap request 1-3). Der komplette Quellcode zu dieser Methode ist im elektronischen Anhang A zu finden.

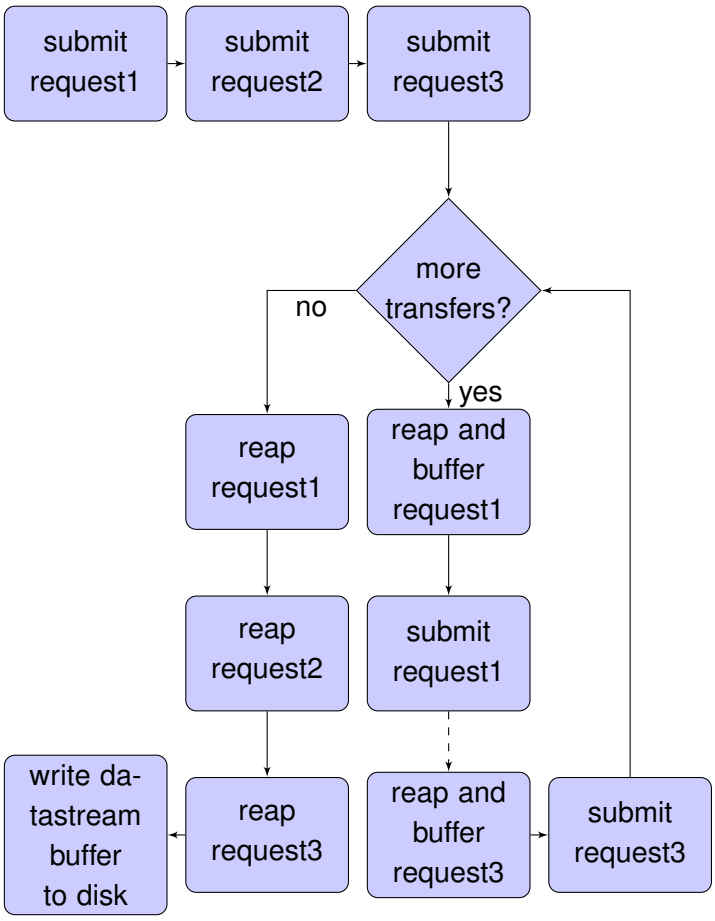

Abbildung 5.16.: Flussdiagramm für die Einlesefunktion

5.4. Verifikation mit CRC32-Prüfsumme

VHDL-Simulationen haben bereits ergeben, dass das Schaltwerk sicher funktioniert. Es musste aber zusätzlich geprüft werden, ob die Ansteuerung des FX2 auch auf der Hardware-Ebene funktioniert. Um die Funktionalität zu überprüfen, ist die Übertragung eines IEEE 802.3 konformen Ethernet-Datenstroms geeignet. Nach dem Empfang dieses Datenstroms wird anschließend die CRC-Prüfsumme jedes Ethernet-Frame überprüft, um beurteilen zu können, ob das System die Daten korrekt überträgt.

Es wurde eine Klasse entwickelt, mit deren Methoden die CRC32-Prüfsumme eines Ethernet-Frames berechnet wird. Das Klassendiagramm ist in Abbildung 5.17 zu sehen. Der verwendete Algorithmus zur Berechnung der Prüfsumme wurde der Quelle [Wikipedia] entnommen. Es wurde eine Klassenhierarchie entworfen, in der die Basisklasse `CalcCRC` den übernommenen Algorithmus beinhaltet. Durch diesen hierarchischen Aufbau wird eine Trennung der selbst entwickelten Elemente und dem übernommenem Algorithmus erreicht.

Basierend auf der Klasse `CalcCRC` wurde eine Klasse mit dem Namen `CalcCRCPcap` abgeleitet. Dem Kontruktor der Klasse wird eine Liste von Frames übergeben, in der Ethernet-Frames in Rohform abgelegt sind. Einzelheiten zu dieser Liste werden im Kapitel 6.2.4 näher erläutert. In der Methode `calcEthernetCRC()` wird zu jedem Frame in der Liste die Prüfsumme berechnet und mit der sich im FCS-Feld befindenden Prüfsumme verglichen. Der Vergleich wird in der Methode `validateCRC()` durchgeführt. Sind die Prüfsummen identisch, trat kein Fehler bei der Übertragung auf. Falls sie nicht identisch sind, liegt ein Fehler im Frame vor, und das Attribut `errors` wird inkrementiert. Der komplette Quellcode dieser Klasse ist im elektronischen Anhang A zu finden.

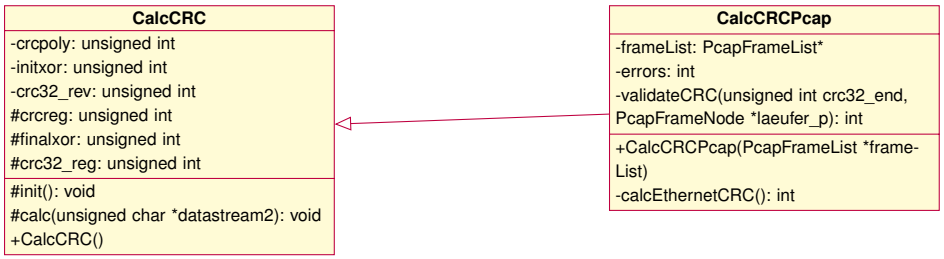

Abbildung 5.17.: Klassendiagramm der Klassenhierarchie CalcCRCPcap

6. Design des Interpretationsmoduls

Das Interpretationsmodul verarbeitet die empfangenen Rohdaten und stellt sie dem Benutzer in Wireshark dar. Es besteht aus zwei Komponenten. Eine Komponente ist ein Konvertierungsprogramm, das die empfangenen Rohdaten in ein Format umschreibt, mit dem Wireshark arbeiten kann. Dieses Format trägt die Bezeichnung pcap und wird im Folgenden erläutert. Die andere Komponente ist der Wireshark-Dissector für das jeweilige Protokoll.

6.1. Das pcap-Format

Eine pcap-Datei besteht aus einem Global-Header-Feld, dem die Kombination aus den Feldern „Packet-Header“ und „Packet-Data“ folgt. Diese Kombination beschreibt einen einzelnen Frame in einer pcap-Datei.

Global-Header	Packet-Header	Packet-Data	Packet-Header	Packet-Data	$\rightarrow$

Global Header: Im Global-Header-Feld wird der Verbindungstyp („network“) spezifiziert, dem die folgenden Nutz-Daten angehören. Für jeden Verbindungstyp, wie zum Beispiel Ethernet oder WLAN gibt es eine bestimmte Zahlenkonstante. Die anderen Parameter dieses Feldes können in der Quelle [Wireshark] nachgeschlagen werden.

Packet Header: Im Packet Header stehen Informationen zu den folgenden Daten. Diese Informationen lauten:

- ts_sec: Zeitstempel in Sekunden
- ts_usec: Zeitstempel in Mikrosekunden
- orig_len und incl_len: Anzahl der Bytes im Packet Data Feld.

Packet Data: In diesem Feld befinden sich die Nutz-Daten mit der im Packet Header angegebenen Länge. Dies sind die zu analysierenden Inhalte.

6.2. Design eines Konvertierungsprogramm

Um die Dokumentation des Konvertierungsprogramms zu strukturieren, orientiert sich dieses Kapitel an dem Weg, den die Daten von ihrem Zustand in Roh-Form bis hin zum Speichern im pcap-Format auf die Festplatte nehmen. Die Quelltexte der Software sind im elektronischen Anhang A zu finden.

6.2.1. Die Rohdaten

Um eine Programmierschnittstelle für den Datenstrom mit den Rohdaten zu schaffen, wurde die Klasse `RawData` erstellt, deren Konstruktor zunächst die Roh-Daten von der Festplatte einliest. Als Schnittstelle zu den eingelesenen Daten gibt es die Methode `unsigned char *nextByte()` mit der ein Byte aus den Roh-Daten zurückgegeben wird, während der Index-Zähler automatisch bei jedem Aufruf der Methode inkrementiert wird. Mit der Methode `unsigned char *getPayload()` erhält man einen Zeiger auf ein zusammenhängendes char-Feld aus den Rohdaten. Die Abbildung 6.1 zeigt das Klassendiagramm.

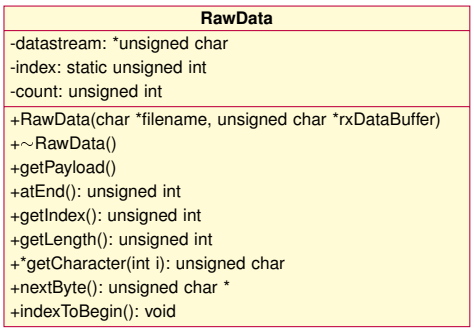

Abbildung 6.1.: Klassendiagramm der Klasse RawData

6.2.2. Konvertierung der Daten mit der HybridToPcapParser-Klasse

Eine Instanz der Klasse RawData benutzt ein Parser, um die Rohdaten in das pcap-Format zu konvertieren. Dieser Parser ist eine Instanz der Klasse `HybridToPcapParser`. Dieses Objekt erzeugt als Ergebnis des Parse-Vorgangs ein `PcapFrameList`-Objekt, das die pcap-konformen Frames in Form einer einfach verketteten Liste von `PcapFrame`-Objekten beherbergt. Jedes `PcapFrame`-Objekt enthält die Informationen, die von dem Pcap-Dateiformat für die Speicherung der Daten vorgesehen sind (vgl. Kapitel 6.1).

6.2.3. Parsen der Daten

Der Parser soll aus dem Datenstrom einzelne Frames extrahieren, um sie in einer pcap-Datei abspeichern zu können. Die Klasse `HybridToPcapParser` verwendet einen Zustandsautomaten für den Parse-Vorgang. Um die Frame-Abgrenzungen in den Rohdaten erkennen können, benötigt der Parser eine definierte Zeichenfolge, die den Start oder das Ende eines Frame kenntlich macht. Die Preambel des hybriden Protokolls ist hierzu geeignet. Die Abbildung 6.2 zeigt das Zustandsdiagramm zur Detektion eines neuen Frame.

Das Eingangssignal des Automaten ist der Datenstrom des hybriden Protokolls. Die Analyse des Datenstroms erfolgt Byte-weise. Der Initialisierungszustand ist der IDLE-Zustand. In diesem Zustand verharrt der Automat so lange, bis ein Byte mit dem hexadezimalen Wert 0x55 als Eingangssignal vorkommt. Dann erfolgt ein Übergang in den Zustand PREABMBLE1. Wenn das folgende Byte ebenfalls den Wert 0x55 besitzt, erfolgt ein Übergang in den PREAMBLE2-Zustand. Wenn nicht, dann erfolgt ein Übergang in den FRAME-Zustand. Mit diesem Verhalten soll erreicht werden, dass der Automat bei der Erkennung von 7 Bytes mit dem Wert 0x55 hintereinander und einem folgenden Byte mit dem Wert 0xD5 signalisiert, dass eine Preambel, also ein neuer Frame erkannt wurde. Bei der Erkennung eines neuen Frame beträgt der Wert des Signals `newFrame` 1, sonst 0.

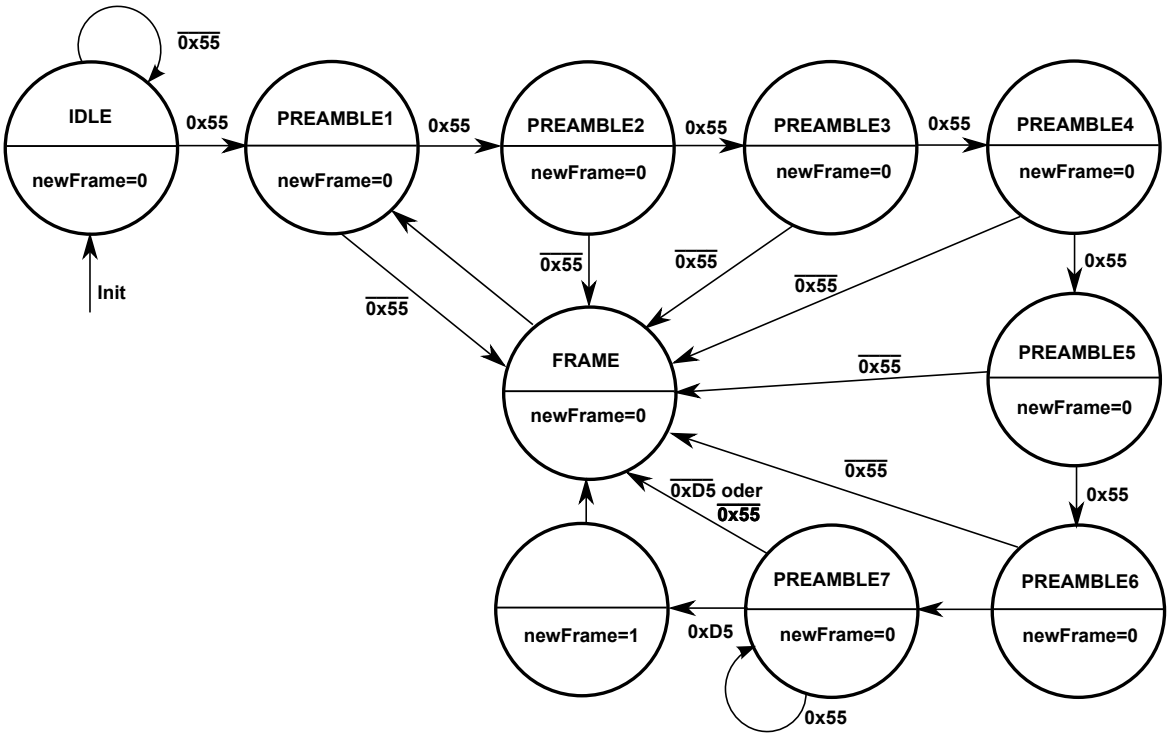

Abbildung 6.2.: Zustandsdiagramm des Datenstrom-Parser

6.2.4. Definition der Frameliste

Wie bereits im Kapitel 6.2.2 beschrieben, erzeugt der Parser eine verkettete Liste von pcap-konformen Frames. Der Vorteil einer verketteten Liste von Frames liegt darin, dass man auf jeden Frame der Rohdaten einzeln zugreifen kann. Durch wenige Änderungen in der Software können so zusätzliche Programm-Funktionen eingefügt werden können. Die Realisierung der CRC-Prüsummenberechnung von IEEE 802.3 konformen Frames für die Verifikation des Einlesemoduls ist ein Beispiel, in dem man von der Listenstruktur profitiert.

Um eine verkettete Liste zu realisieren, bietet es sich an, den Container „list“ der Standard Template Library (STL) von C++ zu verwenden. Mit diesem Template[1] kann eine Liste von Objekten eines bestimmten Typs erstellt werden. Der Vorteil von dieser Lösung ist, dass viele Funktionen für die Verarbeitung einer Liste bereits implementiert sind. Im Laufe der Realisierung der Software kam es zu Abstürzen des Programms. Die Fehlerquelle hierfür lag beim Löschen der Frame-Liste, die mit dem Template der STL realisiert wurde. Da diese Frame-Liste nicht den Schwerpunkt des Analyse-Systems darstellt, wurde die exakte Lokalisierung des Fehlers wegen Mangel an Zeit nicht ausführlicher behandelt. Stattdessen wurde eine eigene verkettete Liste definiert.

[1] Template - engl.: Schablone

Das Klassendiagramm der einfach verketteten Liste ist in Abbildung 6.3 zu sehen. Die Klasse `PcapFrame` beschreibt einen einzelnen Frame mit seinen Informationen. Die Attribute `ts_sec` und `ts_usec` beschreiben zum Beispiel den Zeitstempel eines Frame. `PcapFrameNode` ist eine abgeleitete Klasse von `PcapFrame`. Sie erweitert die Klasse um einen Zeiger auf sich selbst, wodurch die Realisierung einer verketteten Liste möglich wird. Außerdem enthält die Klasse die Methode `setNextNode()`, die einen neuen Knoten an die Liste anhängt.

Um eine Klasse zu realisieren, die die Liste verwaltet, wurde die Klasse `PcapFrameList` erstellt. Die Verwaltungsoptionen begrenzen sind auf das Einfügen eines Elementes in die Liste und das Löschen und die Anzeige aller Elemente. Diese Klasse `PcapFrameList` besitzt vier Zeiger, die auf Objekte der Klasse `PcapFrameNode` zeigen können. Die Beziehung zwischen den beiden Klassen ist eine Komposition, da die Lebensdauer der `PcapNode`-Objekte, auf die die vier Zeiger zeigen, von der Lebensdauer der Klasse `PcapFrameList` abhängt (vgl. [Bernhard Lahres, Gregor Rayman (2009)]).

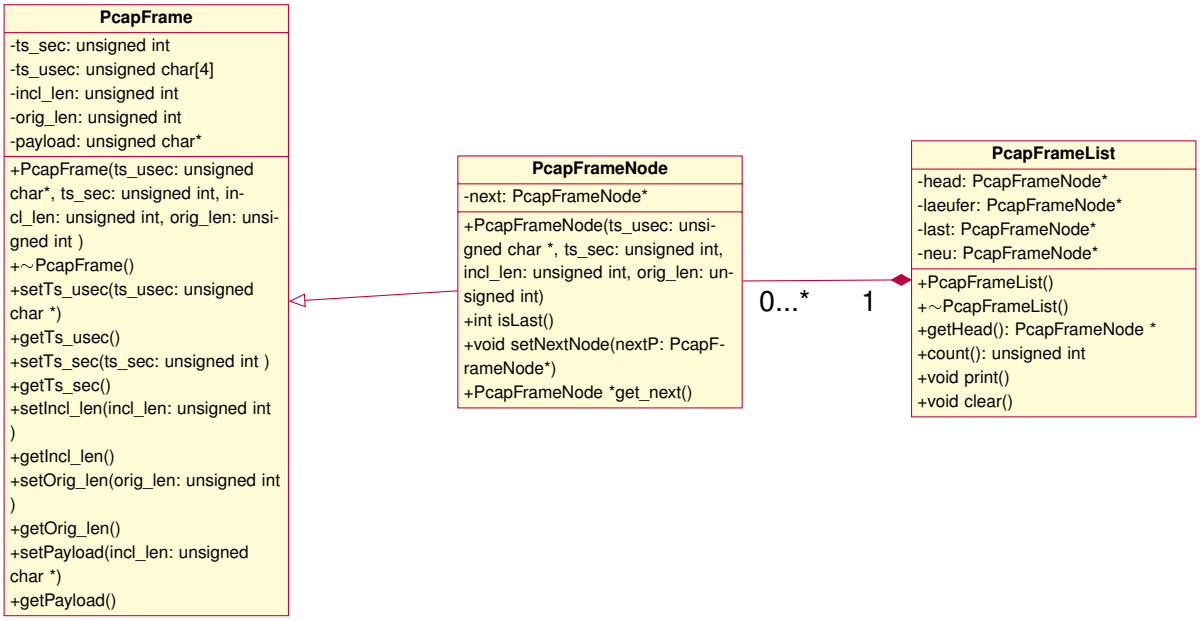

Abbildung 6.3.: Klassenhierarchie der Klasse PcapFrameList

6.2.5. Speichern der Daten mit der PcapFile-Klasse

Für die Beschreibung der zu erstellenden pcap-Datei wurde die Klasse `PcapFile` entworfen. Dem Konstruktor der Klasse `PcapFile` wird ein Zeiger auf die Frameliste übergeben. Es wird nur ein Zeiger übergeben, da die Frameliste den gesamten Datenstrom enthält und bei einer Einlesemenge von 300 MByte das Objekt viel Speicherplatz einnimmt. Um das Kopieren der Daten zu vermeiden, wird ein Zeiger des Objektes übergeben.

Die `PcapFile`-Klasse beherbergt neben dem Zeiger auf die Frameliste auch einen Zeiger auf ein PcapHeader-Objekt, welches den „Global-Header“ einer pcap-Datei repräsentiert. Die PcapFile-Klasse speichert den „Global-Header“ und die Frames mit Hilfe der Methode `saveToDisk(char *filename)` ab. Das Klassendiagramm der Klasse ist in der Abbildung 6.4 zu sehen.

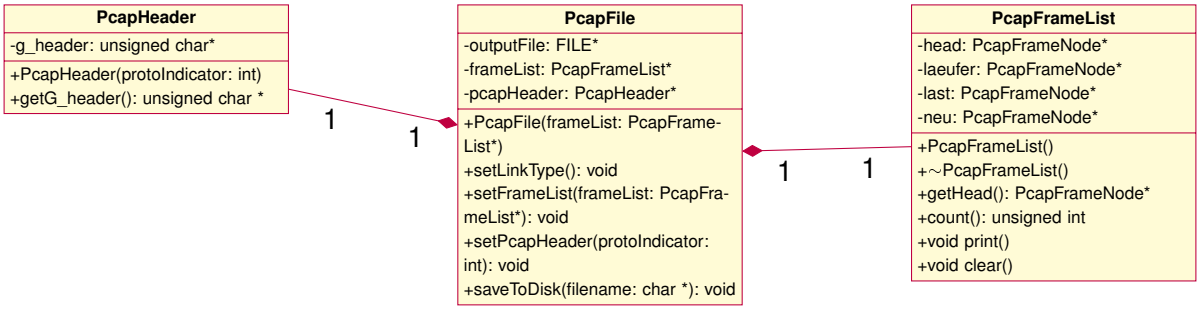

Abbildung 6.4.: Klassendiagramm der Klasse PcapFile

6.3. Design der Wireshark Dissector-Plugins

Für das Design der Dissectors musste zunächst analysiert werden, wie viele Protokolltypen es im hybriden Bussystem gibt. Eine Übersicht über die verschiedenen Protokolltypen lässt sich übersichtlich mit Hilfe einer Baumstruktur darstellen. In Abbildung 6.5 ist sie zu sehen. Von dem Wurzelelement „CIDS Protokolltypen“ (schwarz) gehen zwei Zweige aus, die mit Top- und Middle-Line (braun und grün) bezeichnet werden. Dies sind die zwei verschiedenen Datenbusse des CIDS.

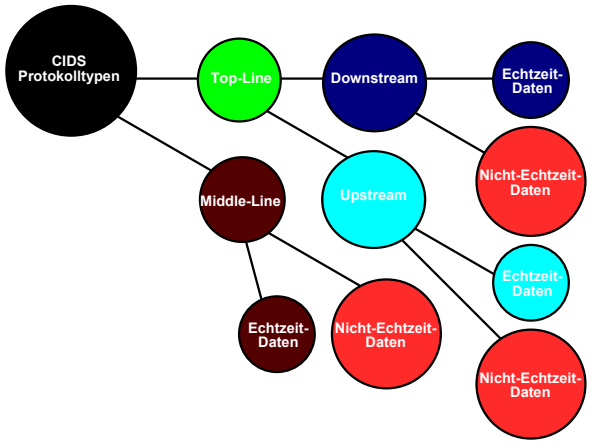

Abbildung 6.5.: Baumstruktur zur Übersicht über die verschiedenen Protokolltypen

Von dem Top-Line Knoten gehen zwei Zweige aus, die zu den Knoten Upstream (hellblau) und Downstream (dunkelblau) führen, da diese sich unterscheiden. Bei der Middle-Line wird diese Unterscheidung nicht gemacht, da die Protokolle für Up- und Downstream identisch sind. Die Endknoten mit der Aufschrift Up- und Downstream zweigen jeweils zwei Knoten ab, die die Echtzeit-Daten-Frames und die Nicht-Echtzeit-Daten-Frames repräsentieren. Die Nicht-Echtzeit-Frames besitzen bei sämtlichen Verbindungstypen das gleiche Protokoll. Für Wireshark müssen deshalb 4 Protokoll-Dissectors für die verschiedenen Protokolle entwickelt werden.

Zwischen den drei Protokolltypen (Middle-Line, Top-Line Uptream und Top-Line Downstream) soll der Benutzer manuell entscheiden können, welches Protokoll für die Verarbeitung der Rohdaten verwendet werden soll. Die Realisierung hierfür wird im Kapitel 7.2.1 (Auswahl der Leitungsverbindung) behandelt.

Innerhalb dieser drei Protokolle können Echtzeit-Frames und Nichtechtzeit-Frames auftreten. Hier muss das Analyseprogramm selbstständig unterscheiden können, um welchen Frame-Typ es sich handelt. Die Realisierung hierfür wird im Kapitel 7.2.2 (Automatische Protokolldetektion) behandelt.

7. Realisierung der Interpretationssoftware

7.1. Realisierung des Zustandsautomaten für den HybridToPcapParser

In diesem Kapitel wird die Realierung des Zustandsautomaten beschrieben, mit dem die Präambel in dem Datenstrom des hybriden Bussystem detektiert wird. Der Zustandsautomat ist in Kapitel 6.2.3 zu sehen. Eine Möglichkeit, einen Zustandsautomaten zu realisieren, ist die Verwendung einer switch-case Anweisung. Um den objektorientierten Programmieransatz zu verfolgen, werden zwei alternative Konzepte vorgestellt.

1. Konzept: State Design Pattern Dies ist ein gebräuchliches Konzept für der Realisierung von Zustandsautomaten in Software mit Hilfe objektorientierter Programmierung. Das Konzept sieht vor, dass das Verhalten eines Objekts abhängig von seinem Zustand ist. Für jeden Zustand (Objekt) wird dabei eine eigene Klasse eingeführt, die das Verhalten des Objekts in diesem Zustand definiert. Jeder dieser Zustände ist für gewöhnlich eine Ableitung einer abstrakten Klasse „State“ (vgl. [Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (2004)]). Ein Vorteil bei der Verwendung von State Design Pattern ist, dass komplexe Bedingungsausdrücke vermieden werden können. Die Wartbarkeit wird erhöht, indem der Quellcode leichter lesbar ist und leichter erweitert werden kann. Der Nachteil ist, dass der Implementierungs-Aufwand für einen Automaten mit einer simplen Struktur, wie sie der hier zu realisierende Automat besitzt, hoch ist.

2. Konzept: Verkettete Objekte vom Typ StateItem Dieses Konzept ist ein proprietäres Konzept, in der die einzelnen Zustände ebenfalls durch eine Klasse beschrieben werden. Diese Klasse beschreibt einen Zustand durch seine Bezeichnung und sein Ausgangssignal. In die Automatenstruktur können im Gegensatz zum 1. Konzept keine Zustände mit anderem Verhalten hinzugefügt werden. Der Grund hierfür wird geklärt, indem das Konzept nun erläutert wird.

Äquivalent zu den 256 möglichen Werten des Eingangssignals besitzt ein StateItem-Objekt 256 Zeiger, die auf ein nächstes StateItem-Objekt zeigen (Zeile 4 des Listing 7.1). Ein Zeiger auf das nächste StateItem-Objekt repräsentiert den Pfeil auf den nächsten Zustand in einem Zustandssymbol eines Zustandsdiagramms.

Beim Erzeugen des Automaten werden die Zustandsübergange mit der Methode `setNextState()` Methode definiert (Zeile 6). Abhängig von dem Parameter `input` wird ein Zustand mit einem anderen Zustand `nextState` über einen Zeiger verbunden (Zeile 8).

Beim Durchlaufen des Zustandsautomaten in der Anwendung wird mit der Methode `getNextState()` (Zeile 10) der Übergangsszustand eines Zustands mit Hilfe des Eingangssignals `input` ermittelt (Zeile 12). Die Ermittlung des nächsten Zustands erfolgt ohne Schachtelung von Abfragen, sondern durch diese Lookup-Tabelle.

```
1  class StateItem : public State
2  {
3    private:
4      StateItem *nextState[256];
5    public:
6      void setNextState(unsigned char input, StateItem *nextState)
7      { // Festlegen des nächsten Zustands.
8        *(this->nextState+input) = nextState;
9      }
10     StateItem *getNextState(unsigned char *input)
11     { // Ermitteln des nächsten Zustands.
12       return *(this->nextState+(*input));
13     }
14 };
```

Listing 7.1: Auszug aus der Definition der Klasse StateItem

Das Listing 7.2 zeigt den Automaten in der Anwendung. Die StateItem-Objekte werden von einer Klasse mit dem Namen `StateMachine` verwaltet.

```
1 int StateMachine::run(unsigned char *input)
2 { // help2 entspricht dem Übergangszustand und help dem aktuellen Zustand.
3   help2 = help->getNextState(input);  // Ermitteln des nächsten Zustands aus dem
        Eingangssignal.
4   newFrame = help2->isNewFrame();    // Liegt ein neuer Frame vor?
5   help  = help2;     // Der aktuelle Zustand wird der nächste Zustand.
6   return newFrame;  // Ergebnis, ob ein neuer Frame vorliegt, zurückgeben.
7 }
```

Listing 7.2: Auszug aus der Definition der Klasse StateMachine

Ein Zustand wird durch die Klasse `State` beschrieben. Die Klassenhierarchie ist in Abbildung 7.1 zu erkennen. Der Aufbau dieser Klasse ist auf das in diesem Kontext zu realisierende Problem zugeschnitten. Der Zustandsautomat ist ein reiner Moore-Automat. Dazu besitzt die Klasse `State` das Attribut `newFrame`, das als Ausgangssignal des jeweiligen Zustands fungiert. Es besitzt den Wert 1 bei der Erkennung eines neuen Frames und den Wert 0 in allen anderen Zuständen. Die Klasse `StateItem` ist eine Vererbung der Klasse `State` und ihre Aufgabe wurde bereits geschildert.

Um eine Vielzahl von diesen Objekten zu verwalten, wurde eine Ableitung der Klasse `StateItem` definiert. Dies ist die Klasse `StateItemNode` und ergänzt ihre Basisklasse um einen Zeiger `nextState`. Durch diesen Zeiger und weiteren Methoden der Klasse ist es möglich diese Objekte in einer Liste zu verwalten. Diese Liste wird durch die Klasse `StateMachine` verwaltet. Der Konstruktor der Klasse übernimmt ein `StateItemNode`-Objekt, dass der Initialzustand ist. Um weitere Zustände in den Automaten einzufügen wurde die Methode `insert()` definiert. Die Klasse `StateMachine` besitzt, wie bereits in Listing 7.2 beschrieben, die Methode `run()`, um den Automaten in der Anwendung zu verwenden. Mit der Methode `clear()` werden sämtliche Zustände gelöscht.

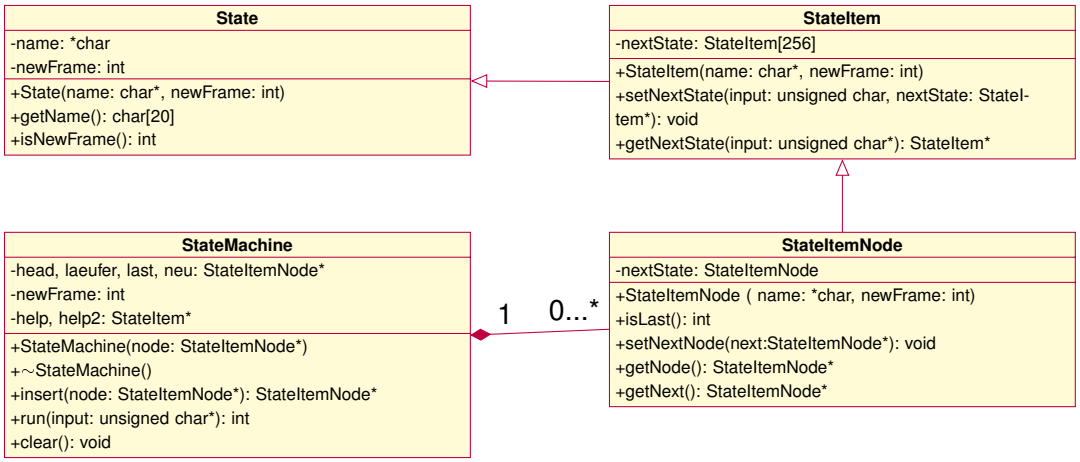

Abbildung 7.1.: Klassenhierarchie der Klasse StateItem

Das Listing 7.3 zeigt die Definition von drei Zuständen und ihren Folgezuständen. Der Zustandsautomat aus Kapitel 6.2.3 in Abbildung 6.2 dient dabei als Grundlage. Zuerst werden die Zustände definiert und anschließend ihre Folge-Zustände festgelegt.

```
1  // Zustände aus StateItemNode-Klasse erzeugen
2  StateItemNode *preamble7 = new StateItemNode((char*)"PREAMBLE7");
3  StateItemNode *framestart = new StateItemNode((char*)"FRAMESTART", 1);  // Eine
       neuer Frame beginnt.
4  StateItemNode *frame = new StateItemNode((char*)"FRAME DATA");
5  // Folgezustände definieren
6  for(int i=0; i<=0xFF; i++)
7  {
8    preamble7->setNextState(i, frame);
9    framestart->setNextState(i, frame);
10   frame->setNextState(i, frame);
11 }
12 preamble7->setNextState(0x55, preamble7);
13 preamble7->setNextState(0xD5, framestart);  // Es wurde eine Preambel gefunden.
14 frame->setNextState(0x55, preamble1); // Ist es eine neue Preambel?
```

Listing 7.3: Definition der Folgezustände

Ein Nachteil von diesem Konzept ist, dass es eine Lösung auf dieses konkrete Problem darstellt und aufwendig erweitert werden muss, um es auf andere Problemlösungen anzuwenden. Das erste Konzept ist in der Hinsicht flexibler, da ein neuer Zustand mit anderem Verhalten in die bestehende Klassen-Konstellation hinzugefügt werden kann. Ein weiterer Nachteil bei dem 2. Konzept ist, dass zu jedem Zustand immer 256 Zeiger gehören, was Speicherplatz kostet. Ein Vorteil von dem 2. Konzept ist, dass die Zustandsübergänge in der Anwendung schnell von statten gehen, da eine Lookup-Tabelle, im Gegensatz zu Abfragen von Bedingungsausdrücken, schneller ist. Um die genannten Vor- und Nachteile erarbeiten zu können, wurde 2. Konzept für die Realisierung der Konvertierungssoftware umgesetzt.

7.2. Realisierung eines Wireshark-Dissectors

Für die Realisierung der Wireshark-Dissectors wurde der „Skeleton-Code"[1] aus der Quelle [Wireshark] erweitert. Die Entwickler-Gemeinde von Wireshark stellt diesen Quelltext für die Entwicklung von Dissectors zu Verfügung. Es wurden drei Dissectors für das Top-Line-Upstream-, Top-Line-Downstream- und Middle-Line-Protokoll erstellt.

[1] skeleton - engl.: Gerüst

7.2.1. Auswahl des Datenbusses

Die Auswahl, mit welchem Protokoll die Daten interpretiert werden sollen, wird vom Benutzer eingestellt. Die Auswahl wird durch den Parameter „network“ des Global Header (vgl. Kapitel 6.1) realisiert. Dieser Parameter stellt eine Zahlenkonstante dar, die den Protokoll-Typ identifiziert. Das Konvertierungsprogramm berücksichtig diesen Parameter in der Klasse `PcapHeader`.

Im Listing 7.4 ist die Funktion `proto_reg_handoff_hybrid_nonrt()` zu sehen (Zeile 1). Diese Funktion hat den Zweck, Wireshark in Kenntnis zu setzten, dass es diesen Dissector gibt. Der Parameter `dissect_hybrid_rt_top_down` in der Parameterliste der Funktion `create_dissector_handle()` ist ein Zeiger auf die Funktion, die einen Frame in seine Bestandteile aufteilt (Zeile 4).

Die Funktion `dissector_add()` definiert, bei welcher Zahlenkonstante, also bei welchem Verbindungstyp der Dissector verwendet werden soll. Der Dissector für das Protokoll der Top-Line-Downstream-Verbindung verwendet die Zahlenkonstante WTAP_ENCAP_USER3, wie im Listing zu erkennen ist (Zeile 5).

```
1 void proto_reg_handoff_hybrid_nonrt(void)
2 {
3   static dissector_handle_t hybrid_rt_top_down;
4   hybrid_rt_top_down = create_dissector_handle(dissect_hybrid_rt_top_down,
        proto_hybrid_rt_top_down);
5   dissector_add("wtap_encap", WTAP_ENCAP_USER3, hybrid_rt_top_down); //
        WTAP_ENCAP_USER3 Konstante, siehe Datei ../wiretap/pcap-common.c
6 }
```

Listing 7.4: Ausschnitt aus der Datei packet-hybrid_topline_downstream.c

Die Datei ./wiretap/pcap-common.c beinhaltet die Auflistung aller in Wireshark verfügbaren Verbindungstypen. In dieser Datei sind die Zahlenkonstanten für die verschiedenen Datenbusse des hybriden Bussystems eingetragen, damit Wireshark ihnen einen bestimmen Dissector zuordnen kann. Das Listing 7.5 zeigt die Zahlenkonstanten für die Verbindungs-Typen des hybriden Bussystems.

```
1   { 148,     WTAP_ENCAP_USER1 }, // REALTIME PROTOCOL MIDDLELINE UP- AND
        DOWNSTREAM
2   { 149,     WTAP_ENCAP_USER2 }, // REALTIME PROTOCOL TOPLINE UPSTREAM
3   { 150,     WTAP_ENCAP_USER3 }, // REALTIME PROTOCOL TOPLINE DOWNSTREAM
```

Listing 7.5: Anpassung der Datei ./wiretap/pcap-common.c

7.2.2. Automatische Frametyp-Detektion

Um innerhalb eines Datentromes erkennen zu können, ob es sich um einen Echtzeit- oder Nicht-Echtzeit-Frame handelt, wird eine automatische Frametyp-Detektion benötigt. Für eine automatische Detektion, muss es einen Mechanismus geben, bei dem zunächst der Inhalt jedes Frames geprüft wird. Nur dadurch ist es möglich, feststellen zu können, ob es sich um Echtzeit- oder Nicht-Echtzeit-Daten handelt. Um dies im hybriden Protokoll festzustellen, gibt es ein Flag, das festlegt, ob die folgenden Daten Echtzeit- oder Nicht-Echtzeit-Informationen sind. Um die automatische Detektion zu realisieren, eignet sich das Prinzip der „heuristischen Dissectoren“, welches durch das Listing 7.6 verdeutlicht wird.

Zu Beginn der Funktion, die das Zerlegen des Frames durchführt, wird der heuristische Dissector mit der Funktion `dissector_try_heuristic()` aufgerufen (Zeile 6). Der heuristische Dissector ist im Fall des hybriden Bussystem der Dissector für die Nicht-Echtzeit-Frames. Falls der Frame von dem heuristischen Dissector zerstückelt wurde, bricht die Funktion durch den Ausdruck `return` die weitere Verarbeitung ab (Zeile 6). Falls der Frame nicht von dem heuristischen Dissector verarbeitet wurde, fährt der Dissector für die Echtzeit-Frames mit der Verarbeitung fort.

```
1 static void dissect_hybrid_rt_top_down(tvbuff_t *tvb, packet_info *pinfo,
      proto_tree *tree)
2 {
3   gint offset = 0;
4   guint8 length = 0;
5
6   if( dissector_try_heuristic(heur_subdissector_list, tvb, pinfo, tree) )
7     return;
```

Listing 7.6: Ausschnitt aus der Datei packet-hybrid_topline_downstream.c

Der heuristische Dissector überprüft einen Frame, ob er ihn zerteilen soll. Dies erfolgt durch die if-Abfrage in Zeile 7 des Listing 7.7. Die Variable `bit` enthält den Wert des Flag, das anzeigt, ob ein Frame ein Echtzeit- oder Nicht-Echtzeit-Frame ist. Die Ermittlung, dieses Wertes erfolgt in der Zeile 6 durch die Funktion `tvb_get_bits8()`. Diese Funktion extrahiert ein Bit aus den Daten, wobei der Offset im Frame 71 beträgt. Dies ist das 7. Bit im 8. Byte jedes Frame. Besitzt dieses Bit den Wert 1, bricht der heuristische Dissector die Verarbeitung ab und gibt dem Standard-Dissector den Wert `FALSE` zurück. Wenn dieses Bit den Wert 0 besitzt, ist dieser Frame ein Nicht-Echtzeit-Frame und der heuristische Dissector fährt mit der Verarbeitung fort, indem er den Frame zerteilt. Am Ende der Funktion gibt der Dissector den Wert TRUE zurück, wodurch dem Standard-Dissector mitgeteilt wird, dass der Frame bereits zerteilt wurde.

```
1 static gboolean dissect_hybrid_nonrt(tvbuff_t *tvb, packet_info *pinfo,
      proto_tree *tree)
2 {
3   gint offset = 0;
4   guint8 length = 0;
5
6   guint8 bit = tvb_get_bits8(tvb, 71, 1); // Dieses Bit ist das RT_Frame Bit (7.
        Bit im 8. Byte des Frames)
7   if (bit == 1)
8     return FALSE;
```

Listing 7.7: Ausschnitt aus der Datei packet-hybrid_nonrt.c

Das Zerteilen eines Frame wird im Listing 7.8 gezeigt. In diesem Beispiel zerstückelt das Plugin einen Teil des „IP-Fragment“ des Nicht-Echtzeit-Protokolls. Die Funktion `proto_tree_add_item()` erzeugt in Zeile 2 des Listing 7.8 den Eintrag für das Timestamp-Feld in Wireshark. Der Parameter `hf_hybrid_timestamp` definiert die Eigenschaften des Feldes. Mit dem Parameter `offset` wandert der Dissector Byte-weise durch den Frame.

```
1    /********** IP Fragment ********/
2    proto_tree_add_item(hybrid_tree, hf_hybrid_timestamp, tvb, offset, 4, FALSE);
         offset += 4;
3    proto_tree_add_item(hybrid_tree, hf_hybrid_src, tvb, offset, 1, FALSE);
4    proto_tree_add_item(hybrid_tree, hf_hybrid_seqnr, tvb, offset, 2, FALSE);
       offset += 2;
```

Listing 7.8: Ausschnitt aus der Datei packet-hybrid_nonrt.c

Die Definition des Parameters `hf_hybrid_timestamp` wird in Listing 7.9 dargestellt. Die Zeichenkonstante `"IP/Timestamp"` gibt die Bezeichnung des Feldes an (Zeile 3). Unter dieser Bezeichnung wird das Feld in Wireshark angezeigt. Der Parameter `BASE_HEX` gibt an, dass das Feld im hexadezimalen Format angezeigt wird (Zeile 4). Der Parameter `FT_UINT32` gibt Wert-Typ für das Feld an. In diesem Fall lautet der Typ 32-bit unsigned integer (Zeile 4). Die anderen Parameter können in der Quelle [Wireshark] nachgeschlagen werden.

```
1    { &hf_hybrid_timestamp,
2      {
3        "IP/Timestamp", "hybrid_non_rt.timestamp",
4        FT_UINT32, BASE_HEX,
5        NULL, 0,
6        NULL, HFILL
7      }
8    },
```

Listing 7.9: Ausschnitt aus der Datei packet-hybrid_nonrt.c

8. Systemintegration und Verifikation

8.1. Integration der Hardwarekomponenten

In diesem Kapitel wird gezeigt, wie in dem Prototyp-Aufbau des Analyse-Systems die Hardwarekomponenten miteinander verbunden wurden. In der Abbildung 8.1 ist dazu ein Blockschaltbild zu sehen. Im Prototypen des hybriden Bussystems werden D-Sub-Steckverbinder zum Verbinden der Komponenten verwendet. Dazu mussten Kabel angefertigt werden, die auf der einen Seite einen D-Sub-Stecker und auf der anderen Seite einen RJ-45-Stecker besitzen (siehe Markierungen 1 und 2). Es wurden Cat5-STP-Kabel als Übertragungsmedium verwendet. Im Laufe der Entwicklung wurden als Adapter (Markierung 3) Einzeladern verwendet.

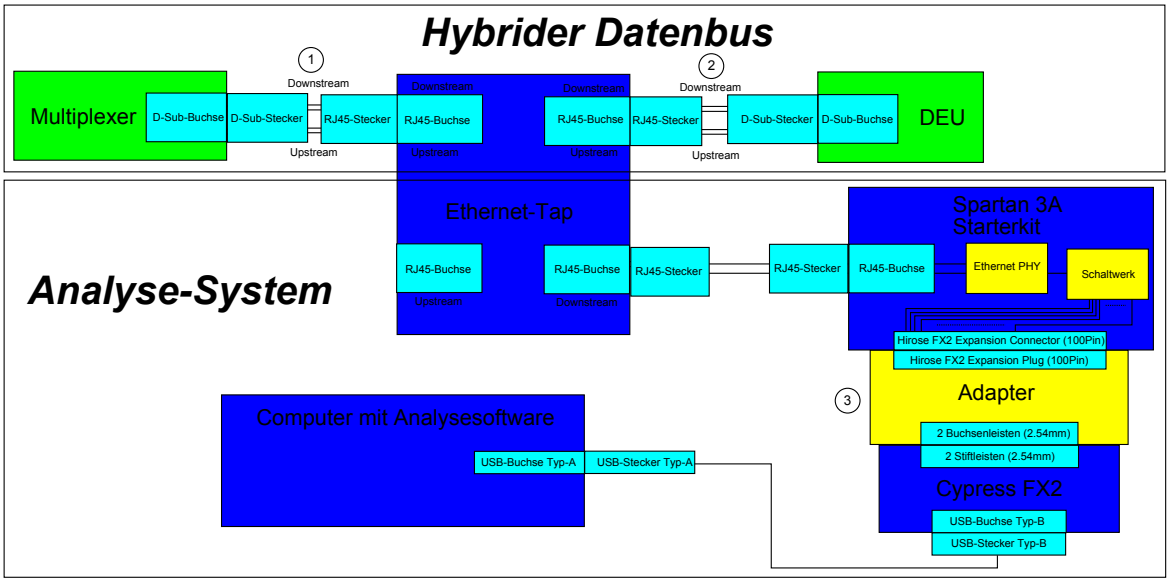

Abbildung 8.1.: Verbindung der Hardwarekomponenten

8.2. Integration der Softwarekomponenten

8.2.1. Integration der Konvertierungssoftware

Es wurde eine Software mit einer graphischen Benutzeroberfläche entwickelt, durch die sich ein Einlesevorgang durchführen lässt. Die Benutzeroberfläche ist in Abbildung 8.2 zu sehen. Die Benutzeroberfläche wurde mit der C++-Klassenbibliothek Qt entwickelt. Die Schaltflächen und Auswahlmenüs wurden mit Elementen dieser Bibliothek realisiert.

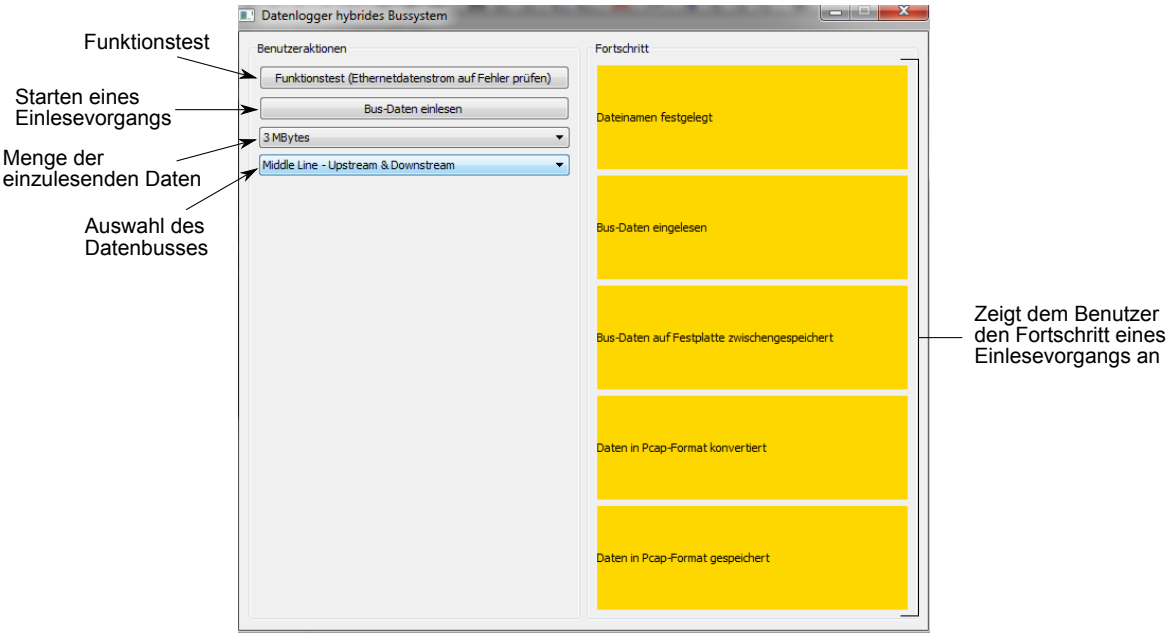

Abbildung 8.2.: Screenshot der grafischen Benutzeroberfläche

Um einen Einlesevorgang zu starten, gibt es die Schaltfläche „Bus-Daten einlesen“. Durch einen Klick auf diese Schaltfläche wird die Methode `readUSBDataSlot()` der Klasse `HybridLoggerGUI` aufgerufen. Ein Auszug aus der Methode ist in Listing 8.1 zu sehen. In dieser Methode werden wie in Kapitel 5.3.2 ausgearbeitet, mit der Methode `read()` die Daten über USB eingelesen (Zeile 6) und mit `saveRxData()` abgespeichert (Zeile 7). Hier endet der Aufgabenbereich des Einlesemoduls.

Ab diesem Punkt beginnt der Parse-Vorgang der Rohdaten. In Zeile 9 wird ein Objekt der Klasse `RawData` erzeugt, während dem Konstruktor der bereits allokierte Speicherplatz als Zeiger übergeben wird. Außerdem wird dem Konstruktor der Dateiname der Rohdaten übergeben. Im Konstruktor wird die Datei eingelesen und im übergebenen Speicherplatz abgelegt.

In Zeile 10 wird ein Objekt der Klasse `HybridToPcapParser` erzeugt und dem Konstruktor die Rohdaten übergeben, die er mit der Methode `parse()` in eine Liste von pcap-Frames konvertiert (Zeile 11). Mit der pcap-Frame-Liste wird ein Objekt der Klasse `PcapFile` erzeugt (Zeile 13). In der Methode `setLinkType()` wird das zu verwendete Protokoll festgelegt, das Wireshark für den „Dissect-Vorgang“ verwenden soll. Dies wird mit einer Zahlenkonstante realisiert, wie in Kapitel 7.2.1 beschrieben. Intern erzeugt die Klasse daraus ein `PcapHeader`-Objekt. Das PcapFile-Objekt speichert seinen Inhalt mit der Methode saveToDisk() auf der Festplatte ab. Anschließend werden die Objekte zerstört.

```
1  QString qFilename = QFileDialog::getSaveFileName(this, tr("Save File"), 0, "
       Wireshark Pcap Files *.pcap");
2  if(!qFilename.isEmpty())
3  {
4    cutFilename(qFilename);          // Dateiendung anpassen (Rohdaten werden mit
       der Endung .bin gespeichert)
5    unsigned int dataSize[]={3, 30, 300};
6    fx2->read( dataSize[dataSizeComboBox->currentIndex()] );// USB-Daten einlesen
7    fx2->saveRxData(filename);                  // USB-Daten abspeichern
8    unsigned char *rxDataBuffer=fx2->getRxDataBuffer();    // Pointer auf den
       reservierten Speicherplatz für die Rohdaten erhalten
9    RawData *rawData=new RawData(filename, rxDataBuffer); // Einlesen der Rohdaten
10   HybridToPcapParser *parser = new HybridToPcapParser(rawData); // Dem Konverter
       die Rohdaten übergeben
11   PcapFrameList *frameList=parser->parse();      // Die Rohdaten parsen
12   delete parser;
13   PcapFile *pcapFile=new PcapFile(frameList);        // PcapFile-Objekt mit der
       erstellten Frame-Liste erzeugen
14   pcapFile->setLinkType(linkTypeComboBox->currentIndex());  // Datenbus festlegen
15   pcapFile->saveToDisk(qFilename.toAscii().data());    // pcapFile Objekt
       abspeichern
16   printf("abgeschlossen\n");
17
18   delete rawData;
19   delete frameList;
20   delete pcapFile;
21 }
```

Listing 8.1: Ausschnitt des Quelltextes der Methode readUSBDataSlot()

8.2.2. Analyse der Daten mit Wirehark

Die mit der Konvertierungssoftware erstellte pcap-Datei lässt sich mit Wireshark öffnen. Die Abbildung 8.3 zeigt einen Screenshot[1], in dem die Frames des hybriden Protokolls dargstellt werden. Im oberen Fensterteil wird die Frame-Liste angezeigt. Hier kann man erkennen, ob es sich um einen Echtzeit-Frame (HYBRID_TOPLINE_RT_UPTREAM) oder um einen Nicht-Echtzeit-Frame (HYBRID_NONRT) handelt.

Im mittleren Fensterteil werden Informationen über die Ankunftszeit des Frame angezeigt. Automatisch berechnet Wireshark die zeitliche Differenz zu dem vorigen Frame. Wie erwartet beträgt die zeitliche Differenz der Frames 31 μs, während bei jedem vierten Frame die Differenz 32 μs beträgt, weil die exakte Differenz 31,25 μs lautet. Außerdem wird die Frame-Länge angezeigt. Unter diesen Informationen wird der Inhalt des Frame angezeigt. Dabei wird der Frame in die Felder des jeweiligen Protokolls, wie das „Type“- oder das „DEU-Address“-Feld, aufgesplittet. Im unteren Fensterteil wird der Frame in Roh-Form wahlweise in Hex- oder Binär-Format angezeigt.

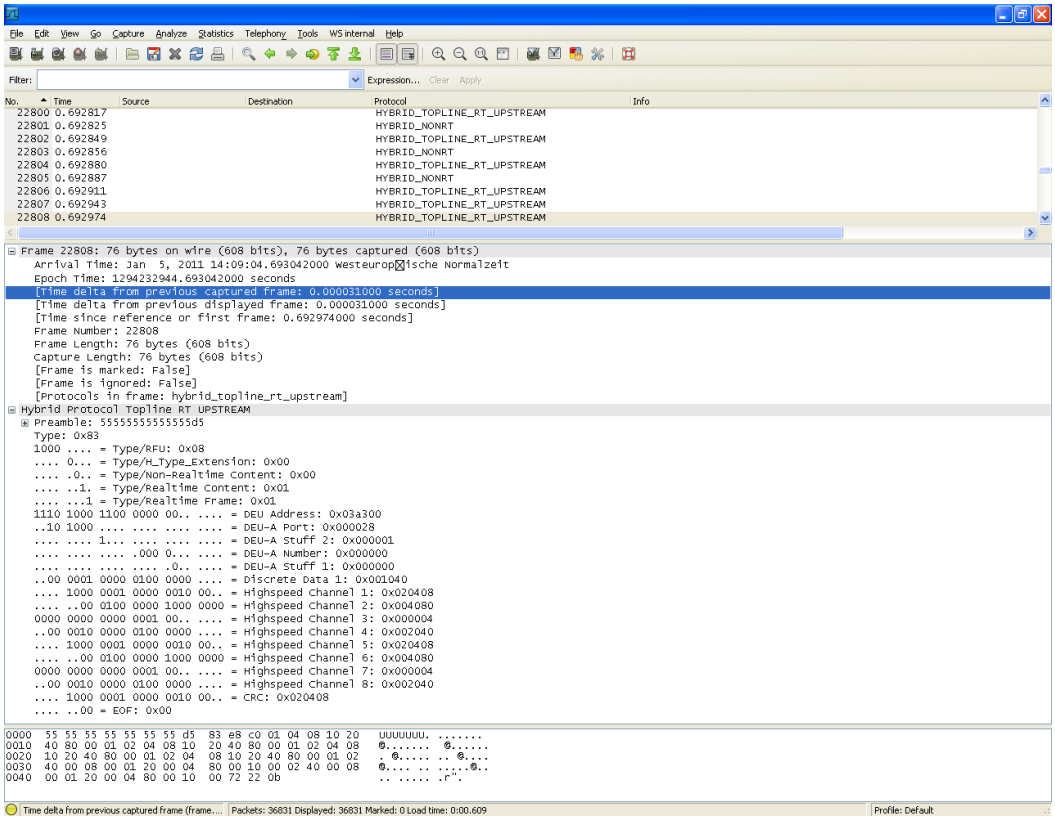

Abbildung 8.3.: Screenshot von Wireshark

[1] screenshot - engl.: Bildschirmausdruck

8.3. Verifikation

Verifikation mit CRC32-Prüfsumme: In der Verifikation wird die Funktionalität des Analysators mit der in Kapitel 5.4 ausgearbeiteten Prüfsummenberechnung nachgewiesen. Wenn die Übertragung von Ethernet-Daten funktioniert, funktioniert auch die Übertragung der Daten des hybriden Bussystems. Zum Starten eines Funktionstests gibt es in der Oberfläche der Software die Schaltfläche „Funktionstest“.

Klickt man die Schaltfläche an, öffnet sich eine Maske, in der man festlegen kann, unter welcher Datei die Rohdaten abgespeichert werden sollen. Der folgende Ablauf ist identisch mit dem Ablauf in der Methode `readUSBDataSlot()`, die in Kapitel 8.2.1 erläutert wurde. Zusätzlich wird hier eine Prüfsummenberechnung mit den Frames durchgeführt. Bei 10 Einlesevorgängen mit je 300 MBytes eingelesenen Daten, wurden keine Fehler im empfangenen Datenstrom entdeckt. Damit ist kein 100%iger Nachweis aber ein hinreichend gutes Indiz erbracht, dass der Datenstrom korrekt übertragen wird.

Detektion von „silent errors“: „Silent errors“ sind Fehler, die während der Datenübertragung passieren und unentdeckt bleiben. In dem vorliegenden Übertragungssystem gibt es eine Fehlerquelle, die ohne weitere Maßnahmen unentdeckt bleiben würde. Diese Fehlerquelle ist das interne FIFO, das überläuft, wenn der USB-Transfer von FX2 zum Computer, langsamer ist, als der Eingangsdatenstrom des Ethernet-Transceivers. In dem VHDL-Schaltwerk ist in der ETH_USB_-Komponente ein Mechanismus implementiert, mit dem diese Fehlerquelle in Form einer leuchtetenden LED bei Überlauf, detektierbar ist. Während der 10 Einlesevorgänge, die als Tests durchgeführt wurden, ist dieser Fehlerfall aber nicht aufgetreten.

9. Zusammenfassung und Ausblick

Diese Arbeit beschreibt die Entwicklung eines Analyse-Systems, mit dem die Untersuchung von proprietären Ethernet-basierten Kommunikationsprotokollen möglich ist. Es werden Möglichkeiten aufgezeigt, wie die Ankopplung an ein zu analysierendes Netzwerk realisiert werden kann. Aus diesen Möglichkeiten hat sich der Ethernet-Tap als optimale Lösung herausgestellt.

Um die Daten in den Computer einzulesen, wurde eine Analyse durchgeführt, in der geprüft wurde, ob man einen Standard-Ethernet-Controller für den Einlesevorgang verwenden kann. Das Ergebnis dieser Analyse ist, dass ein alternatives System nötig ist, um die Daten einzulesen. Dies hat den Grund, dass Ethernet Frames, die nicht dem IEEE 802.3-Standard entsprechen, von einem Ethernet-Controller verworfen werden. Mit dem Xilinx Spartan 3A Entwicklungsboard und dem Cypress FX2 Mikrocontroller wurde ein Einlesemodul realsiert, mit dem die Daten per USB über die Transferart Interrupt-Transfer von dem Computer eingelesen werden.

Es wurden verschiedene Realisierungsmöglichkeiten für die Analyse-Software diskutiert und bewertet. Als optimale Lösung hat sich herausgestellt, das Analyseprogramm Wireshark mit geeigneten Plugins zu erweitern. Die Plugins übernehmen, abhängig von dem Datenstrom, die individuelle Analyse. Um die Daten mit Wireshark untersuchen zu können, müssen die Rohdaten in das pcap-Format konvertiert werden. Hierfür wurde eine Konvertierungssoftware entwickelt, die diesen Vorgang durchführt.

Eine Erweiterungsmöglichkeit für dieses Analyse-System ist das Setzen von Trigger-Bedingungen. Durch einen Mechanismus, der bei einer erfüllten Bedingung automatisch einen Einlesevorgang startet, könnte dies realisiert werden.

Die gleichzeitige Analyse des Upstream- und Downstream-Datenstromes wäre eine weitere Erweiterungsmöglichkeit. Auf diese Weise könnte analysiert werden, wie Netzwerk-Knoten auf bestimmte Anfragen reagieren. Eine Möglichkeit dies zu realisieren, ist, ein Analyse-System an den Sende-Pfad und ein Analyse-System an den Empfangs-Pfad an zu koppeln und beide zeitlich miteinander zu synchronisieren. Nach dem Empfang der Daten beider Datenströme können durch die Zeitinformationen die beiden Datenströme zu einem Datenstrom geschachtelt werden, der anschließend analysiert werden kann.

Literaturverzeichnis

[Bernhard Lahres, Gregor Rayman 2009] BERNHARD LAHRES, GREGOR RAYMAN: *Objektorientierte Programmierung*. Galileo Computing, 2009. – ISBN 978-3-8362-1401-8

[Cypress Datasheet] CYPRESS DATASHEET: *EZ-USB FX2 Datenblatt*. – URL www.keil.com/dd/docs/datashts/cypress/cy7c68xxx_ds.pdf

[Cypress TRM] CYPRESS TRM: *EZ-USB FX2 Technical Reference Manual*. – URL www.keil.com/dd/docs/datashts/cypress/fx2_trm.pdf

[Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides 2004] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, JOHN VLISSIDES: *Entwurfsmuster*. Addision-Wesley, 2004. – ISBN 978-3-8273-2199-2

[Erich Stein 2008] ERICH STEIN: *Taschenbuch Rechnernetze und Internet*. Hanser, 2008. – ISBN 978-3-446-40976-7

[EXFO] EXFO: *Produktbroschüre des Ethernet-Testers AXS-200/850*. – URL http://www.opternus.de/uploads/media/AXS850_dbl1010d.pdf

[HORUS-NET] HORUS-NET: *Beschreibung von Ethernet Taps*. – URL http://www.horus-net.de/de/products/taps/index.html

[IEEE 802.3] IEEE 802.3: *Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications - SECTION ONE*. – URL http://standards.ieee.org/getieee802/download/802.3-2008_section1.pdf

[Jan Axelon 2007] JAN AXELON: *USB 2.0 - Handbuch für Entwickler*. Axelon, 2007. – ISBN 3-936427-45-3

[Johannes Erdfelt] JOHANNES ERDFELT: *libusb-win32 documentation*. http://www.sourceforge.net/. – URL http://sourceforge.net/apps/trac/libusb-win32/wiki/libusbwin32_documentation

[Jürgen Reichard 2009] JÜRGEN REICHARD: *Lehrbuch Digitaltechnik*. Oldenbourg, 2009. – ISBN 978-3-486-58908-5

[Jungo] JUNGO: *JUNGO WinDriver*. – URL http://www.jungo.com/st/windriver_usb_pci_driver_development_software.html

[Klaus Dembowski 2007] KLAUS DEMBOWSKI: *Lokale Netze*. Addision-Wesley, 2007. – ISBN 978-3-8273-2573-0

[Marvel] MARVEL: *Yukon FE+ 88E8040 Product Brief*. – URL http://www.marvell.com/products/pc_connectivity/yukon/Yukon_FE_88E8040.pdf

[synapse] SYNAPSE: *Online-Lexikon zu LAN/WAN-Analyse*. – URL http://www.syn-wiki.de/LAN-WAN-Analysis/htm/ger/_0/LAN-Analyse.htm

[Udo Erhardt 2001] UDO ERHARDT (Hrsg.): *USB 2.0*. Professional Series, 2001. – ISBN 3-7723-7965-6

[Wikipedia] WIKIPEDIA: *Zyklische Redundanzprüfung*. – URL http://de.wikipedia.org/wiki/Zyklische_Redundanzpr%C3%BCfung

[Wireshark] WIRESHARK: *HOWTO for Wireshark developers*. – URL http://anonsvn.wireshark.org/wireshark/trunk/doc/README.developer

Abkürzungsverzeichnis

API	Application-Interface
CIDS	Cabin Intercommunication Data System
CRC	Cyclic Redundancy Check
DEU	Decoder Encoder Units
DIR	Director
FAP	Flight Attendant Panel
FCS	Frame Check Sequence
FPGA	Field Programmable Gate Array
GUI	Graphical User Interface
IEEE	Institute of Electrical and Electronics Engineers
LLC	Logical Link Control
MAC	Media Access Control
OSI	Open Systems Interconnection
PAD	Padding-Bits
PCS	Physical Coding Sublayer
PMA	Physical Medium Attachement
PMD	Physical Medium Dependent
SFD	Start Frame Delimiter
STL	Standard Template Library
STP	Shielded Twisted Pair
TDMA	Time Division Multiple Access
USB	Universal Serial Bus
UTP	Unshielded Twisted Pair

A. Inhalt der CD

./thesis3.pdf Dieses Dokument in PDF-Format

./HybridLogger/ Quelltext des erstellten Programms (Einlese- und Konvertiertungs-Software)

./FX2Firmware/ Firmware für den Cypress FX2 Mikrocontroller

./ETH_USB/ Quelltext und kompletter synthesis report für das VHDL-Schaltwerk des Einlesemoduls

./WSDissectors/ Quelltexte der Wireshark-Dissectors

B. Auszug aus synthesis report

```
Design Summary:
Number of errors:      0
Number of warnings:    2
Logic Utilization:
  Number of Slice Flip Flops:          367 out of  11,776    3%
  Number of 4 input LUTs:              217 out of  11,776    1%
Logic Distribution:
  Number of occupied Slices:           257 out of   5,888    4%
    Number of Slices containing only related logic:     257 out of     257 100%
    Number of Slices containing unrelated logic:          0 out of     257   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:        309 out of  11,776    2%
    Number used as logic:              217
    Number used as a route-thru:        92

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

  Number of bonded IOBs:                38 out of     372   10%
    IOB Flip Flops:                     18
  Number of BUFGMUXs:                    2 out of      24    8%
  Number of RAMB16BWEs:                 16 out of      20   80%

Average Fanout of Non-Clock Nets:                3.65

Peak Memory Usage:  227 MB
Total REAL time to MAP completion:  5 secs
Total CPU time to MAP completion:   5 secs

Timing Summary:
---------------
Speed Grade: -4

   Minimum period: 5.480ns (Maximum Frequency: 182.482MHz)
   Minimum input arrival time before clock: 2.561ns
   Maximum output required time after clock: 6.991ns
   Maximum combinational path delay: 6.884ns
```

Listing B.1: Auszug aus dem synthesis report von Xilinx ISE

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 28. Februar 2011	
Ort, Datum	Unterschrift