Perfection in Automation
www.br-automation.com

X2X Scope

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (BSc)

im Rahmen des Studiums

Technische Informatik

eingereicht von

Martin Kaufleitner
Matrikelnummer 1027229

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao.Univ.Prof. Dr. Wolfgang Kastner
Mitwirkung: Mag.(FH) Ing. Thomas Enzinger

Wien, 31.01.2014

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Martin Kaufleitner
Obere Amtshaussegasse 20-24/Top3, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 31.01.2014 _____

(Unterschrift Verfasser/in)

Abstract

This bachelor thesis describes the realization of a debugger from Bernecker & Rainer Industrie-Elektronik GmbH (B&R) for a specific field bus system. The analyzer tool samples the signal on the bus and transmits it to a PC using Ethernet. Because of the very high data rate of the original bus signal a compress algorithm for transmitting the data had to be designed. Different existing algorithms were compared and finally combined to a new adapted algorithm which can be easily implemented in hardware and has optimal compress characteristics concerning the sample data. Different compress algorithms were tested and the resulsts are presented in this thesis. After the compression the firmware transmits the data to the PC using UDP.

Danksagung

Ich möchte mich zunächst an dieser Stelle bei allen Personen bedanken, die mich bei der Umsetzung dieser Bachelorarbeit unterstützt und mir in Problemsituationen weitergeholfen haben.
In erster Linie möchte ich meinen Projektbetreuer seitens der Technischen Universität Wien Herrn Dr. Wolfgang Kastner erwähnen, welcher mich bei der Fassung der schriftlichen Arbeit immer wieder korrigiert und beraten hat. Ohne sein Korrekturen wäre diese Arbeit nicht in dieser Form zustande gekommen. Außerdem waren seine Vorschläge ausschlaggebend für die Umsetzung der weiterführenden Projektarbeit, welche den Empfang der Daten seitens PC betrifft. Vielen Dank dafür.
Ein ganz besonderer Dank gilt weiteres der Firma Bernecker&Rainer, welche die Realisierung dieses Projektes erst möglich gemacht hat. Im besonderen möchte ich mich hierbei bei meinem Projektbetreuer seitens der Firma Mag.(FH) Ing. Thomas Enzinger bedanken. Bei Problemen halfen mir seine Erfahrung und fachkundigen Ratschläge immer wieder optimale Lösungswege zu finden.
Zu guter Letzt möchte ich mich bei MSc Johannes Pointner für die Umsetzung meines Kompressionsalgorithmus in der Abtasthardware und die hervorragende Zusammenarbeit bedanken. Er hat mir bei der Realisierung der Hardware sehr geholfen und durch seine Vorschläge auch zur Verbesserung der Firmware beigetragen.

Inhaltsverzeichnis

Inhaltsverzeichnis **1**

Abbildungsverzeichnis **4**

Tabellenverzeichnis **5**

I Bachelorarbeit **7**

1 Einleitung **9**
1.1 Motivation . . . 9
1.1.1 X2X Scope . . . 10
1.2 Related Work - X2X Analyzer . . . 10
1.2.1 Funktion . . . 10
1.2.2 Einschränkung . . . 10
1.3 Gliederung . . . 11
1.4 Abkürzungen . . . 12

2 Abtasthardware **13**
2.1 X2X-Link . . . 13
2.1.1 Topologie . . . 13
2.1.2 Frameaufbau . . . 14
2.1.3 Fehlererkennung . . . 16
2.2 Aufbau . . . 16
2.3 Hardware . . . 17
2.3.1 FPGA . . . 17
2.3.2 VHDL . . . 17
2.4 Firmware . . . 20
2.4.1 PX-32C Prozessor . . . 20
2.4.2 Anforderung . . . 22
2.4.3 Realisierung . . . 22
2.5 UDP Kommunikation . . . 24
2.5.1 Paketaufbau . . . 24
2.5.2 Datenübertragung . . . 25

3 Datenkompression **29**
3.1 Allgemeines Prinzip . . . 29
3.1.1 Verlustfreie Datenkompression . . . 29
3.1.2 Verlustbehaftete Datenkompression . . . 30
3.2 Ausgangssituation . . . 30
3.3 Analyse der Daten . . . 30
3.3.1 Logic Analyzer . . . 31
3.3.2 Datenaufzeichnung . . . 31
3.3.3 Datenauswertung . . . 32
3.4 Kompressionsalgorithmen . . . 32
3.4.1 Stringersatzverfahren . . . 33
3.4.2 Entropiekodierung . . . 33
3.4.3 Run Length Encoding (RLE) . . . 33
3.4.4 Compress I . . . 37
3.4.5 Huffman Code . . . 39
3.4.6 Compress II . . . 43

4 Ergebnisse und Ausblick **49**
4.1 Ergebnisse . . . 49
4.1.1 Beschaffenheit der Daten . . . 49
4.1.2 Netzwerkauslastung . . . 50
4.2 Ausblick . . . 51
4.3 Resumé . . . 51

II Projektarbeit **53**

5 Einleitung zur Projektarbeit **55**
5.1 Ausgangssituation . . . 55
5.2 Toolchain . . . 56
5.3 Realisierung . . . 56
5.4 Ergebnis . . . 57

6 Dekomprimieren der Daten **59**
6.1 Einlesen der Daten . . . 59
6.1.1 Beschaffenheit der Daten . . . 59
6.1.2 Online Auswertung . . . 60
6.1.3 Offline Auswertung . . . 60
6.1.4 Thread Konzept . . . 61
6.1.5 Wichtige Programmpassagen . . . 63
6.2 Datendekompression . . . 65
6.2.1 Algorithmus . . . 65
6.2.2 Wichtige Programmpassagen . . . 65

7 Dekodierung der Daten **69**
7.1 X2X Protokoll 69
7.1.1 Datenkodierung 69
7.1.2 Cyclic Redundancy Check CRC 70
7.2 Wichtige Datenstrukturen 72
7.2.1 CDecoder Klasse 72
7.2.2 CX2XFrame Klasse 73

8 Darstellung der Daten **75**
8.1 Omni Peek Plugin 75
8.1.1 PKT File Format 75
8.2 X2X Scope - Oszilloskop 77
8.2.1 Dekodierung 77
8.2.2 Oszilloskop 77
8.2.3 Statistische Auswertung 78
8.3 Zusammenfassung 78
8.3.1 Fehlererkennung auf X2X Protokollebene 78
8.3.2 Fehlererkennung auf X2X Datenebene 79
8.3.3 Oszilloskopfunktion 79

Literaturverzeichnis **81**

Abbildungsverzeichnis

2.1 X2X-Link Topologie [OW02] 14
2.2 Hardware Aufbau des X2X Scope 16
2.3 Y-Modell nach Gajski-Walker [RAW85] 19
2.4 Ethernet Paket nach IEEE 802.3 [Ins08] 27

3.1 Kompressionsergebnisse RLE 36
3.2 Unkomprimierte und komprimierte Sequenz in Compress I 37
3.3 Kompressionsergebnisse Compress I 39
3.4 Huffman Baum zum Text *AAAABBBCCD* 40
3.5 Kompressionsergebnisse Huffman Byte 42
3.6 Kompressionsergebnisse Huffman Stream 42
3.7 Unterschiedliche Sequenzen in Compress II 44
3.8 Beispiel zu Compress II 45
3.9 Vergleich der einzelnen Algorithmen 48

5.1 Erweiterte Toolchain des X2X Scope zur X2X Analyzer Toolchain 56

6.1 Sequentielle Abarbeitung mehrerer Prozesse [Wol09] 61
6.2 Aufbau eines Prozesses [Wol09] 62
6.3 Aufbau eines Multithreading Systems [Wol09] 62
6.4 Klassendiagramm zur X2X Scope Software 63

8.1 Grafische Benutzeroberfläche der X2X Scope Software 77

Tabellenverzeichnis

2.1 LUT eines 2 Bit UND Gatter 18
2.2 Memory Interface HW - SW 23
2.3 X2X Scope Header Aufbau 26

3.1 Anzahl und Länge der 16 Häufigsten `1` und `0` Streams 32
3.2 Parameterwerte 45

4.1 Werte der Kompressionsparamter und entsprechende Kompressionsrate . . . 50

8.1 PKT File Aufbau 76
8.2 X2X Analyzer Frameaufbaus 76

Teil I

Bachelorarbeit

KAPITEL 1

Einleitung

"Der schlimmste aller Fehler ist, sich keines solchen bewusst zu sein."

Thomas Carlyle

Dieses Kapitel liefert einen Einstieg in die Bachelorarbeit. Zuerst wird die Motivation und das bereits bestehende Analyse Tool erklärt, danach erfolgt eine Gliederung des Aufbaus dieser Arbeit.

1.1 Motivation

X2X-Link [OW02] bezeichnet einen firmeninternen Feldbus von Bernecker & Rainer Industrieelektronik GmbH (B&R), der vor allem zur Kommunikation zwischen Speicherprogrammierbaren Steuerungen (SPS) und diversen I/O-Systemen verwendet wird. Die Daten werden dabei in sogenannten Frames mit einer Übertragungsrate von $12MBit/s$ übertragen. Die einzelnen I/O Stationen tasten das Signal 4-fach ab, werten es aus und senden es aufbereitet an die nächste Station weiter. Ein X2X-Link Netzwerk besteht also aus vielen einzelnen Peer-To-Peer Verbindungen. Da bei Neuentwicklungen oder beim Einsatz in rauen Umgebungen verschiedenste Fehler auftreten können, ist es wichtig, auf entsprechende Tools zurückgreifen zu können, um diese Fehler zu finden. Dabei wäre es aber nicht nur interessant den Fehler zu lokalisieren, sondern auch die Ursachen für dessen Auftreten ermitteln zu können. Dies gestaltet sich meist als sehr schwierig, da oft nur geringe Signalstörungen sporadisch auftretende Datenfehler verursachen können, die sehr schwer zu reproduzieren sind. Eine Möglichkeit wäre ein Oszilloskop in das Busnetzwerk zu integrieren, das den Datenstrom auf Signalebene aufzeichnet. Dies ist allerdings äußerst unpraktisch und die anfallenden Datenmengen sind enorm. Daher soll

ein Monitoring-Tool entwickelt werden, welches das Datensignal digitalisiert und zur weiteren Auswertung an den PC sendet.

1.1.1 X2X Scope

Das X2X Scope ist eine Weiterentwicklung des bereits bestehenden X2X Analyzer Tools zur X2X-Link Fehlerdiagnose. Im Gegensatz zum Vorgänger wird dabei das Signal achtfach abgetastet und anschließend ohne weitere Auswertung an den PC gesendet. Dort kann mithilfe dieser Samples das ursprüngliche Signal wiederhergestellt und ausgewertet werden. Wegen der enormen Datenmenge von $96 MBit/s$ müssen die Samples vor dem Senden komprimiert und beim PC wieder dekomprimiert werden. Empfangen werden die Frames wie auch beim X2X Analyzer mithilfe des Network Analyzer Tools, deren Dekompression und Darstellung erfolgt anschließend mit der X2X Scope Software. Dabei kann ein File erzeugt werden, das genauso formatiert ist, wie die Files des X2X Analyzers und somit auch von bereits bestehenden Tools gelesen werden kann. Diese Bachelorarbeit beschäftigt sich mit dem Komprimieren der Daten, sowie der Firmware, welche die Daten aus dem FPGA liest und via UDP an den PC sendet.

1.2 Related Work - X2X Analyzer

Im Jahr 2006 entwickelte Gerald Pichler im Rahmen seiner Diplomarbeit an der Fachhochschule Salzburg ein Analyzer Tool zur Fehlerdiagnose für den X2X-Link [Pic06]. Dieses Tool wird derzeit zu Fehlersuche innerhalb der Firma verwendet.

1.2.1 Funktion

Der X2X Analyzer ist im Grunde ein Busmodul, das für die anderen Slaves, sowie den Master nicht sichtbar ist und die empfangenen Daten unverändert weiterleitet. Das Signal wird dabei 4-fach abgetastet und an die Firmware übergeben. Bei der Abtastung werden die X2X-Link Fehler CRC Error und Sync Error erkannt und gezählt. Diese Daten werden daraufhin in UDP Pakete gepackt und über Ethernet an den PC gesendet. Dabei ist zu beachten, dass immer nur einzelne X2X-Frames übertragen werden. Aufgrund der maximalen X2X Framelänge von $8192 Bytes + CRC32$ kann es allerdings nötig sein, einen X2X Frame auf mehrere Ethernet Frames aufzuteilen. Auf dem PC können diese Frames mithilfe von selbst erstellten Netzwerk Sniffer Plug-ins analysiert und ausgewertet werden.

1.2.2 Einschränkung

Da am PC nur noch Daten der einzelnen X2X Frames ankommen, kann nur noch auf Fehler bezüglich der Daten geschlossen werden. Jegliche Information über das Timing, sowie das Bussignal zwischen den Frames steht zu diesem Zeitpunkt nicht mehr zur Verfügung. Häufig sind allerdings genau solche Timing- und Signalverzerrungen Ursache von Fehlern, deren eigentlicher Ursprung mit dem momentanen Tool nicht ermittelt

werden kann. Aus dieser Erkenntnis ergab sich die Idee einer Weiterentwicklung des Tools, die im Zuge dieser Bachelorarbeit entwickelt und zum Teil auch umgesetzt wurde.

1.3 Gliederung

Teil I Bachelorarbeit

Kapitel 1 liefert einen Einstieg in die Bachelorarbeit. Es wird ein bereits vorhandenes Tool erklärt und darauf aufbauend die Zielvorgabe für das Projekt formuliert.

In Kapitel 2 wird die Hardware zur Abtastung des Datensignals erläutert. Nach einer grundlegenden Einführung wird speziell auf die verwendete Hardware eingegangen. Außerdem wird die UDP Kommunikation zwischen Firmware und PC genauer beschrieben. Hier werden auch die grundlegenden Berechnungen für Datenmenge und die sich daraus ergebende notwendige Kompressionsrate durchgeführt.

Kapitel 3 zeigt auf, welche Möglichkeiten es zur Kompression von Daten gibt und welche sich für dieses Projekt als eine zielführende erwiesen hat. Dabei werden verschiedene Kompressionsalgorithmen genauer erläutert und verglichen.

Kapitel 4 gibt ein Resümee über die Ergebnisse des Projekts, sowie einen Ausblick für mögliche Erweiterungen. Dies betrifft vor allem den Empfang der Daten am PC und deren Auswertung, die im Zuge einer eigenen Projektarbeit durchgeführt wurde.

Teil II Projektpraktikum

Kapitel 5 gibt wiederum eine Einleitung in das, auf die Bachelorarbeit aufbauende, Projektpraktikum. Dabei wird zu Beginn auf die Zielvorgaben eingegangen.

Kapitel 6 befasst sich mit dem Lesen und Dekomprimieren der Daten auf dem PC. Es gibt einen Einblick in die Analyse der empfangenen Daten sowie die in die praktische Umsetzung der in der Bachelorarbeit entwickelten Datendekompression

Kapitel 7 behandelt die Dekodierung der Daten. Hierbei wird auf die Gewinnung der ursprünglichen X2X aus den Sampledaten eingegangen. Dazu ist es nötig, einen tieferen Einblick in das X2X Protokoll zu bereiten.

Kapitel 8 gibt Aufschluss über die verschiedenen Möglichkeiten der Fehleranalyse mithilfe des Debug Tools und erläutert die verschiedenen Features der Oszilloskop Funktionalität.

1.4 Abkürzungen

SPS Speicherprogrammierbare Steuerung

IO Input/Output

FGPA Field Programable Gate Array

UDP User Datagram Protocol

CRC32 Cyclic Redundancy Check 32 Bit

IP67 Internet Protection 67

TCP Transport Control Protocol

IP Internet Protocol

RLE Run Length Encoding

ASCII American Standard Code for Information Interchange

MBit Megabit = 1.000.000 Bit

CPU Central processing unit

IEEE Institute of Electrical and Electronics Engineers

SPARC Scalable Processor Architecture

MIPS Microprocessor without Interlocked Pipeline Stages

LCC Local C Compiler

DMA Direct Memory Access

VHDL Very High Speed Integrated Circuit Hardware Description Language

LUT Lookup-Table

LSB Least Significant Bit

IC Integrated Circuit

(S)RAM (Static) Random-Access Memory

KAPITEL 2

Abtasthardware

"Wer als Werkzeug nur einen Hammer hat, sieht in jedem Problem einen Nagel."

Paul Watzlawick

Dieses Kapitel beschreibt die notwendige Hardware zum Abtasten des X2X-Link Datensignals. Dazu wird auf die X2X-Bus Topologie sowie dessen Frameaufbau genauer eingegangen. Darüber hinaus werden die Firmware zum Senden, sowie die Analyse der UDP Kommunikation näher beschrieben.

2.1 X2X-Link

X2X-Link [OW02] ist ein firmeninternes Feldbussystem, das vor allem für den IO Datenverkehr eingesetzt wird. Es handelt sich dabei um ein Single Master - Slave System, in welchem der Master einen Frame sendet und alle Stationen ihre Daten entsprechend in den Frame einfügen. Die maximale Anzahl an Knoten ist dabei auf 250 begrenzt.

2.1.1 Topologie

Es gibt vier verschiedene Möglichkeiten, IO Stationen über X2X mit der Masterstation (SPS) zu verbinden (siehe Abb. 2.1):

1. IO Stationen können direkt über eine sogenannte Backplane an die SPS gesteckt werden (*central IOs*).

2. Über eine X2X Interface Karte, die in einen der SPS Subslots gesteckt wird, können entfernte IO Stationen mit der SPS verbunden werden (*decentral IOs*).

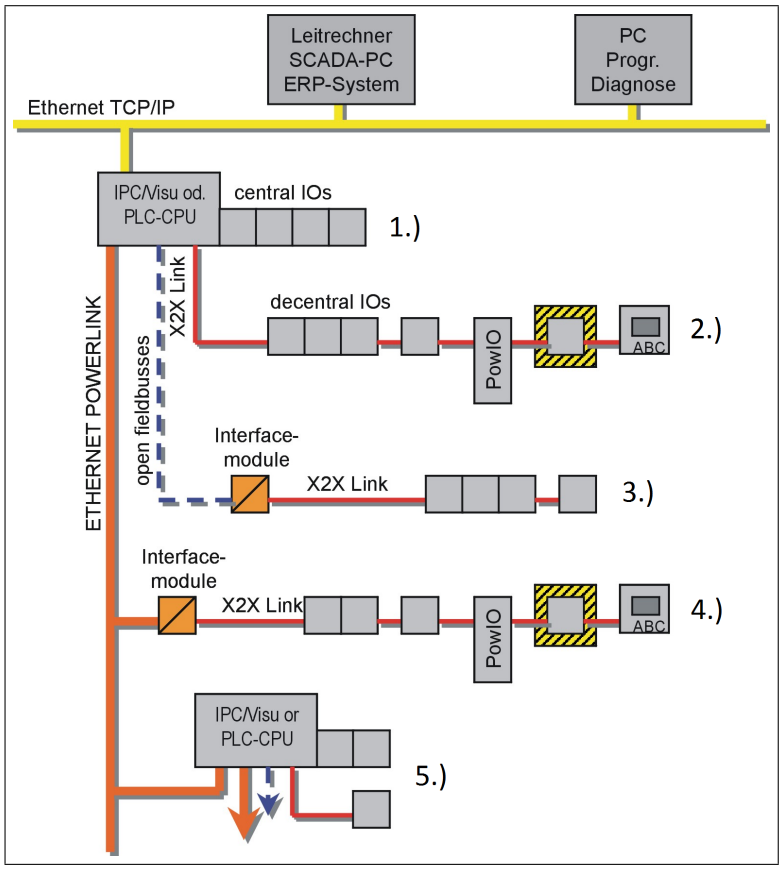


Abbildung 2.1: X2X-Link Topologie [OW02]

3. Über ein Feldbussystem Interface Modul können IO Stationen zum Beispiel auch über Feldbusse, wie DeviceNet, CANopen, Profibus, Profinet, Ethernet/IP, Modbus oder ASi, mit der SPS verbunden werden.

4. Weiteres gibt es die Möglichkeit IO Stationen über die in der SPS integrierte Ethernet Powerlink Schnittstelle und einem entsprechenden Interface Modul mit der Masterstation zu verbinden.

5. Außerdem kann eine CPU oder eine andere Ethernet Slave Station in ein Powerlink Netzwerk eingefügt werden. Hierauf kann entsprechend der verwendeten Hardware das Netzwerk wieder mit einer der bereits genannten Optionen erweitert werden.

2.1.2 Frameaufbau

Das X2X-Link Bussystem ist ein sogenanntes Summenframeverfahren, das bedeutet, dass der Master einen Frame an alle Slaves schickt und diese ihre Daten an der entsprechenden Stelle im Summenframe einfügen. Anschließend wird der Frame wieder an

den Master zurück gesendet, welcher die Daten der einzelnen Stationen auslesen kann. Grundsätzlich werden die Daten innerhalb des X2X-Frames in 8 Bit Blöcken geschrieben, wobei das LSB zuerst übertragen wird. Am Ende eines 8 Bit Blocks wird ein sogenanntes Syncbit angehängt. Dieses ist invertiert zum darauffolgenden Datenbit. Dieses Bit wird zur Synchronisierung verwendet, da aufgrund der Geschwindigkeitsunterschiede der Quarze der einzelnen Module sonst eine zu große Verschiebung entstehen würde.

1. **Start Delimiter**
   Der Start eines neuen X2X Frames wird durch den Start Delimiter angezeigt. Dabei handelt es sich um eine Folge von zehn *0* Bits. Diese Folge kann aufgrund der Syncbits so im Frame nicht vorkommen und kann daher eindeutig erkannt werden.

2. **Frametype**
   Der Frametype besteht aus 14 Bit und gibt die Länge des X2X Frames an. Das erste Bit gibt den Frametype (ASYNC beziehungsweise. SYNC) an. Da die verschiedenen Frametypes in dieser Bachelorarbeit nicht relevant sind, wird darauf nicht näher eingegangen. Eine diesbezügliche Auswertung findet erst während der Datendekodierung in der Projektarbeit statt. Details zum genauen Frameaufbau können in der hausinternen X2X Dokumentation [OW02] nachgelesen werden.

3. **Daten**
   Als nächstes folgen die Datenbytes. Digitale IO Stationen senden ein bis zwei Datenbytes, analoge Module bis zu 32 Bytes. Dies ergibt eine maximale Datenlänge von 250 Stationen * 32 Byte = 8000 Bytes.

4. **CRC**
   Um Fehler zu erkennen und in bestimmten Fällen sogar korrigieren zu können, wird über den X2X Frame, also Frametype und Daten, eine CRC32 Checksumme gebildet. Als Generatorpolynom wird hierzu das IEEE802.3 Ethernet CRC32 Polynom verwendet. Bei der Bildung der Checksumme werden die Syncbits nicht beachtet, Einfluss haben nur die Datenbits von Frametype und Daten. Eine kurze Erklärung zur Berechnung der CRC Checksumme findet sich später in dieser Arbeit (siehe Abschnitt 7.1.2). Die genaue Definition dieses Standards kann in weiterführender Literatur nachgelesen werden. [Ins08]

5. **End Delimiter**
   Nach den 32 Bit der Checksumme folgen wiederum zehn *0* Bits, welche das Ende des Frames kennzeichnen.

6. **Idle**
   Zwischen Start Delimiter und End Delimiter wird ein Idle Signal übertragen. Dabei handelt es sich um eine Folge von abwechselnden *0* und *1* Bits.

2.1.3 Fehlererkennung

Das X2X Protokoll verfügt über folgende Möglichkeiten zur Erkennung von Fehlern.

- Die Länge von Start Delimiter muss zehn *0* Bits betragen.
- Die Länge des End Delimiter muss zehn *0* Bits betragen, gefolgt vom Idle Signal.
- Jedes Syncbit muss invertiert zum folgenden Datenbit sein.
- Die CRC32 Checksumme muss korrekt sein.
- Die Anzahl der Samples für die Darstellung der einzelnen Bits ist ebenfalls vorgegeben.

2.2 Aufbau

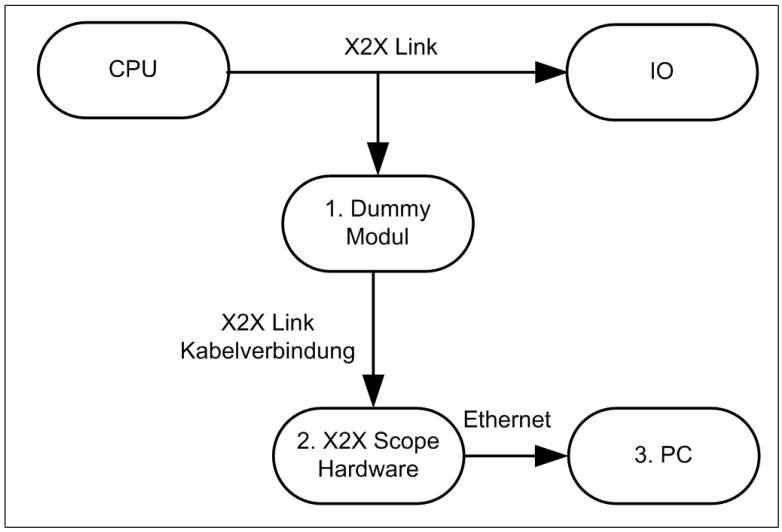


Abbildung 2.2: Hardware Aufbau des X2X Scope

1. Um die Daten vom Bus aufzuzeichnen, wird ein Dummy Modul in das Busnetzwerk gehängt. Dieses Modul leitet das Signal, das gerade am Bus übertragen wird, unverändert an die Abtasthardware weiter. Zu diesem Zeitpunkt ist das Signal noch analog vorhanden. Das Dummymodul ist dabei für die anderen Stationen im X2X Busnetzwerk nicht sichtbar und operiert transparent als Repeater.

2. Das Scope Modul entspricht ursprünglich einem X67 Powerlink Modul, auf welchem die Abtasthardware und die Firmware implementiert sind. Es tastet das vom Dummymodul empfangene Signal mit einer Abtastrate von 96 Megasamples/s ab und komprimiert die so entstandenen Daten.

3. Die komprimierten Daten werden anschließend von der Firmware ausgelesen, in UDP Pakete gepackt, über Ethernet an den PC gesendet und dort ausgewertet. Am PC werden die Frames von einem bereits bestehenden Network Analyzer empfangen und in einem File Ringbuffer gespeichert. Die Auswertung erfolgt mithilfe eines C# Programmes, das in drei parallele Threads aufgeteilt ist. Der erste Thread liest die UDP Frames vom Filesystem und übergibt diese an einen Ringbuffer. Der zweite Thread liest jeweils einen Frame aus dem Buffer, dekomprimiert diesen und schreibt den dekomprimierten Datenstrom in einen zweiten Ringbuffer. Der dritte Thread liest den dekomprimierten Datenstrom, wertet ihn auf etwaige Fehler der ursprünglichen Daten aus und schreibt die Originaldaten in ein File. Genauere Informationen zum Auswerten der Daten können aus Teil II (Projektarbeit) entnommen werden.

2.3 Hardware

Zum Abtasten des Signals wird ein gewöhnlicher X67 Buskoppler verwendet. Dabei handelt es sich um ein X2X Busmodul, das eine Schnittstelle zwischen Ethernet und X2X-Link darstellt. Das Gehäuse ist dabei gemäß IP67 Standard ausgeführt. Das FPGA des Moduls wurde entsprechend der Scope Funktion abgeändert, das bedeutet, es wird das Signal 8-fach abgetastet, komprimiert und im RAM Speicher abgelegt. Für alle anderen Stationen im Bus Netzwerk ist dieses Modul nicht sichtbar und alle Vorgänge verändern in keinerlei Weise die auf dem Bus befindlichen Daten.

2.3.1 FPGA

Bei einem Field Programable Gate Array oder kurz FPGA handelt es sich um einen sogenannten integrierten Schaltkreis, kurz IC. Ein FPGA stellt dem Programmierer eine Vielzahl von programmierbaren Logikelementen zur Verfügung, die abhängig von der Beschaltung und Verknüpfung (dem sogenannten Routing) verschiedene logische Funktionen annehmen können. Bei diesen Logikelementen handelt es sich um sogenannte Look Up Tables (LUT). Eine LUT liefert zu einem vorgegebenen Eingangsmuster den dazu gespeicherten Ausgang. Tabelle 2.1 ist ein Beispiel für ein 2 Bit UND-Gatter. FPGAs werden programmiert, indem man das Programm in einer Hardwarebeschreibungssprache, wie zum Beispiel VHDL, erstellt und dann synthetisiert. Das bedeutet, dass die einzelnen Programm-Anweisungen in Hardware realisiert werden, so wird eine einfache IF Anweisung beispielsweise in einen Multiplexer synthetisiert. Aus diesem Synthese File, der sogenannten Netlist, lässt sich anschließend ein Binary File erstellen, das mithilfe eines Programmers in das FPGA geladen werden kann.

2.3.2 VHDL

VHDL ist eine Abkürzung für Very High Speed Integrated Circuit Hardware Description Language und bezeichnet eine Hardwarebeschreibungssprache. Ursprünglich wurde

Eingangsmuster	Ausgang
00	0
01	0
10	0
11	1

Tabelle 2.1: LUT eines 2 Bit UND Gatter

VHDL nur zu Dokumentations- und Simulationszwecken für elektronische Schaltungen, nicht aber für deren Entwurf entwickelt. Erst mit den Weiterentwicklungen VHDL-93 und VHDL-2001 wurde diese Hardwarebeschreibungssprache immer mehr auch zur Entwicklung eingesetzt.

Modellierung

> *If we are to encompass this range of views of digital systems, we must recognize the complexity with which we are dealing. It is not humanly possible to comprehend such complex systems in their entirely. We need to find methods of dealing with the complexity, so that we can, with some degree of confidence, design components and systems that meet their requirements.* [Ash98]

Ein grundlegendes Prinzip der Modellierung in VHDL ist das sogenannte Y-Modell, das drei verschiedene Sichtweisen auf den Entwurf einer Hardware bietet.

- **Verhalten**
  - Beschreibung der Funktionalität der Komponente
  - Gleichungen, Algorithmen
- **Struktur**
  - Beschreibung des Aufbaus der Komponente
  - Verbindungen zwischen einzelnen Gattern
- **Geometrie**
  - Konkretes Aussehen der Komponente in Hardware
  - Masken, Flurplan

Diese drei Sichtweisen ermöglichen die Beschreibung verschiedenster Komponenten. Dabei können wiederum fünf Abstraktionsebenen voneinander abgegrenzt werden, welche sich in jeder Ebene in den oben genannten Sichtweisen (Verhalten, Struktur und Geometrie) unterscheiden. Es ist wichtig zu verstehen, dass alle drei Sichtweisen auf allen Abstraktionsebenen immer ein und dasselbe System beschreiben. Das Y-Modell dient lediglich dazu, einen passenden Einstieg in den Entwurf der Hardware, den sogenannten Design Entry, zu finden.

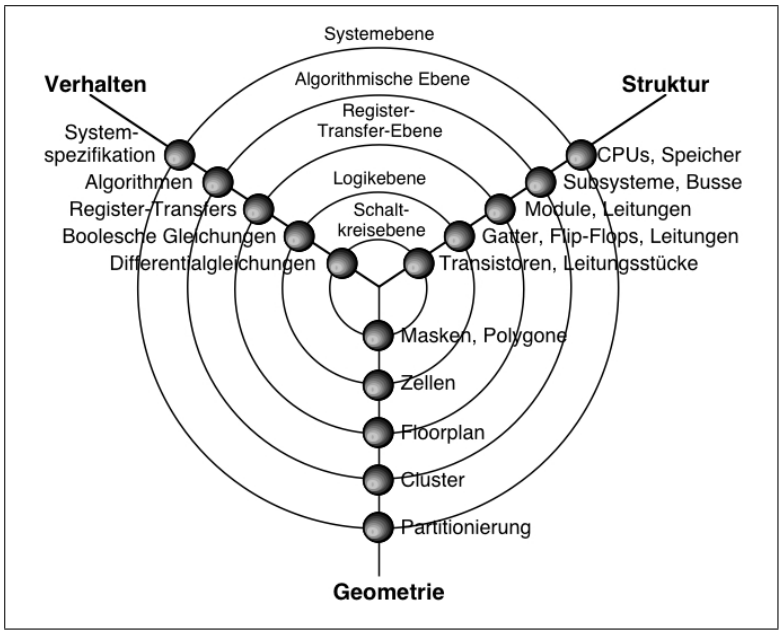


Abbildung 2.3: Y-Modell nach Gajski-Walker [RAW85]

- **Systemebene**
  - Verhalten: funktionale Randbedingungen, Systemspezifikation
  - Struktur: Blockschaltbild (Speicher, CPU, IO,...)
  - Geometrie: Aufbau des Gehäuses, Pinbelegung
- **Algorithmus Ebene**
  - Verhalten: Formulierung des Programmablaufes in einer Hochsprache, wie zum Beispiel C
  - Struktur: Bussysteme, Task-Management, Blockschaltbild
  - Geometrie: Layout des Chips (Placement)
- **Register Transfer Ebene (RTL)**
  - Verhalten: Beschreibung des Verhaltens mithilfe einer HDL, wie zum Beispiel VHDL
  - Struktur: Schaltplan (ALU, RAM, Register,...)
  - Geometrie: Layout der einzelnen Funktionsblöcke (Floorplanning), Routing
- **Logikebene**
  - Verhalten: Beschreibung der Logik durch boolesche Gleichungen
  - Struktur: Schaltplan auf Gatterebene (AND, OR, XOR,...)
  - Geometrie: detailliertes Chiplayout

- **Schaltkreisebene**

  - Verhalten: Beschreibung des Verhaltens mithilfe von Differenzialgleichungen
  - Struktur: Schaltplan auf Transistorebene (Netlist)
  - Geometrische: Masken zur Fertigung des Chips

Design Unit

Hat man sich für eine geeignete Sichtweise auf einer passenden Abstraktionsebene, dem Design Entry, entschieden, kann mit der Hardwarebeschreibung der einzelnen Komponenten (Design Unit) begonnen werden. Grundlegend besteht eine Komponente, welche in VHDL beschrieben wird, aus 3 Teilen:

- **Entity**: Diese bildet eine Art Schnittstelle zwischen den Komponenten. Alle Ein- und Ausgangssignale, die sogenannten Ports, werden hier samt Richtung und Datentyp deklariert. Auch Parameter (Generics) können deklariert werden, in der Entity wird allerdings keinerlei Funktionalität implementiert.
- **Architecture**: In der Architecture wird die Funktionalität der Komponente beschrieben. Dabei kann wiederum zwischen struktureller (Instanzierung anderer Komponenten) und verhaltensorientierter (Prozesse) Beschreibung unterschieden werden.
- **Configuration**: Da zu einer Entity mehrere Architectures zählen können, muss definiert werden, welche Architecture in welcher Instanz einer Entity aktiv ist. Diese Zuweisung erfolgt in der Configuration.

Diese drei Teile einer Komponente können entweder in verschiedenen Files oder auch gemeinsam in einem großen File implementiert werden. Auch die Verwendung von Packages und Libraries ist möglich.

2.4 Firmware

Die Firmware des Buscontrollers wurde in ANSI C mithilfe einer Microsoft Visual Studio Umgebung entwickelt. Sie hat die Aufgabe, die von der Abtasthardware bereitgestellten Daten per UDP an den PC zu senden. Die Firmware läuft dabei auf einem firmeninternen FPGA Prozessor, dem PX-32.

2.4.1 PX-32C Prozessor

Die erste Version des PX-32 Prozessors, genannt PX-16 entstand aus dem xr16 Instruction Set Specifications von Jan Gray.

> *xr16 is a simple 16-bit reduced instruction set computer designed to run integer-only C programs. The xr16 design is optimized for an area-efficient pipelined implementation in a field-programmable gate array and other gate- and interconnect-constrained environments.* [Gra99]

Dabei handelt es sich um ein Instruction Set mit einem 16 Bit Programmzähler und einem Register File mit 16 16-Bit Integer Registern. Vier der 16 Register haben eine spezielle Funktionalität:

- **r0** ist fix auf 0 gemapped und liefert bei einem Lesezugriff immer 0.
- **r13** wird *sp* genannt und dient der Adressierung bei Speicherzugriffen.
  Bsp: *lw r1,8(sp)*
- **r14** ist reserviert für Interrupt Rücksprungadressen und sollte weder gelesen noch beschrieben werden.
- **r15** steht eigentlich zur freien Verfügung, allerdings werden auf dieses Register die *call* Zieladressen gespeichert.

Der Programmspeicher besteht aus einem Array von 2^{16} Bytes. Weiteres sind unter xr16 folgende Datenformate spezifiziert:

- **unsigned char** Byte 8 Bit
- **unsigned word** Word 16 Bit
- **unsigned long** Long 32 Bit

Aus dem PX-16 Prozessor entwickelte sich zuerst der PX-32, bei welchem die Adressbreite auf 32 Bit erweitert wurde und in weiterer Folge der PX-32C. Dabei handelt es sich um einen 32-Bit Prozessor, der im Fall des X2X Scopes in einem Cyclone IV FPGA realisiert wurde. Im Gegensatz zum PX-32 wurden an der Weiterentwicklung unter anderem Änderungen bezüglich der Input Daten Latches, sowie der DMA Signale vorgenommen. Die aktuelle Version kann derzeit nur im Big Endian Mode (Motorola) betrieben werden. Die Grundlage für den C Compiler bildete lcc [CWF95], ein Standard C Compiler für ALPHA, SPARC, MIPS R3000 und Intel x86.

Debugger

Da der PX Prozessor firmenintern entwickelt wurde, konnte dieser weitgehend optimiert und an individuelle Bedürfnisse angepasst werden. Deswegen wurde ein sehr mächtiges Debugger Tool zur Unterstützung bei der Firmwareentwicklung entworfen. Mit dessen Hilfe können sehr einfach Breakpoints gesetzt, Variablenwerte zur Laufzeit ausgelesen und geändert oder auch Speicherstellen beschrieben und ausgelesen werden. Weiteres bietet der Debugger die Möglichkeit, nicht nur den geschriebenen C Code Schritt für Schritt auszuführen, sondern durch einfaches Umschalten auch den daraus resultierenden Assembler Code. Wegen dieser mächtigen Funktionalitäten erwies sich dieses Debugger Tool bei der Entwicklung der Firmware als sehr hilfreich.

2.4.2 Anforderung

Um die gewünschte Oszilloskop Funktionalität am PC nachstellen zu können, ist es notwendig, das X2X-Link Datensignal so hochauflösend wie möglich abzutasten. Man einigte sich hierfür auf 8-faches Oversampling, was einer Abtastrate von 96 MSamples/s entspricht. Das bedeutet, dass jedes übertragene Bit in acht einzelne Samples zerlegt wird. Dadurch können zu lange oder zu kurze High- beziehungsweise Low-Signale erkannt werden. Aufgrund dieser hohen Geschwindigkeit der Hardware entstehen sehr große Datenmengen, die von der Firmware verarbeitet und gesendet werden müssen. Wichtig ist hierbei, dass keine Zeit durch unnötiges Kopieren großer Speicherbereiche verschwendet wird. Es ist von essentieller Bedeutung, dass die Firmware den strengen Timing Vorgaben der Abtasthardware genügt, da mangels Speicher ansonsten Daten verloren gehen würden.

Interface

Firmware und Hardware wurden von verschiedenen Entwicklern implementiert, daher war es notwendig eine Schnittstelle festzulegen, um einen reibungslosen Datenaustausch zu gewährleisten. Für diesen Zweck gibt es zwei wichtige gemeinsame Speicherbereiche, auf welchen der Zugriff kontrolliert erfolgen muss.

1. Kompressionsparameter und Kontrollregister befinden sich ab der Adresse 0x1000 und sind wie in Tabelle 2.2 gespeichert.

2. Der Speicherbereich, der als Datenringbuffer der Hardware verwendet wird, wird von der Firmware alloziert und im Register SRAM (siehe Tabelle 2.2) als Adresse übergeben.

2.4.3 Realisierung

Um den hohen Timing Anforderungen zu entsprechen, wurde ein System mit Ringbuffern entworfen, wobei Schreib- beziehungsweise Lesezugriffe mithilfe von Interrupts gesteuert werden.

Hardware Ringbuffer

Seitens der Abtasthardware werden die Samples abgetastet und komprimiert, darauf werden sie an eine von der Firmware festgelegte Speicherstelle geschrieben. Die ersten 66 Byte werden dabei für den späteren Header freigelassen. Sobald $1518 - 66 = 1452$ Byte an komprimierten Daten geschrieben wurden, entspricht diese Datenmenge genau der maximalen Framelänge eines Ethernet Frames. Danach wird ein X2X Interrupt ausgelöst und das FPGA Programm schreibt die nächsten Daten an die Startadresse mit einem Offset von 1518 Byte. Sind diese Daten wiederum geschrieben, werden die folgenden Daten an Startadresse + 2 * 1518 Byte geschrieben und beim nächsten mal wieder auf

Zugriff (R/W)	**Registername**		**Registernr.** (Offset)
R	Frame Sample Count (15 Bit) + Startbit		$1C / 1D
R	Frame Sample Count (15 Bit) + Startbit		$1A / 1B
R	Frame Sample Count (15 Bit) + Startbit		$18 / 19
R	Buffer Index (Momentaner Schreibindex der HW)		$16 / 17
R	IRQ Count (Interrupt Zähler)		$14 / 15
R	Entire Sample Count LW		$12 / 13
R	Entire Sample Count HW		$10 / 11
R/W	Pattern 6	Pattern 7	$0E / 0F
R/W	Pattern 4	Pattern 5	$0C / 0D
R/W	Pattern 2	Pattern 3	$0A / 0B
R/W	Pattern 0	Pattern 1	$08 / 09
R/W	SRAM Pointer LW		$06 / 07
R/W	SRAM Pointer HW		$04 / 05
R/W	Opt Parameters		$02 / 03
R/W	Control		$00 / 01
	Startadresse		**0x1000**

Tabelle 2.2: Memory Interface HW - SW

die Startadresse. Somit ergibt sich für die Daten ein Ringbuffer mit drei Speicherplätzen, wobei bei jedem Wechsel ein X2X Interrupt ausgelöst wird.

Firmware Ringbuffer

Grundsätzlich gilt, dass Interrupts in einem System so schnell wie möglich bedient, das heißt deren Handler ausgeführt werden müssen, da ansonsten weitere, gleiche Interrupts, verloren gehen können. Demnach muss vor allem in diesem Teil des Programmes auf Effizienz geachtet werden. Jedes mal, wenn in der X2X Scope Firmware ein X2X Read Interrupt ansteht, werden die momentanen Kompressionsparameter aus den entsprechenden Registern (siehe Tabelle 2.2) gelesen und zusammen mit den anderen Headerdaten zu einem fertigen Header zusammengefügt und in einer eigenen Ringbufferstuktur abgespeichert. Darüber hinaus erfolgt die Speicherung eines Index, welcher den Offset der Daten zur Startadresse angibt, gespeichert, um beim Senden die richtigen Paketdaten zu finden. Im Interrupt Handler müssen also nicht die kompletten Daten gespeichert werden, sondern lediglich der Teil des Headers, der sich von Paket zu Paket unterscheidet. Dies resultiert in einer sehr kurzen Ausführungszeit der Interrupt Routine und einem sehr rasch reagierenden System.

Senden der Daten

Innerhalb der Hauptschleife des Programmes, also während der Leerlaufphase, wird zyklisch eine Funktion aufgerufen, welche den Firmware Ringbuffer auf eventuellen Inhalt überprüft. Ist der Ringbuffer nicht leer, so werden kurzzeitig alle Interrupts gesperrt, damit auf die Daten zugegriffen werden kann. Darauf wird in einer Schleife so lange die Sendefunktion[1] mit den entsprechenden Daten aufgerufen, bis der Buffer wieder leer ist. Dabei muss lediglich der Header zu den dazugehörigen Daten kopiert werden, da die gesamten Ethernet Paketdaten bereits formatiert im Speicher liegen. An die Sendefunktion erfolgt daraufhin lediglich die Übergabe der Länge des Ethernet Paketes und der Startadresse der Daten samt Header. Demnach wird auch in dieser Funktion die Ausführungszeit weitgehend optimiert, um ein schnelles Verarbeiten der empfangenen Daten zu gewährleisten. Dank dieses effizienten Vorgehens konnten die strengen Timing-Anforderungen an die Firmware eingehalten werden und die Firmware ist in der Lage den hohen Datendurchsatz in der geforderten Zeit zu bewältigen.

2.5 UDP Kommunikation

Beim User Datagram Protocol (UDP) handelt es sich um ein verbindungsloses Protokoll aus der Familie der Internet Protokolle. Das bedeutet, dass im Gegensatz zum Transport Control Protocol (TCP) kein sogenannter Handshake erfolgt. Nach dem Senden der Daten erfolgt hier keine Rückmeldung, ob alle Daten fehlerfrei angekommen sind. Dies erhöht zwar den Datendurchsatz, da der Protokoll Overhead geringer ist als bei verbindungsorientierten Netzwerkprotokollen, bietet aber keinerlei Sicherheit bezüglich der Vollständigkeit der Daten.

2.5.1 Paketaufbau

Um die Daten über ein Ethernet Netzwerk senden zu können, wird ein UDP Frame in einen Internet Protocol (IP) Frame gepackt (sogenannter IP Pseudo Header) und gesendet. Um die Daten beim Empfänger der richtigen Anwendung zuweisen zu können, werden Ports verwendet.

IPv4 Pseudo Header

Als Header bezeichnet man in Netzwerkprotokollen die ersten Bytes eines Frames, die Informationen zu Sender, Empfänger, Größe und anderen sogenannten Metadaten liefern. Ein IPv4 Pseudo Header besteht aus folgenden Datenfeldern:

[1] Zum Senden der UDP Frames wurde eine bereits fertig erstellte Library des Vorgängerprojektes verwendet. [Pic06]

- **Quell IP Adresse**: 32 Bit IP Adresse des Senders
- **Ziel IP Adresse**: 32 Bit IP Adresse des Empfängers
- **Leerfeld**: 8 Bit
- **Protokoll ID**: 8 Bit für UDP zum Beispiel 17
- **UDP Datagram Länge**: 16 Bit Länge des anschließenden UDP Datagrams

UDP Header

Im Anschluss an die 12 Byte IPv4 des Pseudo Headers wird der UDP Header, bestehend aus folgenden Datenfeldern, übertragen.

- **Quell Port**: 16 Bit Portnummer des sendenden Prozesses, der für eine etwaige Antwort des Empfängers benötigt wird.
- **Ziel Port**: 16 Bit Portnummer des empfangenden Prozesses.
- **Länge**: 16 Bit Länge des Datagrams inklusive Header, Prüfsumme und Daten.
- **Prüfsumme**: 16 Bit Prüfsumme, die über Header und Daten gebildet wird.

X2X Scope Header

Da UDP verbindungslos ist, bleibt der Verlust von Daten am PC unbemerkt. Um diesem Problem entgegen zu wirken, wurde ein eigener Header eingeführt, der diese Fehlererkennung ermöglicht. Weiteres lassen sich damit Informationen zur Komprimierung, sowie die Kompressionsparameter übertragen. Der Aufbau des Headers ist in Tabelle 2.3 angeführt.

2.5.2 Datenübertragung

Zur Abschätzung der notwendigen Kompressionsrate musste im Vorhinein die zu erwartende Datenrate berechnet werden. Dazu wird zuerst der Protokoll Overhead, verursacht durch die Header eines Frames, ermittelt.

Overhead

Als Overhead bezeichnet man in der Netzwerktechnik Daten, die zur Realisierung des Protokolls notwendig sind, aber nicht zu den Nutzdaten zählen. Folglich sollte dieser so gering wie möglich gehalten werden, da sonst ein erheblicher Teil der Bandbreite dafür aufgewendet werden muss und demnach bei der Nutzdatenübertragung fehlt. In unserem Fall wird der Netzwerk Overhead durch die einzelnen Header bestimmt und lässt sich daher einfach berechnen.

Ethernet

In einem Ethernet Netzwerk werden Daten in Paketen übertragen, der Aufbau eines Paketes ist dabei immer gleich (siehe Abb. 2.4) und die maximale Länge beträgt 1526 Byte.

EntireSampleCount Anzahl der bereits aufgezeichneten Samples Erkennung verloren gegangener Samples beim Empfänger	Byte: 0:3
FrameSampleCount Anzahl der im Frame enthaltenen Samples Erkennung verloren gegangener Samples beim Empfänger und Berechnung der Kompressionsrate im aktuellen Frame	Byte: 4:5
minLen Länge, bis zu der Samplefolgen nicht komprimiert werden.	Byte: 6 H
compBits Anzahl der Bits, die zur Darstellung der Folgenlänge verwendet werden	Byte: 6 L
optFlag Flag zeigt, ob Optimierung aktiviert wurde (Special Pattern)	Byte: 7 (Bit 7)
startBit Flag zeigt, mit welchem Bit die erste Sequenz beginnt.	Byte: 7 (Bit 6)
patCnt 2^{patCnt} gibt an, wie viele Bits für Special Pattern verwendet werden.	Byte: 7 (Bit 5:0)
Special Pattern Special Pattern 1 Special Pattern 2 Special Pattern 3 Speical Pattern 4 Special Pattern 5 Special Pattern 6 Special Pattern 7 Speical Pattern 8	 Byte: 8 Byte: 9 Byte: 10 Byte: 11 Byte: 12 Byte: 13 Byte: 14 Byte: 15
Reserve Bytes Für eventuelle Erweiterungen reserviert.	Byte: 16:19
Headerlänge	20 Bytes

Tabelle 2.3: X2X Scope Header Aufbau

Um die maximale Nutzdatenübertragung berechnen zu können, muss vorerst die Bandbreite des verwendeten Netzwerkes betrachtet werden. Im Fall des X2X Scope handelt es sich um ein Standard 100 MBit/s Ethernet Netzwerk. Im Netzwerk können demzufolge maximal $\frac{100*10^6}{8} = 12,5 * 10^6 Byte/s$ übertragen werden, das entspricht $\frac{12,5*10^6}{1526} = 8191$ Ethernet Paketen mit einer maximalen Framelänge von 1518 Byte. Pro Ethernet Paket können höchstens 1500 Byte Nutzdaten übertragen werden, die restlichen 26 Byte sind Overhead.

Die X2X Taktfrequenz beträgt 12 MHz und wird 8-fach abgetastet, das entspricht einer Samplerate von $8*12*10^6 = 96*10^6 Samples/s$. Dieser Datenstrom stellt die anfallenden Nutzdaten dar und muss zusätzlich zum Protokoll Overhead über das Ethernet Netzwerk übertragen werden.

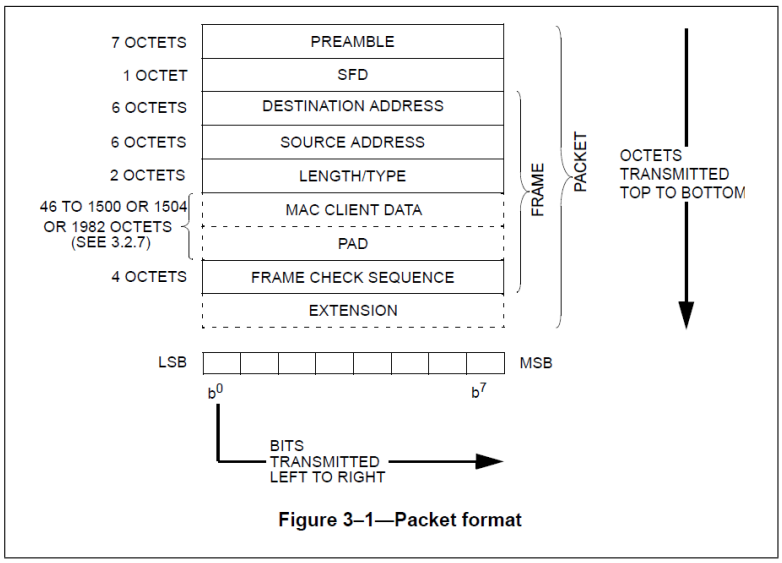


Abbildung 2.4: Ethernet Paket nach IEEE 802.3 [Ins08]

Nicht komprimierter Datenstrom

Es können 1500 Byte Nutzdaten minus 48 Byte Overhead (IP-, UDP- und X2X Header) = 1452 Byte für die Datenübertragung verwendet werden. Ohne eine Komprimierung ist es möglich, im X2X Scope Header auf die Kompressionsparameter zu verzichten. In diesem Fall müssen 10 Byte addiert werden, demnach können 1462 Byte oder $1462 * 8 = 11696$ Samples Nutzdaten pro Paket übermittelt werden.
Bei voller Auslastung des Netzwerkes könnte folglich eine Übertragung von 8191 Paketen/s * 11696 Samples = 95.801.936 Samples/s erreicht werden. Dies ist allerdings nur ein theoretischer Wert, da im Allgemeinen nicht die volle Bandbreite in einem Ethernet Netzwerk zur Verfügung steht. Aufgrund der unzureichenden Übertragungsgeschwindigkeit wurde eine Kompression der Daten in Betracht gezogen.

Komprimierter Datenstrom

Es können $1452 * 8 = 11616$ Samples pro Ethernet Paket gesendet werden. Dies führt wegen der Kompressionsparameter zu einer etwas geringeren Übertragungsrate von $8191 * 11648 = 95.408.768 Samples/s$. Um die Daten übermitteln zu können wurde als Richtwert eine minimale Kompressionsrate von 50% festgelegt[2]. Somit können pro Ethernet Paket circa $11616 * 2 = 23232$ Samples gesendet werden. Das entspricht einer Übertragungsrate von $8191 * 23232 = 190.293.312 Samples/s$. Jetzt kann der Datenstrom problemlos übermittelt werden und es könnten theoretisch auch noch andere Stationen im selben Netzwerk kommunizieren.

[2]Eine spätere Analyse zeigte eine durchschnittliche Kompressionsrate von circa 74%.

KAPITEL 3

Datenkompression

"Der Wege sind viele, doch das Ziel ist eins."

Rumi

Dieses Kapitel befasst sich mit den verschiedenen Möglichkeiten der Datenkomprimierung und der Entwicklung eines geeigneten Verfahrens zur Komprimierung der Sampledaten. Ausgehend von bereits bekannten Algorithmen wurde hierzu eine Methode entwickelt, welche versucht, auf einfache Weise die Vorteile der einzelnen Codes zu vereinen.

3.1 Allgemeines Prinzip

Bei der Datenkomprimierung, oder auch Datenkompression, wird versucht, die Darstellung von Information dahingehend zu verändern, dass die kodierte Darstellung einen sogenannten Kodiergewinn liefert. Das bedeutet, dass die kodierte Darstellung kürzer als das Original ist. Bei genauerer Überlegung kann dieses Ziel nur durch zwei verschiedene Vorgehensweisen erreicht werden.

3.1.1 Verlustfreie Datenkompression

Um diese Art der Datenkomprimierung, auch Redundanzreduktion genannt, zu verstehen, muss zuerst der Begriff der Redundanz genauer erläutert werden. Man spricht von redundanter Information, wenn ein und dieselbe Information in einer Nachricht mehrfach genannt wird, diese Mehrfachnennung aber keinen zusätzlichen Informationsgehalt liefert. Dies lässt sich beispielsweise anhand der deutschen Sprache zeigen, denn viele Wörter wie *z.B.*, *etc.*, *usw.* lassen sich *i.Allg.* einfach durch eine *Abk.* darstellen und werden trotzdem richtig interpretiert, ohne dass dabei Information verloren geht. Dieses

Prinzip wird für die Datenkompression verwendet, folglich wird versucht, redundante Information in Texten zu identifizieren und darauf zu verzichten. Im Zuge dieser Arbeit werden ausschließlich verlustfreie Verfahren in Betracht gezogen, da Sampledaten zwar redundante, aber keine irrelevante Information beinhalten.

3.1.2 Verlustbehaftete Datenkompression

Bei dieser Methode, auch Irrelevanzreduktion genannt, macht man sich die Tatsache zu Nutze, dass gewisse Teilinformationen einer Nachricht für den Empfänger nicht relevant sind und somit außer Acht gelassen werden können. Diese Verfahren kommen vor allem im Bereich der Audio-, Video- und Bildkompression zum Einsatz, da hier auf Kosten der Qualität sehr viel Information eingespart werden kann.

3.2 Ausgangssituation

Die meisten gängigen Kompressionsalgorithmen gehen davon aus, die Daten vollständig analysieren zu können und sind so in der Lage, Tabellen mit verschiedenen Häufigkeiten und Mustern zu erstellen. Diese werden dann dazu verwendet, verschiedene Optimierungen beziehungsweise Ersetzungen auf die Daten anzuwenden.
Der X2X Bus überträgt Daten mit einer $12MHz$ Taktfrequenz. Um etwaige Signalstörungen und Verzerrungen darstellen zu können, wird beim Abtasten des Signals achtfaches Oversampling verwendet. Dies ergibt einen Datenstrom von $8 * 12MBit/s = 96MBit/s$. Die Daten müssen per UDP über Ethernet an den Auswertungsrechner übertragen werden. Da die maximale Bandbreite von Ethernet $100MBit$ beträgt, diese aufgrund von Overhead und Kollisionen allerdings nicht erreicht werden kann, ist es nicht möglich, den $96MBit/s$ Datenstrom unkomprimiert zu übertragen. Aufgrund dieser Ausgangssituation musste man sich für ein Kompressionsverfahren entscheiden, welches einerseits den Datenstrom um mindestens 50% vermindert und andererseits einfach zu realisieren ist.

3.3 Analyse der Daten

Anders als bei der Bild- oder Dateikomprimierung kann bei der Entwicklung dieses Kompressionsalgorithmus nicht davon ausgegangen werden, dass bereits zu Beginn der Komprimierung alle Daten vollständig zur Auswertung bezüglich Häufigkeiten und Muster zur Verfügung stehen. Eine solche Vorgehensweise ist in diesem Fall nicht direkt möglich, da es sich um einen Datenstrom handelt und immer nur ein Teil der Daten zwischengespeichert werden kann. Somit kann keine vollständige Tabelle über die Häufigkeiten eines bestimmten Alphabets erstellt werden.
Um aber trotzdem einen Überblick über die annähernde Beschaffenheit der zu erwartenden Daten zu erhalten, wurden mithilfe eines Logic Analyzers drei Aufzeichnung mit einer Samplerate von $100Megasamples/s$ erstellt. Diese Samplerate entspricht nicht genau der späteren Abtastrate von $96Megasamples/s$, das Ergebnis liefert allerdings einen

guten Richtwert für diverse Überlegungen zum Kompressionsalgorithmus, da der Fehler minimal ist.

3.3.1 Logic Analyzer

Ein Logic Analyzer ermöglicht das Aufzeichnen des zeitlichen Verlaufs eines logischen Signals. Im Gegensatz zu einem Oszilloskop, welches den genauen Signalverlauf wiedergibt, können also nur logische Pegel betrachtet und ausgewertet werden. Dafür bietet der Logic Analyzer gegenüber dem Oszilloskop mehr Eingangskanäle und eine Vielzahl an Funktionen und Einstellungen zum Auswerten der Daten. Grundsätzlich unterscheidet man bei der Aufzeichnung zwei verschiedene Methoden:

1. Timing Mode
   Im Timing Mode wird bei der Abtastung eine interne Clock-Frequenz verwendet, zu deren Flanken das Signal abgetastet wird. Diese Funktion ähnelt der eines Speicheroszilloskops.

2. State Mode
   Im State Mode wird als Abtastfrequenz die Clock-Frequenz des Targets verwendet.

Bei der Aufzeichnung der Daten wurde der Logic Analyzer im Timing Mode verwendet, da dies auch dem Abtastverfahren der IO Module entspricht und man somit die gleichen Sampledaten erhält.

3.3.2 Datenaufzeichnung

Der X2X Bus wurde während der Aufzeichnung mit einigen wenigen IO-Modulen und zufälligen Daten belastet. Insgesamt lieferten die Aufzeichnungen drei unabhängige Testfiles (*samples1.txt*, *samples2.txt*, *samples3.txt*) mit je 100.000 Samples (ausschließlich `0` und `1` Folgen), die auf dem PC gespeichert wurden. Die so erhaltenen Daten konnten darauf auf bestimmte Häufigkeiten und besondere Eigenheiten analysiert werden. Da allerdings in der Hardwareimplementierung der Abtastung der Speicher begrenzt ist und deshalb nicht 100.000 Samples gleichzeitig komprimiert werden können, musste zusätzlich untersucht werden, welche Ergebnisse der Kompressionsalgorithmus bei kleineren, eventuell einseitig verteilten Daten liefert, wobei der Algorithmus aber auf die 100.000 Sample Testfiles optimiert wurde. Zu diesem Zweck wurden zusätzlich Teilausschnitte aus den drei großen Files separat abgespeichert. *part1.txt*, *part2.txt* und *part3.txt* mit 18.000 bis 20.000 Samples und *part4.txt*, *part5.txt* und *part6.txt* mit jeweils nur 1024 Samples.
Da pro Frame voraussichtlich ca. 20.000 komprimierte Samples übertragen werden, konnte man mit diesen Testfiles den Algorithmus unter realen beziehungsweise Worst Case Verhältnissen testen. Mithilfe der so erhaltenen Informationen konnte daraufhin begonnen werden, einen passenden Kompressionsalgorithmus zu entwickeln.

Bit 1		Bit 0	
Länge	**Anzahl**	**Länge**	**Anzahl**
11	1149	8	1591
10	610	66	213
9	216	67	195
8	178	9	157
17	111	10	78
25	51	77	66
19	44	16	65
67	39	17	50
13	37	33	42
44	36	25	42
27	24	19	31
58	22	58	22
26	21	18	22
59	19	11	18
33	15	83	18
12	12	24	16

Tabelle 3.1: Anzahl und Länge der 16 Häufigsten `1` und `0` Streams

3.3.3 Datenauswertung

Es stellte sich schnell heraus, dass in den Samples meist längere Folgen mit gleichen logischen Werten vorkommen. Mithilfe eines kleinen Analyseprogramms wurden diese Folgen gezählt und das Ergebnis in einer Tabelle dargestellt (Tabelle 3.1). Die Tabelle bezieht sich auf das Testfile *samples1.txt*, bei *samples2.txt* und *samples3.txt* ist das Ergebnis allerdings nahezu identisch. Eine spätere Analyse der tatsächlich auftretenden Sampledaten bestätigte die Richtigkeit dieser Häufigkeitsabschätzung. Darüber hinaus wurden Analysen bezüglich bestimmter Muster, wie zum Beispiel Flankenübergänge (`0000000111111111`) und ähnliche Sequenzen durchgeführt. Diese stellten sich jedoch als nicht besonders vielversprechend heraus.

3.4 Kompressionsalgorithmen

Im Anschluss an die Analyse der Testdaten konnte dank der daraus erhaltenen Ergebnisse begonnen werden, ein zielführendes Kompressionsverfahren zu entwickeln. Dabei wurde zuerst auf bestehende Verfahren Bezug genommen, um im nächsten Schritt Vorteile einzelner Verfahren zu kombinieren. Grundlegend gibt es zwei verschiedene Ansätze zur Komprimierung von Daten.

3.4.1 Stringersatzverfahren

Beim Stringersatzverfahren, oder auch Wörterbuchkompression, werden sich häufig wiederholende Zeichenfolgen durch ein kürzeres Ersatzsymbol eines anderen Alphabetes ausgetauscht. Ein Beispiel dafür wäre der klassische Run Length Encoding Algorithmus, bei dem Folgen von gleichen Zeichen durch die Anzahl des Zeichens ersetzt werden, so wird zum Beispiel *aaaa* zu *4a* beziehungsweise zu *a4*, je nach Implementierung.

- Vorteil: Stringersatzverfahren sind in der Regel relativ einfach zu implementieren.

- Nachteil: Die Kompressionsrate ist meistens nicht sehr gut, was dazu führt, dass solche Kompressionsalgorithmen häufig zusammen mit anderen Verfahren, wie zum Beispiel der Entropiekodierung, kombiniert werden.

3.4.2 Entropiekodierung

Bei der Entropiekodierung wird, im Gegensatz zum Stringersatzverfahren, jedem Zeichen eines Textes, entsprechend seiner Häufigkeit, eine unterschiedlich lange Folge von Bits zugeordnet. Je häufiger ein Zeichen vorkommt, desto kürzer ist seine Bitkodierung. So wird versucht, zu einem vorgegebenen Text eine optimale Kodierung zu finden. Grundsätzlich unterscheidet man bei der Entropiekodierung die Kodierung mit ganzen Bits, wie zum Beispiel den Huffman Code (siehe Abschnitt 3.4.5), und die sogenannte arithmetische Kodierung, bei welcher Kompressionsraten von $< 1 Bit/Zeichen$ erreicht werden. Weiteres kann zwischen zwei Modellen unterschieden werden:

- **statisches Verfahren**: Dabei wird die Ersetzungstabelle im Vorhinein statisch durch Analyse des Textes erstellt.

- **dynamisches Verfahren**: Hier wird, ausgehend von einem statischen Modell, die Tabelle während des Kompressionsvorganges laufend anhand der bereits komprimierten Daten angepasst und optimiert.

3.4.3 Run Length Encoding (RLE)

Der Run Length Encoding Algorithmus zählt zu den Stringersatzverfahren, da dabei lange, gleiche Zeichenketten durch die Anzahl der Zeichen ersetzt werden.

Beschreibung

Aufgrund der Ergebnisse aus der Datenanalyse erschien es vernünftig, einen klassischen Run Length Encoding Algorithmus zu verwenden. Hierbei handelt es sich um einen sehr einfachen Algorithmus zum Komprimieren von Zeichenketten, in denen gleiche Zeichen mehrmals hintereinander vorkommen. Es wird anstelle der vollständigen Zeichenkette lediglich die Anzahl an Zeichen, gefolgt vom Symbol selbst, ausgegeben. So wird beispielsweise aus der Zeichenfolge `aaabbbbbccc` die komprimierte Folge *`3a4b2c`*.

Implementierung

Im Fall der Samples besteht die Zeichenfolge lediglich aus `1` und `0`, deshalb muss bei der Implementierung zu der Anzahl kein dazugehöriges Zeichen angegeben werden, da angenommen werden kann, dass sich `1` und `0` immer abwechseln. Der komprimierte Code *`1234`* steht folglich für *0110001111*. Allerdings müssen auch die Zahlenangaben im komprimierten Code als Binärzahlen dargestellt werden. Hierbei muss man sich auf eine geeignete Größe festlegen. Da die Anzahl Null nicht dargestellt werden muss, wird ein Offset von 1 verwendet. So kann man mit vier Bit theoretisch Folgen mit maximaler Länge von $2^4 = 16$ Zeichen kodieren, obwohl *`1111`* dem Wert 15 entspricht. Wenn aber eine Zeichenfolge länger als 2^{Bits} ist, muss diese in zwei einzelne Folgen aufgeteilt werden. In diesem Fall ginge allerdings die Annahme verloren, dass sich `1` und `0` Folgen stets abwechseln. Aufgrund dieses Problems muss in solchen Fällen der maximalen Länge einer Folge die Information über die Zeichen der nächsten Sequenz mitgegeben werden. Dazu wird das letzte Bit verwendet, dementsprechend beträgt die längste Folge, welche noch kodiert werden kann, $2^4 - 1 = 15$ Zeichen, wobei:

- *`1110`* bedeutet 15 aktuelle Zeichen, als nächstes folgt eine `0` Sequenz
- *`1111`* bedeutet 15 aktuelle Zeichen, als nächstes folgt eine `1` Sequenz

Wenn eine Folge also aus 13 `1` besteht, kann sie mit *`1100`* dargestellt werden, was einer Kompressionsrate von $1 - \frac{4}{13} = 69,23\%$ entspricht.

Pseudocode

```
Input:
INT compBits;
Data:
BOOL nextBit;
BOOL oldBit;
INT count;
count := 0;
oldBit := readBit();
while (nextBit := readBit()) != NULL do
    if nextBit = oldBit then
        count++;
        if count = 2^(compBits-1) then
            if (oldBit = readBit()) = 1 then
                write count+1;
            else
                write count;
            end
            count := 1;
        end
    else
        write count;
        write oldBit;
        oldBit := nextBit;
        count := 0;
    end
end
```

Algorithm 1: Pseudocode RLE

Auswertung und Analyse

Mit diesem Algorithmus wurden die in Abb. 3.1 dargestellten Ergebnisse erzielt. Die Folgen wurden dabei mit 5 Bit kodiert, da sich diese Anzahl anhand mehrerer Versuche als optimal herausstellte. Das Ergebnis scheint auf den ersten Blick zufriedenstellend, doch analysiert man die Beschaffenheit der späteren Daten genauer, stößt man bei diesem Algorithmus auf ein Problem. Da das X2X Scope vor allem im Fehlerfall zum Aufzeichnen von Daten verwendet wird, kann es vorkommen, dass es sich dabei nicht nur um lange Folgen, sondern auch um kurze, alternierende Sequenzen handelt. Diese Tatsache stellt für den Algorithmus ein beachtliches Problem dar. Die Folge `000` würde zum Beispiel mit fünf Bit, also *`00010`* kodiert und wäre plötzlich als komprimierter Code länger als die ursprüngliche Zeichenfolge. Solche extrem kurzen Zeichenfolgen kommen zwar selten vor (*samples1.txt* – *samples3.txt*), aber generell gilt, je kürzer die Folgen, desto schlechter die Kompressionsrate (*part4.txt* – *part6.txt*). Da im Worst Case, bei alternie-

renden Samples, die komprimierten Daten um den Faktor der verwendeten Bits größer als die ursprünglichen Daten werden, muss hier eine Änderung vorgenommen werden. Man könnte dem Algorithmus beispielsweise eine bestimmte Mindestlänge vorgeben, ab welcher kodiert wird und kleinere Folgen nicht kodiert weitergeben. Diese Überlegung führte zur Implementierung der ersten Version des späteren Kompressionsalgorithmus.

Laufzeitabschätzung Beim Vergleich verschiedener Algorithmen werden oft Laufzeitabschätzungen als Maß für die Güte verwendet. Dabei wird die Laufzeit anhand von n Elementen angegeben, welche der Algorithmus verarbeiten muss. Eine Schleife, welche von *1* bis n läuft, hat also eine Laufzeit von $\Theta(n)$. Zwei verschachtelte Schleifen, welche jeweils bis n laufen, haben insgesamt eine Laufzeit von $\Theta(n^2)$. Dabei werden konstante Faktoren nicht berücksichtigt. So werden zwei Schleifen, welche von *1* bis n beziehungsweise von *1* bis *3n* iterieren, beide als $\Theta(n)$ angegeben. Die Bezeichnung $\Theta(n)$ gibt dabei an, dass der Algorithmus n Durchgänge ausführt. Weiteres gilt, dass sich solche Abschätzungen nicht addieren, es gilt immer die größte Laufzeit. Besitzt ein Algorithmus also ein Konstrukt, welches in $\Theta(n^2)$ läuft und außerdem, davon entkoppelt, eine Schleife von *1* bis n, so läuft der Algorithmus insgesamt in $\Theta(n^2)$ und nicht etwa $\Theta(n^2 + n)$. Genauere Erläuterungen zur Laufzeitabschätzung von Algorithmen können in der Fachliteratur nachgelesen werden [Sed02]. Die Laufzeitabschätzung bei RLE gestaltet sich als relativ einfach. Jedes der n Zeichen muss einmal durchlaufen und gezählt werden, wodurch sich insgesamt eine Laufzeit von $\Theta(n)$ ergibt.

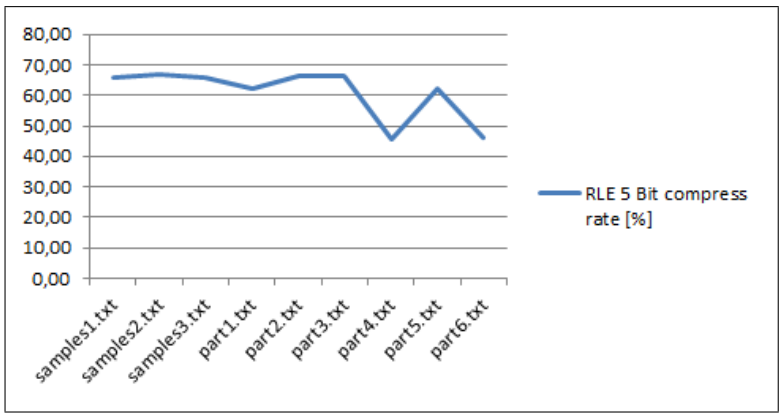


Abbildung 3.1: Kompressionsergebnisse RLE

3.4.4 Compress I

Die erste Version des späteren Kompressionsalgorithmus ist eine abgeänderte Version von RLE und zählt deshalb zu den Stringersatzverfahren.

Beschreibung

Bei Compress I handelt es sich im Prinzip um eine Art RLE, mit dem Unterschied, dass sich die Anzahl der zum Kodieren verwendeten Bits und die Mindestlänge der Zeichenfolge einstellen lassen. Ist eine Bitfolge kürzer als die Mindestlänge, wird diese unkomprimiert ausgegeben. Dazu ist allerdings ein zusätzliches Bit notwendig, das Aufschluss darüber gibt, ob die folgende Bitfolge komprimiert oder unkomprimiert ist.

Implementierung

Die Daten werden also wie beim RLE durchlaufen und die Längen der einzelnen Folgen gezählt. Ist eine Folge allerdings zu kurz, wird diese nicht komprimiert, sondern unkomprimiert kodiert. Der so entstandene Code besteht demnach aus zwei verschieden Arten von Sequenzen:

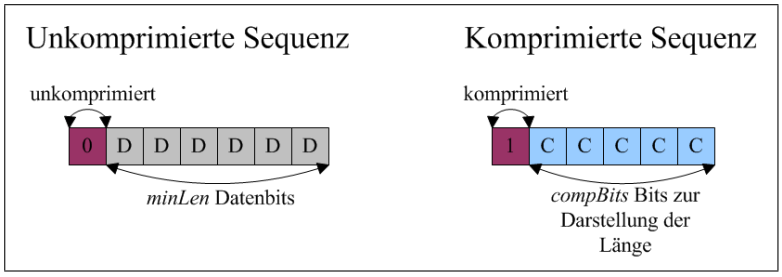


Abbildung 3.2: Unkomprimierte und komprimierte Sequenz in Compress I

- unkomprimiert: ein `0` Bit, gefolgt von einer einstellbaren Anzahl an unkomprimierten Bits
- komprimiert: ein `1` Bit, gefolgt von einer einstellbaren Anzahl an Bits, die die Länge der Folge darstellen

Pseudocode

Der Pseudocode baut auf dem RLE Pseudocode auf, allerdings ist eine zusätzliche Abfrage auf die Länge der komprimierten Folge nötig. Um im Fall der unkomprimierten Ausgabe die Samples ausgeben zu können, muss ein Buffer der Größe *minLen* angelegt werden, welcher immer die aktuellen Samples abspeichert.

```
Input:
INT compBits;
INT minLen;
Data:
INT count := 0;
INT index := 0;
BOOL buffer[minLen];
buffer[index] := readBit();
index++;
while (buffer[index] := readBit()) != NULL do
    if buffer[index] = buffer[index-1] then
        count++;
        if count = 2^compBits − 1 then
            write 1;
            index := 0;
            if (buffer[index] := readBit()) = 1 then
                write count+1;
            else
                write count;
            end
            count := 1;
            index++;
        end
    else
        index := 0;
        if count > minLen then
            write 1;
            write count;
        else
            write 0;
            for i := count to minLen do
                buffer[i] = readBit();
            end
            write buffer;
            buffer[index] := readBit();
            count := 1;
            index++;
        end
    end
end
```

Algorithm 2: Pseudocode Compress I

Auswertung und Analyse

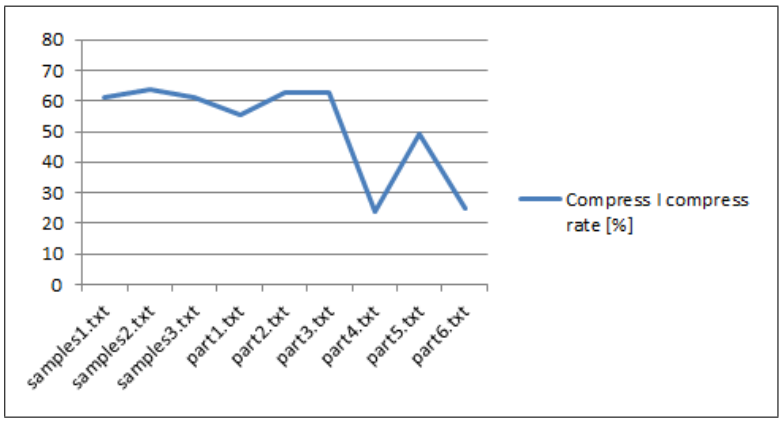


Abbildung 3.3: Kompressionsergebnisse Compress I

Da unkomprimierte Folgen mit Zeichen enden, welche völlig unabhängig von der vorhergehenden Folge sind, müsste die Annahme der abwechselnden `0` und `1` Sequenzen wieder verworfen werden. Um dieses Problem zu lösen, wurde vereinbart, dass das letzte Bit der unkomprimierten Folge gleich dem ersten Bit der darauffolgenden Sequenz sein muss. Ist diese Bedingung nicht gegeben, wird noch einmal eine unkomprimierte Folge ausgegeben, solange bis die Anforderung erfüllt ist. Wegen dieser Vereinbarung und des zusätzlichen Bits bei der Kodierung verschlechtert sich die Kompressionsrate um nur ein paar Prozent gegenüber dem Standard RLE Algorithmus, allerdings konnten die Worst Case Szenarien, welche beim Standard RLE Algorithmus auftreten können, weitgehend verhindert werden. Im Worst Case, wenn also alle Folgen kleiner gleich der vorgegebenen *minLen* sind, beträgt die Größe der Daten das $1 + \frac{1}{minLen}$-fache der ursprünglichen Daten.

Laufzeitabschätzung Wie auch bei RLE muss im Algorithmus jedes Zeichen einmal durchlaufen werden, so ergibt sich auch hier eine Laufzeit von $\Theta(n)$.

3.4.5 Huffman Code

Aufgrund der immer noch nicht ganz zufriedenstellenden Ergebnisse von Compress I wurden weitere Überlegungen zur Optimierung angestellt. Dabei fiel der Gedanke auf die bereits erwähnte Entropiekodierung, genauer auf einen weitverbreiteten Algorithmus, welcher dieser Familie angehört, den Huffman Code.

Beschreibung

Der Huffman Code benötigt zum Erstellen der Ersetzungstabelle als Ausgangssituation einen Text, welcher aus einem Alphabet mit einer endlichen Anzahl an Symbolen be-

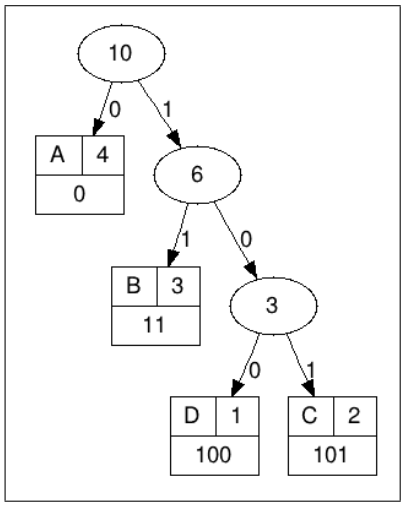


Abbildung 3.4: Huffman Baum zum Text *AAAABBBCCD*

steht. Daraus können, wie bereits erwähnt, statisch oder dynamisch Codewörter erstellt werden, welche, anders als bei arithmetischen Verfahren, aus ganzen Bit bestehen. Dabei wird zuerst die Anzahl der einzelnen Symbole gezählt und diese werden entsprechend ihrer Häufigkeit sortiert. Danach wird nach vorgegebenen Algorithmen ein Baum erstellt, in dessen Blättern die Codewörter stehen. Die Knoten enthalten Bits, die beim Durchlaufen des Baumes von der Wurzel bis zum jeweiligen Blatt zusammengefügt werden und somit das zum Blatt gehörige Codewort bilden.
Beispiel: Die Analyse eines (sehr einfachen) Textes `AAAABBBCCD` ergibt, dass folgende Symbole in absteigender Häufigkeit vorkommen: `A`, `B`, `C`, `D`. Wird darauf der Huffman Algorithmus zur Erstellung des Codebaumes angewendet, so erhält man folgenden Baum (Abb. 3.4) und die entsprechende Kodierung:

- *A = '0', B = '11', C = '100', D = '101'*

Im Gegensatz zur Anordnung der Knoten ist die Beschriftung der Kanten nicht eindeutig, deshalb können verschiedene Kodierungen erzeugt werden.
Wichtig ist allerdings, dass die Kodierung so aufgebaut ist, dass aus kodierten Folgen wieder eindeutig der Originaltext erstellt werden kann. Um dies zu gewährleisten, darf es nicht vorkommen, dass die Kodierung eines Symbols identisch mit dem Präfix der Kodierung eines anderen Symbols ist, da man sonst nicht mehr entscheiden könnte, um welches Originalsymbol es sich bei der Kodierung handelt. In vorhergegangenen Beispiel wären also, bei entsprechender Kantenbeschriftung, auch folgende Kodierungen gültig:

- `A = '0', B = '11', C = '101', D = '100'`
- `A = '1', B = '00', C = '010', D = '011'`
- `A = '1', B = '01', C = '001', D = '000'`

Insgesamt gibt es zu diesem Baum acht verschiedene Möglichkeiten der Kodierung.
Der Text, der mithilfe der im Baum (Abb. 3.4) ersichtlichen Kodierung nach erstem Schema komprimiert wurde, lautet:

- `0 0 0 0 11 11 11 100 100 101`

Dies entspricht einer mittleren Codewortlänge von $\frac{19}{10} = 1,9 \frac{Bit}{Zeichen}$.
Betrachtet man dazu als Vergleich die Originalkodierung von ASCII, in welcher 1 Zeichen mit 8 Bit dargestellt wird, so ergibt sich eine Kompressionsrate von $1 - \frac{1,9}{8} = 76,25\%$.

Implementierung

Im Falle der Sampledaten gibt es zwei mögliche Ansätze, ein Alphabet, welches den Anforderungen genügt, zu implementieren.

1. **Byte als Zeichen**: Es werden immer 8 Samples als Byte gelesen und der daraus resultierende Wert ergibt dann das Symbol. Die Samplefolge `10000001` wird also als Zahlenwert 65 interpretiert und dem ASCII Zeichen A zugewiesen. Demnach können in der Zeichenfolge maximal 256 verschiedene Symbole entstehen. Diese werden entsprechend ihrer Häufigkeiten kodiert und entsprechend entweder dynamisch oder statisch eine Ersetzungstabelle erstellt. Wegen der zusätzlichen Berechnung der Tabelle benötigt die dynamische Methode mehr Ressourcen, kann die Komprimierung dafür aber besser an die Daten anpassen. Für die statische Methode wurde die Tabelle anhand des Testfiles *samples1.txt* erstellt und darauf zur Komprimierung aller anderen Testfiles verwendet.

2. **Streamlängen als Zeichen**: Bei der zweiten Variante werden die verschiedenen Längen der Bitfolgen jeweils als Zeichen kodiert. Da in den Testfiles keine Längen größer 255 vorkommen, können dazu wieder 8 Bit ASCII Zeichen verwendet werden. Auch hier gibt es die dynamische und statische Möglichkeit der Tabellenerstellung.

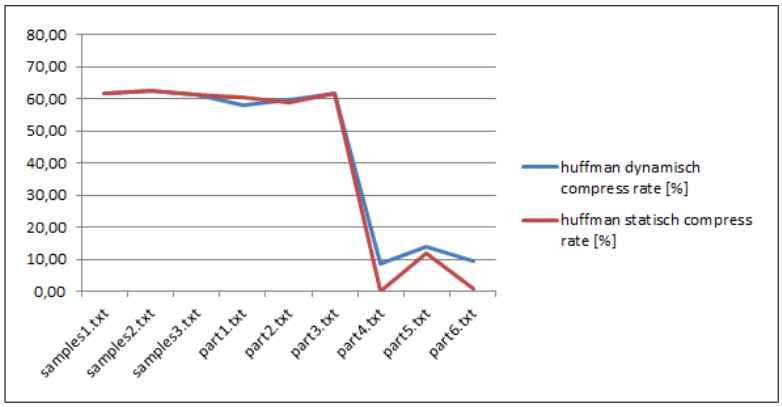


Abbildung 3.5: Kompressionsergebnisse Huffman Byte

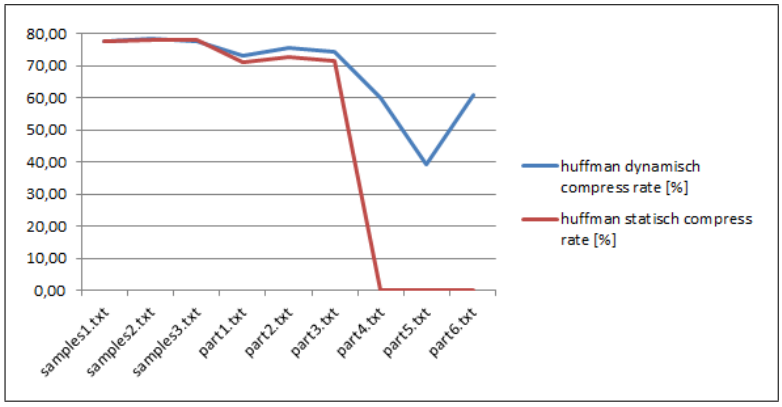


Abbildung 3.6: Kompressionsergebnisse Huffman Stream

Pseudocode

Input:
Alphabet C;
Data:
INT n;
Alphabet Q;
n := $|C|$;
Q := C;
for *i := 1* ***to*** *n - 1* **do**
  erstelle neuen Knoten z;
  left[z] := x := get_min(Q);
  right[z] := y := get_min (Q);
  f[z] := f[x] + f[y];
  INSERT (Q,z);
end
Return get_min (Q);

Algorithm 3: Pseudocode Huffman

Da sich der Huffman Code, wie bereits gezeigt, direkt aus dem dazugehörigen Huffman Baum ableiten lässt, genügt es einen Algorithmus für die Erstellung des Huffman Baums zu erstellen (bekannte Häufigkeiten vorausgesetzt). Der hier beschriebene Pseudocode ist stark vereinfacht und zeigt nur das grundlegende Prinzip. Im Rahmen dieser Bachelorarbeit wurde keine eigene Implementierung dieses Algorithmus entwickelt, sondern auf bereits vorhandene zurück gegriffen. Das Alphabet wird mit C dargestellt und somit ist $c \in C$ ein Zeichen aus dem Alphabet mit der Häufigkeit *[c]*. Q ist die sogenannte min-priority Queue, welche anfänglich mit dem gesamten Alphabet gefüllt wird. Nun werden immer zwei Elemente mit kleinster Priorität(=Häufigkeit) ausgewählt und "verbunden" also einem gemeinsamen Vaterknoten zugeordnet. Dieser hat nun die Priorität der Summe der beiden einzelnen Knoten und wird wieder in Q eingefügt. Auf diese Weise wird der Reihe nach der Huffman Baum erzeugt. Am Ende wird der Root Knoten des Baumes retourniert.[1]

Auswertung und Analyse

- **Byte als Zeichen**: Wie aus Abb. 3.5 und 3.6 ersichtlich, führt diese Variante, egal ob statisch oder dynamisch, nicht zu dem gewünschten Ergebnis. Offensichtlich ist, vor allem in kurzen Ausschnitten, das Alphabet mit seinen möglichen 256 Symbolen, in Relation zu den einzelnen Häufigkeiten, zu groß, um eine ausreichende Kompressionsrate zu erhalten.

- **Streamlängen als Zeichen**: Aufgrund der schlecht optimierten Tabelle versagt die statische Methode bei den kurzen Testfiles vollends. Der komprimierte Code ist um über 60% größer als die ursprünglichen Bitfolgen (hier als 0% compress rate dargestellt). Die dynamische Methode liefert für die großen Testfiles sehr gute Werte von annähernd 80% und knickt bei den kleineren Files zwar ein, allerdings nicht so stark wie die statische Methode. Das Problem ist, dass sich dieser Algorithmus aufgrund seiner Komplexität und der Ressourcenanforderungen nicht in der Abtasthardware implementieren lässt. Doch die grundsätzliche Idee des Huffmancodes, häufige auftretende Bitfolgen kürzer zu kodieren, wurde aufgegriffen und in die verbesserte Version des späteren Algorithmus implementiert.

Laufzeitabschätzung Das Aufsummieren aller n Zeichen aus Q liegt in $\theta(n)$. Operationen in einem Heap, wie es bei Q der Fall ist, benötigen im Worst Case eine Laufzeit von $\theta(lg(n))$, woraus eine Gesamtlaufzeit von $\theta(n * lg(n))$ resultiert.

3.4.6 Compress II

Hierbei handelt es sich um den fertig implementierten Algorithmus, einer Erweiterung zu Compress I, in welcher die Vorteile des Huffman Codes zumindest teilweise übernommen wurden.

[1] http://www.graviton.de/ai/algoan/referate/HuffmanCodes_Ala_ws0304.pdf

Beschreibung

Die Idee, die sich dahinter verbirgt, ist es, häufig auftretende Muster, in diesem Fall Bit Längen, kürzer zu kodieren, um somit die Kompressionsrate zu verbessern. Im Prinzip handelt es sich dabei um einen Algorithmus, der Teile von RLE mit der Worst Case Einschränkung von Compress I und grundsätzlichen Optimierungsideen des Huffmancodes kombiniert. Compress II lässt sich somit nicht mehr eindeutig einer Kategorie wie Stringersatzverfahren oder Entropiekodierung zuordnen.

Implementierung

Unkomprimierte Folgen, also Folgen die <= *minLen* sind, werden genauso dargestellt, wie in Compress I, also mit einem führenden `0` Bit und anschließend *minLen* Datenbits. Bei kodierten Folgen wird allerdings unterschieden zwischen sogenannten *specialPattern*, also häufig auftretenden Folgen und normal komprimierten Folgen. Die Unterscheidung erfolgt mit einem zusätzlichen Bit, `1` steht für *specialPattern*, `0` für normal. Die *specialPattern*-Liste wird im Vorhinein vereinbart und kodiert. Durch Analysieren der Testsequenzen lassen sich schnell die häufigsten `0` und `1` Folgen finden, welche dann kodiert werden. Dazu wird ein neuer Parameter eingeführt, *patCnt*. Dieser gibt die Anzahl der Bits an, die zur Kodierung der *specialPattern* verwendet werden. Da wiederum angenommen wird, dass sich 0 und 1 Folgen abwechseln, muss diese Information nicht zusätzlich dargestellt werden. Mit $patCnt = 2$ können demnach zum Beispiel $2^2 = 4$ *specialPattern* für `0` und vier *specialPattern* für `1` kodiert werden. Diese speziellen Folgenlängen werden im *specialPattern* Array gespeichert, wobei die erste Hälfte für die `1` Folgen und die zweite Hälfte für die `0` Folgen steht.
Es gibt also drei verschiedene Arten kodierter Folgen:

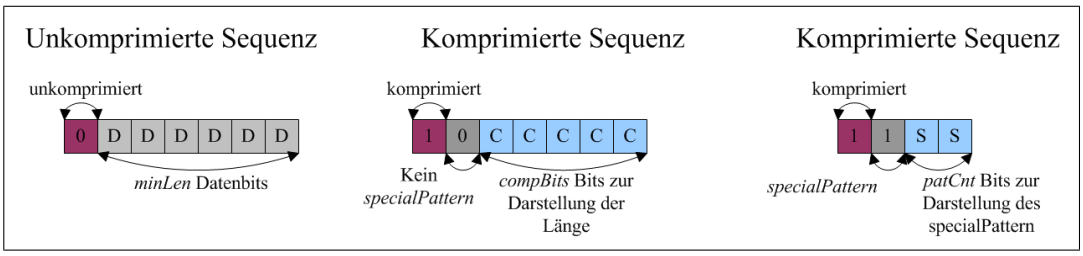


Abbildung 3.7: Unterschiedliche Sequenzen in Compress II

- **unkomprimiert**: eine `0` gefolgt von einer einstellbaren Anzahl an nicht kodierten Bits
- **komprimiert**, kein *specialPattern*: eine `1` gefolgt von einer `0` und einer einstellbaren Anzahl an Bits, welche die Länge der Folge darstellen
- **komprimiert**, *specialPattern*: eine `1` gefolgt von einer weiteren `1` und einer einstellbaren Anzahl an Bits, welche die Länge der Folge darstellen

Parameter	Wert
compBits	6
minLen	6
oFlag	TRUE
patCnt	1
specialPattern [0]	10
specialPattern [1]	11
specialPattern [2]	8
specialPattern [3]	9

Tabelle 3.2: Parameterwerte

Beispiel

Der Algorithmus wird anhand des folgenden Beispiels erklärt. Die Daten wurden im Vorfeld analysiert und die *1* Folgen 10 beziehungsweise 11 und *0* Folgen 8 und 9 als die Häufigsten ermittelt. Somit wird der Algorithmus mit den Parametern aus Tabelle 3.2 durchgeführt.

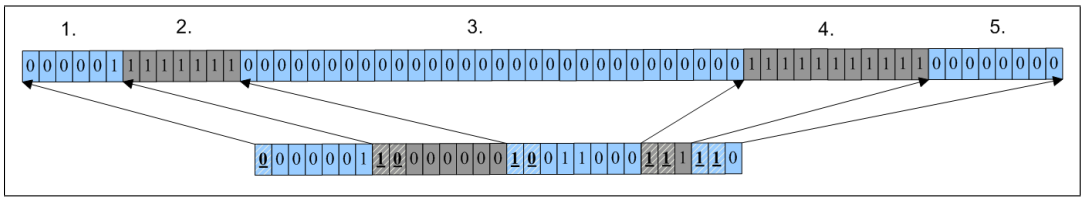


Abbildung 3.8: Beispiel zu Compress II

1. Die erste Folge ist <= *minLen* und demnach zu kurz für eine Komprimierung, deshalb wird ein *0* Bit und danach 6 Datenbits übertragen. Das sechste Bit = *1* würde eigentlich schon zur nächsten Folge gehören, da aber eine unkomprimierte Folge immer *minLen* Zeichen lang ist, wird es zur ersten Sequenz hinzugefügt. Anhand der Analyse der zu erwartenden Folgen wird *minLen* so gewählt, dass dieser Fall nur äußerst selten auftritt, da hierbei der komprimierte Code um $\frac{1}{minLen}$ % größer wird als der ursprüngliche Code.

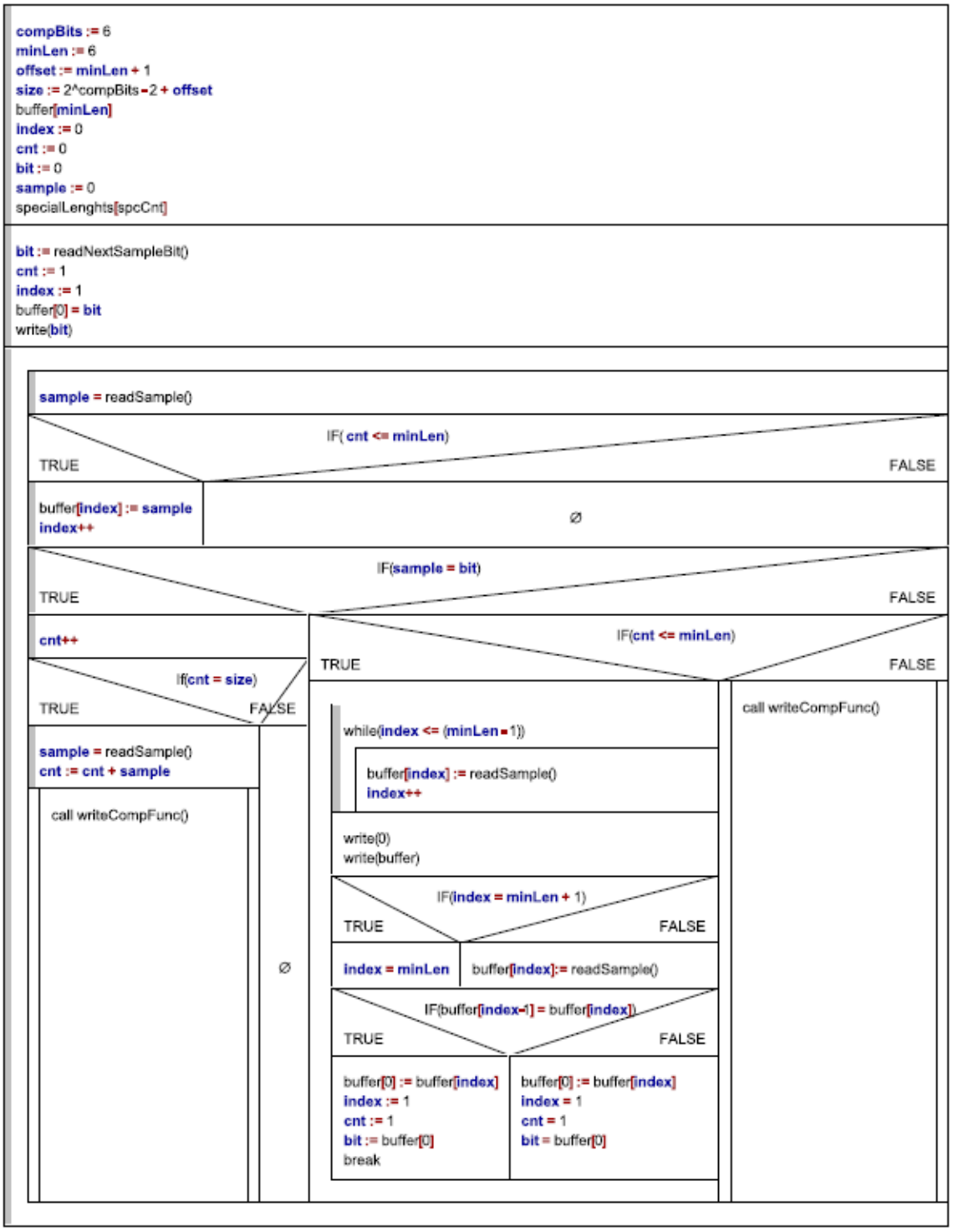
2. Die zweite Folge entspricht genau der Mindestlänge, aber nicht einem der *specialPattern*. Auch hier wird mithilfe der Analyse versucht, diese sehr kurzen Folgen entweder durch Erhöhung der *minLen* oder durch Einführen eines *specialPattern* möglichst optimiert zu kodieren. Das erste Bit = *1* signalisiert, dass es sich um eine komprimierte Folge handelt, das zweite Bit = *0* zeigt, dass es sich um kein *specialPattern* handelt. Die Länge wird mittels dem Parameter *compBits* dargestellt. In diesem Fall ergibt sich für die Länge: $6 - minLen = 0 = 000000$.

3. Bei der dritten Folge handelt es sich um eine Folge, die länger als *minLen* ist, aber kein *specialPattern*. Hierbei kann der Algorithmus die Folge beträchtlich verkürzen, da Folgen mit einer Länge von bis zu $2^{compBits} + minLen - 1$ mit nur $compBits + 2$ Bits dargestellt werden können. Die Länge beträgt $30 - minLen = 24 = 011000$.

4. Die vierte Folge stellt ein *specialPattern* für *1* Folgen dar. Das erste Bit = 1 signalisiert wiederum eine komprimierte Folge, das zweite Bit = *1* zeigt, dass es sich um ein *specialPattern* handelt und das dritte Bit = *1* gibt an, dass es sich um eine Folge der Länge 11 handelt, *0* würde für Länge 10 stehen.

5. Die letzte Folge zeigt ein *specialPattern* für `0` Folgen. Das dritte Bit = *0* zeigt, dass es sich um das *specialPattern* der Länge 8 handelt (Bit = *1* für Länge 9).

Strukturdiagramm



Auswertung und Analyse

Wie gut zu erkennen ist, liefert der Compress II Algorithmus bei großen Files und statischer, vorangegangener Analyse annähernd so gute Ergebnisse wie die statische Huffman Stream Kodierung. Wenn allerdings kleinere Datenmengen mit den gleichen Parametern komprimiert werden, zeigt sich die Überlegenheit der Worst Case Eingrenzung gegenüber dem Huffmancode. Die Kompressionsrate bleibt auch bei den kleinen Testfiles immer über 50%. Trotz dieser sehr guten Kompressionsergebnisse lässt sich der Compress II Algorithmus einfach implementieren und kann auch auf ressourcenschwachen Systemen ausgeführt werden. Damit wurde ein geeigneter Algorithmus für die Komprimierung der Sampledaten gefunden, der die Vorteile der bisher betrachteten Algorithmen kombiniert und so den Anforderungen genügt.

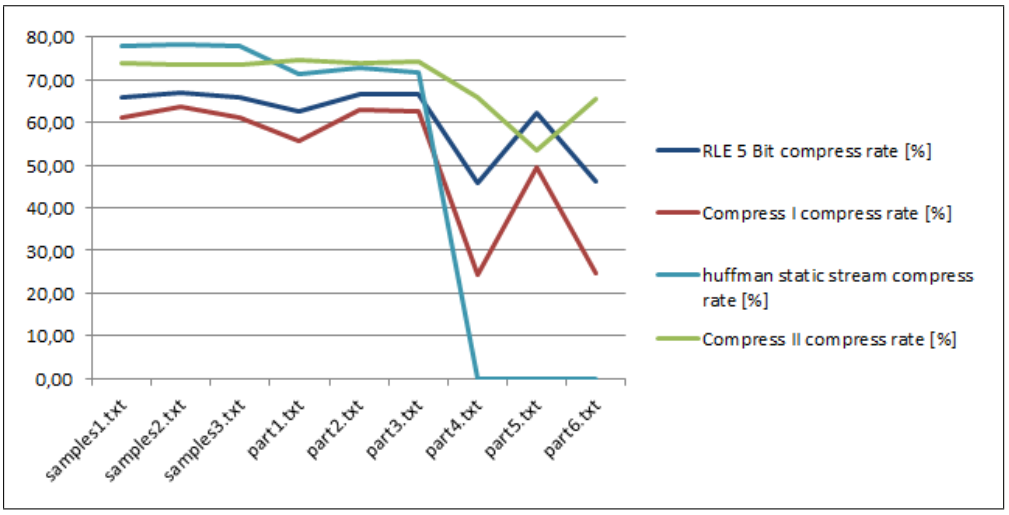


Abbildung 3.9: Vergleich der einzelnen Algorithmen

Laufzeitabschätzung Das zusätzliche *specialPattern* Array benötigt zwar zusätzlichen Speicher und einen etwas komplizierteren Programmfluss, die Laufzeit beträgt aber trotzdem immer noch $\theta(n)$.

KAPITEL 4

Ergebnisse und Ausblick

"Ein Problem sieht, wenn es einmal gelöst ist, immer einfach aus. Der große Sieg, der heute leicht errungen scheint, ist das Ergebnis einer Reihe kleiner, unbemerkter Siege."

Paulo Coelho

Dieses Kapitel befasst sich mit den Resultaten des Bachelorprojektes. Die Ergebnisse werden präsentiert, bewertet und etwaige Verbesserungsmöglichkeiten angegeben.

4.1 Ergebnisse

Die grundlegende Anforderung an das Bachelorprojekt war die Möglichkeit der Sampledatenübertragung des Originalsignales über UDP. Aufgrund der hohen Datenrate wird dazu ein Kompressionsalgorithmus benötigt. Für die Evaluierung des Projektes ist also die Betrachtung der Kompressionsrate und der damit verbundenen Netzwerkauslastung ausschlaggebend.

4.1.1 Beschaffenheit der Daten

Eine Analyse der empfangenen Daten zeigte eine sehr gute Übereinstimmungen mit den bereits bekannten Testdaten aus den Logic Analyzer Aufzeichnungen. Die Optimierungen des Kompressionsverfahrens und der entsprechenden Parameter konnten daher große Wirkung zeigen und führten zu sehr guten Kompressionsergebnissen.

patCnt	1					
specialPattern[0]	10					
specialPattern[1]	11					
specialPattern[2]	8					
specialPattern[3]	7					
compBits	7	5	5	6	7	6
minlen	7	6	5	5	6	6
Kompressionsrate	**69**%	**70**%	**71**%	**72**%	**73**%	**74**%

Tabelle 4.1: Werte der Kompressionsparamter und entsprechende Kompressionsrate

Kompressionsrate

Die durchschnittliche Kompressionsrate im Normalbetrieb beträgt nach Messungen circa 74%. Tabelle 4.1 zeigt verschiedene Parametereinstellungen und die entsprechenden Kompressionsraten, welche im Test gemessen wurden.
Dieses Ergebnis liegt über dem erhofften Grenzwert von in etwa 50% und war somit sehr zufriedenstellend. Durch den parametrierbaren Aufbau des Algorithmus kann die Kompressionsrate auch abgeändert werden, wenn zum Beispiel aus Timinggründen die *specialPattern* nicht möglich sind. So ist es zum Beispiel möglich, keine *specialPattern*, $compBits = 7$ und $minLen = 7$ zu wählen. Diese Einstellung würde bedeuten, dass alle Sequenzen, egal ob komprimiert oder unkomprimiert, genau acht Bit, also ein Byte, lang sind, weshalb beim Dekomprimieren byteweise eingelesen werden kann. Trotz diesbezüglicher Maßnahmen wurde allerdings kein schlechterer Wert als die vorgegebenen 50% erreicht.

4.1.2 Netzwerkauslastung

Laut Messungen mit einem Netzwerk Sniffer werden UDP Frames maximaler Länge durchschnittlich im Abstand von $465\mu s$ empfangen. Dies entspricht einer Datenrate von circa $1452 * (1/0,000465) = 3122580 Byte/s$, was wiederum bei einer Kompressionsrate von 74% eine Samplerate von $3122580 * 8 * (100/26) = 96.097.384 Samples/s$ ergibt. Es ist sofort erkennbar, dass dieses Ergebnis annähernd der Abtastrate der Hardware entspricht und somit zeigt, dass Kompressionsrate und Netzwerkauslastung, ohne den Verlust von Sampledaten, eingehalten werden, da beim Empfangen am PC die gleiche Datenrate wie beim Abtasten anfällt. Die geringe Differenz liegt an der etwas ungenauen Messung der Frame-Zeitabstände. Hierbei wirken sich bereits geringe Unterschiede im μs Bereich sehr stark auf das Endergebnis der Berechnung aus.

Hinweis

Trotz der äußerst positiv ausfallenden Analysen der Ergebnisse muss stets vor Augen gehalten werden, dass diese Kompressionsraten auf Annahmen bezüglich der Häufigkeiten der einzelnen Folgen basieren. Demnach könnte eine völlig falsche Einschätzung diesbezüglich und eine extrem ungünstige Beschaffenheit der Daten die Kompressionsrate erheblich verschlechtern und eventuell sogar einen Betrieb des Scopes unmöglich machen. Daher wird in weiterer Folge, nach der Fertigstellung eines ersten lauffähigen Prototyps, eine Erweiterung des Projekts in Erwägung gezogen. Diese sollte es ermöglichen, sich ständig die momentane Kompressionsrate und die dazugehörigen Parameter anzeigen zu lassen und dadurch die Möglichkeit bereitstellen, die Parameter zur Laufzeit zu verändern, um sich an die richtige Einstellung herantasten zu können. Auch eine eventuelle Anzeige der derzeit häufigsten Folgen, zur Unterstützung bei der Wahl der richtigen Parameter, wäre denkbar.

4.2 Ausblick

Mit dem Senden der UDP Frames ist dieses Projekt noch nicht abgeschlossen. Die Daten müssen anschließend auf dem PC empfangen, dekomprimiert und daraufhin dekodiert werden. Im Anschluss an die Dekodierung stehen die originalen X2X-Framedaten mit Informationen zu etwaigen CRC Fehlern zur Verfügung. Diese Informationen müssen hierauf ausgewertet und in eine aufschlussreiche Darstellung gebracht werden. Weiteres müssen auch Tools zur Auswertung der Samples entwickelt werden. Konkret geht es dabei um eine Art Oszilloskop-Tool, welches die ungefähre Darstellung des Originalsignals auf dem PC ermöglichen soll. Dabei sollte es auch möglich sein, Funktionalitäten wie Zoom, Zeitmessung, Triggerpunkte etc. benutzen zu können. Da dies, zusammen mit der bisherigen Arbeit, allerdings den Umfang eines Bachelorprojekts überschreiten würde, wurde auf die Thematik nur kurz eingegangen. Details können in der Ausarbeitung der Projektarbeit in Teil II nachgelesen werden.

4.3 Resumé

Diese Bachelorarbeit beschreibt den Entwurf eines X2X Bus Analyzers, genannt X2X Scope, welcher die Möglichkeit der Oszilloskop Aufzeichnung auf dem PC imitiert. Die beiden ersten Komponenten des späteren Tools, die Abtasthardware mit Kompressionsalgorithmus und die Firmware samt UDP Kommunikation, wurden fertig implementiert. Für den weiteren Verlauf des Projektes wurden Konzepte und Lösungsstrategien entworfen und zum Teil auch bereits getestet.
Vor allem die Suche nach einem passenden Kompressionsverfahren stellte sich, entgegen aller Erwartungen, als nicht trivial heraus. Es musste ein guter Mittelweg zwischen Kompressionsrate und Laufzeit gefunden werden, da bei einer solch hohen Datenrate die Performance der Hardware bei der Kompression sowie die des PCs bei der Dekompression an ihre Grenzen stoßen. Das Hauptaugenmerk lag deshalb, während des gesamten

Projektverlaufes, auf der effizienten Gestaltung der Software bezüglich des Timings. Darüber hinaus wurde die Parametrierbarkeit des Tools berücksichtigt, damit es in verschiedenen Einsatzgebieten seine Funktion erfüllen kann.
Abschließend kann man sagen, dass die Auswertung der Ergebnisse eine Erfüllung aller im Vorhinein gestellten Anforderungen zeigt und daher die Grundlage für den weiteren Projektverlauf gegeben ist.

Teil II

Projektarbeit

KAPITEL 5

Einleitung zur Projektarbeit

"Eine gute Information bedeutet nicht nur einen Schritt weiter kommen, sondern schon den halben Weg hinter sich haben."

Julian Scharnau

Dieses Kapitel liefert einen Einstieg in die Projektarbeit, aufbauend auf dem Projektstand nach Abschluss der Bachelorarbeit.

5.1 Ausgangssituation

Nach dem erfolgreichen Aufzeichnen, Komprimieren und Senden der X2X Bus Signaldaten müssen diese auf dem PC interpretiert werden. Grundsätzlich geht es dabei darum, die empfangenen UDP Pakete zu entpacken und den Datenstrom zu dekomprimieren. Dadurch entsteht eine Folge von 0 und 1 Samples, welche den Abtastwerten der achtfachen Abtastung entsprechen. Diese Werte können einerseits zur Interpretation des Signals in einer Art Oszilloskop-Darstellung verwendet werden, andererseits müssen aus diesen Samples auch die ursprünglichen X2X Daten zurückgewonnen werden. Es ist äußerst wichtig, dass bei der Rückgewinnung der Daten aus den Samples die gleiche Vorgehensweise angewendet wird, wie auch in den tatsächlichen IO Modulen. Nur so kann gewährleistet werden, dass etwaige Signalverzerrungen zu den gleichen Fehlern in der Auswertung führen, zu denen sie auch im realen Einsatz führen würden. Zu diesem Zweck wurde das X2X Protokoll genauestens analysiert und Schritt für Schritt in Software simuliert. Die so entstandenen dekodierten Daten müssen in einem vorgegebenen Format abgespeichert werden, sodass sie wiederum von den bestehenden Tools verarbeitet werden können.

5.2 Toolchain

Die bereits bestehende Toolchain wird lediglich um das neue Tool ergänzt, es ergeben sich ansonsten keinerlei Änderungen.

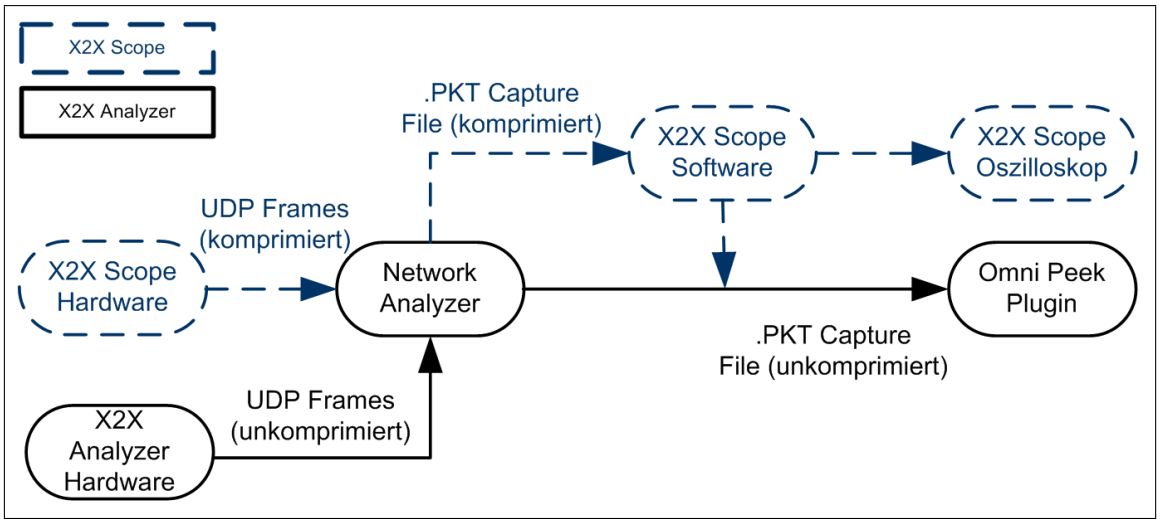


Abbildung 5.1: Erweiterte Toolchain des X2X Scope zur X2X Analyzer Toolchain

5.3 Realisierung

Das gesamte Software Projekt wurde in C# realisiert, welches, aufgeteilt in drei parallele Threads, die Daten entpackt, dekomprimiert, dekodiert und graphisch aufbereitet. Grundlegend ist das Programm dabei aus drei Teilen aufgebaut:

- Dekomprimieren der Daten
- Dekodieren der Daten
- Darstellen der Daten

Vor allem auf die Darstellung der Daten wurde ein besonderes Augenmerk gelegt, da diese Funktion den entscheidenden Vorteil gegenüber dem alten X2X Analyzer bringt. Es wurde deshalb viel daran gearbeitet, die vorhandenen Informationen so gut und vollständig wie möglich dem Anwender zur Verfügung zu stellen, um den maximalen Nutzen aus dem Tool zu erhalten.

5.4 Ergebnis

Im Rahmen dieser Projektarbeit konnten alle drei Teilziele, aufbauend auf den Arbeiten während der Bachelorarbeit, erfüllt werden. Der Dekompressionsalgorithmus konnte nahezu unverändert aus dem, bereits in der Bachelorarbeit erstelltem C Testprogramm, übernommen werden und stellte somit keine weiteren Probleme dar. Vor allem aber die Datendekodierung und im Besonderen die fehlerfreie, parallele Ausführung bereiteten den größten Implementierungsaufwand. Für ersteres musste das X2X Protokoll genauestens analysiert und softwaremäßig simuliert werden. Auch die Parallelität der einzelnen Programmkonstrukte führte immer wieder zu Schwierigkeiten, welche letztendlich aber alle beseitigt werden konnten. Auch die Darstellung der Daten, in Form eines Oszilloskopes, gestaltete sich als schwieriger als erwartet, doch auch diese Hürde wurde am Ende gemeistert. Das X2X Scope stellt letztendlich ein voll funktionstüchtiges Tool dar, welches ab sofort im Feld zu Debug- und Analysezwecken eingesetzt werden kann.

KAPITEL 6

Dekomprimieren der Daten

"Wenn etwas kleiner ist als das Größte, so ist es darum noch lange nicht unbedeutend."

Lucius Annaeus Seneca

6.1 Einlesen der Daten

Bevor die Daten dekomprimiert werden können, müssen diese aus den einzelnen Capture Files gelesen werden.

6.1.1 Beschaffenheit der Daten

Um die notwendige Software zum Empfangen und Lesen der Daten erstellen zu können, muss über die genaue Beschaffenheit der ankommenden Daten Kenntnis gewonnen werden. Für diesen Zweck wurde das Netzwerk Analyse Programm "Wireshark" eingesetzt. Mit dessen Hilfe konnten

1. die von der Hardware gesendeten Daten kontrolliert und
2. das Timing der ankommenden Pakete genau bestimmt werden.

Wireshark

Wireshark ist ein sogenanntes Netzwerk Sniffer Programm, welches den gesamten, auf dem Netzwerk befindlichen, Datenverkehr aufzeichnet. Darüber hinaus stellt es verschiedene Filteroptionen bezüglich Paketdaten, wie zum Beispiel einen Filter nach Source-IP, Protokoll, etc. zur Verfügung. Die Daten der Pakete werden in die einzelnen Header aufgespalten, was eine übersichtliche Darstellung ermöglicht. Mithilfe dieses Tools wurden

die von der Hardware gesendeten Daten evaluiert und anfänglich zu Debugging Zwecken ausgewertet. Zusätzlich gibt es bereits ein Omni Peek Plugin[1], welches, wie bereits in Kapitel 2 erwähnt, die X2X Rohdaten übersichtlich darstellt.

6.1.2 Online Auswertung

Dieser Ansatz beschreibt die Verarbeitung der UDP Frames zur Laufzeit. Als erstes müssen die UDP Frames am PC empfangen werden. Dazu wurde ein C# Projekt entwickelt, welches, von Interrupts getriggert, die Frames in einen Ringbuffer speichert. Zum Empfangen wird ein eigener Thread verwendet, welcher gleichzeitig alle Kompressionsparameter aus dem Header liest und dem weiteren Programmverlauf zur Verfügung stellt. Demnach ist es möglich, dass sich die Kompressionsparameter zur Laufzeit ändern, da die neuen Einstellungen beim Empfang des ersten Frames sofort übernommen werden. Zusätzlich wird eine Lesefunktion bereitgestellt, welche es ermöglicht, Sample für Sample Daten aus dem Byte-Array zu lesen, dies ist für den Dekompressionsalgorithmus vonnöten. Da die Daten bitweise gelesen werden müssen, ist der Rechenaufwand für den auswertenden PC enorm. Im Laufe der Projektentwicklung stellte sich heraus, dass alleine der Dekompressionsvorgang zur Laufzeit eine zu große Belastung für den PC darstellt. Deswegen wurde ein Ansatz gewählt, bei dem die Daten nicht online, sondern erst zu einem späteren Zeitpunkt dekomprimiert werden.

6.1.3 Offline Auswertung

Dieser Ansatz benutzt ein bereits vorhandenes Tool, den sogenannten Network Analyzer, zum Empfangen der Frames. Jene werden in sogenannten Capture Files in einer Ringbufferstruktur abgelegt. Die Daten in den Frames sind zu diesem Zeitpunkt allerdings noch komprimiert. Es ist daher nicht möglich, die Aufzeichnung mit Triggern zu steuern, welche sich auf die X2X Daten beziehen. Es können lediglich, wie beim alten Analyzer, vier Digitale Eingänge, welche extra im Header angeführt sind, zum Triggern verwendet werden. Dadurch entsteht also keineswegs ein Nachteil gegenüber dem Vorgänger Tool. Dieser Ansatz bietet allerdings folgende Vorteile:

- Geringere Timinganforderungen
  Daraus ergibt sich die Möglichkeit mächtigere graphische Darstellungs- und Auswertungs-Features zu implementieren, da diese nicht mehr zur Aufzeichnungszeit durchgeführt werden müssen.
- Capture Files
  Diese können einfach gespeichert, gesendet und zu einem späteren Zeitpunkt auf einem anderen PC ausgewertet werden.
- Toolchain
  Die neue Toolchain lässt sich weitestgehend ohne Aufwand in die bereits vorhandene integrieren (siehe Abb. 5.1).

[1] Bei Omni Peek handelt es sich ebenfalls um einen Netzwerk Sniffer

6.1.4 Thread Konzept

Um die Performance des Programmes zu steigern, wurde ein Multithreading Ansatz gewählt, wobei wichtige Programmteile parallel abgearbeitet werden können. Es wird darauf hingewiesen, dass in den folgenden Erläuterungen auf das Thread Handling unter Windows und im speziellen C# eingegangen wird. Die spezifischen Implementierungen variieren zwischen verschiedenen Betriebssystemen.

Allgemein

Grundsätzlich benutzt ein Betriebssystem verschiedene Prozesse, um verschiedene Aufgaben zu erledigen. Dabei beansprucht ein Prozess immer die gesamte CPU. Wenn in einem Multitasking System mehrere Prozesse gleichzeitig ausgeführt werden müssen, so bekommen die einzelnen Prozesse abwechselnd die CPU zugewiesen (siehe Abb. 6.1).

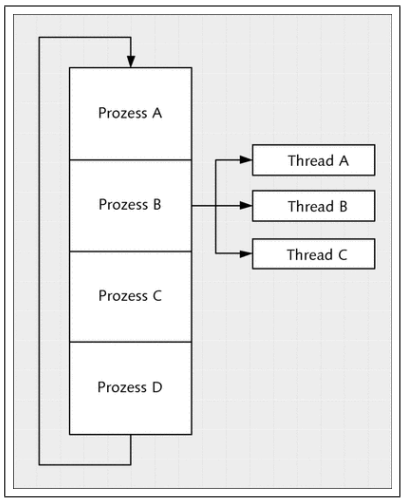


Abbildung 6.1: Sequentielle Abarbeitung mehrerer Prozesse [Wol09]

Ein einzelner Prozess ist wie folgt aufgebaut:

- Codesegment mit dem auszuführenden Programmcode.
- Datensegment mit Daten, welche zur Ausführung des Codes nötig sind.
- Stacksegment mit dem zur momentanen Abarbeitung gehörenden Stack.
- Dateideskriptoren mit eventuell geöffneten Dateien.

Beim Wechsel zwischen zwei Prozessen muss der gesamte, sogenannte Kontext, gewechselt werden, was einen erheblichen Aufwand für die CPU darstellt. In einem Multithreading System ist es deshalb möglich, innerhalb eines Prozesses mehrere Threads parallel laufen zu lassen. Dabei können alle Threads innerhalb eines Prozesses auf dasselbe Datensegment und Codesegment, sowie auf dieselben geöffneten Dateien zugreifen.

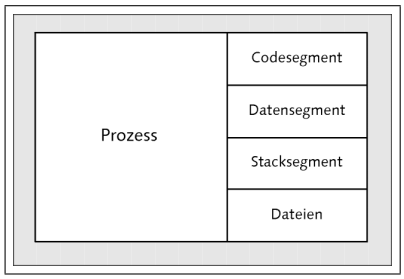


Abbildung 6.2: Aufbau eines Prozesses [Wol09]

Die einzelnen Threads teilen sich also einen gemeinsamen Adressraum, lediglich Stack und Befehlszähler werden für jeden Thread extra verwaltet. Auf diese Weise kann derselbe Programmcode in mehrere Threads aufgeteilt und parallel abgearbeitet werden (siehe Abb. 6.3). Der Zugriff auf gemeinsam benutzte Daten wird dabei nicht mehr durch die vom Betriebssystem bereitgestellte Speicherverwaltung geregelt, weshalb explizit Synchronisationsmechanismen eingeführt werden müssen, um den reibungslosen Ablauf paralleler Threads und die Datenkonsistenz gewährleisten zu können.

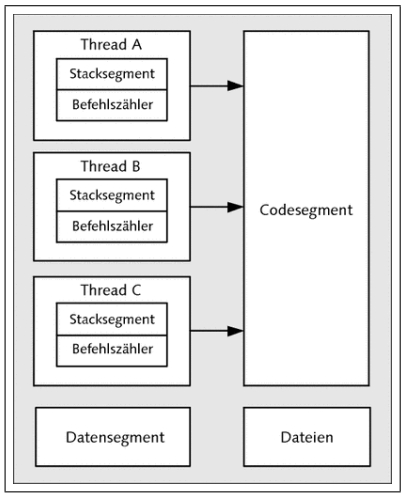


Abbildung 6.3: Aufbau eines Multithreading Systems [Wol09]

Implementierung in C#

Im .NET Framework werden Betriebssysteme noch weiter in sogenannte *application domains* unterteilt. Ein oder mehrere Threads können in einer oder mehreren dieser Domains ausgeführt werden. So ist es möglich, dass ein einzelner Thread zwischen verschiedenen *application domains* innerhalb eines Prozesses wechselt. Es können daher auch einzelne Teile eines Programmes, wie zum Beispiel for-Schleifen oder ähnliches, parallelisiert werden. Im Falle dieser Projektarbeit macht eine dahingehende Parallelisierung

allerdings keinen Sinn, da im Beispiel der Datendekompression das Ergebnis des *i+1-ten* Schleifendurchganges vom Resultat des *i-ten* abhängt. Selbiges gilt für die Datendekodierung. Aufgrund dieser Tatsache wurde das Programm in drei grobe Teilaufgaben zerlegt, welche aber dann für sich sequentiell abgearbeitet werden. Eine detaillierte Beschreibung zur Implementierung von Threads in C# kann der Microsoft Developer Network Homepage entnommen werden.[2]

6.1.5 Wichtige Programmpassagen

Im folgenden Abschnitt werden die für das Lesen der Daten relevanten Codepassagen angeführt und kurz erläutert. Abb. 6.4 stellt einen groben Überblick über die X2X Scope Software dar und dient zur besseren Übersicht.

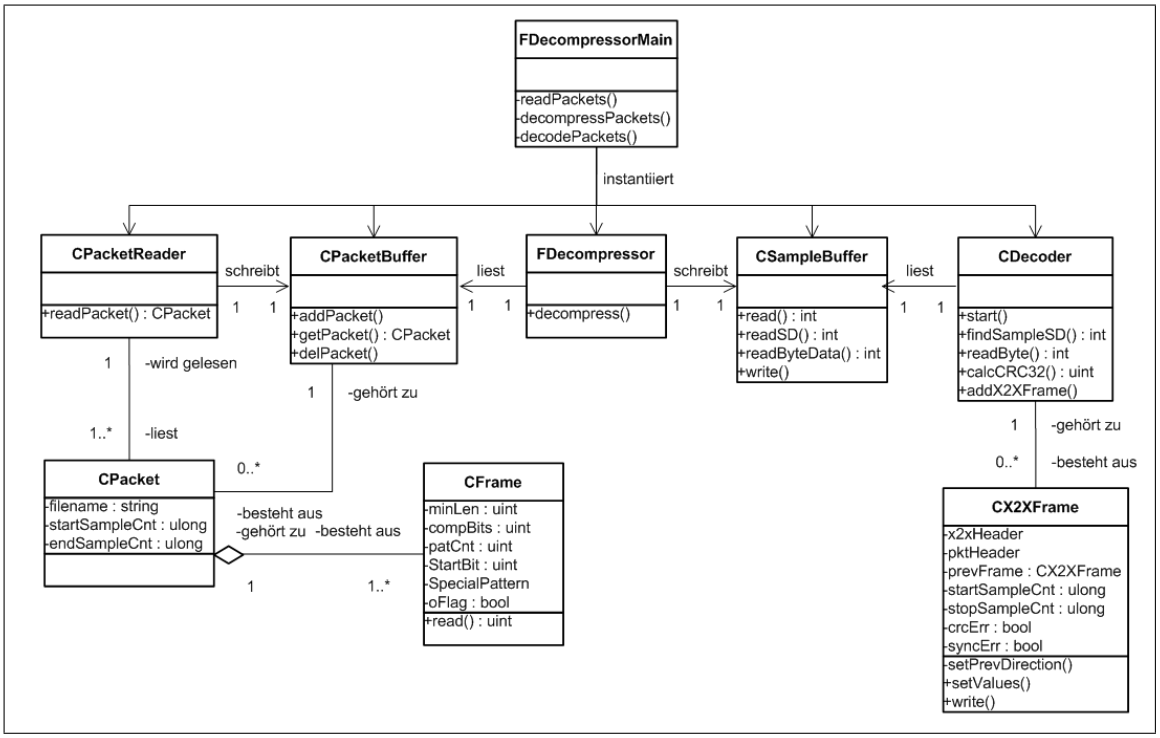


Abbildung 6.4: Klassendiagramm zur X2X Scope Software

FDecompressorMain Klasse

Grundsätzlich wird das Lesen der Daten sowie das Dekomprimieren und Dekodieren von der *FDecompressorMain* Klasse gesteuert, dazu werden drei Threads verwendet. Zuerst wird vom Benutzer der Pfad zu den Capture Files angegeben, danach wird mit

[2] http://msdn.microsoft.com/en-us/library/aa720724

der Ausführung begonnen. Die *FDecompressorMain* Klasse übernimmt dabei das Starten und Beenden der einzelnen Threads, sowie den Datenaustausch dazwischen. Als Parameter im Konstruktor wird die *FMain()* Klasse übergeben, damit kann in der *FDecompressorMain* Klasse nach Beendigung der Dekodierung eine entsprechende Callback Funktion der *FMain()* Klasse aufgerufen werden. Die drei Threads werden mit den folgenden Methoden initialisiert und gestartet:

- ***private void readPackets()***
  Dieser Thread liest mithilfe der *CPacketReader.readPacket()* Methode ein File nach dem anderen und legt die darin befindlichen Daten im *CPacketBuffer*, sowie in einer Liste ab. Sobald das letzte File gelesen wurde wird ein Finish Flag gesetzt.

- ***private void decompressPackets()***
  In diesem Thread werden die einzelnen *CPacket* Objekte der Reihe nach an die *FDecompressor.decompress()* übergeben und somit dekomprimiert. Die dekomprimierten Samples werden im *CSampleBuffer* abgelegt. Sind alle Files verarbeitet, wird der Thread wieder beendet.

- ***private void decodePackets()***
  Mithilfe der *CDecoder.start()* Methode werden alle Samples im *CSampleBuffer* dekodiert und das Ergebnis zur weiteren Darstellung bereitgestellt.

CPacketReader Klasse

Diese Klasse wird im *FDecompressorMain.reackPackets()* Thread instantiiert und ermöglicht das Lesen der einzelnen Frames aus einem Verzeichnis mit mehreren Capture Files (PKT Files). In einem Capture File befinden sich, je nach Konfiguration des Network Analyzers, mehrere tausend UDP Frames. Um diese zu lesen, werden folgende Programmkonstrukte verwendet:

- ***public CPacket readPacket(String filename)***
  Diese Methode liest aus dem, als Parameter übergebenen File, alle Frames und speichert diese als *CFrame* Objekte innerhalb eines *CPacket* Objektes ab. Zusätzlich werden PKT File Header und Ethernet Header gelesen und global gespeichert.

- ***CFrame Klasse***
  Ein *CFrame* Objekt entspricht genau einem empfangenen UDP Frame. Dazu werden die Kompressionsparameter, Sample Counter sowie die eigentlichen, komprimierten Daten gespeichert. Zusätzlich werden einige Methoden bereitgestellt, um bequem auf die Parameter, sowie auf einzelne Samples in den Daten zugreifen zu können.

- ***CPacket Klasse***
  Ein *CPacket* Objekt enspricht genau einem Capture File (PKT File). Es besteht aus einem Start- und Stop Sample Counter, einem Filenamen und einer Liste von

CFrame Objekten, welche als UDP Frames im dazugehörigen File abgespeichert sind. Die Start- und Stop Sample Counter werden zur späteren Analyse benötigt, um feststellen zu können, welche Frames sich in welchen Files befinden.

- ***CPacketBuffer Klasse***
  Dieser Bufferspeicher entspricht dem gesamten Verzeichnis, mit allen sich darin befindlichen Files. Dank dieser Datenstruktur kann sehr einfach, gezielt auf die Daten eines bestimmten Frames zugegriffen werden. Damit stehen die Daten in bequemer Form für die weitere Verarbeitung zur Verfügung.

6.2 Datendekompression

Für die Dekompression der Daten wird ebenfalls ein eigener Thread verwendet. Dieser liest mithilfe der bereitgestellten Lesefunktion *CFrame.read()* die komprimierten Sample Daten und schreibt die dekomprimierten Sequenzen in einen zweiten Ringbuffer (*CSampleBuffer*). Dieser Ringbuffer stellt hierbei wieder Lese- und Schreibfunktionen zur Verfügung, welche den Zugriff auf Sample-Ebene zulassen. Das Ende der Dekompression eines Frames kann anhand des Sample Counters, welcher mitgeschickt wird, erkannt werden. Dieser gibt an, wie viele komprimierte beziehungsweise unkomprimierte Sequenzen sich im empfangenen Frame befinden. Sobald ein UDP Frame vollständig dekomprimiert wurde, wird der nächste aus dem *CPacket* Objekt gelesen.

6.2.1 Algorithmus

Der Algorithmus zur Datendekompression ist relativ einfach zu implementieren.

- Zuerst wird das erste Bit gelesen, ist dieses *0*, handelt es sich um eine unkomprimierte Folge und es werden die nächsten *minLen* Bit gelesen und in den Buffer geschrieben. Ist das gelesene Bit *1*, so muss das nächste Bit gelesen werden.
- Ist das nächste Bit wiederum 1, so handelt es sich um ein *specialPattern*. Es müssen also die nächsten *patCnt* Bits gelesen werden. Danach kann im *specialPattern* Array die entsprechende Folgenlänge gesucht und in den Buffer geschrieben werden. Ist das zweite Bit *0*, so handelt es sich um eine normale komprimierte Folge, somit müssen die nächsten *compBits* Bits gelesen und als Länge interpretiert werden.
- Ist die so ermittelte Länge gleich der maximalen Länge, so muss das Bit für die folgende Länge auf das letzte Bit des Längencodes gesetzt werden. Danach kann die ermittelte Folgenlänge in den Buffer geschrieben werden.

6.2.2 Wichtige Programmpassagen

Auch die Datendekompression wird in der *DecompressorMain* Klasse gestartet, zuständig ist hierfür der *decompressPackets()* Thread. Dieser dekomprimiert mithilfe der *FDecompressor.decompress()* Methode ein *CPacket* Objekt nach dem anderen (siehe Abb. 6.4).

FDecompressor Klasse

Diese Klasse ermöglicht das Dekomprimieren eines *CPacket* Objektes. Zusätzlich wird dabei der Stop Sample Counter gesetzt und es wird eine Fortschrittsanzeige aktualisiert.

- ***public void decompress(CPacket packet, CSampleBuffer sampleBuffer)***
  Die eigentliche Dekompression erfolgt durch den Aufruf dieser Methode. Daraufhin wird für jedes *CFrame* Objekt der Algorithmus durchgeführt, wobei zuerst die dafür notwendigen Parameter ausgelesen werden. Die dekomprimierten Samples werden in einer eigenen Ringbuffer Struktur, dem *CSampleBuffer* gespeichert.

CSampleBuffer Klasse

Diese Klasse speichert den dekomprimieren Sample Stream und ermöglicht das bequeme Schreiben und Lesen einzelner Samples. Sie dient somit zur Entkopplung zwischen Dekompressions- und Dekodierungsvorgang.

- ***public void write(int bits, int cnt, bool opt)***
  Diese Methode ermöglicht es dem Dekompressionsthread, dekomprimierte Samples in den Buffer zu schreiben. Dabei gibt es zwei Möglichkeiten:

  1. *opt = true*: Diese Option bedeutet, dass *cnt* mal der Wert von *bits & 1* geschrieben wird. Da beim Dekomprimieren sehr oft lange 1 beziehungsweise 0 Folgen auftreten, wurde diese Methode erstellt, um elegant eine bestimmt Anzahl an 1 oder 0 schreiben zu können.
  2. *opt = false*: In diesem Fall werden die niederwertigen *cnt* Bits von *bits* geschrieben. Dies wird benötigt, um zum Beispiel ein unkomprimiertes Bitmuster auszugeben.

- ***public int read(uint cnt)***
  Mithilfe von *read()* kann der Dekodierungsthread eine beliebige Anzahl an Samples vom Datenstrom lesen. Der Parameter *cnt* gibt dabei an, das wievielte Sample gelesen werden soll. Den Hintergrund dazu liefert die Implementierung des X2X Protokolls, bei welchem des öfteren nur das zweite und vierte Sample benötigt werden. Dabei wurde bereits berücksichtigt, dass die Abtastrate, im Vergleich zum originalen X2X, 8-fach durchgeführt wurde. Wenn demnach *read()* mit *cnt = 4* aufgerufen wird, so bedeutet das, dass das vierte Sample in der ursprünglichen Abtastrate benötigt wird. In diesem Fall wird um 7 Samples weiter gelesen und anschließend das achte Sample retourniert.

- ***public int readSD(uint cnt)***
  Bei dieser Methode handelt es sich im Prinzip um die gleiche Vorgehensweise wie bei *read()*, mit dem Unterschied, dass nicht nur jedes zweite Sample betrachtet wird. Dies war erforderlich, um die korrekte Suche nach dem Startdelimiter bei der Datendekodierung zu ermöglichen. Die normale *read()* Methode ist in diesem Fall zu ungenau.

- ***public int readDataByte(int prevSyncBit, CX2XFrame x2xFrame)***
  Hiermit wird das Lesen eines vollständigen Datenbytes aus den dekomprimierten Samples ermöglicht. Die Methode hält sich dabei an das vorgegebene X2X Protokoll und führt während des Lesens gleichzeitig eine *Syncbit* Kontrolle durch. Da im X2X Protokoll immer genau ein Byte mit einem *Syncbit* abgeschlossen wird und die einzelnen Bits immer gleich kodiert sind, bietet diese Methode eine elegante Möglichkeit, ein ganzes Datenbyte zu lesen und somit den Code in der Hauptschleife des Dekompressionsalgorithmuses zu vereinfachen. Darüber hinaus werden eventuelle *Syncbit* Fehler aufgezeichnet und im *CX2XFrame* Objekt zur späteren Darstellung abgelegt.

KAPITEL 7

Dekodierung der Daten

"Gute Informationen sind schwer zu bekommen. Noch schwerer ist es, mit ihnen etwas anzufangen."

Sir Arthur Conan Doyle

Der Thread zur Dekodierung der Daten ist über einen Ringbuffer von der Dekompression entkoppelt, sodass beide Algorithmen gleichzeitig durchgeführt werden und parallel dazu die Daten aus den Files gelesen werden können. Bei der Dekodierung geht es darum, aus den Sample Sequenzen wieder die ursprünglichen X2X Daten zurückgewinnen zu können. Es ist wichtig, dass dabei genauso vorgegangen wird, wie es auch ein normales Hardware Modul macht, damit im Fehlerfall auch die gleichen Fehler reproduziert werden können. Anschließend können die fertigen X2X Daten in ein File gespeichert werden, welches schlussendlich von bereits vorhandenen Tools analysiert werden kann.

7.1 X2X Protokoll

In Kaptitel 2 wurde bereits auf die Implementierung des X2X-Links bezüglich Topologie, Frameaufbau und Fehlererkennung eingegangen. Für die Dekodierung der Daten ist es wichtig, das X2X Protokoll beziehungsweise die genaue Darstellung der Daten zu analysieren.

7.1.1 Datenkodierung

Wie bereits beschrieben, wird das X2X Signal von den einzelnen Stationen 4-fach abgetastet. Der Frame beginnt mit dem sogenannten Startdelimiter, und endet mit dem Enddelimiter. Beide entsprechen genau 10 *0* Bits. Dazwischen wird am Bus ein Idle Signal übertragen, das mittels alternierender *0* und *1* Folgen dargestellt wird.

Starterkennung

Um mit dem Auswerten des Frames starten zu können, muss zuerst dessen Anfang erkannt werden. Die Bedingung lautet hierfür wie folgt:

- Zu Beginn müssen viermal abwechselnd vier *0* und vier bis sechs *1* Sample gelesen werden. Das entspricht dem letzten Teil des Idle-Signals. Beim Lesen wird allerdings immer nur das erste und letzte Sample gelesen. So würde beispielsweise die Folge *0110* als vier *0* Samples interpretiert. Die originale Abtasthardware in den Modulen würde allerdings genauso reagieren.
- Nach den vier *01* Sequenzen folgt der Startdelimiter mit insgesamt $10 * 4 = 40$ *0* Samples. Da sich bei der Starterkennung immer wieder Probleme zeigten, entschloss man sich nicht nur jedes zweite Sample, sondern jedes einzelne Sample zu betrachten (siehe *CSampleBuffer.readSD()*).

Das Suchen der Startbedingung wurde im Programmcode in der Funktion *findSD()* implementiert.

Datenauswertung

Nach den 10 *0* des Startdelimiter folgt ein sogenanntes *Syncbit*. Dieses ist vier bis sechs Samples lang und immer invertiert zum darauffolgenden Datenbit. Es wird zu Synchronisationszwecken verwendet. Nach dem *Syncbit* folgen acht Datenbits, welche jeweils vier Samples, beziehungsweise bei achtfach Oversampling, 8 Sample lang sind. Es wird dabei zuerst das dritte Sample des ersten Datenbits gelesen und darauf acht mal um vier (acht) Samples weiter geschoben und der Wert ausgelesen. Bei der $8 * 4 = 32$ Sample langen Datenfolge bestimmen also nur die markierten Samples den Wert der Bits.

- *11**1**111**1**100**0**011**1**111**1**100**0**000**0**011**1**1*

Das gelesene Datenbyte lautet somit: *11011001*. Dieser Teil der Dekodierung wurde in der Methode CDecoder.readByte() beziehungsweise in der CSampleBuffer.readDataByte() Methode implementiert. Hier musste außerdem auch eine Fehlererkennung bezüglich Korrektheit (Länge und Wert) des *Syncbit* vorgenommen werden, da diese im Normalbetrieb auch an dieser Stelle durchgeführt wird. Dabei wird überprüft ob das *SyncBit* invertiert zum ersten Datenbit des nächsten Bytes ist.

7.1.2 Cyclic Redundancy Check CRC

Im Anschluss an die Daten folgt eine 32 Bit Checksumme, welche berechnet und verglichen werden muss.

> *Mit der CRC-Technik (Cyclic Redundancy Check), auch zyklischer Redundanz-Prüfsummentest genannt, können mit relativ wenig Aufwand Mehrfach- und Burst-Fehler (mehrere Fehler kurz hintereinander) in Datenblöcken mit sehr*

> *großer Sicherheit entdeckt werden. Diese Technik wird heute sehr verbreitet in der Datenspeicherung (Disk) und der Datenübertragung (Computernetzwerke) angewendet.* [MKGK09]

Um für Daten die entsprechende Checksumme zu erhalten, muss zuerst ein sogenanntes Generatorpolynom festgelegt werden.

Berechnung

Das Generatorpolynom wird im Vorhinein festgelegt, dabei wird ein sogenanntes dualydisches Polynom verwendet. Dabei handelt es sich um ein, zu einer Binärzahl gehöriges Polynom, welches wiederum nur zwei Werte (*1* oder *0*) annehmen kann. Das Polynom zu Bitfolge *10010110* lautet daher: $1*x^7+0*x^6+0*x^5+1*x^4+0*x^3+1*x^2+1*x^1+0*x^0 = 1*x^7+x^4+x^2+x^1$. Die zu übertragenden Daten werden als Bitfolge dargestellt und durch das Generatorpolynom modulo 2 dividiert. Der so entstandene Rest wird an die Daten angehängt.

Überprüfung

Beim Empfang der Daten werden diese samt angehängtem CRC Wert, wiederum durch das Generatorpolynom dividiert. Bleibt bei dieser Division kein Rest, so ist bei der Übertragung entweder kein Fehler aufgetreten, oder der Fehler führte zu einem Wert, der genau ein Vielfaches des Polynoms darstellt. Dieser Fall ist bei großen Polynomen allerdings äußerst unwahrscheinlich.

Implementierung

Für die CRC Implementierung im X2X Bus wird das standardisierte IEEE CRC-32 [Ins08] Polynom $G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + 1 = 0x104C11DB7$ verwendet. Dieses Polynom wird unter anderem auch bei der Datenübertragung über Ethernet verwendet. Die zu übertragende Bitfolge muss jetzt mit x^{32} multipliziert und anschließend durch $G(x)$ dividiert werden. Außerdem wurden in dieser Implementierung die ersten 32 Bit invertiert, dies entspricht einer gängigen Vorgehensweise. Der genaue Berechnungsalgorithmus wurde aus verschiedenen Implementierungen im Internet kombiniert, nähere Informationen dazu liefert unter anderem die Seite *www.zorc.breitbandkatze.de*, welche unter anderem eine online Berechnung ermöglicht. Grundlegend handelt es sich bei der Implementierung aber um die vorher beschriebene Vorgehensweise. Im Programmcode lautet die entsprechende Methode *calcCRC32()*.

7.2 Wichtige Datenstrukturen

7.2.1 CDecoder Klasse

Die *CDecoder* Klasse übernimmt die dekomprimierten Sampledaten und gewinnt daraus die ursprünglichen X2X Daten, welche in einem formatierten File abgespeichert werden (siehe Abb. 6.4). Zusätzliche werden Informationen zu Position und Anzahl von Frames und Fehlern aufgezeichnet.

Wichtige Methoden

- ***private int findSampleSD(int sample)***
  Diese Methode durchsucht den Stream nach der Startbedingung. Dabei wird permanent nach dem *01010101* Muster und den darauf folgenden zehn *0* geparsed. Wichtig bei der Implementierung ist, dass im Falle von gefundenen 01010101 aber nicht gefundenen Startdelimiter im Stream nur um zwei Stellen weitergegangen werden darf (siehe Algorithmen zum Suchen in Texten [Sed02]).

- ***private int readByte()***
  Dank dieser Methode ist es möglich, mithilfe der von *CSampleBuffer()* zur Verfügung gestellten Methode *public int readDataByte()* ein komplettes Datenbyte samt *Syncbit* zu lesen und zu erkennen ob es sich dabei um wirkliche Daten oder bereits um den Enddelimiter handelt. Dies wird durch 8 *0* Datenbit und einem *0 Syncbit*, welches länger als 6 Sample ist, erkannt. Diese Folge stellt eine, für Daten, nicht mögliche Sequenz dar, welche somit einfach als Enddelimiter erkannt werden kann.

- ***private uint calcCRC32()***
  Um die Korrektheit der Daten überprüfen zu können, muss eine Checksumme berechnet werden. Dazu werden alle dekodierten Datenbits in ein Array gespeichert und am Ende jedes X2X Frames daraus die Checksumme nach vorgegebenen Algorithmus berechnet.

- ***public void start(CSampleBuffer sampleBuffer, String filename)***
  Der gesamte Dekodiervorgang wird mit dieser Methode gestartet. Als Parameter werden der Ringbuffer mit den dekomprimierten Daten und ein File zum schreiben der fertig dekodierten X2X Daten übergeben. Im Programm wird zunächst das Startdelimiter gesucht, danach solange Daten gelesen bis das Enddelimiter gefunden wurde. Im Anschluss werden die Checksumme überprüft, die Daten geschrieben und erneut nach dem Startdelimiter gesucht.

- ***public void addX2XFrame(bool write)***
  Das Omni Peek Plugin wurde zum Lesen von PKT Files erstellt, in welchen die einzelnen UDP Frames gespeichert sind. Pro UDP Frame wurde genau ein X2X Frame gespeichert. Dieser wurde mit einem zusätzlichem Header versehen damit die Frames in einem Netzwerk Sniffer dargestellt werden können. Um ein gültiges

File erstellen zu können, müssen zu Beginn die komprimierten UDP Frames gelesen, dekomprimiert, dekodiert und daraufhin im richtigen Format abgespeichert werden. Die *addX2XFrame()* Methode übernimmt diese Aufgabe. Sie erzeugt aus den dekodierten Daten ein neues *CX2XFrame()* Objekt und fügt es zu einer Liste hinzu. Wurden alle Frames gelesen, werden sie in ein File geschrieben. Hierzu bietet jedes *CX2XFrame()* Objekt eine write Methode, um Daten und Header in richtiger Reihenfolge bequem schreiben zu können.

7.2.2 CX2XFrame Klasse

In dieser Klasse werden sämtliche Informationen zu einem X2X Frame gespeichert. Dazu zählen Headerdaten, X2X Daten, Start- und Stop-Sample-Counter, sowie eventuelle Fehler. Eine weitere wichtige Aufgabe dieser Klasse ist die Bestimmung des Frametypes. Ein X2X Zyklus besteht aus vier verschiedenen Frames: Analog Output (AO), Analog Input (AI), Digital Output (DO) und Digital Input (DI), welche immer in dieser Reihenfolge gesendet werden.

Wichtige Methoden

- ***private void setPrevDirection()***
  Ob es sich bei einem Frame um einen analogen oder digitalen Frame handelt, kann aus dem X2X Header gelesen werden. Daraufhin muss noch bestimmt werden, ob es sich um einen Output- oder Input-Frame handelt. Dazu wird, aufgrund der gesammelten Information, die Richtung des vorherigen Frames gesetzt. Diese Aufgabe erfüllt die Methode mithilfe eines Algorithmus.

- ***public void setValues(byte[] data, List <CX2XFrame> x2x_frames)***
  Diese Methode setzt im Objekt alle Flags, Header, Frametype und Daten auf die entsprechenden Werte. Nach deren Aufruf ist der Frame richtig initialisiert und kann geschrieben werden.

- ***public void write(BinaryWriter bwDecode)***
  Mithilfe dieser Methode wird der Frame in ein File geschrieben. Zuerst werden die benötigten PKT-Tags, gefolgt von der Länge, dem Frame Header, dem X2X Header und den Daten geschrieben. Am Ende werden vier Byte als Ethernet Checksumme angehängt, diese wird allerdings nicht ausgewertet.

KAPITEL 8

Darstellung der Daten

"Willst du dich am Ganzen
erquicken, so musst du das Ganze
im Kleinsten erblicken."

Johann Wolfgang von Goethe

Die Auswertung der Daten ist der essentielle Teil dieses Projektes. Erst nach deren Implementierung kann das X2X Scope im Feld zu Debug Zwecken eingesetzt werden, dazu müssen die X2X Daten interpretiert werden. Diese Daten enthalten Knotennummern, analoge und digitale Ein- und Ausgänge und viele andere Daten, abhängig von den angeschlossenen Modulen.

8.1 Omni Peek Plugin

Die X2X Framedaten werden vom Auswerte-Tool dekodiert und anschließend in einem definierten Format gespeichert, damit sie von einem bereits implementierten Omni Peek Plugin ausgewertet werden können. Dieses Plugin dient zur Fehleranalyse auf Datenebene. Es werden nicht mehr CRC Fehler auf X2X Frameebene wie bei der Dekodierung, sondern eventuelle Fehler in den Daten, wie zum Beispiel falsche Adressen, Werte und ähnliches, erkannt. Aufgrund eigens eingeführter Header werden CRC und *Syncbit* Fehler allerdings trotzdem im Plugin angezeigt. Dies entspricht exakt der gleichen Auswertung, wie die des Vorgänger X2X Anaylzers, wessen Funktionalität somit in den neuen Analyzer integriert wurde.

8.1.1 PKT File Format

Damit die dekodierten Files von Omni Peek geöffnet werden können, müssen sie in einem festgelegten File Format gespeichert werden. Dieses ist wie folgt aufgebaut:

PKT File Aufbau	
PKT Header	Header im xml Format. Beinhaltet Länge des Files sowie Informationen über Übertragungsgeschwindigkeit, Versions ID,... Dieser wird aus einem der ursprünglichen Capture Files kopiert und angepasst.
ETH/IP/UDP Header	Standard Netzwerk Header. Werden ebenfalls kopiert und angepasst.
X2X Analyzer Header	Header des alten X2X Analyzers (siehe Tabelle 8.2).
X2X Daten	Daten
ETH Checksumme	Ethernet Checksumme, wird nicht verwendet aber vom Tool ausgelesen.
Danach folgen wieder ETH/IP/UDP Header, Analyzer Header, Daten und Checksumme des zweiten Frames usw. bis zum letzten Frame.	

Tabelle 8.1: PKT File Aufbau

Frameaufbau		
X2X - Header & Daten		
Status	1 Byte	X2X-Link Status Bit15 ... 0=Out, 1=IN Bit9 .... CRC Fehler Bit8 .... Sync Fehler
Reserve	1 Byte	derzeit nicht verwendet
Paketzähler	1 Byte	laufender Zähler
Triggereingänge	1 Byte	Zustand der lokalen Triggereingänge
Startzeit	1 Long	Startzeit des Frames am X2X-Link
Endzeit	1 Long	Endzeit des Frames am X2X-Link
Daten	8192 Byte	X2X-Link Frame

Tabelle 8.2: X2X Analyzer Frameaufbaus

8.2 X2X Scope - Oszilloskop

Diese Funktion zeigt die ganze Stärke des neuen X2X Scopes gegenüber dem alten Analyzer Tools.

8.2.1 Dekodierung

Zuerst muss im Tool das Verzeichnis, mit den sich darin befindenden Files, ausgewählt und die Dekodierung der Daten gestartet werden. Dabei werden die einzelnen Files gelesen, dekomprimiert und dekodiert. Zusätzlich werden Informationen über Position und Anzahl eventuell auftretender Fehler, sowie der einzelnen Frames gesammelt. Diese Informationen können im Anschluss im Oszilloskop grafisch dargestellt werden. Die Dekodierung wird durch Klicken auf den Reiter *File* $->$ *Decode* gestartet.

8.2.2 Oszilloskop

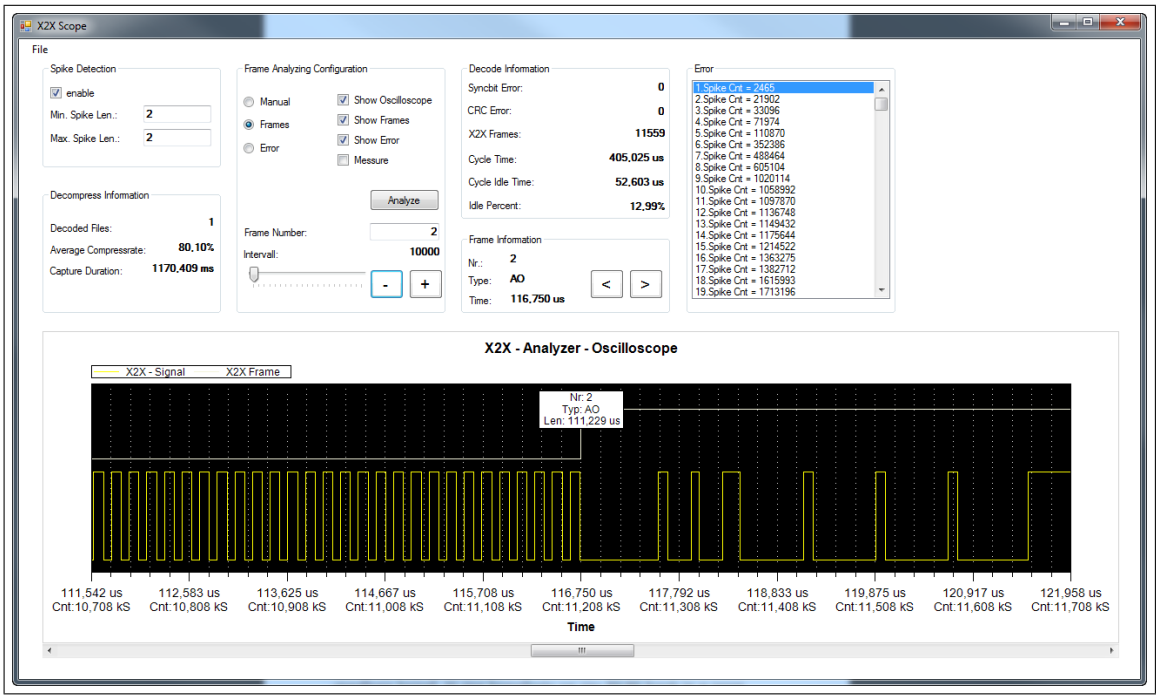


Abbildung 8.1: Grafische Benutzeroberfläche der X2X Scope Software

Nach der Dekodierung kann das X2X Signal grafisch dargestellt werden. Signal, Frames und Fehler können separat ein- und ausgeblendet werden. Weiteres besteht die Möglichkeit, zwischen den einzelnen Frames beziehungsweise eventuellen Fehlern zu wechseln. Die Darstellung erfolgt dabei immer auf Basis einer vorgegebenen Position und einem dazu einstellbaren Intervall. Ab einer Intervallbreite von etwa 200.000 Sample ist es ratsam, das Signal auszublenden, da sonst ein sehr hoher Rechenaufwand

entsteht, welcher das Programm verlangsamt. Neben der Möglichkeit zwischen Frames und Fehlern zu wechseln kann auch manuell eine Position und ein dazugehöriges Intervall angegeben werden, um bestimmte Ausschnitte im Signal genauer betrachten zu können.

Spike Detektion

Ein weiteres Feature ist die Detektion von Spikes. Dazu müssen vor der Dekodierung die minimale und maximale Länge der zu suchenden Spikes angegeben werden. Diese werden nach der Dekodierung dargestellt und in der Fehler Liste abgelegt. Diese Funktion ist sehr hilfreich, da nicht nur nach Sync und CRC Fehlern, sondern auch nach fehlerhaftem Signalverhalten automatisch gesucht werden kann.

Messfunktion

Es ist außerdem möglich zwischen einzelnen Samples den Abstand zu messen. Auch dieses Feature macht das X2X Scope zu einem sehr nützlichen und praktischen Analyse Tool.

8.2.3 Statistische Auswertung

Die Dekodierung liefert neben den Datenfile für das Omni Peek Plugin bereits im Vorfeld eine statistische Auswertung mehrerer Kenngrößen. So werden zum Beispiel gleich nach der Dekodierung Kompressionsrate, Anzahl diverser Fehler und Frames, sowie Zyklus- und Leerlaufzeit angegeben. Diese Informationen machen in vielen Fällen eine weitere Betrachtung mit dem Omni Peek Plugin nicht mehr notwendig.

8.3 Zusammenfassung

Mit dem X2X Scope wurde ein sehr mächtiges Debug Tool entwickelt, welches allen im Vorfeld gestellten Anforderungen entspricht und seine volle Funktionsstärke im Feld beweisen kann. Es beinhaltet folgende drei verschiedene Ebenen zur Analyse des X2X Signales:

8.3.1 Fehlererkennung auf X2X Protokollebene

Das X2X Scope ist imstande, Länge und Wert des *Syncbit*, sowie die korrekte Darstellung von Leerlauf (Idle), Start- und Enddelimiter auf Signalebene zu überprüfen. Weiteres ist auch eine CRC Überprüfung nach standardisierten Vorgaben realisiert worden. Auf diese Weise ist eine komplette Fehlererkennung, wie sie auch in normalen X2X Modulen implementiert ist, gegeben. Diese Fehler werden einerseits im Auswerte Tool und andererseits im Omni Peek Plugin angezeigt.

8.3.2 Fehlererkennung auf X2X Datenebene

Auch auf Datenebene können eventuelle Datenfehler durch Analyse im Omni Peek Plugin erkannt werden. Dabei handelt es sich um Fehler, die nicht an der Übertragung oder dem Signal liegen, sondern von vornherein falsch gesendet wurden, wie zum Beispiel diverse Zeitstempel und Zyklenzähler. Diese Fehler konnten bereits bei der Vorgängerversion erkannt werden. Damit wurde auch die volle Funktionalität des ursprünglichen X2X Analyzers in der erweiterten Version realisiert.

8.3.3 Oszilloskopfunktion

Die Oszilloskopfunktion spiegelt die ganze Stärke des X2X Scopes wieder. Ein Zitat aus der Diplomarbeit des Vorgängermodells zeigt das Problem, welches mit diesem Tool gelöst werden sollte.

> *... Allerdings wurden auch die Grenzen eines solchen Analysators aufgezeigt. Bei einem Einsatz wurden aufgrund schlechter Kabelverbindungen zwischen den X2X-Link Stationen die notwendigen Timinganforderungen an die Übertragungsstrecke nicht mehr eingehalten. Der Analysator arbeitet aber mit der selben Abtastgeschwindigkeit wie die X2X-Link I/O-Stationen und konnte aus diesem Grund keine Mehrinformation zur Fehlerursache liefern. Erst eine Messung mit dem Oszilloskop konnte die wahre Ursache für diese Übertragungsfehler zeigen.* [Pic06]

Genau an dieser Stelle soll das neue X2X Scope, welches im Zuge dieses Projektpraktikums, sowie der vorangegangenen Bachelorarbeit entworfen und umgesetzt wurde, Abhilfe schaffen, indem es zwar kein vollwertiges Oszilloskop ersetzt, aber Dank der achtfach Abtastung so gut wie möglich nachahmt. Damit schafft es die Möglichkeit, zu lange oder zu kurze Pegel im Signal, welche eventuelle Fehler verursachen, zu analysieren.

Literaturverzeichnis

[Ash98] P.J. Ashenden. *The Student's Guide to Vhdl.* Morgan Kaufmann Series in Systems on Silicon. Morgan Kaufmann Publishers, 1998.

[CWF95] David R. Hanson Christopher W. Fraser. *A Retargetable C Compiler: Design and Implementation.* Addison-Wesley, 1995.

[Gra99] Jan Gray. *The xr16 Specification.* Gray Research LLC, 1999.

[Ins08] The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA. *IEEE Standard for Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Specific requirements*, Dezember 2008.

[MKGK09] T. Müller, H. Käser, R. Gübeli, and R. Klaus. *Technische Informatik I: Grundlagen der Informatik und Assemblerprogrammierung.* Technische Informatik. vdf, Hochschulverlag an der ETH Zürich, 2009.

[OW02] Gerald Pichler Otto Wagner. *X2X - Link.* Bernecker und Rainer Industrie Elektronik, 1999 - 2002.

[Pic06] Gerald Pichler. *X2X Link Protokollanalysator*, September 2006.

[RAW85] Donald E. Thomas Robert Allen Walker. *A Model Of Design Representation And Synthesis.* Proceeding DAC '85 Proceedings of the 22nd ACM/IEEE Design Automation Conference, 1985.

[Sed02] R. Sedgewick. *Algorithmen.* Informatik - Pearson Studium. Addison-Wesley-Longmann, 2002.

[Wol09] Jürgen Wolf. *C von A bis Z.* Galileo Computing, 2009.