Fachhochschule Jena
Fachbereich Elektrotechnik

Diplomarbeit

Das Bussystem IEEE 1394 (FireWire)

Untersuchung des Bussystems IEEE 1394 und dessen Realisierungsmöglichkeiten in Hinblick auf embedded Lösungen im Automatisierungsbereich.

eingereicht von Stephan Linz
geb. am 01. 04. 1973 in Jena

Matrikel-Nr.: 95201021
Seminargruppe: 952ET-IT

Hochschulbetreuer: Prof. Dipl.-Ing. H. Wagner
Mentor: Dipl.-Ing. F. Reklies

Datum der Themenausgabe: 01. 11. 1999
Abgabedatum: 01. 02. 2000

Danksagung

Ich möchte mich bei meinen Betreuern und allen Mitarbeitern des GNU/Linux IEEE 1394 Projekts für ihre tatkräftige Unterstützung bedanken. Besonderer Dank gebührt meiner Frau sowie meiner gesamte Familie für ihr Verständnis und ihre Rücksichtnahme.

Zusammenfassung

Durch das neue Buskonzept des Standards IEEE 1394, das auf einem komplexen aber fortschrittlichen Kommunikationsmodell beruht, erhält man erstmals ein preiswertes und leistungsstarkes System für den Aufbau von Verbindungsgliedern zwischen verschiedensten Geräteklassen.

Im Rahmen der Vorlaufentwicklung wird bei der MAZeT GmbH immer wieder das Ziel verfolgt, neue Technologien und Entwicklungsstufen in das Firmenknowhow einzuflechten. Aus diesem Zusammenhang heraus soll das in dieser Diplomarbeit behandelte Thema mit dazu beitragen, die durch den Einsatz von IEEE 1394 entstehenden Vor- und Nachteile bzw. die damit verbundenen Mehraufwendungen erfassen und abschätzen zu können. Mit Berücksichtigung der bisherigen Betätigungsfelder der MAZeT GmbH wird dabei speziell die Realisierung von embedded Lösungen und die Nutzung im industriellen Bereich als möglicher Feldbusersatz betrachtet. Dazu wird das Buskonzept erläuternd vorgestellt und anhand eines Vergleichs mit dem InterBus, der schon seit längerer Zeit ein wesentlicher Bestandteil der Forschungs- und Entwicklungsarbeiten bei der MAZeT GmbH ist, auf dessen momentane Industrietauglichkeit hin untersucht. Exemplarisch für den Prozeß der Systemintegration von IEEE 1394 dient der dritte Teil dieser Arbeit zur Vorstellung eines universellen Chipsets und dessen Vergleich mit anderen Möglichkeiten. Dabei wird mit den zur Verfügung stehenden Mitteln ein Kommunikationsdemonstrator aufgebaut, mit dem Ziel, die Erkenntnisse speziell im Softwareteil dieser Applikation auch anderweitig zu nutzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen und Begriffe

1394TA	1394 Trade Association
ANSI	American National Standards Institute — amerikanisches Institut für Normung
API	Application Programming Interface
Arbitration	Buszuteilungsverfahren
ASCII	American Standard Code for Information Interchange
asynchron	Ungleichheit der zeitlichen Abstände zwischen verschiedenen Arbeitsschritten
AUX	Auxiliary — engl. für Hilfs-
A/V	Audio und Video
AV/C	Audio Video Control
Backplane	engl. für Rückverdrahtung
Big-Endian	Bit- bzw. Byteorientierung mit Zählbeginn an der höchstwertigsten Stelle (MSB)
Branch	engl. für Zweig — bei IEEE 1394 ein Knoten mit mehr als einem Nachbarn
`bus_ID`	eine 10 Bit Zahl, die eindeutig einen Teilbus innerhalb eines Verbundsystems verschiedener Busse identifiziert
Child Node	der Knoten nach einem Knoten (Folgeknoten)

CORBA	Common Object Request Broker Architecture — ein Standard, der beschreibt, wie verteilte Objekte Informationen austauschen können
CPU	Central Processing Unit — zentrale Verarbeitungseinheit eines Computers
CRC	Cyclic Redundancy Check — Mittel zur Erkennung von Bündelfehlern
CSR	Control and Status Register architecture — eine Adreß- & Registerstruktur als Basis für IEEE 1394 (siehe [IEEE91])
Daisy-Chain	aneinandergereihter Geräteverbund
DBC	Device Bay Controller — Steuerung und Koordination der einheitlichen Schnittstelle Device Bay
DC	Direct Current — Gleichstrom
DDK	Driver Development Kit
Device Bay	eine Schnittstellenbeschreibung, die USB und IEEE 1394 vereinheitlicht
DMA	Direct Memory Access — direkter Speicherzugriff
DPP	Direct Printing Protocol
EEPROM	Electric Eraseable Programable Read Only Memory
EMV	Elektro-Magnetische Verträglichkeit
FIFO	First In First Out — spez. Speicherkonstruktion (Schlauchspeicher); auch als Pipe bekannt
FireWire	Synonym für IEEE 1394 — eingetragenes Warenzeichen von Apple Computers Inc.
gap	engl. für Lücke oder Kluft

GUI	Graphic User Interface
hop	engl. für Sprung — bei IEEE 1394 die Verbindung von einem zum nächsten Knoten
IEEE	Institute of Electrical and Electronical Engineers — amerikanische Vereinigung der Elektro- und Elektronikingenieure
IETF	Internet Engineering Task Force — offene internationale Gemeinschaft für Netzwerkdesigner
I/F	ein Kürzel für Interface
IICP	Instrument and Industrial Control Protocol
i.Link	Synonym für IEEE 1394 — eingetragenes Warenzeichen von Sony Corp.
I/O	ein Kürzel für Input und Output
IP-over-1394	Internet Protocol über IEEE 1394
ISA	Industrial Standard Architecture
ISO	International Standard Organisation — Internationaler Normen-Ausschuß
isochron	Gleichheit der zeitlichen Abstände zwischen verschiedenen Arbeitsschritten
Knoten	ein Teilnehmer am Bus
Leaf	engl. für Blatt — bei IEEE 1394 ein Knoten mit nur einem Nachbarn
Link Layer	Verbindungsschicht
Little-Endian	Bit- bzw. Byteorientierung mit Zählbeginn an der niederwertigsten Stelle (LSB)
LLC	Link Layer Controller — Steuerung der Verbindungsschicht

LSB	Least Significant Bit — niederwertiges Bit
LWL	Licht-Wellen-Leiter
MSB	Most Significant Bit — höchstwertiges Bit
`node_ID`	eine 6 Bit Zahl, die einen Knoten von allen anderen innerhalb eines Busses unterscheidet
Node	siehe Knoten
NRZ	Non Return to Zero — Codierungsverfahren, Leitungscode
OEM	Original Equipment Manufacturer — ursprünglicher Gerätehersteller
OHCI	Open Host Controller Interface — Schnittstellenbeschreibung für IEEE 1394
OPC	OLE for Process Control
OSI	Open Systems Interconnection — von ISO entwickeltes Modell für die Kommunikation zwischen offenen Systemen
PAP	Programm-Ablauf-Plan — Mittel zur strukturierten Darstellung von Algorithmen
Parent Node	der Knoten vor einem Knoten (Vorgängerknoten)
PCI	Peripheral Component Interconnect
PCL	Programm Control List
Peer-to-Peer	Möglichkeit der Datenübertragung ohne Benutzung eines Mastergeräts — direkte Datenübertragung von Sender zu Empfänger
PHY	Steuerbaustein für die physikalische Schicht
Physical Layer	physikalische Schicht
PLL	Phase Locked Loop — System aus Phasenvergleicher und VCO zum phasenstarren Synchronisieren zweier Frequenzen

RAM	Random Access Memory
ROM	Read Only Memory
Root Node	der oberste Knoten in einem Baum (logisch)
RTOS	Real Time Operating System — Echtzeitbetriebssystem
SBP-2	Serial Bus Protocol 2
SCSI	Small Computer System Interface
SDK	Source Development Kit
SPS	Speicher-Programmierbare Steuerung
Tcl	Tool Command Language — weit verbreitete Scriptsprache
Tk	Tool Kit — GUI Erweiterung für Tcl
TPA	Signalleitungspaar im IEEE 1394 Kabel — angeschlossen an DATA Receiver, STROBE Driver
TPB	Signalleitungspaar im IEEE 1394 Kabel — angeschlossen an DATA Driver, STROBE Receiver
TPBIAS	Gleichspannungssignal zur Busanschalterkennung
Transaction Layer	Ausführungsschicht
USB	Universal Serial Bus
VCO	Voltage Controlled Oscillator — spannungsgest. Oszillator
WDM	Win32 Driver Modell
XOR	exklusives ODER
μC	Mikrocontroller
μP	Mikroprozessor

Symbole, Formelzeichen und Einheiten

A Ampére; Einheit der elektrischen Stromstärke

BB Formelzeichen für Bandbreite

Bit binary digit — kleinste Informationseinheit in der Datenverarbeitung; mögliche Werte:
`0` logisch falsch (Lowpegel)
`1` logisch wahr (Highpegel)
`Z` nicht definiert (hochohmig)

Byte = 8 Bit

dB Dezibel — zehnte Teil eines logarithmischen Zahlenverhältnisses; dient der Angabe von Verstärkung und Dämpfung

Doublet = 2 Byte = 16 Bit — IEEE Std.-Notation

f Formelzeichen für Frequenz

Hz Herz — Schwingungen pro Sekunde; Einheit der Frequenz

kB = 1024 Byte; Kilobyte

kHz = 10^3 Hz; Kiloherz

m Meter

mA = 10^{-3} A; Milliampére

Mbps megabit per seconds — Megabit pro Sekunde

MBps megabyte per seconds — Megabyte pro Sekunde

MHz $= 10^6$ Hz; Megaherz

mV $= 10^{-3}$ V; Millivolt

n math. Symbol für ganze Zahlen

mm $= 10^{-3}$ m; Millimeter

ms $= 10^{-3}$ s; Millisekunde

ns $= 10^{-9}$ s; Nanosekunde

Octlet = 8 Byte = 64 Bit — IEEE Std.-Notation

Quadlet = 4 Byte = 32 Bit — IEEE Std.-Notation

s Sekunde; Einheit der Zeit

TB $= 2^{48}$ Byte; Terabyte

V Volt; Einheit der elektrischen Spannung

μ**s** $= 10^{-6}$ s; Mikrosekunde

Ω Ohm; Einheit des elektrischen Widerstand (Ohmscher Widerstand)

1. Einführung

1.1. Zweck und Intention dieser Arbeit

Seit einigen Jahren sind die Titelblätter namhafter deutscher Fachzeitschriften voll mit Schlagzeilen wie *„Ethernet in der Fabrik“* oder *„Vom Büro bis ins Feld via TCP/IP“*. Ohne solchen Aufmachern kommt fast keine Ausgabe mehr aus. Doch hat der sich angesprochen gefühlte Kunde gerade erst begonnen das Potential von Ethernet zu erkennen, taucht am Horizont wieder ein neues System mit der Behauptung auf, das klassische „Feld“ der Automatisierungstechnik revolutionieren zu wollen. Die Rede ist von dem aus der Multimediatechnik bekannten Bussystem IEEE 1394.

Was macht diese System eigentlich so interessant, daß zusehends immer mehr Produzenten industrieller Ausrüstung in Arbeitsgruppen zusammentreten, um Entwürfe für die Nutzbarmachung von IEEE 1394 in der Industrie zu erarbeiten? Einige Antworten auf diese Frage kann man sehr schnell erahnen. Zum einen ermöglicht die um ein vielfaches höhere Übertragungsrate endlich die Einführung von breitbandigen Kommunikationsmodellen und zum anderen erhält man mit IEEE 1394 ein Bussystem, das auch harten Echtzeitanforderungen im Millisekundenbereich und kleiner gerecht wird. Doch was steckt hinter den kontrovers geführten Diskussionen über die generelle Einführung dieses Systems? Manche Autoren sprachen in den Anfängen sogar davon, daß IEEE 1394 für die Industrie gänzlich uninteressant wäre, da das System nur einen kleinen Aktionsradius aufweist und somit keine größeren Strecken überwinden kann. Die eigentlichen Antworten auf diese Fragen wird man wohl erst durch den Blick hinter die Kulissen beantworten können. So ist es Aufgabe dieser Arbeit, anhand einer Analyse des sehr umfangreichen Standardwerks IEEE 1394 die Funktionsweise dieses Busses verständlich zu machen und die damit verbundenen Eigenschaften zu unterstreichen. Ferner soll der hypothetische Einsatz von IEEE 1394 als Feldbussystem betrachtet werden, um abschätzen zu können, inwieweit die Nut-

zung schon möglich ist oder in absehbarer Zeit erst noch wird.

Die stetig wachsende Produktion von industrietauglichen IEEE 1394 Anschaltbaugruppen zwingt zu diesen Auseinandersetzungen. Ebenso läßt der rasante Anstieg an IEEE 1394 Integrationen im Audio- und Videobereich sowie der Computerindustrie und die zunehmende Nachfrage in den Bereichen der Meßtechnik und Datenerfassung die Frage nach Systemlösungen laut werden. Es ist einleuchtend, daß der Aufwand wesentlich höher sein muß als die Implementation einer einfachen seriellen Schnittstelle. Doch, wie hoch? Was für Arbeit steckt hinter dem Hardwaredesign? Wieviel Software muß entwickelt werden und was kann ich käuflich erwerben? All diese Fragen sollen durch eine allgemeingültige Betrachtung der bis jetzt existierenden Lösungsvarianten beantwortet werden. Wichtig ist dabei die Betrachtung von Unterschieden und Gemeinsamkeiten der verschiedenen Ansätze sowie die Extraktion von technologischen Voraussetzungen.

Um Erfahrungen auf dem Gebiet der Softwareimplementierung zu gewinnen, ist die Bearbeitung einer Demoapplikation vorgesehen. Hierüber kann anhand der System- und Anwendungsprogrammierung der Aufwand viel besser abgeschätzt werden und die prinzipielle Funktionsweise von IEEE 1394 nachgewiesen werden.

2. Das Bussystem IEEE 1394

In diesem Kapitel werden die wichtigsten Eigenschaften und Funktionen des Bussystems IEEE 1394 erläutert, die für das Verständnis der nachfolgenden Kapitel wichtig sind. Ergänzende und weiterführende Informationen findet man in der Literatur [And98] und [IEEE95][1]. Für einen Überblick verweise ich auf [Tee99], [Pei99] und [Dem97], [Urb97], [Kro97].

2.1. Der Standard IEEE 1394

Der Standard IEEE 1394 beschreibt ein serielles Hochgeschwindigkeitsinterface. Dieses entstand aus einer Gemeinschaftsentwicklung der Firmen Apple, Adaptec, Sony u.a. und wurde 1995 zum IEEE 1394-1995 [IEEE95] erhoben. Einige Firmen, wie Apple oder Sony, haben dem Bus einen Produktnamen gegeben. So existieren auch Bezeichnungen wie FireWire[2] oder i.Link[3] für dieses System.

In diesem Standard werden Kabel und Steckverbinder sowie das gesamte Kommunikationsmodell fest definiert. Dabei kommt eine weitere Spezifizierung, die IEEE 1212, als CSR-Modell[4] für die Beschreibung des Adreßraumes und der Zugriffsoperationen Lesen, Schreiben und Sperren[5] zur Anwendung. Durch sie wird die Registerstruktur eines jeden Teilnehmers teilweise vorgegeben und der Inhalt bzw. die Benutzung jener Register durch das darauf aufbauende Bussystem bestimmt. Neben IEEE 1394 bauen die Standards „IEEE 896-1991, FutureBus+“ und “IEEE 1596-1992, SCI“ auf dieses CSR-Modell auf, wodurch die Verschmelzung aller drei

[1] für die Ausarbeitung stand mir nur ein Draft von IEEE 1394 zur Verfügung [P95]
[2] FireWire ist ein eingetragenes Warenzeichen von Apple Computers Inc.
[3] i.Link ist ein eingetragenes Warenzeichen von Sony Corp.
[4] CSR steht für Control and Status Register; siehe [IEEE91]
[5] Read, Write und Lock

genannten Systeme über Bridges schon von der Standardisierung her vereinfacht wird. Ferner ist nach [T10] der Einsatz von IEEE 1394 als eines von vielen Übertragungsmedien innerhalb des neuen SCSI-3 Architekturmodells vorgesehen.

Für die Pflege, Weiterentwicklung und Verbreitung von IEEE 1394 hat sich die *1394 Trade Association*[6] (*1394TA*) gegründet, die durch zahlreiche Firmen der herstellenden Industrie unterhalten wird. Unter ihrer Leitung treffen verschiedene themenbezogene Arbeitsgruppen zusammen, um die Belange neuer Einsatzgebiete von IEEE 1394 in Form von Spezifizierungen zu definieren.

2.2. Wesentliche Eigenschaften

Im IEEE 1394 sind viele Eigenschaften vereint, die sonst andere Busse nur zum Teil oder überhaupt nicht bieten. Das hier noch näher zu erläuternde Kommunikationsmodell kann in zwei verschiedenen physikalischen Umgebungen betrieben werden: Backplane und Kabel. Die Übertragungsgeschwindigkeiten können dabei je nach Medium Werte zwischen 25 Mbps und 400 Mbps annehmen. Neben den durch CSR standardisierten asynchronen Zugriffsmechanismen Lesen, Schreiben und Sperren bietet der Bus umfassende zeitbasierte Zusatzfunktionen, die das Übertragen von isochronen Datenströmen ermöglichen und eine genaue globale Zeitbasis im Mikrosekundenbereich bereit stellen. Hinzu kommen noch diverse bauliche Vorkehrungen und Automatismen innerhalb des Kommunikationsmodells, die dem Bus einen gewissen Grad an Eigenintelligenz verleihen, wie z.B. eine Selbstverwaltung von Busteilnehmern.

Zusammenfassend sind folgende Eigenschaften zu unterstreichen:

- *hohe Geschwindigkeit — große Bandbreite*
  Momentan werden Übertragungsraten von 100, 200 und 400 Mbps vom Standard her unterstützt. Diese sind gemischt nutzbar.
  Die Backplanevariante erlaubt nur Raten von 25 und 50 Mbps.

[6] http://www.1394ta.org/

- *asynchrone und isochrone Übertragung — Echtzeitfähigkeit*
  In einem Großteil der zur Verfügung stehenden Bandbreite werden unquittierte isochrone Daten verschickt. Sie unterliegen einem festen Zyklustakt und sind somit Echtzeitdaten. Für einen sicheren Datentransport werden quittierte asynchrone Pakete benutzt, deren Zustellungszeit nicht garantiert werden kann.

- *Peer-to-Peer Kommunikation — flexible Topologie*
  Die räumliche Anordnung der Busteilnehmer kann in beliebiger Baum- und Kettenstruktur erfolgen. Jeder Teilnehmer fungiert als Repeater.
  Die Datenübertragung erfolgt unabhängig von einem Master und damit immer direkt vom Sende- zum Empfangsgerät.

- *Ein- u. Auskoppeln im laufenden Betrieb — Hot Plug-and-Play*
  Durch die umfangreiche Selbstidentifizierung können Busteilnehmer dynamisch abgetrennt oder eingefügt werden.

- *Stromversorgung über den Bus*
  Geräte ohne eigene Stromversorgung können für die Aufrechterhaltung der Kommunikation über den Bus gespeist werden.

- *Skalierbarkeit*
  Der Bus ist für zukünftige Weiterentwicklungen in den Bereichen Übertragungsgeschwindigkeit und physikalische Medien ausgelegt, wobei die Abwärtskompatibilität erhalten bleibt.

Die obigen Geschwindigkeiten sind Näherungen, da sie genauer nach der Formel $f_{\ddot{U}} = f(n = \{13, 14, \ldots, 17\}) = 2^n \cdot 3000Hz$ berechnet werden, also: 24,576 Mbps, 49,152 Mbps bis 393,216 Mbps. Diese Formel ergibt sich nach detailierten Vorgaben des Zeitregimes und ist im Anhang, Abschnitt A.3 auf Seite 99, hergeleitet. Um die Schreibweise zu vereinfachen, werde ich so, wie im Standard, die Kürzel S25 für 24,576 Mbps, S50 für 49,152 Mbps usw. benutzt.

2.3. Kommunikationsmodell

Der Protokollstapel von IEEE 1394 setzt sich aus drei wesentlichen Schichten zusammen: *Physical Layer (PHY)*, *Link Layer (LLC)* und *Transaction Layer*. Alle drei Schichten werden über das *Serial Bus Management* zusammengefaßt. Wie man in Abbildung 2.1 sehen kann, sind nicht alle Teile in Hardware ausgelegt. Die Ausführungsschicht und das gesamte Busmanagement, also der Großteil des Kommunikationsmanagements, müssen in Form von Firmware als Software ausgelegt werden. Die Nutzbarmachung von IEEE 1394 für OEM Anwendungen wird über eine implementationsabhängige API-Schicht mit den Teilen *BuMI — Bus Management Interface*, *AsTI — Asynchron Transfer Interface* und *IsTI — Isochronous Transfer Interface* hergestellt ([And98] Seite 42ff. und [P95] 3.4 Seite 22ff.).

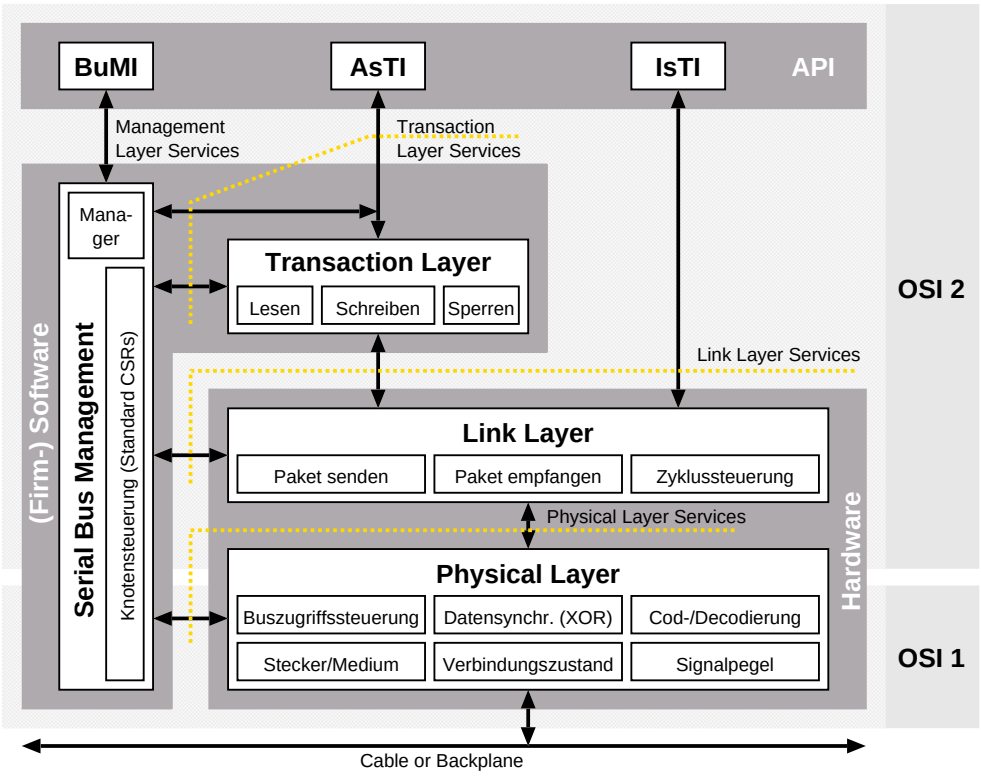

Abbildung 2.1.: Schichtenmodell von IEEE 1394 eingegliedert in OSI

Die Transaction Layer stellt alle notwendigen Funktionen für die asynchronen Protokolle Lesen, Schreiben und Sperren zur Verfügung. Diese Funktionen sind „bestätigte Funktionen“, das heißt: wird an einen Kommunikationspartner ein Da-

tenpaket für eine Schreibanforderung geschickt, so wird der Empfang quittiert. Es ist ein *Request-Response* Verfahren. Durch die Transaction Layer wird kein Service für die Übertragung von isochronen Daten bereitgestellt. Diese müssen direkt von einer höheren Instanz an die darunter liegende Link Layer übergeben werden.

Die Link Layer faßt die Forderungen nach asynchroner Übertragung von der Transaction Layer mit den benötigten isochronen Verbindungen der höheren Schichten zusammen und versucht nach dem Aufbau der jeweiligen Datenpakete diese mit Hilfe der physikalischen Schicht nach Erhalt des Busses[7] zu verarbeiten. Die isochronen Daten werden dabei in Kanälen verwaltet, die wahlweise empfangen oder gesendet werden können. Zudem erfolgt in dieser Schicht die Sicherung der Daten durch die Bildung und Kontrolle einer 32 Bit CRC-Summe.

Im Physical Layer werden alle für eine Datenübertragung notwendigen Vorkehrungen getroffen. Das sind im einzelnen die Steuerung über den Buszugriff zur Kollisionsvermeidung, Businitialisierung und -identifizierung nach einem Busreset, Leitungscode generieren bzw. decodieren, empfangene Daten auf den eigenen Takt synchronisieren sowie Signalpegel erzeugen und identifizieren. Am unteren Ende stellt der PHY seine Ports für den Anschluß an eine Kabel- bzw. Backplane-Umgebung bereit.

2.3.1. Abbildung von IEEE 1394 auf das OSI-Modell

Für einen späteren Vergleich von IEEE 1394 mit anderen Bussystemen ist es sehr hilfreich, wenn es dafür ein Referenzsystem als gemeinsame Basis gibt. Dazu eignet sich das OSI-Modell, beschrieben in [ISO84], durch das moderne Kommunikationssysteme in einer Struktur aus 7 Schichten abgebildet werden können.
Da in [P95] kein Bezug auf das OSI-Modell genommen wird, versuche ich aufgrund der folgenden Überlegungen nach [Ost97] 2.5.1 Seite 63 das Bussystem IEEE 1394 in das OSI-Modell abzubilden:

1. Unterschiedliche Funktionen in verschiedene Schichten und ähnliche Funktionen in gleiche Schichten einordnen.

2. Dann eine Schicht schaffen, wenn sich die Funktionen auf unterschiedlichen Ebenen der Abstraktion der Datenbearbeitung unterscheiden.

[7] erfolgreiche Buszuteilung

3. Grenzen der Schichten dorthin legen, wo die Dienstbeschreibung und die Anzahl der Aktionen über die Grenzen hinweg gering sind und wo in Zukunft die Wahrscheinlichkeit einer Schnittstellennormung absehbar ist.

Im Physical Layer von IEEE 1394 werden alle elektromechanischen Bestandteile beschrieben, wie sie im OSI-Modell Schicht 1 *Physical Layer* definiert sind. Selbst die Repeater-Funktion des Kabel-PHYs (siehe Abschnitt 2.4) wird unter OSI als *Ph-Transitsystem* bezeichnet.

Die Datensicherung, Verbindungsauf- und abbau sowie die Flußkontrolle sind Bestandteile der Schicht 2 *Data Link Layer*. Diese Aufgaben werden unter IEEE 1394 durch das Zusammenspiel von Transaction Layer, Link Layer und Serial Bus Management bewältigt. Das unter OSI beschriebene *DL-Transitsystem* ist unter IEEE 1394 gleichbedeutend mit dem Verbinden verschiedener Teilbusse über Bridges.

In Abbildung 2.1 auf Seite 22 sind die Zuordnungen der IEEE 1394 Schichten zu den zwei unteren OSI-Schichten durch die Bereiche OSI1 und OSI2 dargestellt. Den Zugang für die höheren OSI-Transportschichten stellt die darüber liegende API bereit.

2.4. Topologie und Adressierung

Geräte für den IEEE 1394 können entweder über ein Kabel oder eine Backplane miteinander verbunden werden. Hauptbestandteil des Standards ist die Kabelvariante, welche in meiner Arbeit Berücksichtigung findet und die Grundlage bildet.

Ein IEEE 1394 Gerät wird meist als Knoten bzw. Node bezeichnet. Die topologische Anordnung kann gleichzeitig als Kette (*daisy-chaining*) und Baum erfolgen, solange kein geschlossener Ring entsteht. Dabei wird immer ein Knoten über je ein einzelnes Kabel mit seinen Nachbarknoten verbunden. Eine solche Verbindung wird unter IEEE 1394 als *hop* (Sprung) bezeichnet und darf als Summe zwischen zwei beliebigen Knoten nicht größer als 16 werden (Abbildung 2.2 auf der nächsten Seite). Letztlich die Repeaterfunktion im PHY ermöglicht jedem Knoten die direkte Datenübertragung vom Sender zum Empfänger (*Peer-to-Peer*).

Ein Knoten, der mehr als nur einen Nachbarn besitzt, wird als *Branch* und einer mit nur einem als *Leaf* bezeichnet. Der Nachbar eines Knoten in Richtung

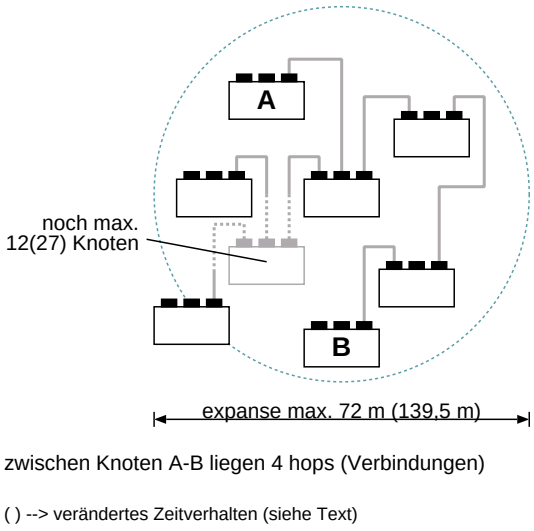

Abbildung 2.2.: Baumstruktur von IEEE 1394

logische Root ist der *Parent Node* und der entgegengesetzte der *Child Node*. Der *Root Node* stellt den logischen Ausgangspunkt des gesamten Baums dar und besitzt in gewissen Funktionsbereichen einen Sonderstatus. Dieser Knoten muß nicht zwingend die topologische (körperliche) Wurzel sein.

Die Entfernungen zwischen zwei Teilnehmern beträgt laut Standard bis zu 4,5 m. Durch das Aneinanderreihen von max. zulässigen 17 Geräten hintereinander (16 hops) werden insgesamt 72 m überwindbar[8] (siehe [P95] 4.2 Seite 49ff.). Durch die Anpassung der zeitlichen Verhältnisse nach [P95] Annex B.1 Seite 271ff. kann die Geräteanzahl zwischen den Endpunkten des Busses auf 32 erhöht werden und es entsteht eine Reichweite von 139,5 m.

Das so entstehende Netzwerk ist dank des PHY-Aufbaus eines jeden Knotens ein Diffusionsnetzwerk, da jeder Knoten in der Lage ist, Datenpakete, die an einem Port empfangen werden, an alle anderen Ports weiterzuleiten. Wird ein solcher PHY mit Energie aus dem Bus betrieben, ist die Weiterleitung auch bei „ausgeschaltetem“ Knoten vorhanden. Ohne ihn ist die Erreichbarkeit nachfolgender Knoten nicht mehr gegeben!

Durch diesen Repeater-Mechanismus erklärt sich unter anderem auch die Limitierung auf 16 bzw. 31 hops, denn jeder Knoten verursacht beim Weiterleiten von

[8] entspricht der bevorzugten Einstellung des Zeitregimes im PHY mit 16 hops nach einem Busreset

Datenpaketen eine zur Laufzeit zusätzliche Verzögerung, die so in Summe begrenzt und deterministisch wird. Das ist wichtig für das Buszuteilungsverfahren und die globale Synchronisation des Busses.

2.4.1. Das CSR-Modell und IEEE 1394

Die Adressierung des IEEE 1394 beruht auf dem CSR-Modell für 64 Bit breite Adressen[9], wovon die obersten 16 Bit repräsentativ für die Codierung der Knoten stehen. Diese 16 Bit sind durch IEEE 1394 noch einmal unterteilt in eine 6 Bit breite `node_ID` und eine 10 Bit breite `bus_ID` (Abbildung 2.3). Jedes der beiden Felder kennt eine Broadcast Adresse für spezielle Anwendungen (alle Bits logisch 1). Somit können in je einem der 1023 Teilnetze max. 63 Knoten existieren.

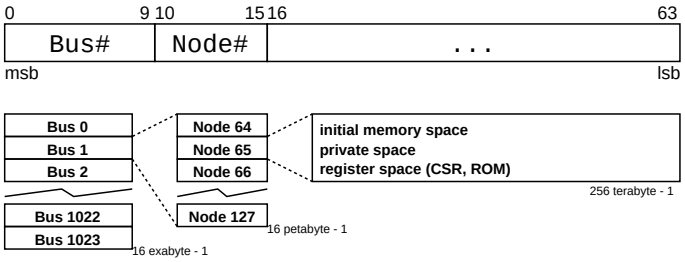

Abbildung 2.3.: 64 Bit breite CSR-Adresse unter IEEE 1394

Die restlichen 48 Bit spannen innerhalb des Knotens einen 2^{48} Byte (256 TB) großen Bereich auf, der in die Teile *Register Space*, *Privat Space* und *Initial Memory Space* zerlegt ist. Die Begriffe sind in [IEEE91] genauer definiert. Der Register Space wird noch einmal unterteilt in einen 2 kB großen *Initial Register Space* für wichtige Register, die teilweise durch das CSR-Modell vorgegeben sind, und der restliche Bereich, *Initial Unit Space*, für Register von implementationsabhängigen Teilfunktionen eines Knotens (Units, siehe weiter unten). Im *Initial Register Space* befinden sich die Kerninformationen der CSR-Architektur und die von IEEE 1394 abhängigen *Serial-Bus-Register* sowie die ersten 1024 Byte eines *Configuration-ROM*, auch *ROM-ID* genannt. Hier sind geräte-, funktions- und hard-/softwarespezifische Daten und Parameter in Form eines hierarchisch organisierten Baums, ähnlich einem

[9] unter IEEE 1394 der Datentyp *Octlet*

Filesystem mit Verzeichnissen und Dateien, hinterlegt (z.B. Herstellerkennungen, Versionsnummern).

Anzumerken ist, daß die Byteorder aller Daten und Adressen, die durch das Bussystem IEEE 1394 verarbeitet werden, durchgängig als *Big-Endian* definiert sind. Ankopplungen an ein *Little-Endian* System müssen zwangsläufig für ein Byte-Swapping sorgen.

Ebenfalls durch das CSR-Modell sind die Begriffe *Module*, *Node* und *Unit* vorgegeben. Diese Terminologie wird benötigt, um bestimmte Teile einer Geräteeinheit funktional abzugrenzen.

- *Module* — stellt ein physikalisches Gerät dar, das an den Bus angeschlossen ist.
- *Node* — ist ein logischer Teil eines Moduls und beinhaltet die CSR- und ROM-Einträge. Diese Knoten werden bei IEEE 1394 über die `bus_ID` und die `node_ID` adressiert.
- *Unit* — ist ein Funktionsteil eines Knotens, der einen Prozeß, Speicher oder I/O-Bereich darstellt. Die Arbeit mit ihnen muß über OEM Software erfolgen und wird bislang nicht definiert.

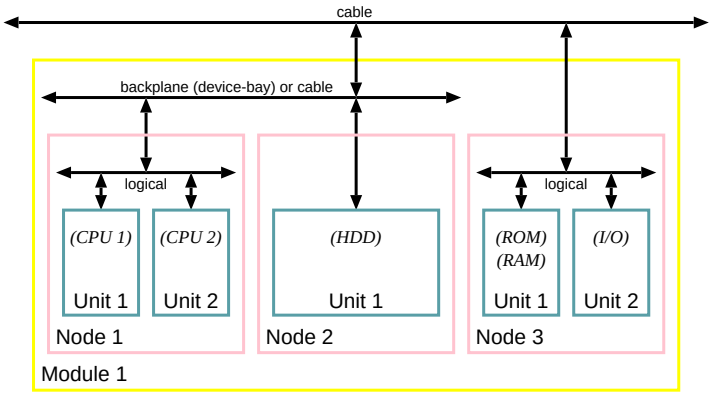

Abbildung 2.4.: Module, Node und Unit Architektur

Zu beachten ist jedoch, daß durch IEEE 1394 lediglich die Funktionsweise auf Node-Ebene beschrieben wird. Aussehen, Funktion und Anzahl von Units oder Art des Zusammenschluß zu Modulen ist implementationsabhängig.

In Abbildung 2.4 auf der vorherigen Seite ist als Beispiel eine herkömmliche Computerarchitektur auf das CSR-Modell mit IEEE 1394 als Kommunikationsmedium abgebildet. Das Modul ist dabei gleichbedeutend mit dem „Gerät Computer“ und die Units stellen die einzelnen Funktionsbausteine dar. Als physikalisches Medium kann dabei je nach räumlicher Anordnung die Backplane- oder Kabelvariante zum Einsatz kommen.

2.5. Elektromechanik

Laut Spezifizierung können an einem Kabel PHY zwischen 1 bis 27 Kabelanschlüsse vorhanden sein. Die Kabelvariante von IEEE 1394 benutzt dazu je einen 6-poligen Steckverbinder (verpolungssicher). Näheres dazu findet man im Anhang, Abschnitt A.1 auf Seite 97. Das Kabel besteht aus einem Bündel von zwei geschirmten twisted-pair Leitungen für zwei Datensignale (TPA, TPB) und zwei Leitungen für die Stromversorgung (VP, VG). Beide Leitungspaare sind im Kabel gekreuzt und beidseitig im Knoten mit 110 Ω abgeschlossen. Die genauen Kabeldaten für die verschiedenen Übertragungsgeschwindigkeiten können dem Anhang, Abschnitt A.2 auf Seite 98, entnommen werden.

2.5.1. Leitungscodierung

Über die beiden Datenleitungspaare TPA und TPB werden Datenpakete *synchron* von einem zum nächsten Knoten übertragen. Dafür wird nicht wie üblich ein Daten- und ein Taktsignal benutzt, sondern auf TPB der Datenstrom und auf TPA ein Strobesignal gesendet (*Data-Strobe-Encoding*[10]). Das Strobesignal wird dabei immer dann seinen logischen Pegel ändern, wenn das Datensignal in seinem Pegel gleich bleibt. Durch eine XOR-Verknüpfung beider Signale kann der für die synchrone Übertragung notwendige Takt wieder zurückgewonnen werden, da im Strobesignal die im Datensignal fehlenden Taktflanken für die Bitsynchronisation enthalten sind (vgl. Abbildung 2.5 auf der nächsten Seite). Der Vorteil dieses Verfahrens liegt

[10] „data-strobe bit-level encoding“ ist ein U.K. Patent (Nr.: 9011700.3; *DS-Link bit-level encoding*) von INMOS Limited und mit Lizenzansprüchen behaftet

darin, daß der hochfrequente Takt nicht direkt über eine Leitung gesendet wird[11] und die Jitter-Effekte sich auf zwei Leitungen aufteilen, also halbieren ([Kro97]).

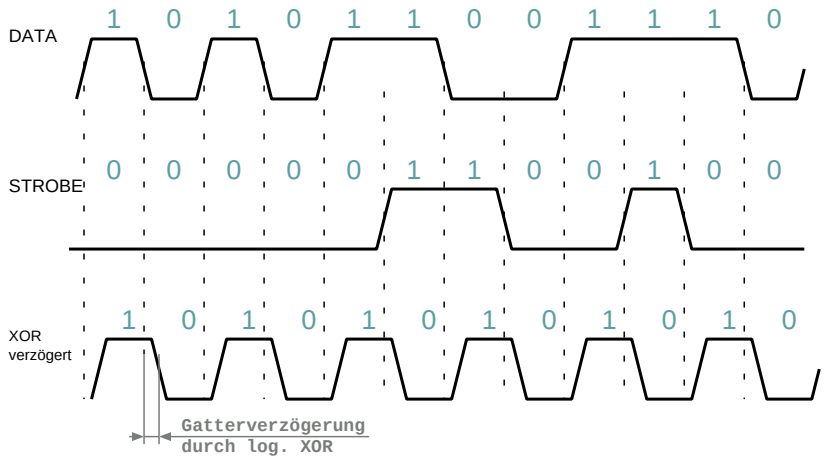

Abbildung 2.5.: Taktrückgewinnung aus DATA und STROBE

Beide Signale sind NRZ kodiert und müssen vor der Übertragung auf das Differenzsignal der Leitungen entsprechend umcodiert werden. Hierfür wird jede logische 0 mit einem negativen und jede logische 1 mit einem positiven Pegel umgesetzt[12]. Die für die Leitungspaare genutzte Differenzspannung kann sich je nach Busgeschwindigkeit in einem Bereich von 172 mV bis 265 mV bewegen ([And98] Seite 76ff. und [P95] 4.2.2.1 Seite 75ff.).

2.5.2. Status- und Handshakesignale (Arbitration Signals)

Zur Signalisierung verschiedener Betriebszustände sind den Differenzsignalen Gleichsignale überlagert. Eine zusätzlich aufaddierte Gleichspannung von ≥ 1 V, der *TPBIAS*, auf die Signalspannung von TPA zeigt dem gegenüberliegenden Knoten eine existierende physikalische Verbindung zwischen beiden an.
Will ein Knoten mit einer Geschwindigkeit größer als S100 mit seinem Nachbarn kommunizieren, dann wird dem Differenzsignal bei der Paketübertragung ein zusätzlicher Gleichstrom von 3,5 mA (S200) bzw. 10 mA (S400) aufgeprägt.

[11] vorausgesetzt der Pegel des Datensignal ist nicht gleichbleibend
[12] TPA*/TPB* sind dabei die Bezugspotentiale

Die Differenzsignale werden nicht nur für die Übertragung von Datenpaketen genutzt, wobei immer beide Leitungspaare gebraucht werden, sondern auch für den Austausch von wichtigen Protokollinformationen während der automatischen Konfiguration des Busses und der Buszuteilung. Dabei liegen an den beiden Leitungspaaren entsprechende Handshakesignale bidirektional[13] an, um die jeweiligen Anforderungen und Antworten zwischen den Knoten auszutauschen. Diese Signale werden *Arbitration Signals* genannt und sind als *Z*, *1* und *0* definiert. Durch die gekreuzten Leitungen stehen sich immer die Treiberausgänge von TPA und TPB gegenüber. Beide Ausgänge sind in der Lage ihren wahren Signalpegel auf der Leitung rückwärtig zu empfangen und mit ihrem getriebenen Wert zu vergleichen. Dabei können Abweichungen enstehen. Wird z.B. eine `1` gegen eine `0` getrieben so entsteht `Z`. Durch eine geeignete Reihenfolge solcher Signalspiele können Verhandlungen über die Buszuteilung oder die Systeminitialisierung erreicht werden. Mehr dazu findet man in [And98] Seite 78ff..

2.6. Übertragungsverfahren

Als Grundlage der kontrollierten Datenübertragung existiert ein fester Buszyklus, der genau 125 μs lang ist und durch den *Cycle Master — CM* mit dem Senden eines *Cycle Start* Pakets als Broadcast eingeleitet wird. Nur der Root Node kann den *CM* bereitstellen, da er nach den Regeln der Arbitrierung die höchste Priorität besitzt und sich so bei jeder Neuverhandlung über den Buszugriff immer durchsetzen kann, also zu jedem Startzeitpunkt eines Zyklus. Genaueres dazu findet man in [And98] Seite 309ff..

In den Zeitrahmen eines Zyklus teilen sich die beiden voneinander unabhängigen Übertragungsverfahren, *asynchron* und *isochron*. Dabei kann die isochrone Datenmenge bis zu 7/8 der zur Verfügung stehenden Bandbreite in Anspruch nehmen. Der Rest verbleibt für asynchrone Pakete, um den Transport von Kommandos und Steueroperationen höherer Protokollschichten auch bei primär isochronem Betriebsverhalten sicherzustellen. In Abbildung 2.6 auf der nächsten Seite ist ein Beispiel gezeigt.

[13] die Leitungspaare sind im Kabel gekreuzt

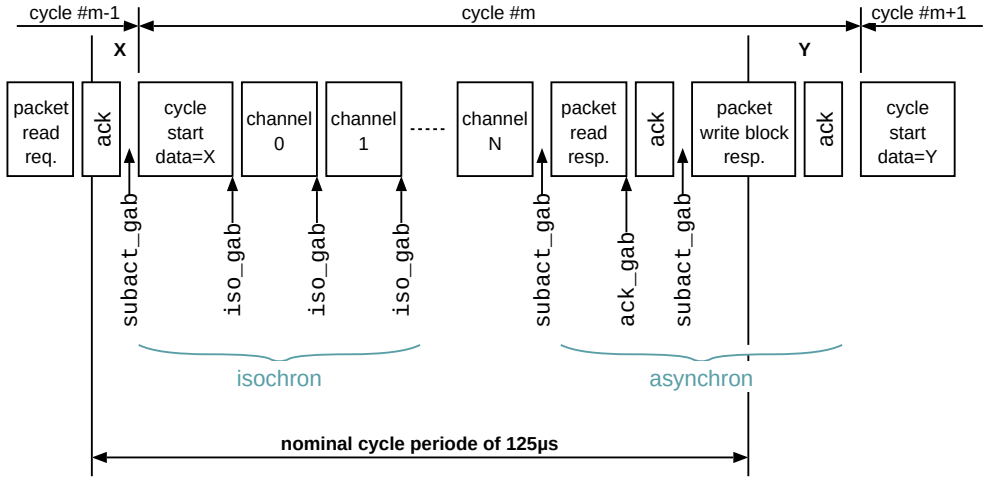

Abbildung 2.6.: Buszyklus mit iso- und asynchronen Datenpaketen

Isochrone Datenpakete werden am Anfang des Zyklus gesendet. Daran schließen sich die asynchronen an. Wie zu sehen ist, kann die letzte asynchrone Paketübertragung den Startpunkt des Folgezyklus überschreiten (X und Y). Eine solche Abweichung (*delay*) wird indirekt ausgeglichen, indem durch das folgende *Cycle Start* Paket die Verzögerung allen Knoten mitgeteilt wird. Solche Knoten, die gerade isochrone Kanäle unterhalten, können sich so auf die Zeitabweichung einstellen und die Isochronität bleibt erhalten. Die maximal mögliche Latenzzeit für isochrone Daten kann die Zeit eines Zyklus nicht überschreiten.

2.6.1. Asynchron

Asynchrone Datenpakete werden von einem Sender zu einem Empfänger übertragen und quittiert. Nur Broadcast-Pakete, wie z.B. das *Cycle Start* Paket, werden nicht bestätigt, da sie an alle Knoten gerichtet sind. Die Reaktion auf eine mißlungene Aktion wird durch die Wahl eines *Retry Protocols* innerhalb der Transaction Layer bestimmt (siehe [P95] 7.3.5 Seite 189ff.). Zusätzlich werden die Datenpakete über eine 32 Bit CRC-Summe abgesichert. Die Paketgröße hängt von der jeweiligen Bandbreite ab und wächst mit ihr linear an (von 512 Byte bei S100 bis 2048 Byte bei S400). Der Hintergrund dafür ist in der Gewährleistung der fairen Buszuteilung zu suchen. Jede asynchrone Transaktion, inklusive der Bestätigung und Buszuteilung, darf nicht länger als 62 μs andauern, wodurch die Sendezeit begrenzt wird.

Die kleinste zu übertragende Datenmenge ist ein Quadlet (4 Byte). Die asynchrone Kommunikation beruht auf einem Request-Response Verfahren und stellt die Funktionen Lesen, Schreiben von einzelnen Quadlets oder ganzen Blöcken und Sperren einzelner Quadlets bereit. Zusammen mit dem *Cycle Start* Paket, was ebenfalls als asynchrones Datenpaket angesehen wird, existieren insgesamt zehn verschiedene asynchrone Paketformate. Die notwendige Bestätigung der Pakete erfolgt durch ein 1 Byte *Acknowledge Packet*. Die Pakete sind in [And98] Seite 136ff. oder [P95] 6.2 Seite 142ff. genau erklärt.

2.6.2. Isochron

Isochrone Kommunikation wird erst mit dem Vorhandensein eines *Isochronous Resource Manager — IRM* möglich, der nur einmalig vorhanden sein darf. Dieser übernimmt die Zuteilung der Bandbreite an sendewillige Knoten. Hierfür kann er innerhalb eines Busses bis zu 64 Kanäle verwalten. Isochrone Datenpakete werden von einem Sender an alle empfangswilligen Knoten in zyklischen Abständen unterbrechungsfrei und ungepuffert übertragen. Sie werden nicht bestätigt, sind aber ebenso über eine 32 Bit CRC-Summe abgesichert. Sie besitzen im Gegensatz zu asynchronen Daten eine garantierte Bandbreite und Zustellungszeit. Durch die fehlende Empfangsbestätigung gehen auf dem Transport beschädigte Daten verloren, wenn sie nicht durch die CRC-Summe restauriert werden können.

Die kleinste zu übertragende Datenmenge ist ein Quadlet. Die Paketgröße ist je nach Bandbreitenzuteilung durch den *IRM* beliebig groß. Es ist möglich die gesamten 7/8 (siehe oben) für einen Kanal in Anspruch zu nehmen. Das Paketformat ist in [And98] Seite 170ff. oder [P95] 6.2.3 Seite 153ff. beschrieben.

2.7. Zugriff auf den Bus

2.7.1. Paketabtrennung durch Gaps

Nach jedem übertragenen Paket entstehen Lücken (*Gaps*) unterschiedlicher Länge, in denen der Bus keine Signale führt — er ist still. Folgend auf diese Lücken wird je nach Übertragungsart (asynchron oder isochron) eine Neubewerbung um den Bus

ausgelöst. Die Gaps sind in Abbildung 2.6 auf Seite 31 als Zwischenräume dargestellt und werden in folgende vier Typen unterschieden:

- *Subaction Gap*:
  Die Übertragung aller isochronen Kanäle und die Abwicklung eines asynchronen Datenblocks (Senden + Bestätigung) werden als Aktionen behandelt. Die Trennung voneinander erfolgt durch diesen Typ. Er kennzeichnet den Abschluß der isochronen Kanalübertragung und leitet die Busbewerbung für asynchrone Pakete ein.

- *Acknowledge Gap*:
  Diese Lücke trennt ein asynchrones Datenpaket von dessen Bestätigungspaket. Es ist sehr viel kürzer als ein Subaction Gap, wodurch bei einem ausbleibenden Acknowledge Packet sofort der Abschluß dieser (fehlgeschlagenen) Aktion interpretiert wird.

- *Isochronous Gap*:
  Zwischen den einzelnen isochronen Kanälen muß ebenfalls eine Trennung durchgeführt werden, da sie nicht alle von einem einzelnen Knoten aus versendet werden. Zu Beginn eines Buszyklus befindet sich das Kommunikationssystem in der Phase (Aktion) der isochronen Übertragung. Ein Isochronous Gap innerhalb dieser Phase hat eine Neubewerbung um den Bus für weitere isochron sendewilligen Knoten zur Folge. Der Abschluß dieser Phase wird durch den oben schon beschriebenen *Subaction Gap* gekennzeichnet.

- *Arbitration Reset Gab*:
  Das ist ein spezieller Gap, der für die Steuerung der Fairneß bei der Buszuteilung für asynchronen Übertragungswunsch benötigt wird. Siehe dazu weiter unten.

Die absoluten Längen der Gaps sind je nach Konfiguration des Zeitregimes im PHY unterschiedlich, jedoch ist die Relation zwischen ihnen immer gleich:

$$\text{Acknowledge Gap} < \text{Subaction Gap} < \text{Arbitration Reset Gap}$$
$$\text{Acknowledge Gap} = \text{Isochronous Gap}$$

2.7.2. Buszuteilungsverfahren

Die Buszuteilung bei IEEE 1394 basiert auf einem fairen Verfahren. Jeder Knoten erhält innerhalb eines garantierten Zeitrahmens wenigstens einmal Zugang zum Bus. Wird das Ende eines isochronen Datenpakets (Kanal) durch ein Isochronous Gap oder der Abschluß einer asynchronen Transmission durch ein Subaction Gap erkannt, bewirbt sich ein sendewilliger Knoten mit seiner Forderung den Bus zu erhalten bei seinem Parent Node. Gleichzeitig weist er gleichartige Anfragen an ihn selbst durch vorhandene Child Nodes zurück. Existiert nun kein weiterer Knoten in der Hierarchie über ihm, der ebenfalls Anspruch auf den Bus angemeldet hat, so wird ihm der Buszugriff durch seinen Parent Node erlaubt, da dieser durch die gleiche Funktionalität die Bestätigung erhalten hat. Andernfalls muß der Knoten warten und diesen Vorgang zum nächsten Arbitrierungszeitpunkt wiederholen. Die Fairneß besteht nun darin, daß sich ein Knoten mit bereits erhaltenem Zuschlag an einer weiteren Bewerbung nicht mehr beteiligen darf, bis das Ende des *Fairness Intervalls* erreicht wurde. So werden alle Knoten vom Root Node aus immer tiefer in den Baum absteigend in die Lage versetzt, Daten zu senden. Für die Signalisierung zwischen den Knoten werden die in Abschnitt 2.5.2 auf Seite 29 beschriebenen *Arbitration Signals* benutzt. Der genaue Ablauf wird in [And98] Seite 104ff. gezeigt.

Isochrone und asynchrone Arbitrierung

Der Unterschied zwischen isochroner und asynchroner Arbitrierung besteht in der Länge des Fairness Intervalls. Alle Knoten mit dem Wunsch nach isochroner Kommunikation werden durch die Bandbreitenaufteilung innerhalb eines Buszyklus dafür Gelegenheit erhalten. Das gilt nicht für Knoten mit asynchronem Sendewunsch. Hier kann sich das Fairness Intervall über mehrere Buszyklen erstrecken.

Bei der isochronen Arbitrierung kann das Ende des Fairness Intervalls durch das erste Auftreten eines Subaction Gaps innerhalb des Zyklus erkannt werden. Bei der asynchronen Arbitrierung erfolgt das gleiche durch ein Arbitration Reset Gap. Danach beginnen wieder alle Knoten, sich um den Bus zu bewerben.

2.8. Initialisierung und Identifizierung

Zu Beginn dieses Kapitels wurde ein gewisser Grad von Eigenintelligenz des Bussystems IEEE 1394 erwähnt. Eines der wichtigsten davon ist die Fähigkeit der Selbstkontrolle und -verwaltung des topologischen Aufbaus. Das ist ein substantieller Bestandteil des Kommunikationsmodells und zwingende Voraussetzung für die Funktion des Busses. Ohne die Kenntnis über den Aufbau und Zustand von Knoten in der unmittelbaren Nachbarschaft bis hin zur gesamten Ausdehnung des Busses können Funktionen wie Arbitrierung und Busmanagement nicht realisiert werden.

In diesem Abschnitt soll beispielhaft der gesamte Ablauf einer Reinitialisierung vom Auslösen eines Busresets bis zur vollen Funktionstüchtigkeit erläutert werden. Dazu wird der in Abbildung 2.7 auf der nächsten Seite gezeigte Beispielaufbau angenommen. Es sei darauf hingewiesen, daß dieser Abschnitt einen sehr komplexen und komplizierten Funktionsteil von IEEE 1394 beschreibt. Für ausführliche und verbindliche Informationen über diese Thematik möchte ich auf den Standard selbst bzw. [And98] Seite 253ff. verweisen.

2.8.1. Businitialisierung

Ausgelöst durch ein Ab- oder Zuschalten eines Knotens an den Bus oder der Identifizierung eines Stromausfalls im Buskabel (*power-fail*) wird automatisch ein busweites Reset erzeugt. Dieses spezielle *Arbitration Signal* veranlaßt alle Knoten ihre aktuellen Daten über die Infrastruktur des Busses zu verwerfen und eine Neuinitialisierung ihrer Ports vorzunehmen. Dabei wird über die Analyse von überlagerten Gleichspannungen der Verbindungszustand eines jeden Ports ermittelt und wie folgt registriert:

1. Alle vorhandenen Ports werden durchnumeriert.

2. Jeder nicht verbundene Port wird isoliert und nicht weiter betrachtet.

3. Bleibt nur ein Port übrig, dann ist der Knoten ein ***Leaf***. Bei mehr als einem verbundenen Port ist der Knoten ein ***Branch*** (Abbildung 2.7 auf der nächsten Seite).

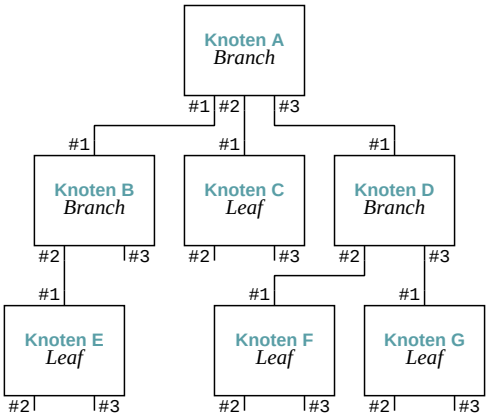

Abbildung 2.7.: IEEE 1394 Baumtopologie nach Businitialisierung (Bsp.)

Nach Abschluß der Businitialisierung (*Bus Initialize*) kennen alle Knoten die Anzahl ihrer Ports und wissen, welche aktiv sind. Sie wissen auch ob sie Leaf oder Branch sind.

2.8.2. Baumidentifizierung

Nach Beendigung der Businitialisierung folgt die Baumidentifizierung (*Tree Identify*). Jetzt wird die existierende Netzwerkstruktur in einen logischen Baum überführt. Dazu werden alle aktiven Ports mit einer Richtungskennung versehen, also einer Markierung, die aussagt, ob ein Port in Richtung Root Node oder weiter abwärts in den Baum verzweigt. Die Knoten in Richtung Root Node werden als Parent Node und in anderer Richtung als Child Node bezeichnet. Die Kennzeichnung der Ports in diese Richtungen erfolgt genauso nach dem folgenden Schema:

1. Jeder Knoten mit nur noch einem ungekennzeichneten Port versucht sein Parent Node zu finden. Er sendet ein *parent_notify* über dieses Port aus. Zu Beginn werden dies alle Leafs tun, da sie lediglich ein aktives Port besitzen.

2. Der Parent Node beantwortet das *parent_notify* mit einem *child_notify* und markiert dieses Port als *child*.

   Für den Spezialfall, daß ein Knoten unbedingt Root Node werden will, verweigert er diese Quittierung für einen längeren Zeitraum (160 μs). Damit erzwingt er, daß an allen seinen Ports ein *parent_notify* entsteht.

3. Empfängt ein Knoten als Bestätigung auf sein *parent_notify* das erwartete *child_notify*, so wird dieser Port als *parent* gekennzeichnet und der Knoten hat seine Baumidentifizierung abgeschlossen.

   In manchen Fällen, speziell bei nur zwei Knoten im System, senden zwei verbundene Ports gegenseitig ein *parent_notify*, um den vermeintlichen Parent Node zu finden. Jetzt müssen beide ihre Signalisierung zurücknehmen und nach Ablauf zweier unterschiedlicher Zufallstimer die Suche nach dem Parent Node wieder aufnehmen. Dieser *Root Contention* kann sich u.U. wiederholen und ist erst abgeschlossen, wenn einer der beiden Knoten schneller sein *parent_notify* senden kann und der gegenüberliegende Knoten dies mit einem *child_notify* bestätigt.

4. Jetzt sind wieder Knoten entstanden, bei denen alle Ports, bis auf eins, gekennzeichnet sind. Die Identifizierung setzt sich mit Punkt 1 fort.

   Sollten alle Knoten ihre Ports gekennzeichnet haben, so ist die Baumidentifizierung abgeschlossen und es existiert ein Knoten, bei dem alle Ports als *child* markiert sind. Dieser Knoten erhebt sich selbst zum Root Node.

Nach zwei Durchläufen der obigen Regeln ist die Beispieltopologie wie in Abbildung 2.8 fertig initialisiert und identifiziert. Interessant ist, daß die Benennung eines Knotens zum Root Node unabhängig von der realen Topologie ist. Selbst ein Leaf kann es erzwingen, im Gesamtsystem Root Node zu werden.

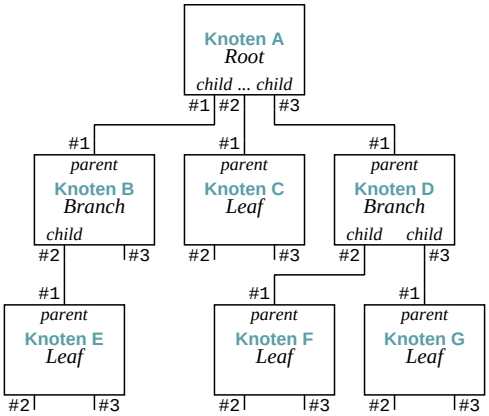

Abbildung 2.8.: IEEE 1394 Baumtopologie nach Baumidentifizierung (Bsp.)

2.8.3. Selbstidentifizierung

Mit dem Abschluß der Baumidentifizierung kennt jeder Knoten erst seine unmittelbare Umgebung. Die Arbitrierung funktioniert jedoch schon, da die Hierarchie des Baums festgelegt ist. Bereits etablierte isochrone Kanäle aus einer früheren Buskonfiguration können ab jetzt schon wieder gesendet und empfangen werden, da deren Adressierung unabhängig von Knotenadressen ist.

Um auch wieder asynchrone Pakete verschicken zu können, muß jedem einzelnen Knoten eine eindeutige `node_ID` zugeteilt werden. Dazu können alle Knoten in einer geordneten und vorherbestimmten Reihenfolge ihre Konfigurationsdaten, wie Übertragungsgeschwindigkeit der Ports oder Energiekonsum aus dem Bus, mit ihren bisherigen Initialisierungsdaten an alle anderen Knoten senden. Parallel dazu werden die eindeutigen `node_ID` vergeben:

1. Durch das Busreset wurde in jedem Knoten der `self_ID` Zähler auf Null zurückgesetzt. Als Start für die Selbstidentifizierung legt der Root Node an seinen ersten Child Port ein *grant* an. Dieses Zeichen signalisiert die Sendefreigabe. Alle übrigen Child Ports erhalten ein *data_prefix*, wodurch die dahinter liegenden Knoten für ein Senden gesperrt sind und Daten nur empfangen können.

2. Der nächst tiefere Knoten, der das *grant* empfangen hat, reicht dieses auf die gleiche Weise wie der Root Node an seinen ersten Child Node weiter. Das wiederholt sich solange, bis das *grant* ein Leaf erreicht.

   Der Leaf kann das *grant* seinerseits nicht mehr weiterreichen und antwortet auf die Sendeaufforderung mit einem *data_prefix*, worauf der Root Node sein *grant* zurück nimmt und mit einem *data_prefix* bestätigt. Jetzt warten alle Knoten auf Daten, die der gerade aufgeforderte Knoten (Leaf) senden muß, worauf dieser den Stand seines `self_ID` Zählers als seine `node_ID` übernimmt und zusammen mit den Informationen über seine Portkonfiguration als *Self ID Packet* sendet.

   Der Abschluß des *Self ID Packet* wird durch ein *data_end* gekennzeichnet, worauf alle anderen Knoten ihren `self_ID` Zähler inkrementieren. Der Port, an dem der soeben identifizierte Child Node hängt, wird vom übergeordneten

Parent Node als „identifiziert“ gekennzeichnet und in den weiteren Prozeß nicht mehr einbezogen.

3. Der unter 1. und 2. beschriebene Ablauf setzt sich jetzt mit jedem weiteren noch nicht als „identifiziert“ gekennzeichnetem Child Port fort, bis alle, außer dem Root Node, ihre `node_ID` erhalten haben.

4. Der Root Node sendet als letzter Knoten sein *Self ID Packet* aus und verharrt dann für die Zeitspanne eines *Arbitration Reset Gap* ohne Aktion am Bus. Somit wird sichergestellt, daß die Knoten ihre gewöhnliche asynchrone Arbitrierung wieder aufnehmen können.

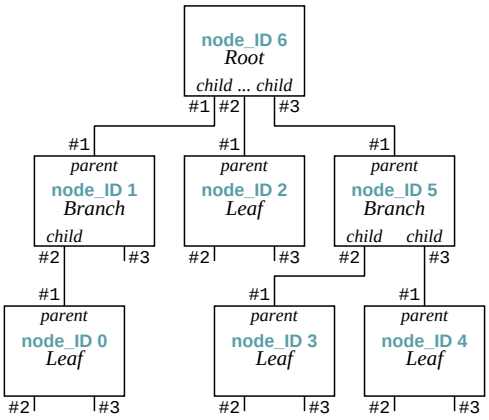

Abbildung 2.9.: IEEE 1394 Baumtopologie nach Selbstidentifizierung (Bsp.)

Jetzt sind alle Knoten wie in Abbildung 2.9 adressiert. Die *Self ID Packets* mit den jeweiligen `node_ID` können von jedem Knoten mitgeschrieben werden, wodurch diese in der Lage sind, eine *Topology Map* und eine *Speed Map* des aktuellen Baums aufzubauen. Die Knoten erhalten also die Möglichkeit neben ihrer unmittelbaren Umgebung die gesamte Busstruktur zu kennen. Tun sie das nicht, können sie sich immer noch an einen *Busmanager* wenden, der diese Daten mit geschrieben haben muß.

2.9. Erweiterungen von IEEE 1394

IEEE 1394a ergänzender Standard und beseitigt Unstimmigkeiten und Mißdeutungen:

- beschleunigte Arbitrierung — zusammen mit einem *Acknowledge Packet* auch Daten senden, *Arbitration Reset Gap* verkürzt
- schnelleres Reset und Begrenzung von Reset Storming (Störungen durch an- und abstecken von Geräten an den Bus)
- kontrollierte Energieeinsparung (Power Reducing) durch Suspend/Resume Verfahren — ein PHY kann vom Bus her durch ein Power Management suspendiert und bei Bedarf wieder aktiviert werden
- PHY „pinging“ — automatische Konfiguration und Feinjustage des PHY Zeitregimes
- zusätzlich einen 4-poligen Stecker definiert[14] — es wird die Stromversorgung vernachlässigt
- PHY-LLC fest definiert — in IEEE 1394 bisher nur als “informativ“ enthalten
- die Arbeitsgruppe schloß ihre Arbeiten zu Q2/99 ab; in diesem Jahr wird der Standard IEEE 1394a-2000 verabschiedet
- Quellen: [P99a]

IEEE 1394b ergänzender Standard zu neuen Übertragungsmedien und -raten:

- Kabelvariante wird um die beiden Medien Lichtleiter und UTP5 (wie Fast Ethernet) erweitert
- Übertragungsrate vergrößert sich auf 800, 1600 und 3200 Mbps
- Entfernung zwischen zwei Knoten kann auf 50, 70 oder 100 m ansteigen
- neuer Leitungscode: 8B10B
- die Arbeitsgruppe besteht seit 3 Jahren; Spezifizierung hält an
- Quellen: [P99b]

[14] Original von Sony

IEEE 1394.1 Netzwerkspezifizierung zu IEEE 1394:

- Erweiterung um Bridges zum Zusammenführen verschiedener IEEE 1394 Busse zu „einem Netz“
- erweiterte Definition von CSR-Einträgen und der *ROM-ID*
- global agierendes Managementsystem: *Cycle Master* bis *Bus Master* arbeiten für das gesamte Netz
- neues asynchrones Paketformat eingeführt: *Global Asynchronous Stream Packet* (*GASP*) — wird für die Implementation verschiedener höherer Netzwerkprotokolle benötigt (z.B. Internet Protokoll)
- Quellen: [P97.1]

3. Feldbusbetrachtung

Der Standard IEEE 1394 findet bisher in den Bereichen der Multimediatechnik seine Haupeinsatzgebiete. Seine Stärken spielt er innerhalb der Audio- u. Videoschnitttechnik aus, wenn datenintensive Informationen in Form von *digitalen* Bitströmen zwischen verschiedenen Geräteeinheiten wie Kamera, Speicher und Sichtgeräten ausgetauscht werden müssen. Dieser Umstand ist letztendlich nicht nur auf die besonders dafür geeignete Funktionalität, sondern auch auf die Herkunft von IEEE 1394 aus den Entwicklungslabors namhafter Hersteller der Konsumgüter- und Entertainmentindustrie zurückzuführen.

Mit der zunehmenden Verbreitung von IEEE 1394 durch den Konsumermarkt ist auch die Diskussion über den möglichen Einsatz in anderen Zweigen der Industrie gewachsen. Hier besteht ein großes Interesse für die Nutzbarmachung in Büroanwendungen und Lösungen für die Gebäude- und Anlagenautomatisierung. Grund dafür ist ebenfalls die Suche nach preiswerteren Ersatzsystemen, um bisherige proprietäre bzw. aufgabenspezifische Einrichtungen abzulösen **und** zu vereinen. Für einen solchen Wandel sind die Forderungen an einen universellen Bus folgendermaßen zu formulieren.

Elektromechanik und Topologie

- Übertragungsstrecken von wenigen Zentimetern für Modulkomponenten bis zu 100 m zwischen zwei räumlich getrennten Geräten im Fernbereich
- Installation parallel zu Stromversorgungskabeln mit beliebigen Verzweigungen
- hohe Übertragungsgeschwindigkeiten für die Integration von Breitbandanwendungen
- problemloses Hinzufügen und Entfernen von Geräten

- extrem hohe Störsicherheit gegenüber elektromagnetischen Einflüssen (EMV)
- keine externen Leitungsabschlüsse

Kommunikationsmodell

- autonom funktionsfähige Geräte, also unabhängig von Kopf- oder Masterstationen
- nachrichtenorientierte Dienste für die Unterstützung von Aufgabenteilung (verteilte Objekte)
- datenorientierte Dienste für deterministische Informationsverteilung — der Automatisierungstechniker verlangt hier nach 8 kHz Buszyklen
- keine manuelle Adreßvergabe

Neben diesen technologischen Forderungen spielen auch wirtschaftliche Aspekte für den Einsatz eines Bussystems eine große Rolle. So sollte für das System ein offener Standard existieren, um die Austauschbarkeit von Komponenten zu gewähren. Die Anschaltkosten müssen durch eine hohe Verfügbarkeit der Produkte klein ausfallen. Stützend auf die genannten Punkte soll im weiteren Verlauf ein von mir erdachtes Einsatzbeispiel von IEEE 1394 in der industriellen Automatisierung vorgestellt werden. Dazu wird dieses System zunächst einem näheren Vergleich mit InterBus unterzogen, ja sogar mit der provokativen Frage der „Ersetzung“ konfrontiert. Die für den InterBus angesprochenen technischen Einzelheiten sind meiner Praktikumsarbeit aus dem 5. Semester entnommen [Lnz98].

3.1. IEEE 1394 als Ersatz von InterBus ?

Es scheint als etwas abwegig, einen für multimediale Anwendungen konzipierten Bus gleich als Austauschmedium für einen Feldbus anzusehen. Obwohl IEEE 1394 in seiner Realisierung und Spezifizierung noch nicht so weit vorangeschritten ist, einen bereits seit Jahren etablierten Feldbus ohne Abstriche zu ersetzen, lassen sich in Hinblick auf InterBus einige interessante Parallelen im Buskonzept finden. Eine zusammenfassende Gegenüberstellung beider Systeme ist im Anhang, Abschnitt B.1 auf Seite 100, zu sehen.

Baum-Topologie

Obwohl dem InterBus eine Ringstruktur zugrunde liegt, wird der physikalische Aufbau wieder auf einen Baum zurückgeführt. Der Baum hat sich in der Automatisierungstechnik als die anpassungsfähigste Verkabelungsvariante für Busse herausgestellt. Nicht zuletzt deshalb, um herkömmliche Kabelbäume zu ersetzen. Im InterBus werden zwei wichtige elektromechanische Aufbauvarianten unterschieden. Der *Fernbus* stellt die serielle Datenübertragung durch zwei twisted-pair Leitungen zwischen den einzelnen Knoten her. Innerhalb eines Knotens, einer sogenannten Busklemme, kann der Ring über den *Peripheriebus* in Form einer kurzen Stichleitung aufgetrennt und verzweigt werden. Ein solcher lokal begrenzter Abzweig wird ebenso wie die Backplaneausführung von IEEE 1394 für die Modularisierung der Geräteeinheiten benutzt. Im Gegensatz zu IEEE 1394 kann der Fernbus nicht beliebig stark verzweigen, bietet aber mit bis zu 400 m Kabellänge zwischen zwei Geräten eine weitaus höhere Wegstrecke. Selbst für die Ausdehnung eines 13 km langen InterBus kann der max. 72 m lange IEEE 1394 *noch* keine Konkurrenz darstellen. Das wird sich bald ändern, wenn der Erweiterungsstandard IEEE 1394b, der sich momentan noch in Arbeit befindet, die Kabellänge auf bis zu 100 m erhöht. Dann wird es auch möglich sein, LWL Technik, wie sie bei InterBus schon längere Zeit nutzbar ist, für den IEEE 1394 einzusetzen.

Buszyklus

Beide Systeme, sowohl IEEE 1394 als auch InterBus, bieten durch eine fest definierte Synchronisation in Form des Buszyklus die Voraussetzung für die zyklische Übertragung echtzeitrelevanter Daten. Dafür existieren jedoch zwei unterschiedliche Ansätze, die im Wesentlichen auf die Strukturierung der Bustopologie zurück-

zuführen sind.
Beim InterBus ist es immer notwendig, daß die für nur einen Busteilnehmer bestimmten Daten durch **alle** anderen Teilnehmer geführt und weitergereicht werden müssen. Diese Technik begründet den Aufbau des verwendeten *Summenrahmenprotokolls*. Wie der Name schon sagt, wird sich der Rahmen für die Daten mit zunehmender Geräteanzahl in Summe vergrößern. Das hat zur Folge, daß der Buszyklus stark vom jeweiligen Busaufbau abhängig sein wird und die Periodenlänge nicht allgemein definiert werden kann. Die Zeitdauer für ein Zyklus ist also eine Funktion in Abhängigkeit der Geräteanzahl und kann letztendlich Werte zwischen 1,2 ms und 17 ms erreichen. Genaueres dazu ist in [Lnz98] 7.2.6 Seite 36 nachlesbar.
Der Buszyklus von IEEE 1394 ist anders organisiert. Er basiert auf einer gleichzeitigen systemweiten Synchronisation aller Geräte durch ein Verwaltungsgerät, daß alle 125 μs, also mit einer Frequenz von 8 kHz, den Beginn eines Buszyklus markiert. Diese Periode ist absolut unabhängig von der tatsächlichen Geräteanzahl oder deren Anordnung und kann somit als allgemeingültig angesehen werden.

Art der Daten und deren Sicherung

Trotz der unterschiedlichen Zykluszeiten gleichen sich die zwei Arten enthaltener Daten. Ein Buszyklus bietet immer ausreichend Möglichkeiten, neben den für die Automatisierungstechnik wichtigen Prozeßdaten auch Parameterdaten zu übertragen. Es wird sichergestellt, daß alle Prozeßdaten innerhalb einer Periode aufgefrischt werden können und wenn nötig zusätzlich, also asynchron, Daten für die Parametrisierung und Kommandierung von Geräten übertragen werden können.
Beim InterBus sind die Prozeßdaten Inhalt der *Datentelegramme*, wo hingegen bei IEEE 1394 diese Daten als isochron angesehen werden müssen und immer zu Beginn eines Zyklus **garantiert** übertragen werden. Die Sicherung erfolgt in beiden Fällen über eine CRC Summe, da wie bei allen Daten-*Übertragungs*-Systemen davon ausgegangen werden muß, daß verfälscht empfangene Daten nachgefordert werden können ([Dank94]). Diese Sicherungsform dient der Erkennung von Bündelfehlern und berührt nicht die weiteren Möglichkeiten durch die inhaltliche Abstraktion der Daten wie es bei InterBus realisiert wird (Test auf Reihenfolge, Handshake-Signale, Start-Stop-Bits und Loopbackword). Bei IEEE 1394 müssen solche erweiterten Test- und Kontrollfunktionen wenn nötig in höheren Protokollschichten realisiert werden. Eine Möglichkeit wäre die Nutzung des Synchronisationscodes, der durch IEEE 1394

als anwendungsspezifisch definiert ist.
Die Kommunikation über den Parameterkanal bei InterBus ist im Wesen dem des asynchronen Request-Response-Verfahren von IEEE 1394 gleich. Es existiert immer ein Server-Client-Verhältnis zwischen zwei Kommunikationspartnern, nur mit dem Unterschied, daß bei InterBus die Abwicklung dieses Datenverkehrs gezwungenermaßen über das Mastergerät erfolgen muß, hingegen bei IEEE 1394 jedes Gerät in der Lage ist, **direkt** eine Verbindung zu einem anderen aufzubauen.

3.2. Vorteile und Probleme beim industriellen Einsatz

Wie diese drei Gemeinsamkeiten zeigen, lassen sich beide Systeme aus Sicht des Buskonzepts einfach miteinander betreiben ohne dabei die auszeichnenden Eigenschaften der Echtzeitfähigkeit und der zwei unterschiedlichen Datenbehandlungen zu verlieren. Es ist aber offensichtlich, daß in vielen Bereichen, speziell der Sensorik und Aktorik, der InterBus durch seine Schlichtheit ein breiteres Einsatzgebiet findet. Die Kosten für eine einfache digitale I/O Baugruppe mit InterBus fällt bei weitem geringer aus, als unter Nutzung von IEEE 1394. Der InterBus bietet zusätzlich noch die Möglichkeit, neben den standardisierten Kupferkabeln oder LWLs auch Hohlleiter oder Schleifringläufer als Übertragungsmedium zu nutzen. Diesen Bereich wird IEEE 1394 niemals ersetzen können, was auch nicht sein primäres Ziel ist.

Vielmehr kann man durch den zusätzlichen Einsatz von IEEE 1394 die Funktionsvielfalt und die Leistungsfähigkeit einer Automatisierungsanlage steigern. Erst durch Ausnutzung der hohen Übertragungsrate kann die immer stärker werdende digitale Bildverarbeitung integraler Bestandteil von Feldbusanwendungen werden. Viele Lösungen in diesem Bereich waren bisher nur auf lokale Steuereinrichtungen beschränkt. Das kann sich durch IEEE 1394 insofern ändern, daß die digitalen Bildinformationen global verfügbar werden und damit von vielen Einzelgeräten interpretiert werden können. Ein erstes brauchbares Beispiel dafür ist eine industrietaugliche digitale Kamera mit IEEE 1394 Anschluß von Sony (DCM 250).

Mit der Einführung von IEEE 1394 in industrielle Bereiche sehen die meisten Kritiker die viel zu geringen Ausdehnung dieses Systems als Problem an (max. 72 m).

Dieses Hindernis wird jedoch mit dem bevorstehenden Abschluß der Standardisierung IEEE 1394b beseitigt sein. So wird man bald mit einfachen UTP5 Kabeln und einer damit verbundenen Übertragungsrate von 100 Mbps eine Ausdehnung von 16 km erreichen. Durch LWL Technik kann man auch noch längere Strecken mit höheren Datenraten überwinden, was sich für den Aufbau von zuverlässigen High-Speed Verbindungen in höheren Schichten der industriellen Automatisierung gut eignet. Doch die Entwicklung bis zu einsatzfähigen Komponenten wird noch einige Zeit in Anspruch nehmen. Erste Erfolge wurden vorerst nur von der Firma NEC Electronics erziehl, die seit Anbeginn maßgeblich an der Standardisierung des Einsatzes von LWL Technik beteiligt ist. Sie vertreibt als Ergebnis ihrer Forschungen einen LWL-Umsetzer, der die elektrischen Signale eines IEEE 1394 Kabel-PHY auf LWL-Kabel umsetzt (und umgekehrt). So könnten schon heute größere Distanz als nur 4,5 m überwunden werden, jedoch mit einem erheblichen Mehrkostenaufwand.

Interessant ist die Tatsache, daß die aktuellen Diskussionen und Einsatzbeispiele von Ethernet in der Industrie wesentliche Wegbereiter von IEEE 1394 sein könnten. Die durch Switched-Ethernet aufgebaute Infrastruktur in Industrieanlagen könnte ohne gravierendem Eingriff an der Verkabelung auch von IEEE 1394 genutzt werden, denn wie oben schon beschrieben, kann IEEE 1394 zukünftig auch auf UTP5-Basis arbeiten. An die Stelle der Switchtechnologie des Ethernets kann die noch in Bearbeitung befindliche Standardisierung IEEE 1394.1 treten.

Ein weiterer Kritikpunkt in bezug auf den Einsatz von IEEE 1394 als Feldbus wird in den fehlenden höheren Protokollschichten gesehen. Das würde ich differenzierter betrachten wollen. Richtig ist, daß weder durch den Standard IEEE 1394 noch durch irgend einen seiner Folgestandards eine höhere Protokollschicht vorgeschrieben wird. Nach dem OSI-Modell schließen die Definitionen an der Schicht 2 ab. Doch diese Eigenschaft bringen auch andere Systeme mit sich. Ich glaube eher, daß eine Anbindung an renommierte und etablierte Netzwerkprotokolle, wie es z.B. das häufig eingesetzte *Internet Protokoll* (*IP*) ist, den Schritt zur funktionalen Integration in bestehende Systeme bedeutet. Diese Entwicklungstendenz wurde schon 1997 durch das IETF rechtzeitig erkannt, so daß seit Ende 1999 die *IP-over-1394* Spezifikation existiert, welche beschreibt, wie Datenströme nach dem Internet Protokoll über den IEEE 1394 Bus transportiert werden. Der daraus entstehende Vorteil ist offensichtlich. Mit dem stetig wachsendem Verlangen nach verteilten Funktionsobjekten für

die gleichzeitig konspirative und autonome Problemlösung von Automatisierungsaufgaben müssen die dafür notwendigen Unterbaukonstruktionen für die Verständigung zwischen den verteilten Objekten einem akzeptablen, weit verbreiteten und offen gelegten Standard entsprechen, um die Interoperabilität zwischen den verschiedensten Produkten auch in Zukunft sicherzustellen. Mit *IP-over-1394* erhält man dann den Vorteil, Querentwicklungen, z.B. aus den Bereichen der Informationstechnik, auch in die Belange der industriellen Anlagensteuerung einfließen zu lassen. Ich denke hier speziell an die in letzter Zeit immens zunehmenden Entwicklungen rund um CORBA. Der Weg hin zur Integration von IEEE 1394 als Transportmedium in die Welt von OPC ist somit auf gleiche Weise geebnet.

Für den Bereich der Prozeßdatenprotokollierung durch Drucker oder Massenspeicher und der zusätzlichen Übertragung von digitalen Audio- und Videosignalen existieren bereits busspezifische Protokolle. Die wichtigsten davon möchte ich kurz nennen. Alle aufzuzählen währe mir angesichts der großen Vielfahlt und dem immensen Umfang existierender Protokollspezifizierungen nicht möglich. Einen tieferen Einblick kann hier die Literatur vermitteln [Tee99].

- Das ***Seriall Bus Protocol 2*** (*SBP-2*) entstammt dem SCSI-3 Architekturmodell [T10] und kann für die Kommunikation mit Computerperipherie, wie Festplatten, Drucker oder Scanner, genutzt werden. Der Zugriff auf solche Geräte erfolgt hierbei durch einen direkten Speicherzugriff über IEEE 1394 in oder aus dem jeweiligen Gerät.

- Das ***Direct Printing Protocol*** (*DPP*) ist eines der drei verschiedenen Möglichkeiten, die Kommunikation mit einem Drucker herzustellen. Der Zugriff gestaltet sich dabei als sehr einfach, da im Gegensatz zu SBP-2 oder AV/C die Notwendigkeit anderer Protokollschichten nicht gegeben ist.

- Durch die Spezifizierung ***Audio Video Control*** (*AV/C*) wird der Versuch unternommen, die stark verwucherten Einzeldefinitionen der verschiedenen Arbeitsgruppen aus der A/V-Welt wieder zu einem in sich geschlossenen Dokument zusammen zu fassen. Momentan existieren noch über 30 individuelle Spezifikationsschriften für die A/V Geräteklassen Aufnahme (Kameras), Speicherung (DVD oder CD) und Wiedergabe (Bildschirme, Drucker).

Wie man sieht, ist das Spektrum an höheren Protokollschichten unübersichtlich groß und teilweise überlappend. Daß nur Teile davon in industriellen Umgebungen genutzt werden können, zeigen die eingeschränkten Geräteklassen auf die obige Spezifizierungen zielen. Daher haben sich drei vielversprechende Arbeitsgruppen innerhalb der 1394TA gebildet, um den Einsatz von IEEE 1394 in der Industrie, Instrumentierung und Kfz-Elektronik vorzubereiten. Für die Instrumentierung ist im Oktober 1999 eine erste Spezifizierung für den Transport von IEEE 488.2 Befehlen über den IEEE 1394 Bus verabschiedet worden, das *Instrument and Industrial Control Protocol* (*IICP*) [Kro99]. Für den Industrieeinsatz wird an einem neuen robusteren Steckersystem gearbeitet und im Kfz-Bereich laufen die Bestrebungen zu Diagnose- und Kommunikationssystemen auf Basis von IEEE 1394 in Verbindung mit LWL Technologie auf Hochtouren.

All diese Bestrebungen stecken noch in den Anfängen, so daß eine sofortige Nutzung von IEEE 1394 als Übertragungsmedium in der Automatisierungstechnik schnell zu eigenständigen Lösungen führen kann. Ein ernstzunehmendes Einsatzgebiet ist die Antriebstechnik. Hier hätte IEEE 1394 eine gute Chance so eigenständige Systeme wie SERCOS abzulösen, denn neben der Steuerung und Synchronisation von Servoantrieben kann IEEE 1394 zusätzlich übrige I/O Kommunikation und multimediale Datenverbreitung innerhalb **einer homogenen** physikalischen Umgebung koordinieren. Beispielsysteme dazu päsentieren die beiden Firmen ORMEC[1] und Nyquist Industrial Control[2], die unabhängig voneinander ein Kommunikationsmodell für die Synchronisation und Reglung ihrer Servoantriebe unter Nutzung von IEEE 1394 entwickelten. Dabei wird der Regelkreis auf dem Weg von der Regeleinrichtung zur Regelstrecke durch IEEE 1394 ersetzt, also ortsunabhängig gestaltet. Somit ist das erziehlen einer 8 kHz schnellen Abtastfrequenz für den digitalen Regler unter gleichzeitiger Nutzung eines „Feldbusses“ möglich. Zu hoffen ist, daß die Ergebnisse und Erfahrungen beider Produkte in eine zukünftige Spezifizierung münden.

[1] http://www.ormec.com/
[2] siehe Elektronik Heft 12 1999, Seite 30

3.2.1. Mögliche Einsatzgebiete

Aus den bisherigen Betrachtungen resultieren einige interessante Anwendungsfälle, bei denen IEEE 1394 als wichtiger Baustein zum Einsatz kommen könnte:

1. Industrielle Bildverarbeitung an Orten mit ausergewöhnlichen Umwelt- und Arbeitsbedingungen (z.B. extreme Hitze bei der Glasfertigung oder EMV Testlabore).

2. Antriebstechnik für Mehrachsrobotor und Werkzeugmaschinen, wobei die dafür notwendigen Regelkreise über den Bus betrieben werden.

3. Echtzeitdatenbanken können direkt im gleichen Netzwerk wie die Feldgeräte existieren und passiv alle Aktivitäten aufzeichnen aber auch aktiv Informationen bereit stellen.

4. Eine SPS kann durch das gleiche Bussystem, wie es für die angeschlossenen Feldgeräte benutzt wird, modular gestaltet werden, um die Funktionsmöglichkeiten zur Laufzeit zu erweitern (z.B. Speicher, Hardwaretimer, Sichtgeräte).

5. Im Flug- und Fahrzeugbau kann neben den Kommunikationswegen zwischen einzelnen elektronischen Steuerbausteinen das gleiche Bussystem für die Integration von Audio- und Videosystemen genutzt werden. Auch weitere Informationssysteme aus dem Internet sind denkbar.

3.3. Fiktives Anwendungsbeispiel

Nach den zahlreichen Abschätzungen und Möglichkeiten möchte ich meine Überlegungen über den Einsatz von IEEE 1394 in der Industrie anhand eines fiktiven Anwendungsbeispiels konkretisieren. In Abbildung 3.1 auf der nächsten Seite ist eine Produktionszelle dargestellt, in der ein beliebiger abgrenzbarer Teilprozeß eines Produktionsflusses gesteuert werden soll. Die Steuerung und Kontrolle übernimmt dabei eine SPS, die aus Sicherheitsgründen redundant aufgebaut werden kann (Warte 1 und Warte 2). Die für diese Steuerung notwendigen digitalen und analogen Sensorwerte werden durch „Datensammler“ aus der Teilprozeßumgebung herausgeholt und anschließend über den IEEE 1394 an beide Warten verteilt. Daß diese Daten

über einen InterBus extrahiert werden, kommt den niedrigeren Anschaltkosten zugute und sichert die weitere Nutzung bestehender Installationen. Komplexere Steuer- und Regelaufgaben, wie die Koordination des mehrachsigen Antriebes für die Be- und Entladung oder den Teiletransport, werden durch die direkte Regelung über IEEE 1394 realisiert.

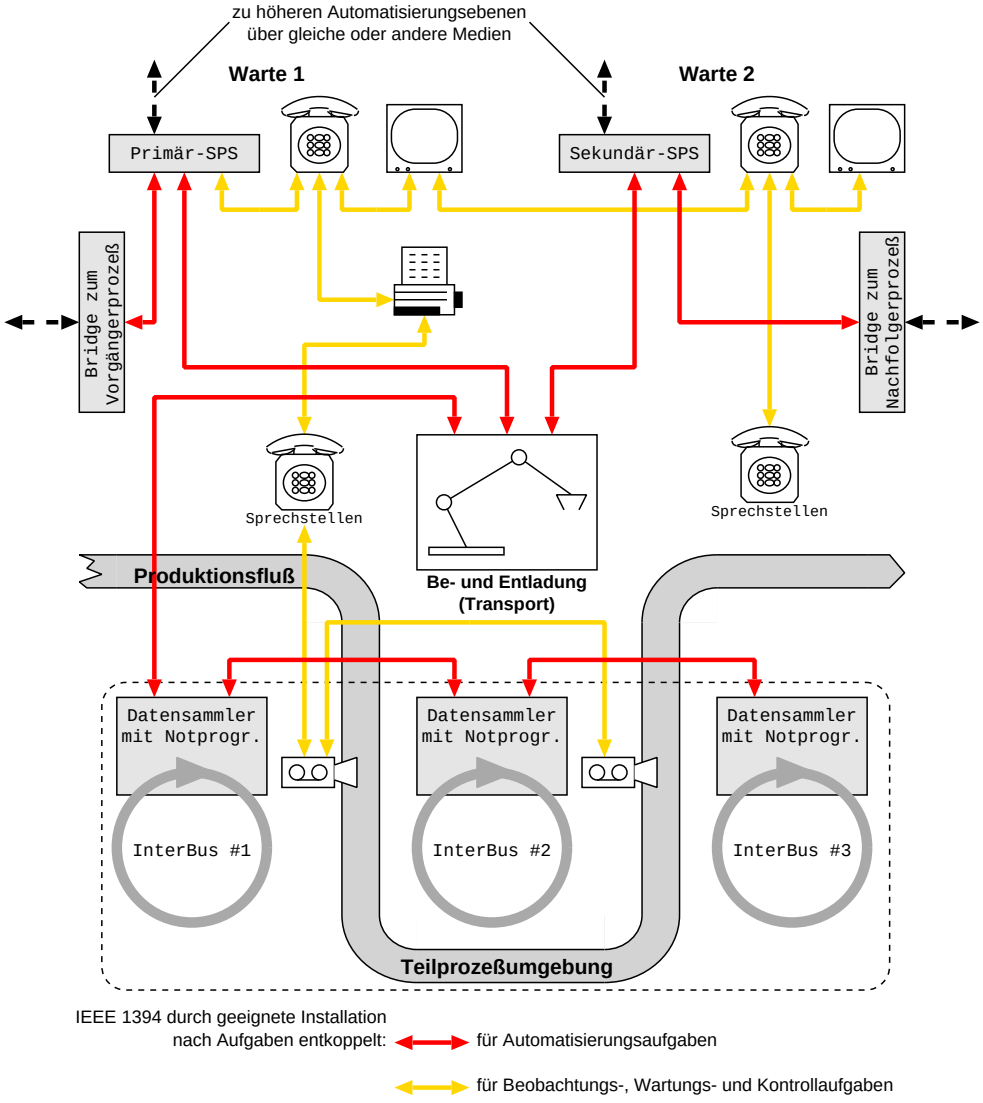

Abbildung 3.1.: IEEE 1394 und InterBus in einer Produktionsanlage

Sollte eine Verständiung mit benachbarten Produktionszellen notwendig sein, so kann durch Einsatz einer geeigneten Bridge nach IEEE 1394.1 der Informationsaustausch mit dem Vorgänger oder Nachfolger realisiert werden. Eine notwendige

Abstimmung mit höheren Automatisierungsebenen würde auf gleiche oder ähnliche Art und Weise geschehen.

Da bereits in der Steuerebene der IEEE 1394 Bus zum Einsatz kommt, kann die Leistungsfähigkeit dieser Ebene rapide Gesteigert werden. Der riesige Unterschied zwischen den Übertragungsgeschwindigkeiten von InterBus und IEEE 1394 lassen eine einfache Bündelung der Datenströme zu. Man könnte mit einem 100 Mbps schnellen IEEE 1394 Bus bis zu 43 InterBus Kanäle mit je 2 Mbps zusammenfassen. Ein weiterer Vorteil eröffnet sich in der parallelen Nutzung des IEEE 1394 Busses durch Anlagenteile, die im klassischen Sinn der Automatisierungstechnik nicht in direktem Zusammenhang mit dem zu steuernden Prozeß stehen. So können Sprechverbindungen (mit oder ohne Bild) und Beobachtungskameras in einer größeren Anlage durch das gleiche Kommunikationsmedium miteinander verbunden werden. Das spart Kosten. Die Bilddaten von Beobachtungskameras können aber auch gleichzeitig für eine digitale Bildverarbeitung genutzt werden, um den Produktionsprozeß zu beeinflussen oder eine Qualitätssicherung bereits innerhalb der Steuerung zu realisieren.

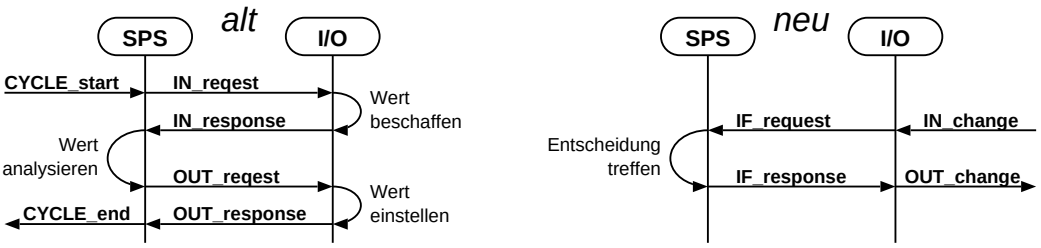

Abbildung 3.2.: altes und neues Kommunikationsmodell für Steuerungen

Durch den Einsatz eines modernen und flexiblen Kommunikationsmediums zwischen SPS und I/O Baugruppe kann eine effektive Ablaufform für die Verarbeitung von Eingangsinformationen zu daraufhin notwendigen Ausgangsaktionen entstehen. Bisher wurden die Eingangsinformationen durch die SPS angefordert und aufgrund deren Zustände eine Ausgangsbeschaltung ermittelt und ausgegeben. Mit zunehmender Intelligenz in den I/O Baugruppen können diese selbstständig bei Veränderungen ihrer Eingänge reagieren, indem sie entweder selber schon wissen, was zu tun ist, oder sie ihre übergeordnete SPS befragen. Die SPS verändert sich somit zu einem Wissensspeicher, eine Art Server für Regelwerke. In Abbildung 3.2 auf der

vorherigen Seite sind die klassische und die neue Ablaufform gegenübergestellt. Man kann leicht erkennen, daß der notwendige Kommunikationsaufwand bei der neuen Variante halb so groß ist, als bei der alten. Das ist sehr günstig, da mit steigender Anzahl an I/O Baugruppen auch der Kommunikationsaufwand linear anwächst, der so gedämpft werden kann.

4. Systemintegration von IEEE 1394

Neben der genauen Untersuchung der Funktionsweise des Bussystems und der Konfrontation dieser mit den Belangen von automatisierungstechnischen Einsatzgebieten sollen in diesem Kapitel die bisher existierenden Integrationsmöglichkeiten in embedded Systeme aufgezeigt werden. Dabei möchte ich im allgemeinen auf die wichtigsten Komponenten und Teilsysteme kurz eingehen, um somit die Prinzipien genannt zu haben. Im weiteren Verlauf wird der Chipset „PCILynx“ von Texas Instruments genauer vorgestellt. Dieser bietet Eigenschaften, die speziell für den Einsatz in Kompaktgeräten mittlerer Komplexität von Nutzen sein können.

Wie sich noch herausstellen wird, nimmt die Firmware einen nicht zu unterschätzenden Anteil der gesamten IEEE 1394 Integration ein. Auf dem Weg zum funktionstüchtigen Gerät ist sie aber letztendlich nur ein notwendiger Baustein. Sie wird in einem Gesamtkonzept bei der Problemlösung mittels IEEE 1394 immer eine untergeordnete Rolle spielen. Trotzdem kommt man in den meisten Fällen ohne sie nicht aus. Dazu möchte ich die existierenden Varianten in Abhängigkeit und Unabhängigkeit zu einem Betriebssystem nennen.

Der letzte Abschnitt wird sich mit dem Aufbau eines IEEE 1394 Demonstrators beschäftigen. Er soll das Zusammenspiel von aktuell verfügbaren Hard- und Softwarekomponenten veranschaulichen und somit den entstehenden Aufwand bei der Systemintegration von IEEE 1394 aufzeigen.

4.1. Hardware — Chipsets

Schon mit der Darstellung des Kommunikationsmodells in Abbildung 2.1 auf Seite 22 wurde gezeigt, daß die Hardware einer IEEE 1394 Anbindung aus zwei Teilen besteht, dem Physical Layer und dem Link Layer. Die reale Ausführung

dieser Teile wird ebenfalls durch zwei Controllerbausteine umgesetzt. Es handelt sich um den *PHY*, mit den Aufgaben der physikalischen Schicht, und den *Link Layer Controller* (*LLC*). Beide Bausteine sind substantiell notwendige Komponenten und über das LLC-PHY-Interface miteinander verbunden. Dieses Interface wurde erst durch IEEE 1394a fest definiert. Durch die zunehmende Integrationstiefe werden momentan immer öfter beide Controller in einem Chip zusammengefaßt.

4.1.1. Historischer Verlauf

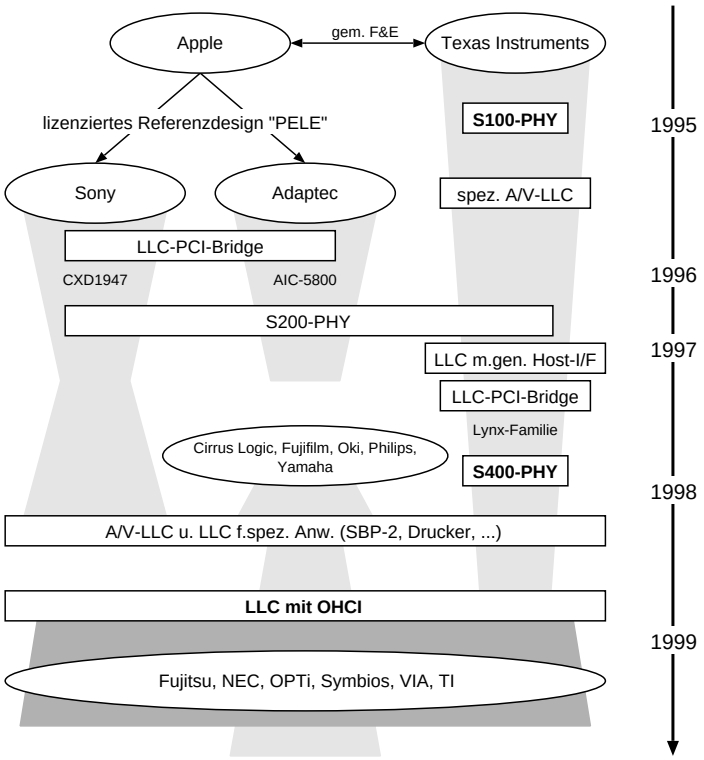

Abbildung 4.1.: Mainstream der wichtigsten Chiphersteller

Seit den ersten funktionsfähigen Labormustern der späten 80er Jahre bis hin zur heutigen industriellen Großfertigung von IEEE 1394 Baugruppen hat die Entwicklung von Hard- und Softwareunterstützung für dieses Bussystem mehrere Etappen durchlaufen. Angesichts der großen Komplexität des Standardwerkes ist zu verstehen, daß am Anfang dieser Entwicklung nur ein paar wenige Firmen der Hardwarebranche das Knowhow und die Möglichkeiten für die Produktion der ersten Chips

als Grundlage der Systemintegration besaßen.

Im Jahr 1995, zum Zeitpunkt der Verabschiedung von IEEE 1394, existierte nur ein S100 PHY und ein paar wenige spezielle LLC für Audio- und Videoanwendungen (A/V) von Texas Instruments, welche aus der gemeinsamen Forschung und Entwicklung mit Apple hervorgegangen sind. Wie ich aus diversen Email-Listen entnehmen konnte, existierte zum damaligen Zeitpunkt ein von Apple lizenziertes Referenzdesign für den Aufbau eines LLC-Cores mit dem Kunstnamen *„Pele“*. Dieses wurde in Form von Derivaten durch Sony und Adaptec in Chips umgesetzt, so daß die ersten IEEE 1394-PCI-Bridges entstanden.

Mit der Bereitstellung verschiedener S200 PHYs kamen 1996 und 1997 die ersten PCI-Add-On Karten von Adaptec auf den Markt. Diese basieren noch bis heute auf dem „Pele“ Chipset AIC-5800. Parallel dazu entstanden bis Anfang 1998 bei Texas Instruments mehrere LLC Chips mit unterschiedlichen Schnittstellen zum jeweiligen Hostsystem. So existiert heute eine vielfältige Palette für den LLC Anschluß an PCI-Umgebungen und proprietären Mikrocontrollerbussen.

1998 stellte Texas Instruments den S400 PHY fertig, womit erstmals die von vielen datenorientierten Geräten geforderte Durchsatzrate von mehreren 10 MBps auch mit IEEE 1394 erreichbar war. Das legte den Grundstein für die Projektierung und Fertigstellung von Massenspeichermedien, wie Festplatten, und Datenein- bzw. ausgabeeinheiten, wie Scanner, Digitalkamera und Drucker. Die dazu notwendigen LLCs mit den hierfür abgestimmten Zusatzfunktionen betreffend der höheren Protokollschichten, werden seitdem von den Chipherstellern Cirrus Logic, Philips, Sony, Yamaha uvm. bereitgestellt.

Im Anhang, Abschnitt C.1 auf Seite 104, sind alle mir bekannten Chiphersteller mit ihren IEEE 1394 Produkten aufgelistet. Die Daten entstammen einer detailierten Internetrecherche, wobei ich Wert auf den Vergleich und die Verdeutlichung von Unterschieden zwischen den Chipsets in Bezug auf die Verwendbarkeit gelegt habe. Diese Übersicht soll als einfacher Leitfaden und Einstiegspunkt in die Auswahl von Chipsets dienen. Man wird nicht umhinkommen, sich bei besonderen Problemfällen mit einigen Chipsets tiefgreifender auseinanderzusetzen.

Die für das Betreiben dieser Chipsets notwendige Firmware wurde ausschließlich von den Hardwareherstellern bereitgestellt und gepflegt, so daß eine direkte Austauschbarkeit und Interoperabilität zwischen verschiedenen Hardware- und

Softwareumgebungen nicht immer gegeben war. In aufgabenorientierten embedded Systemen, wie z.B. Kameras, werden ohnehin spezielle Chipsets zusammen mit darauf vom Systemintegrator abgestimmter Firmware zum Einsatz kommen. Dagegen existieren eine Vielzahl von Chipsets für die Einbettung in bestehende Computersysteme; größtenteils über IEEE 1394-PCI-Bridges mit jeweils unterschiedlichen Programmierschnittstellen. Diesen Umstand haben Microsoft, als bisheriger Marktführer im Bereich IT-Betriebssysteme, zusammen mit weiteren namhaften Computerherstellern, wie Sun, Compaq und Apple, rechtzeitig erkannt und sind 1998 gezielt an die existierenden Chip-Hersteller herangetreten, um für die Bindestelle zwischen Hardware und Firmware eine einheitliche Schnittstelle zu definieren und zu realisieren. Das so entstandene *Open Host Controller Interface*, kurz *OHCI* oder *OpenHCI*, war letztendlich Mittel zum Zweck, die Auswüchse von Treiberneuentwicklungen für ein und dasselbe System schon im Vorfeld der Gesamtentwicklung zu vereiteln.

Der Erfolg dessen brachte in der gesamten IT-Welt einen schlagartigen Boom der Hard- und Softwareentwicklung mit sich. Jetzt war es für viele weitere Chiphersteller möglich, durch die Stabilität eines Standardwerkes für den Bus und einer eindeutigen Spezifikation für das Aussehen von Hard- und Software in die Entwicklung und Produktion zu investieren. Um die Bereitstellung der notwendigen Firmware bzw. Treiber kümmern sich nun ausschließlich Softwarehäuser. Dieser gewollte Trend in Richtung Interoperabilität zwischen Hard- und Software spiegelt sich auch in den Zahlen der heute existierenden Chiphersteller für IEEE 1394 wider. So stellen etwa die Hälfte aller Anbieter ausschließlich Chipsets auf Basis des OHCI her.

4.1.2. Aufbau der Kernkomponenten

4.1.2.1. Der PHY — Physical Interface

In der Regel besitzt ein Kabel-PHY 1 oder 3 Ports. Innerhalb von Schaltkreisfamilien existieren auch PHY-Chips mit 2, 4 und 6 Ports. Ein PHY besteht im wesentlichen aus den in Abbildung 4.2 dargestellten Funktionsblöcken. Kern eines PHY bildet die Zustandsmaschine für Arbitrierung und Bildung bzw. Auswertung der *Arbitration Signals*. Hinzu kommen neben der Datencodierung, -decodierung die intern gesteuerten Strom- und Spannungsquellen für die Generierung der Geschwindigkeitscodierung und des TPBIAS als Linkkontrolle.

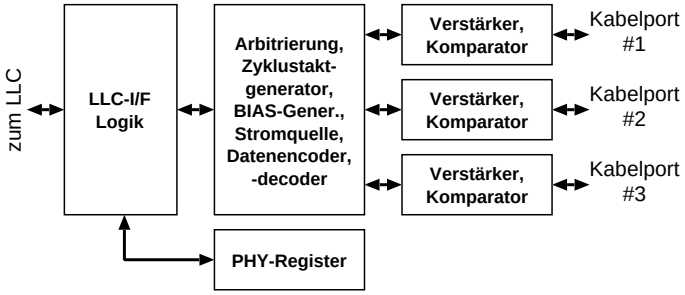

Abbildung 4.2.: Funktionsblöcke eines Kabel-PHY

Durch die in jedem Kabelport vorhandenen Verstärker werden Sendedaten an die Leitungspegel angepaßt. Die Komparatoren dienen dem detektieren von *Arbitration Signals*. Beide Funktionen sind im Empfangs- bzw. Sendeteil eines jeden Ports untergebracht.

Ein Kabel-PHY benötigt immer einen externen 24,576 MHz Quarz, dessen Takt über eine interne PLL den jeweiligen Sende-/Empfangstakt stabilisiert (S100. . .S400). Empfangene Daten werden ebenso auf den internen Arbeitstakt von ca. 50 MHz (genauer S50 = 49,152 MHz) synchronisiert und für den Weitertransport zum LLC zwischengepuffert. Diese Pufferung erfolgt nur, wenn der serielle Datenstrom aus dem Kabel auf 2, 4 oder 8 Datenpfade des LLC-PHY-I/F umgesetzt werden soll. Diese Auslegung ist Implementationsabhängig. Je mehr Datenpfade zum LLC existieren, desto höher ist die Bandbreite an dieser Stelle. Bei einem S400 PHY wird man alle 8 Pfade benötigen, da 8 multipliziert mit dem Arbeitstakt S50 eine Bandbreite von S400 ergibt.

Durch das LLC-PHY-I/F kann man über den LLC auf die PHY-Register zugreifen. Sie beinhalten Informationen über die Anzahl verfügbarer Ports, deren Linkstatus und Geschwindigkeit sowie Stromversorgungseigenschaften. Manche Daten, wie z.B. jene für das Powermanagement, müssen durch Firmwarefunktionen des Hostsystem vorgegeben werden. Die Struktur und der Umfang dieses Registerfeldes wurde durch den Erweiterungsstandard IEEE 1394a verändert. Entsprechende Treiber müssen das berücksichtigen.

galvanische Trennung

Die bestehende Möglichkeit, einen PHY nicht nur durch das Hostsystem, sondern auch durch entfernte Geräte über das Buskabel mit Energie zu versorgen, verlangt zwingend eine galvanische Entkopplung des PHY, da durch die Nutzung der gemeinsamen „Bus-Masse“ als Bezugspotetial für alle PHYs bei fehlender Entkopplung Masseschleifen über die Erdungspunkte der einzelnen Geräte entstehen würden. Um das zu verhindern, wird die lokale Stromversorgung über einen DC/DC-Wandler an den PHY herangeführt und zusätzlich mit den entsprechenden Leitungen des Buskabels verbunden. Wird der PHY nur über den Bus gespeist, dann kann der DC/DC-Wandler entfallen. Eine Längsreglung der Stromversorgung aus dem Bus ist dennoch notwendig.

Wahlweise können auch die Signalleitungen des LLC-PHY-I/F über Induktivitäten oder Kapazitäten galvanisch getrennt werden. Die letztere Möglichkeit wird normalerweise bevorzugt, da der Platzbedarf viel geringer ausfällt. Für den Erhalt der Pegel auf diesen Leitungen werden Pull-Up und Pull-Down Widerstände oder Busholder Schaltkreise benutzt.

4.1.2.2. Der LLC — Logical Link Control

Ein LLC hat grundsätzlich das Aussehen wie in Abbildung 4.3 auf der nächsten Seite. Die Kernkomponente stellen die Teile Sender, Empfänger, CRC Bildung und Kontrolle, Buszyklussteuerung sowie das LLC-PHY-I/F dar. Innerhalb der Anbindung zum PHY werden wenn nötig eine Parallel-Seriell- bzw. Seriell-Parallel-Wandlung vorgenommen. Die Buszyklussteuerung übernimmt die Arbeiten, die für die Generierung von *Cycle Start* Paketen im 8 kHz Takt durch einen *Cycle Master* benötigt werden. Die Taktbasis für den hierzu existierenden Zeitgeber wird entweder

durch den Arbeitstakt des PHY über das LLC-PHY-I/F oder eine weitere externe Taktquelle bereitgestellt.

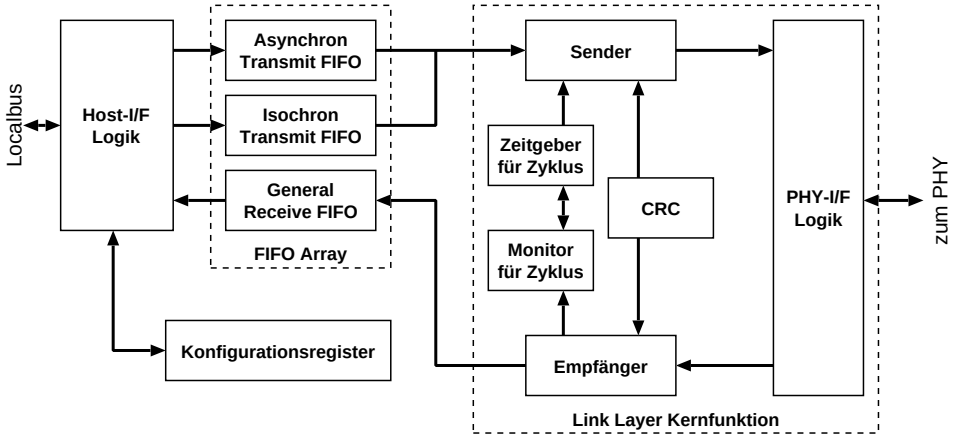

Abbildung 4.3.: Funktionsblöcke eines LLC (allgemein)

Dem Link Layer Kern folgend besitzen so gut wie alle LLC-Chips ein FIFO-Array. Mir ist nur ein Chipset von Oki Semiconductor bekannt, bei dem dieser FIFO optional als externer Baustein benutzt wird. Die FIFO-Bank ist immer in drei separat und autonom kontrollierbare FIFOs untergliedert. In manchen Chips ist die jeweilige Größe programmierbar in anderen fest vorgegeben. Grundsätzlich dienen sie der Beschleunigung und Synchronisation des Datenaustauschs zwischen dem übergeordneten Hostsystem und dem obersten Berührungspunkt des IEEE 1394 Systems. Dazu existieren in Senderichtung je ein FIFO für asynchronen und isochronen Verkehr und ein gemeinsam genutzter FIFO für die Empfangsrichtung, da diese Daten ohnehin seriell empfangen werden, also je Buszyklus erst isochrone und anschließend asynchrone.

Der Anschluß des LLC-Chips an das Hostsystem erfolgt über das Host-I/F. Dieser Funktionsblock ist hier in der Abbildung symbolisch zu verstehen, denn je nach Integrationsumgebung ist diese Schnittstelle verschiedenartig aufgebaut. Es gibt Auslegungen für den Anschluß an einen Mikrocontroller oder Mikroprozessor über generische 8, 16 oder 32 Bit breite Datenbusse bis hin zu kompletten PCI-Architekturen. Daneben existieren sehr viele spezielle Schnittstellen für den Anschluß an DSP-Systeme oder Audio- und Videoprozessoren, die über ein abgetrenntes Interface für isochrone Datenströme mit dem LLC kommunizieren. Dazu mehr

im folgenden Abschnitt.

Über die Konfigurationsregister wird die Funktionweise des Chips und der Datenaustausch über das Hostinterface kontrolliert und gesteuert. In diesen Registern werden auch Link Layer Informationen bereitgestellt, welche innerhalb der Firmware für die Präsentation im CSR-Modell benötigt werden (z.B. die Status- und Zählerregister der Buszyklussteuerung). Die oben genannten PHY-Register werden über diese Konfigurationsregister zugänglich gemacht.

4.1.3. Die zwei wichtigsten Hardwaremodelle

Bei der Betrachtung der verschiedenen Möglichkeiten, eine IEEE 1394 Anbindung herzustellen, kristallisieren sich immer wieder zwei grundlegende Strukturen heraus (Abbildung 4.4). Sie unterscheiden sich im wesentlichen durch die Handhabung der asynchronen und isochronen Datenflüsse. Je nach Gestaltung des Hostinterfaces ist der Zugriff auf die zwei verschiedenen Formen entweder gebündelt über ein und denselben Bus (links) oder getrennt in einen asynchronen und einen isochronen Kanal möglich (rechts).

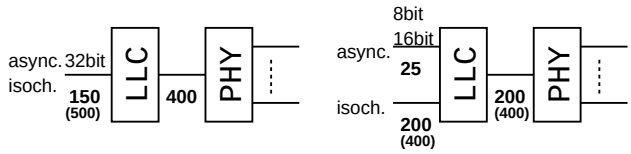

Abbildung 4.4.: zwei wesentliche Modelle der Hardwareintegration

Der Grund dafür liegt in der proportional abnehmenden Bandbreite bei abnehmender Anzahl von Datenbits in Verbindung mit dem zulässigen Bustakt am Hostinterface. Hinzu kommen noch Zeitverluste durch die Zerlegung der kleinsten durch IEEE 1394 übertragbaren Dateneinheit von 32 Bit in die jeweilige Bitbreite des Interface und der immer vorhandenen Verzögerung eines Schreib-/Lesezugriffs. Das Fehlen von DMA-Kanälen verschlechtert diese Situation zusätzlich. Dazu zwei einfache Rechenbeispiele zu unterschiedlichen LLC-Chips von Texas Instruments:

1. **16 Bit Hostinterface des TSB12LV31 (GPLynx)**:

   - zulässige Taktfrequenz am Host-I/F $f_{BClk} = 50MHz$
   - für die Umsetzung von 32 auf 16 Bit und umgekehrt werden etwa $n \approx 40$ Takte benötigt (laut Handbuch ist dies ein variabler Wert, der adreßabhängig ist)
   - die resultierende Bandbreite BB_{16} bei $n_{Bit} = 16Bit$ ergibt sich somit durch:

$$BB_{16} = n_{Bit} \cdot \frac{f_{BClk}}{n} \approx 16Bit \cdot \frac{50 \cdot 10^6 s^{-1}}{40} \approx 20Mbps$$

2. **32 Bit Hostinterface des TSB12C01 (High Speed Controller)**:

   - zulässige Taktfrequenz am Host-I/F $f_{BClk} = 33MHz$
   - für einen Buszugriff werden bis zu $n = 9$ Takte benötigt
   - die resultierende Bandbreite BB_{32} bei $n_{Bit} = 32Bit$ ergibt sich somit durch:

$$BB_{32} = n_{Bit} \cdot \frac{f_{BClk}}{n} = 32Bit \cdot \frac{33 \cdot 10^6 s^{-1}}{9} \approx 111Mbps$$

Allein diese zwei Rechenbeispiele zeigen, daß durch die richtige Wahl des Hostinterface, als Anbindung an das übergeordnete System, Geschwindigkeitssteigerungen um den Faktor 5 erreicht werden können. Durch den Einsatz von PCI-Technologie mit 33 MHz Bustakt, 32 Bit Datenbreite und den Vorzügen des DMA durch PCI-Masterfunktionalität kann man Bandbreiten von bis zu 500 Mbps erreichen.

Unabhängig davon, wie groß die Bandbreite des IEEE 1394 PHY für die Datenübertragung ausgelegt wird, der Flaschenhals entsteht oft auf dem Weg zu höheren Protokollschichten zwischen LLC und Hostsystem. Existiert nun so, wie in Beispiel 1, zwischen beiden Werten eine unverhältnismäßig große Kluft, so wird die isochrone Kommunikation schon hardwareseitig von der asynchronen getrennt. Interfacearchitekturen mit einem gewöhnlichen 8 oder 16 Bit breiten μC/μP-Bus weisen in der Regel eine solche Struktur auf. Diese LLC-Chips werden für die Anbindung isochroner Datenströme an Encoder bzw. Decoder oder sonstige datenverarbeitende Hardwarekomponenten benutzt. Der notwendige asynchrone Verkehr für

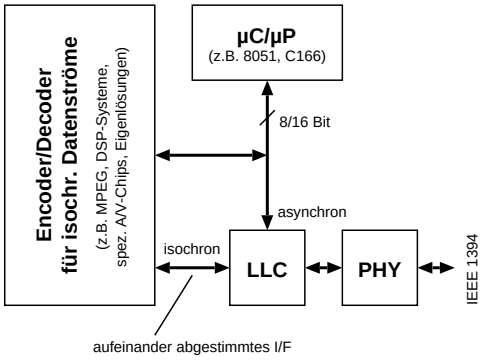

Abbildung 4.5.: Prinzipaufbau eines IEEE 1394-Links mit μC/μP

die Flußkontrolle erfolgt durch die Firmware im jeweiligen Mikrocontroller. Ein solcher Aufbau sieht dann prinzipiell so, wie in Abbildung 4.5 aus. Die Aufgaben des Mikrocontrollers kann u.U. auch „der“ DSP übernehmen, der für die Verarbeitung der isochronen Daten existiert.

4.1.4. Das Open Host Controller Interface — OHCI

Die Spezifizierung des Open Host Controller Interface dient der Vereinheitlichung der Programmierschnittstelle von IEEE 1394 LLCs und definiert hauptsächlich das Aussehen und den Aufbau des Hostinterfaces aus Sicht der Software (Firmware) [ACIM99], [Pei99]. Alle LLC-Chips von verschiedenen Herstellern mit einem nach OHCI gestaltetem Hostinterface können über die gleiche Firmware betrieben werden. Für die Realisierung im LLC muß die jeweils benutzte Hardwarebasis, sprich der Bus des Hostsystems, vier Eigenschaften zwingend erfüllen:

1. Der Open Host Controller benötigt direkten Systemspeicherzugang — **DMA**.

2. Der Open Host Controller muß den Systemspeicher byteweise ändern können.

3. Der Open Host Controller muß Ereignisse melden können — **Interrupts**.

4. Der Open Host Controller erlaubt keine 8 oder 16 Bit Zugriffe, da jeder 32 Bit Zugriff atomar erfolgen muß — **32 Bit Bus**.

Auf Grund dieser Kriterien hat sich bisher nur der PCI-Bus als Hardwarebasis behaupten können. Es gibt sicherlich weitere Bussysteme (z.B. VME), die diese Voraussetzungen erfüllen, jedoch sind mir Entwicklungen in diese Richtung unbekannt.

4.1.5. Device Bay

Neben IEEE 1394 entwickelten sich auch andere serielle Hochgeschwindigkeitsbusse mit dem Ziel, möglichst viele I/O-Geräte mit digitalen Verarbeitungseinrichtungen durch ein einheitliches Kommunikationskonzept zu einem Gerätemodell zu vereinen. Einer der erfolgreichsten Umsetzungen ist der *Universal Serial Bus* (*USB*), der zunehmend in embedded Systemen wiederzufinden ist. Mit ihm war es erstmals realistisch möglich, Kleingeräte wie Tastatur, Maus, Joystick, Monitor, Modems usw. über ein und dasselbe Medium zu verbinden und von einer zentralen Arbeitsstation aus anzusprechen. Mit fortschreitender Entwicklung von USB entstanden auch Geräte zur Anbindung verschiedener teilweise proprietärer Kommunikationslösungen; an dieser Stelle sei auch an die USB-InterBus-Anbindung erinnert.

Gegenüber USB bietet IEEE 1394 weitere Vorzüge. Das sind zum einen die wesentlich höhere Bandbreite und zum anderen die Möglichkeit von Querkommunikation zwischen zwei Geräten ohne Nutzung einer Mastereinheit (Peer-to-Peer). Eine kurze Gegenüberstellung zeigt die folgende Tabelle, deren Inhalt aus beiläufigen Internetrecherchen stammt.

	USB	**IEEE 1394**
Bandbreite	12 Mbps	100, 200 oder 400 Mbps
Geräteanzahl	127 Geräte	63 Geräte in jedem der max. 1023 Busse
Kabel	4 Adern; max. 5 m	4/6 Adern; max. 4,5 m
Querkommunikation	Mastercontroller benötigt	Peer-to-Peer

Tabelle 4.1.: Gegenüberstellung USB, IEEE 1394

Durch die Existenz von vielfältigen USB-Geräten und dem schnellen Zuwachs an weiteren datenintensiven und intelligenten IEEE 1394 Knoten sah und sieht man sich gezwungen, beide Kommunikationsmedien für die Systemintegration zu vereinen.

Das Ergebnis dieser Bestrebung verbirgt sich hinter der Spezifizierung *Device Bay* der Firmen Compaq, Intel und Microsoft [CIM98], [Kro98].

Aufbau

Durch Device Bay wird ähnlich wie bei PCMCIA ein einheitliches Einschubsystem für beide Gerätearten definiert. Pro Moduleinschub findet entweder ein USB oder IEEE 1394 Gerät Platz. Die Module können dabei drei verschiedene Formfaktoren annehmen, wobei sich diese im Wesentlichen durch die Bauhöhe unterscheiden (13, 20 und 32 mm). Die Breite und Tiefe entspricht in etwa einem herkömmlichen $5^1/_2$“ Gerät.

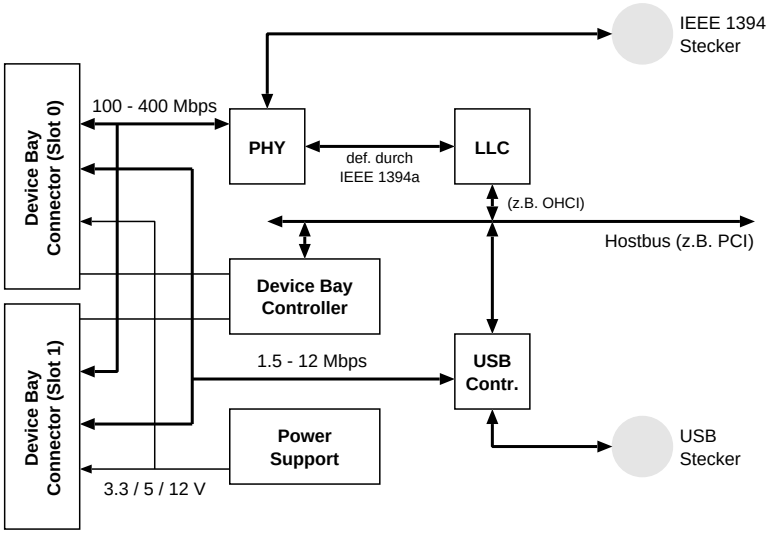

Abbildung 4.6.: Signalwege für zwei Device Bay Slots

USB und IEEE 1394 bieten die Möglichkeit, Geräte während des Betriebes hinzuzufügen bzw. zu entfernen (hot plug). Um diesen unterschiedlich realisierten Vorgang zu vereinen wird ein Steuerbaustein benötigt. Dieser übernimmt die Kontrolle über die Präsentation bereitstehender Ressourcen beider Bussysteme und sorgt für die Steuerung der Energieversorgung. Dieser *Device Bay Controller* (*DBC*) kann entweder separat aufgebaut oder Teil eines USB-Controllers bzw. LLC von IEEE 1394 sein. Der letztere Fall wird z.B. durch den Chipset „VIAFire“ von VIA Technologies umgesetzt. In ihm sind ein S400 PHY, ein LLC mit OHCI und ein DBC für vier Device Bay Slots enthalten. Zusammen mit einem USB-Controller ließe sich der in Abbildung 4.6 dargestellte elektrische Aufbau für zwei Modulsteckplätze realisieren.

4.1.6. PCILynx als autonome Steuerzentrale

Der Kunstname *PCILynx* steht für den LLC-Chip TSB12LV21A von Texas Instruments [TI97]. Sein Nachfolger *PCILynx2* besitzt lediglich einen größeren FIFO und ist sonst pin- und registerkompatibel [TI98]. Die Schaltkreisfamilie „Lynx“ umfaßt viele verschiedene LLCs mit aufgabenbezogenen Ausstattungen: *OHCILynx* für OHCI-Konformität, *GPLynx* für einfache Mikrocontrolleranschaltungen oder *DV-Lynx* bzw. *MPEG2Lynx* für Audio/Video Implementationen.

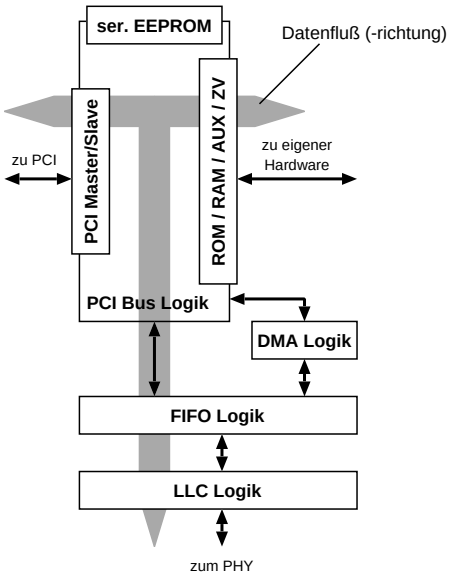

Abbildung 4.7.: Funktionsblöcke des PCILynx von TI

In Abbildung 4.7 sind die wichtigsten Funktionsblöcke des PCILynx dargestellt. Neben den primär notwendigen Teilen *LLC-Logik*, *FIFO* und *PCI Bus Logik* existieren die *DMA-Logik* für den beschleunigten Datenaustausch zwischen LLC und Hostsystem und die Schnittstelle zu einem seriellen EEPROM. Im EEPROM sind wichtige Informationen wie Vendor-ID oder Product-ID hinterlegt, welche über die Firmware im ROM-Bereich des CSR-Modells bereitgestellt werden. Diese Komponenten würden für die Bereitstellung eines IEEE 1394 Anschlusses grundsätzlich ausreichen.

Für die Einbindung applikationsabhängiger Ein- und Ausgabebaugruppen (AUX) bzw. RAM oder ROM als Arbeitsspeicher besitzt der PCILynx einen *Local Bus* und einen *Zoom Video Port* (*ZV-Port*). Über den ZV-Port können isochrone Datenströme

mit Videoinformationen direkt in einen dafür vorgesehenen Videospeicher des Hostsystems eingeblendet werden. Durch den Local Bus wird ein 16 Bit breiter Daten und Adreßbus mit drei Chip-Select, einem Out-Enable und zwei Write-Enable Signalen eröffnet. Weiter existiert ein Interrupteingang sowie ein Resetausgang und vier *General Purpose I/Os*. Mit ihm können eigene Hardwareerweiterungen angeschlossen werden. Einen Beispielaufbau zeigt Abbildung 4.8.

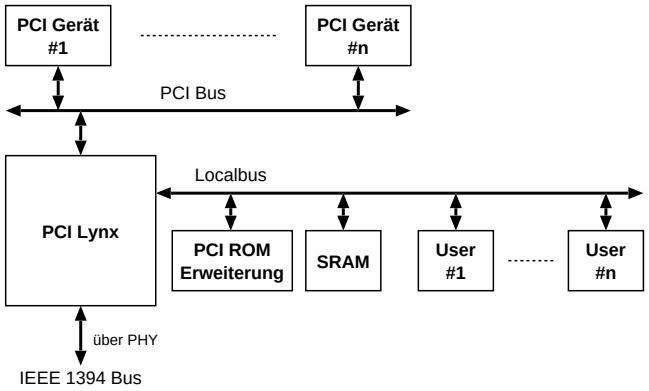

Abbildung 4.8.: PCILynx als lokaler Prozessor

Zwei der herausgeführten Chip-Select Signale aktivieren je einen Adreßbereich im ROM und RAM. Der RAM kann als Ablage der *Program Control List* (*PCL*) für die fünf DMA-Kanäle des PCILynx dienen. Eine PCL ist eine Art verkettete Liste mit Zeigern zur nächsten Anweisung für die Ablaufsteuerung einer DMA-Maschine. Jeder DMA-Kanal besitzt seine eigene PCL, die in einem direkt zugänglichen Speicherbereich hinterlegt ist. Die Nutzung des lokalen RAM am Local Bus umgeht die Zeitverzögerung, die durch die Multimasterfunktionalität des PCI Bus hervorgerufen wird, wenn sich eine solche PCL im Speicherbereich eines anderen PCI Gerätes befindet (z.B. Hauptspeicher). Die genaue Funktionsweise der PCL kann [TI97a] entnommen werden. Im ROM kann ein Boot-Up Programm in Form einer PCL hinterlegt werden, durch das bei aktiviertem *Autoboot Mode* in erster Linie der PCILynx konfiguriert und parametrisiert werden muß und letztlich weitere Systemkomponenten sowohl am Local Bus als auch am PCI Bus aktiviert werden können. der PCILynx kann also unabhängig von einem Hostsystem autonom arbeiten.

Die Existenz des Local Bus und der im PCILynx integrierte Autoboot Mode las-

sen diesen Chip für die Integration von IEEE 1394 in embedded Systeme mit kleinem bis mittlerem Leistungsumfang interessant werden. Der PCILynx fungiert dabei als ein „lokaler Prozessor“ und tritt gegenüber den verschiedenen Funktionseinheiten am Local Bus wie ein „Master“ auf. Er koordiniert mit Hilfe verschiedener PCLs den asynchronen und isochronen Verkehr zwischen IEEE 1394 und den Funktionseinheiten sowie den Datenaustausch dieser untereinander. Die zusätzliche Nutzung von etwaigen PCI Geräten bleibt weiterhin als Option bestehen. Das so entstehende Gerätekonzept wirkt in bezug auf die dynamische Erweiterbarkeit etwas starrer, unflexibler und unhandlicher, als wenn zusätzlich eine übergeordneten CPU für die Abarbeitung der Firmware und eines Anwendungsprogramms zum Einsatz käme. Aber gerade das ist wesentlich an einem embedded System: klein, kompakt und aufgabenbezogene Problemlösung mit möglichst wenig Aufwand.

Man könnte sich somit folgenden Aufbau vorstellen: Die in Abbildung 4.8 auf der vorherigen Seite dargestellten Kästchen *User #1* bis *User #n* sind Ein- und Ausgabebausteine für analoge und digitale Signale, stellen einfache Mensch-Maschine-Schnittstellen oder Prozeßsteuerbausteine dar. Sie können auch Brücken zu anderen Kommunikationsmedien sein. So ließen sich verschieden viele InterBus-Masteranschaltungen an dieser Stelle plazieren. Der so entstandene Aufbau entspräche dann den Datensammlern aus Abbildung 3.1 auf Seite 51, die für die Steuerung und Regelung eines Produktionsprozeß Daten gewinnen und über IEEE 1394 an die zugehörige SPS zur Weiterverarbeitung übergeben. Bei Kommunikationsproblemen bzw. Netzstörungen am IEEE 1394 Bus kann der eigene Prozeßsteuerbaustein die Kontrolle übernehmen.

Es gäbe sicher viele weitere Einsatzbeispiele. Dieses hier sollte zur Veranschaulichung dienen und die Idee aus dem Kapitel der Feldbusbetrachtung technologisch untermauern.

4.2. Software — Betriebssystemintegration

Die Software taucht in Form der Firmware für die Realisierung aller Kontroll-, Steuer- und Transportvorgänge innerhalb der Transaction Layer und des Serial Busmanagements auf und ist ein äußerst wichtiger Bestandteil des Gesamtaufbaus. Für embedded Systeme steht immer die Kompaktheit und Abhängigkeit

zu Betriebssystemen (Selbstständigkeit) einer Firmwarerealisierung im Vordergrund. Die Belange der Automatisierungstechnik gehen da weiter. Hier kommen neben Feldgeräten auch leistungsfähige Workstation bzw. Datenserver für Steuerung, Reglung und Bedienung zum Einsatz. Die bestehnden Softwarelösungen stellen sich genau so dar.

Die verbreitetsten Desktop-Betriebssysteme, wie Windows, MacOS, und UNIX, stellen die Firmwareintegration ohne weiteres bereit. Die Umsetzung von IEEE 1394 unter Microsoft Windows ist für den Bereich der Industrieautomatisierung besonders interessant, da dort dieses System am häufigsten zum Einsatz kommt. Sei es als NT-Server für eine Prozeßdatenbank oder als WindowsCE innerhalb eines Maschinenpults. MacOS findet man kaum in den Umgebungen der Automatisierungstechnik. Ebenso spielt UNIX eher eine untergeordnete Rolle. Linux nimmt jedoch einen Sonderplatz ein. Dieses „junge“ Betriebssystem befand sich bisher noch in keiner Nische und kann somit nicht nur in Zusammenhang mit IEEE 1394 für die Automatisierungs- und Reglungstechnik interessant werden. Dies ist unter anderem mit ein Grund, weshalb ich mich mit dem in Entstehung befindlichen IEEE 1394 Stack von Linux intensiver beschäftigt habe.

Für die Aufgabenumsetzung in embedded Systemen stehen oft nicht so mächtige Betriebsysteme zur Verfügung — manchmal auch gar keine. Ein IEEE 1394 Anschluß ist oft nur zusätzliches Beiwerk zum Gesamtergebnis einer Lösung und die Entscheidung für oder gegen ein Betriebssystem wird das zu lösende Problem bestimmen. Somit existieren hierfür betriebssystemunabhängige und skalierbare Produkte.

4.2.1. Microsoft Windows

Microsoft hatte die Entwicklung der OHCI-Spezifizierung im vergangenen Jahr sehr stark vorangetrieben. Das spiegelt sich schließlich in den Konzepten der Systemunterstützung von IEEE 1394 in den verschiedenen Versionen von MS Windows wieder. Windows 98 unterstützt neben OHCI konformen Chipsets **noch** die PCI-Chips PCILynx von TI und AIC-5800 von Adaptec. Zukünftig werden aber nur noch IEEE 1394 Baugruppen mit einem OHCI Chip unterstützt. Das wird Windows 2000 und Windows NT 5.0 betreffen. Ein herkömmliches Windows NT oder Windows 95 muß immer mit zusätzlichen Treibern des Hardwareherstellers erweitert werden.

Seit Windows 98 werden Treiber für IEEE 1394 durch das neue *Win32 Driver Modell* (*WDM*) bereitgestellt. Die höheren Protokollschichten für A/V-Anwendungen werden dann in Form von *WDM Stream Classes* bereitgestellt. Unterhalb der Stream Class Implementation liegen die eigentlichen Hardwaretreiber für die jeweiligen Chipsets in Form von kleinen *Minidrivers*. Sie stellen die durch IEEE 1394 definierte und notwendige Firmware dar und sind in einer Treiberklasse, dem *1394 Bus Class Driver*, zusammengefaßt. Die Implementation des SBP-2 erfolgt ähnlich transparent. Alle Zugriffe auf Massenspeicher oder Drucker werden bei Windows durch verschiedene *Class Driver* durchgeführt. Diese verzweigen dann zum notwendigen *Port Driver*, der hier Zugriffsmöglichkeiten auf angeschlossene Drucker oder Festplatten am IEEE 1394 Bus bietet.

Für die Erweiterung und Ergänzung dieses Treibermodells durch eigene Protokolle oder Hardwaretreiber benötigt man das *WDM DDK for IEEE 1394*, *Windows 98 DDK* oder *Windows NT DDK* sowie das *Win32 SDK*. Mehr Informationen zur Integration von IEEE 1394 in Microsofts Windows findet man im Internet unter: http://www.microsoft.com/hwdev/1394/.

Die Softwareunterstützung von IEEE 1394 unter WindowsCE wird genauso gehandhabt, wie bei allen anderen RTOS im embedded Bereich. Dennoch gibt es ein *1394 Developers Kit* speziell für WindowsCE von der Firma 3A[1].

4.2.2. Embedded OS und OS freie IEEE 1394 Stacks

Vor kurzem las ich, daß es ca. 30 verschiedene Echtzeitsysteme gibt, die für die Lösung von mehr oder weniger großen und kleinen Softwareaufgaben genutzt werden können. Wichtige Vertreter dieser Branche sind QNX, VxWorks, LynxOS und pSOS. Hinzu kommen noch die vielen teilweise selbst entwickelten Systeme für Mikrocontroller, die die Anzahl von verfügbaren Betriebssystemen im embedded Bereich schnell anwachsen lassen. Zusätzlich ist die Zielplattform, also die Prozessorfamilie, von Aufgabe zu Aufgabe eine andere.

Um in einer solchen Situation den Programmieraufwand für jede neue Entwicklung nicht ins Unermeßliche zu treiben, entstanden ein paar wenige Treibermodelle für die Implementation von IEEE 1394 in RTOS und proprietäre Betriebssysteme.

[1] http://www.3a.com/3awince.htm

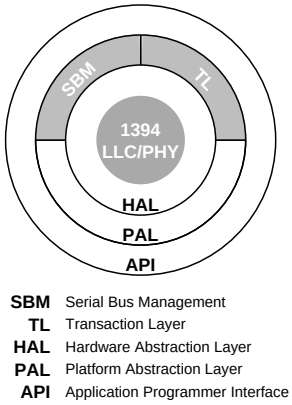

Abbildung 4.9.: Schalenmodell von eigenständigen IEEE 1394 Subsystemen

Prinzipiell sind sie alle so wie in Abbildung 4.9 gezeigt als Schichtenmodell aufgebaut. Alle diese Entwicklungen haben folgendes gemeinsam:

1. Unterstützung vieler verschiedener IEEE 1394 Chipsets über eine *Hardware Apstraction Layer* (*HAL*). Hiermit werden die Abhängigkeiten zum Chipset umgesetzt.

2. Bereitstellung aller notwendigen IEEE 1394 Funktionen, wie Serial Bus Management und Transaction Layer, als Kernstück des Treibermodells.

3. Anpassungsfähigkeit an verschiedene Plattformen (Prozessor und Betriebssystem) durch eine *Platform Apstraction Layer* (*PAL*).

4. Der Datenaustausch zwischen IEEE 1394 Funktionen und der Anwendung erfolgt über eine API.

5. Zusätzlich stehen höhere Protokollschichten, wie SBP-2 oder DPP, als integraler Bestandteil des Stacks zur Verfügung.

6. Diese Stacks sind als Quellcode erhältlich und daher sehr teuer. Zum Beispiel würde der „FireStack“ von Intoto, Inc. zusammen mit der Realisierung des SBP-2 ca. US$ 50.000,- kosten.

In vielen Fällen ist die Funktionalität von IEEE 1394 ohne Einsatz eines Betriebssystem wünschenswert. Für solche Fälle ist die Abstraktionsschicht für die Plattform

(PAL) entsprechend umzusetzen und um genau die Komponenten zu erweitern, die für die Vollfunktion des IEEE 1394 Stacks benötigt werden. Somit werden fehlende Systemaufrufe „nachgeschrieben“.

Da ich hier nicht jeden Stack im einzelnen vorstelle, verweise ich auf die Ergebnisse meiner Internetrecherche im Anhang, Abschnitt C.3 auf Seite 107. Dort befindet sich eine kurze Auflistung aller mir bekannten Softwareimplementationen des IEEE 1394 Protokollstapels mit höheren Protokollschichten für die Integration in verschiedene Betriebsysteme.

4.3. Der Demonstrator als Integrationsbeispiel

Im Verlauf dieser Arbeit stand immer wieder der Aufbau eines Demonstrators mit IEEE 1394 zur Diskussion, um die eigentlichen Aufwände im Hard- und Softwarebereich bei einer Systemintegration kennenzulernen. Damit wäre man in der Lage, spätere Realisierungen nach Kosten und Laufzeit vernünftig abzuschätzen. Nach tiefer gehenden Betrachtungen der Hard- und Softwarekomponenten wurde schnell deutlich, daß der Aufbau beider Teile gleichzeitig den zeitlichen und inhaltlichen Rahmen dieser Diplomarbeit übersteigen würde, so daß eine Teillösung eingegangen wurde. Es soll ein einfacher Systemaufbau aus zwei IEEE 1394 Kommunikationspartnern mit Kaufkomponenten hergestellt werden, deren Programmierung zu einer Applikation führt, die beide Geräte als eine einfache digitale I/O-Einheit erscheinen läßt. Dabei soll zumindest eines der Geräte auf Basis des schon beschriebenen Chipset „PCILynx“ arbeiten.

In erster Linie war die Wahl des hierfür geeigneten IEEE 1394 Stack und des damit verbundenen Betriebssystem zu lösen. Dabei stand für mich die Verfügbarkeit der Quellen des Stacks im Vordergrund, da man nur durch sie die Funktionsweise einer IEEE 1394 Realisierung nachvollziehen kann und wenn nötig in Zukunft auch Anpassungen und Veränderungen vornehmen kann, wenn ein solcher Stack in Kundenprojekte einfließen soll. Hierzu habe ich einige Angebote der verschiedenen Hersteller eingeholt. Das Ergebnis war sehr ernüchternd, denn wider erwarten stiegen die Kosten für einen Softwarestack im Sourcecode schnell in die US$ 50.000,-. Einen solchen Preis kann man ohne Kundenprojekt im Rücken nicht aufbringen.
Eine andere Möglichkeit tat sich durch Geschäftsgespräche mit einer mit der MA-

ZeT GmbH in Kooperation befindlichen Ilmenauer Firma, der Thesycon GmbH, auf. Diese Firma stellt einen vielseitig verwendbaren IEEE 1394 Treiber für die Betriebssysteme Windows 98 und Windows 2000 bereit, durch den es möglich ist, über eine sehr einfach gehaltene API (Öffnen, Lesen, Schreiben und Schließen von IEEE 1394 Geräten) eigene Softwarelösungen zu integrieren. Die Erkenntnisse durch den Aufbau des Demonstrators sollen jedoch in Projekte des embedded Bereichs einfließen. Dort hat man in der Regel diese beiden Betriebssysteme nich zur Verfügung; wenn überhaupt Windows, dann nur WindowsCE.
Ein weiteres Problem, das ich sah, war die eingeschränkte Wissensbasis aller kommerziellen Lösungen, denn entweder man erhält für viel Geld die Quellen und muß dann in speziellen Fällen selber zusehen, wie man seine Integration realisiert, oder man könnte auf eine gute Zusammenarbeit, wie im Fall Thesycon GmbH, bauen, kann aber nicht mehr Wissen und Kompetenz von seinem Partner verlangen, als er selber besitzt.

Um aus diesem Dilemma einen Ausweg zu finden, bot sich das junge „GNU/Linux IEEE 1394 Project“ an. Frei nach den Grundsätzen von Richard M. Stallman, dem Wegbereiter der ältesten Open Sources Lizenz *GNU Public License* (*GPL*), sind die Entwicklungen und Ergebnisse dieses IEEE 1394 Stacks frei von Ansprüchen jeglicher Art. Die Quellen sind ohne Auflagen frei zugänglich und können somit den von Stallman schon 1983 beabsichtigten und von mir heute gesuchten Lerneffekt von freier Software voll und ganz erfüllen. Das sich dieses Projekt gerade unter Linux angesiedelt hat, verwundert nicht, trägt doch Linux den Löwenanteil am globalen Zuspruch an Open Sources Projekten. Dieses System ist eines der erfolgreichsten GNU Projekte und setzt sich, neben den schon eroberten Serverumgebungen, zunehmend auch in industriellen Bereichen als embedded OS (auch RTOS) durch. Grund genug, in die System- und Kernelprogrammierung von Linux einzusteigen, denn die Kundenanfragen steigen.

Man mag sich nun Fragen, was sich an der Kritik in Bezug auf den Support kommerzieller Quellen durch den Einsatz von Linux geändert hat? Ganz einfach, die Wissensbasis ist global geworden und die Integrationsproblematik wird entweder von vielen Leuten gleichzeitig behandelt, oder man kann auf eine große Programmiergemeinde mit gleicher Interessenlage bauen, die einem mit Rat und Tat zur Seite steht. Auch ich habe diese Informationsquellen rege benutzt und rückwertig

durch meine Erfahrungen und Lösungen ergänzt.

Ein weiterer Anreiz für die Nutzung des GNU/IEEE 1394 Stacks lag im Erkenntnisgewinn des Realisierungsaufwandes. Man bekommt hierdurch ohne zusätzlichen finanziellen Aufwand eine Beispiellösung in die Hand und kann mit ihr seine eigene Stackintegration unter anderen Betriebssystemen abschätzen und schneller umsetzen, ohne das Rad neu zu erfinden. Dieser ist wohl der wichtigste Punkt gewesen, den Demonstrator unter Linux zu programmieren. Die dabei vorgenommene Softwareanalyse des Stacks in Bezug auf benutzte Systemfunktionen und Algorithmen kann die Portierung erleichtern.

Neben der Nutzung eines bestimmten IEEE 1394 Stacks, ob kommerziell oder Open Sources, muß auch ein Anwendungsprogramm für die Bedienung des Demonstrators erstellt werden. Da die Entscheidung nun schon für Linux gefallen war, standen mir verscheidene Möglichkeiten offen, angefangen bei der Programmierung grafischer Oberflächen mit Linux spezifischen GUI Bibliotheken bis hin zu einfachen Programmen für die Kommandozeile in ANSI-C. Für die bessere Bedienbarkeit und Visualisierung war ein grafisches Frontend geplant. Um nun dem Aspekt der angestrebten Portierbarkeit gerecht zu werden, fiel die Wahl auf die Verwendung der Scriptsprache *Tcl/Tk*. Diese Programmiersprache ist auf vielen weiteren Betriebssystemen verfügbar (Windows, MacOS, QNX, VxWorks, ...). Somit erstellte Applikationen können schnell und einfach portiert werden.

4.3.1. GNU/Linux IEEE 1394 Projekt

4.3.1.1. Zur Geschichte

Das GNU/Linux IEEE 1394 Projekt[2] wurde Anfang 1998 von Emanuel Pirker, ein Student der Universität Klagenfurt (Österreich), im Rahmen des *„Enthusiastic Carinthian Linux Project“* (*ECLiPt*) ins Leben gerufen. ECLiPt ist eine nicht kommerzielle Initiative von Linux Usern in Klagenfurt für die Unterstützung von Linux Projekten. Mit Beginn meiner Untersuchungen von IEEE 1394 und dem Entschluß der Mitarbeit an diesem Projekt existierte ein Programmgerüst für den IEEE 1394 Stack mit einem ausformulierten Hardwaretreiber für den Adaptec Chipset AIC-5800 und ein paar wenigen funktionierenden Teilen im Kern. Dieses Gerüst basierte auf

[2] http://eclipt.uni-klu.ac.at/ieee1394/

dem bereits existierenden SCSI Modell unter Linux. Wenig später, im Sommer 1999, als ich den Versuch unternahm, den Hardwaretreiber für den TI Chipset PCILynx zu implementieren, stieß Andreas Bombe aus München zum Projekt hinzu. Er hatte schon einigen Kernelcode für den Zugriff auf den Local Bus des PCILynx entwickelt und bot sich für die Fortführung des gesamten Projekts an, da Emanuel vorerst zu SGI nach Kanada zog und seitdem durch den Interessenkonflikt seiner dortigen Arbeit nicht mehr aktiv am Linux Projekt teilnehmen kann. Andreas war von Anfang an mit der Strukturübernahme aus dem SCSI Modell unzufrieden und hat dann den Kern des IEEE 1394 Stacks neu geschreiben und den PCILynx Hardwaretreiber um die asynchrone Kommunikation erweitert. Das hieß für mich doppelte Arbeit, denn ich bagenn mich von Neuem in den Stack einzuarbeiten. Bis Herbst 1999 wurde der Adaptec Treiber von Emanuel angepaßt und von Sebastien Rougeaux aus Australien und Gorden Peters der Treiber für OHCI Chipsets hinzugefügt. Man sieht, die noch andauernden Entwicklungen gestalten sich international.

Mit fortschreitender Ergründung des Kernelcodes stellte ich schnell einige Fehlfunktionen und notwendige Ergänzungen fest. So hatte ich anfangs Probleme, den PCILynx Treiber auf meiner PCI Karte zum Laufen zu bekommen. In diesem Zusammenhang erweiterte ich den Code um wichtige Grundfunktionen für die DMA Behandlung, die Arbeit mit den neuen IEEE 1394a konformen PHYs und die korrekte Behandlung von SelfID Paketen. Ohne diese Vervollständigungen würde der PCILynx Treiber noch nicht auf anderen Plattforme wie Apples iMac funktionieren.

Der meiner Arbeit zugrundeliegende Softwarestand des IEEE 1394 Stacks stammt vom November 1999. Seit dem hat das Softwaremodell einige Erneuerungen erfahren, die durch mich nicht weiter betrachtet wurden. Das ist auch der Grund dafür das mein Anwendungsprogramm auf veralteten Programmierbibliotheken aufsetzt. Mehr dazu weiter unten.

4.3.1.2. Der Stackaufbau

Der IEEE 1394 Stack für Linux ist erst ab der Kernelversion 2.2.x lauffähig und gilt als *experimentell*. Alle Neuentwicklungen am und für den Stack sind so ausgelegt, daß sie die teilweise veränderten Programmierumgebungen des Entwicklerkernels 2.3.x unterstützen und damit das gesamte Projekt mit Erscheinen der nächsten stabilen Version in den Kernel einfließen kann. Seit Januar 2000 ist der

Arbeitsstand des GNU/Linux IEEE 1394 Projects offizieller Bestandteil der experimentellen Kernelversion 2.3.40 und somit patch-frei sofort nutzbar.

Der Linuxkernel ist trotz seiner gewachsenen und umfangreichen Funktionalität ein *monolithischer Kernel* geblieben. Er büst aber aufgrund seines einfachen Modulhandlings nichts von der Dynamik eines Mikrokernels ein. Der IEEE 1394 Stack nutzt diese Modularisierung von Kernelkomponenten für alle seine Teile aus. So läßt sich der Stack anhand der Abbildung 4.10 zu drei Hauptbestandteilen zusammenfassen:

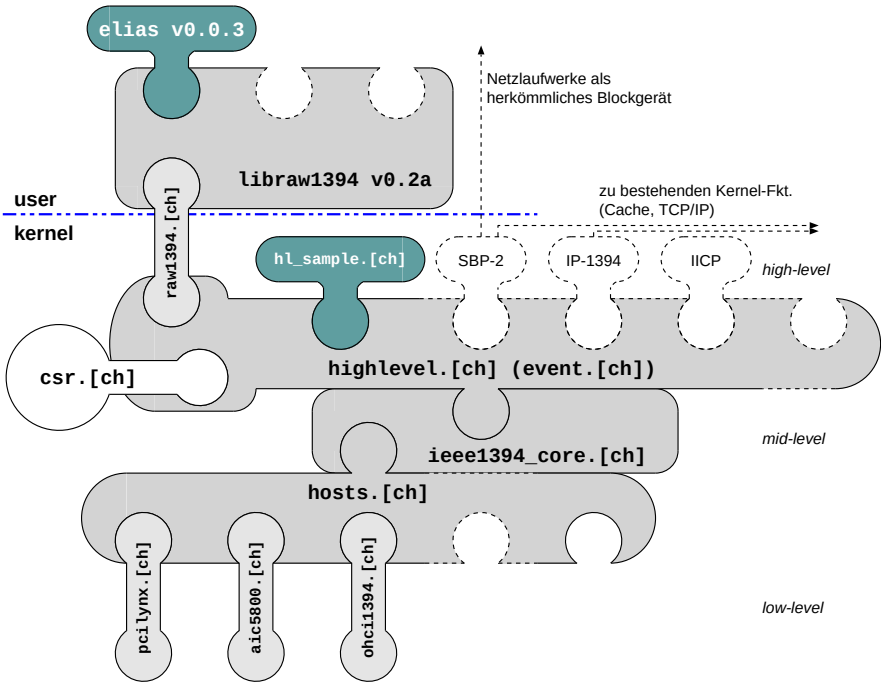

Abbildung 4.10.: Funktionsblöcke des GNU/Linux IEEE 1394 Stacks

1. **Lov-Level**: Alle drei Hardwaretreiber (innerhalb des Stacks auch Host-Treiber genannt), sind als eigenständige Module übersetzbar. Sie stellen dem Kern des Stacks die Basisfunktionen für den Hardwarezugriff auf das jeweilige Chipset bereit: Initialisieren, Senden, ROM auslesen, IO-Control und Beenden (für die Entfernung des Treibers aus dem Kernel notwendige Arbeiten).

   Unterstützt werden Hostadapter mit den Chipsets PCILynx (`pcilynx.[ch]`), AIC-5800 (`aic5800.[ch]`) und OHCI (`ohci1394.[ch]`)

2. **Mid-Level**: Der Kern des Stacks, bestehend aus `ieee1394_core.[ch]` und weiteren Hilfskomponenten, bildet das zentrale Modul. Es bearbeitet jede Kommunikationsanforderung vom und zum Bus. Alle Pakete, sowohl asynchrone als auch isochrone, werden durch diesen Kern vermittelt. Hier werden die Hauptaufgaben der Transaction Layer und des Serial Busmanagements erledigt.

   Zur Verwaltung der Low-Level Treiber dienen Hilfsfunktionen aus `hosts.[ch]`. Durch diese kann ein Hardwaretreiber seine wichtigen Funktionspointer an den Kern übergeben und umgekehrt der Kern notwendige Folgeaufrufe für die Registrierung eines neuen Lov-Level Treibers in höheren Schichten auslösen (High-Level Treiber informieren).

   Für die High-Level Treiber steht eine ähnliche Funktionalität in Form von `highlevel.[ch]` bereit. Hierüber können höhere Protokollschichten so implementiert werden, daß sie für ganz bestimmte CSR Adreßbereiche die Funktionen Lesen, Schreiben und Sperren beim Stack registrieren. Möglichkeiten für isochronen Verkehr von und zu High-Level Treibern sind noch nicht vorhanden.

3. **High-Level**: Im Bereich der höheren Protokollschichten hat sich bisher sehr wenig entwickelt, da die API hierzu erst ein paar Monate alt ist. Ein wesentlicher Bestandteil des Stacks ist der RAW I/O Treiber (`raw1394.[ch]`), der zusammen mit der RAW Library (`libraw1394 v0.2a`) das Bindeglied zwischen User- und Kernelspace für IEEE 1394 basierende Anwendungen darstellt. In der Grafik ist der Zusammenhang zwischen RAW I/O Treiber und Stack schon etwas vorauseilend geschildert. Bei dem für meinen Demonstrator zugrundeliegenden Entwicklungsstand des Stacks wird dieser Treiber noch nicht wie ein High-Level Treiber verwaltet. Das ist aber geplant.

   Eine Besonderheit stellt die Implementation der CSR Register für das Serial Busmanagment dar (`csr.[ch]`). Sie werden zwar wie ein High-Level Treiber verwaltet, befinden sich dennoch mit im Kernmodul des Stacks.

   Über die API der High-Level Verwaltung können zukünftig weitere Module, wie SBP-2 (in Entstehung), IP-over-1394 oder IICP entstehen.

Das High-level Modul `hl_sample.[ch]` und die Anwendung `elias v0.0.3` bilden den von mir entwickelten IEEE 1394 Demonstrator. Die Eigenschaften beider

Teile werden gesondert beschrieben, da zunächst durch eine Analyse des bis jetzt bestehenden Softwaremodells in punkto benutzter Systemfunktionen die geforderte Aufwandsabschätzung für die Implementation eines IEEE 1394 Stacks dargestellt werden soll.

4.3.1.3. Kernelalgorithmen und Kernelfunktionen

Der gesamte IEEE 1394 Stack unter Linux stellt eine Ansammlung von verteilten Datenstrukturen und zugehörigen Funktionen für die Bearbeitung der Inhalte dieser Strukturen dar. Für die Bewältigung dieser Aufgaben bedient sich der Stack einer Vielzahl bereits bestehender Kernelfunktionen. Hier sollen diese benutzten Kerneleigenschaften in Verbindung mit ihrer Anwendung im Stack analysiert und vorgestellt werden.

In meinen Betrachtungen setze ich jedoch voraus, daß die hier nicht angesprochenen Programmteile des Kernels eine vollständige Systeminitialisierung der zugrundeliegenden Plattform und das Aufrechterhalten aller sonstigen Betriebssystemfunktionen gewährleisten. Hierbei ist eine etwaige Kooperation mit einem BIOS zum Systemstart, die komplette Parametrisierung und Aktivierung des PCI Bus und die linuxspezifische Modulverwaltung inbegriffen. Unterlagen dazu lassen sich in den Dokumentationen der Kernelquellen selbst oder in den Büchern [Beck99], [Card98] und [Rub98] finden.

Dynamische und virtuelle Speicherverwaltung

Unter Linux wird der verfügbare physikalische Speicher in Seiten zu je 4 KB verwaltet. Dieser Wert variiert für verschiedene Plattformen. Für die Anforderung einer oder mehrerer Seiten gibt es die Funktionen `__get_free_pages()`, `__get_free_page()` und `get_free_page()`, wobei letztere die „neue“ Speicherseite zusätzlich löscht. Da der so reservierte Speicher bei Platzmangel durch den Kernel ausgelagert werden kann, wird bei Normalbenutzung dieser Funktionen der rufende Prozeß blockiert, wenn während der Reservierung ausgelagert werden muß. Um das Blockieren innerhalb eigener atomarer Funktionen zu verhindern (z.B. ISR), kann eine Prioritätsmaske übergeben werden. Durch diese wird nicht nur die Blockade verhindert (`GFP_ATOMIC`), sondern kann auch sichergestellt werden, daß der zurückgegebene Speicherbereich für DMA nutzbar ist (`GFP_DMA`) oder nach der Zuweisung noch wei-

tere Speicherseiten frei sind (GFP_BUFFER). Einmal reservierter Speicher muß über free_pages() bzw. free_page() wieder freigegeben werden.

Zur besseren Handhabung existieren äquivalent zu ANSI-C die beiden Funktionen kmalloc() und kfree(), wobei kmalloc() ein weiteres Argument für die Priorität besitzt. Mit diesen beiden Funktionen werden fast alle Speicherbedürfnisse des Stack koordiniert.

Die Speicherreservierung über kmalloc() ist begrenzt. So können nicht mehr als 32 Seiten auf einmal angefordert werden (also 128 KB), da immer hintereinander liegende Seiten gesucht und verwaltet werden. Abhilfe schafft hier die virtuelle Speicherverwaltung. Der über die Funktionen vmalloc() und vfree() benutzte Speicher kann in mehreren, über den physikalischen Adreßraum verteilten, Seiten enthalten sein. Diese Bereiche werden durch den Kernel nicht ausgelagert und sind somit permanent verfügbar. Diese Art von Speicherreservierung wird vom IEEE 1394 Stack benötigt, um ressourcenschonend zu arbeiten, da einige Datenstrukturen, speziell die Hardwaretreiberverwaltung, erst viel später nach der Erstinitialisierung aufgebaut werden.

Bei PCI-Geräten wird der Speicher für Register, RAM oder ROM an hohe Adressen gelegt, die über dem physikalischen Speicher liegen. Diese Adreßbereiche müssen zusätzlich in den virtuellen Speicher über die Funktionen ioremap() und ioremap_nocach() eingebunden werden, um mit ihnen arbeiten zu können. Die zweit genannte Funktion wird für Register benutzt, um die Cache-Speicher zwischen Prozessor und PCI-Registerbereich zu umgehen. So eingehangene Bereiche werden durch iounmap() wieder entfernt. Diese Funktionen sind zusammen mit den Verwaltungsfunktionen für den PCI Bus notwendig, um überhaupt die Zugriffsfunktionen auf die PCI Geräte innerhalb eines Hardwaretreibers zu implementieren.

PCI Geräteverwaltung

Für die Implementation der Hardwaretreiber, die PCI Geräte ansprechen sollen, ist eine einzige Funktion wichtig: pci_find_device(). Mit ihr wird über den Vendor- und Devicecode ein ganz bestimmtes PCI Gerät am Bus gesucht. Ist die Suche erfolgreich, so erhält man für alle gefundenen Geräte dieser Art eine einfach verkettete Liste mit allen wichtigen Konfigurationsdaten zurück. Die vorherige Buskonfiguration durch das BIOS oder neuerdings durch Linux selbst wird hierbei vorausgesetzt.

Sperrvariablen

Für die gemeinsame Benutzung von Variablen durch einen Interrupt-Handler und andere Funktionen, muß der Zugriff auf diese atomar ablaufen. Hierzu bietet der Kernel für Integer-Variablen die Funktionen `atomic_add()`, `atomic_sub()`, `atomic_inc()`, `atomic_dec()` und `atomic_read()` und `atomic_set()`. Diese Funktionen garantieren einen atomaren Zugriff auf eine Integer-Variable und werden sehr häufig für Ressourcenzählungen eingesetzt. Im Hardwaretreiber für den PCILynx wird über diese Funktionen ein Interruptzähler verwaltet. Es handelt sich dabei um den Interrupt des Local Bus, der von diesem Chip aus eröffnet wird.

Interprozeßkommunikation

Linux ist ein Multitasking-Betriebssystem, wodurch zwangsläufig mehr als nur ein Prozeß quasiparallel auf Ressourcen zugreifen kann. Dabei ist in vielen Fällen der gleichzeitige Zugriff auszuschließen. Diese Wettbewerbsbedingung (*race condition*) kann durch Verwendung verschiedener Puffer und/oder der Kombination mit Synchronisationsmethoden ausgeschlossen werden. Für die Synchronisation zwischen den einzelnen Komponenten des IEEE 1394 Stacks und Anwenderprozessen benutzt der Stack Warteschlangen (*Waitqueues*), Samaphoren und spezielle Semaphoren für den „gegenseitigen Ausschluß“ (Mutex), die im Kernel als *Spin Locks* bezeichnet werden.

Spin Locks werden für den Schutz von kritischen Bereichen benutzt. Ein solcher Bereich benötigt die Garantie, daß er ohne Unterbrechung durch einen Interrupt ablaufen kann und/oder die Datenbasis, die er bearbeiten will, nur durch ihn verändert werden kann. Die bisherigen Linuxkernel haben für diesen Fall lediglich alle Interruptquellen gesperrt. Damit war der Timerinterrupt und folglich der Scheduler für die Prozeßwechsel ausgeschaltet. Mit der Unterstützung von Mehrprozessorsystemen existieren aber trotzdem je nach Prozessoranzahl mehrere parallel arbeitende Prozesse und der gegenseitige Ausschluß muß erweitert werden. Dazu dienen die Spin Lock Funktionen `spin_lock()`, `spin_lock_irq()`, `spin_lock_irqsave()` und ihre Gegenstücke. Sie können wahlweise eine Ressource mit oder ohne Interruptverbot sperren. Die Sperrung selbst erfolg atomar (ein Muß).

Im IEEE 1394 Stack können das Versenden von Paketen, das Warten auf Ereignisse (z.B. Busreset) oder die Abarbeitung von DMA Programmen längere Zeit dauern. Um dadurch das System insgesamt nicht zu blockieren, werden Waitqueu-

es genutzt. Sie sind einfach verkettete Ringlisten mit Zeigern auf die jeweils eingetragenen Prozeßtabellen. Zur Bearbeitung dieser Liste dienen die Funktionen `init_waitqueue()`, `add_wait_queue()` und `remove_wait_queue()`. Das eigentliche Blockieren und Warten wird mit weiteren drei Funktionen realisiert: mit `sleep_on()` unterbrechungsfrei, mit `interruptible_sleep_on()` unterbrechbar durch Signale anderer Prozesse und mit `interruptible_sleep_on_timeout()` für eine gewisse Zeit. Die Aufweckfunktionen als Gegenstück sind: `wake_up()` für alle Prozesse einer Warteschlange und `wake_up_interruptible()` für alle als unterbrechbar hinterlegten Prozesse.

Mit Hilfe der Waitqueues werden Semaphoren aufgebaut. Nach der Initialisierung kann der Semaphorenzähler mit `up()` herauf und mit `down()` herab gezählt werden. eine Semaphore gilt bei einem Zählerstand von 0 als belegt und der zugreifende Prozeß wird automatisch durch einhängen in eine Warteschlange blockiert. Innerhalb des IEEE 1394 Stacks kommen auch Semaphoren zum Einsatz, jedoch bin ich über deren genaue Nutzung nicht ausausreichend informiert. Meines Erachtens nach dienen sie lediglich als einfache Auf- und Abwärtszähler.

Task Queues

Die *Task Queue* erfüllt einen ähnlichen Zweck für Funktionen wie die Waitqueue für Prozesse. Über Sie können beliebig viele Funktionen in eine Warteschlange (Task Queue) eingereiht werden und zu einem späteren Zeitpunkt ausgeführt werden. Im IEEE 1394 Stack wird dieser Mechanismus beim Versenden von Paketen genutzt um nach einem Timeout feststellen zu können, ob die Übertragung fehlgeschlagen oder erfolgreich verlaufen ist. Eine ausgebliebene Bestätigung eines asynchronen Datenpakets wird somit feststellbar.

Virtuelles File System

Das *Virtuelle File System* (*VFS*) ist eine Zusammenfassung aller denkbaren, unter Linux schon zahlreich implementierten, Filesysteme zu einem einzigen „neutralen“ Filesystem. Das ist in erster Linie mehr für die Integration zusätzlicher Filesysteme interessant, jedoch werden dem Anwender durch dieses System auch Gerätetreiber in Form von Gerätedateien bereitgestellt. Das Öffnen, Schließen, Lesen und Schreiben, als Grundfunktion für Gerätezugriffe, gestaltet sich dadurch so wie ein Dateizugriff. Im Treiber werden für diese und weitere Zugriffsarten einzel-

ne Funktionen geschrieben und dem Kernel durch die Treiberregistrierung in Form einer Zeigerstruktur übergeben.

Hilfsfunktionen

Für die Arbeit mit DMA müssen die durch den Gerätetreiber reservierten Speicherbereiche über physikalische Adressen angesprochen werden. Der Programmcode kennt aber nur virtuelle Adressen. Dazu kommt, daß in vielen Plattformen die Adressen der Bussysteme über einen Brückenschaltkreis in den Arbeitsspeicher abgebildet werden (z.B. PCI, ISA). Dann können die für DMA wichtigen Busadressen nicht mit den physikalischen Adressen übereinstimmen. Zur Umsetzung der Adressen zwischen diesen drei möglichen Adreßräumen sind im Kernel die Konvertierungsfunktionen `virt_to_bus()`, `bus_to_virt()`, `phys_to_bus()` und `bus_to_phys()` definiert.

Mit den Konvertierungsfunktionen `cpu_to_be32()`, `cpu_to_le32()` und umgekehrt können Mehrbyte-Werte, von denen bekannt ist in welcher Byteorder sie vorliegen, in die richtige Byteorder des Hostsystems, der CPU, überführt werden. Der Hardwaretreiber für den AIC-5800 von Adaptec benötigt diese Hilfsfunktionen, da der Chip nicht in der Lage ist, das Byteswaping hardwareseitig vorzunehmen.

Für die Verwaltung der High-Level Treiber wird vom Stack eine durch Kernelfunktionen verwaltete doppelt verkettete Liste genutzt (eine Standard-Linuxliste). Der Kernel stellt dazu eine ganze Reihe an Makros und Funktionen bereit, die das aufwendige Ein- und Aushängen von Elementen erleichtern.

Eine für Debugging sehr wichtige Funktion des Kernels ist `printk()`. Mit dieser Funktion kann der Kernel entweder über eine Konsole direkt auf den Bildschirm oder über einen Anwendungsprozess, dem *syslog demon*, in eine Datei beliebige Meldungen ausgeben. Der Aufruf folgt dem ANSI-C Pendant `printf()` in allen Regeln.

4.3.2. High-Level CSR Speichermodul — hl_sample.[ch]

Grundlage für den Demonstrator bildet das Kernelmodul `hl_sample.[ch]` (siehe Abbildung 4.10 auf Seite 76). Dieses registriert unter Nutzung der High-Level-API des IEEE 1394 Stacks einen unbenutzten Speicherbereich innerhalb des CSR-Modells und stellt für diesen Bereich die asynchronen Zugriffsaktionen Lesen und Schreiben bereit. Diese Aktionen werden in Form von Funktionspointern (`hls_addr_read()` und `hls_addr_write()`) während der Registrierung dem Kernmodul (Mid-Level)

bekanntgegeben und durch dieses bei Bedarf genutzt, wenn ein beliebiges am Bus existierendes Geräte auf genau diesen Speicherbereich zugreifen will. Das gleichzeitige Sperren und Schreiben (Lock) ist nicht implementiert, nur vorbereitet.

Der so verfügbare Speicherbereich wird in genau so vielen Speicherblöcken, wie Hostadapter bekannt sind, verwaltet. Das ist nötig, da jede im Kernmodul registrierte IEEE 1394 Anschaltung einen eigenständigen Knoten am Bus darstellt und damit einen eigenen Speicherbereich in der Adreßkonvention von CSR besitzt. Die Speicherverwaltung wird durch eine einfach verkettete Liste realisiert, an die bei Bedarf ein weiterer Speicherblock angehangen oder abgehangen wird. Dieser Vorgang wird für jeden schon registrierten Hostadapter ebenso beim Laden und Entfernen des Speichermoduls in den Kernel wie auch beim Laden und Entfernen eines Hostadaptermoduls bei schon existierendem Speichermodul durch das Kernmodul angestoßen. Dazu werden dem Kernmodul zum Zeitpunkt der High-Level Treiberregistrierung entsprechende Funktionspointer (`hls_host_added()`, `hls_host_removed()` und `hls_host_reseted()`) übergeben. Die dritte dieser drei Funktionen kann dazu benutzt werden, um eventuelle Ureinstellungen wieder einzunehmen, wenn im IEEE 1394 Bus ein Busreset ausgelöst wurde. In dem von mir geschriebenen Speichermodul werden als Reaktion auf ein Busreset alle Speicherzellen des jeweiligen Speicherblocks[3] gelöscht. In Abbildung 4.11 auf der nächsten Seite ist als repräsentatives Beispiel der Programm-Ablauf-Plan (PAP) abgebildet, der der Funktion `hls_host_added()` zugrunde liegt. Dabei wurde angenommen, daß die Liste der Speicherblöcke schon einen Eintrag besitzt — ein Speicherelement für einen AIC-5800 Hostadapter. Die abzuarbeitenden Schritte gestalten sich dann wie folgt:

1. Der Funktion wird ein Referenzzeiger auf die einzigartige Datenstruktur des Hostadapters übergeben (im Beispiel die eines PCILynx Hostadapters). Nach der erfolgreichen Reservierung von genug Speicher für ein neues Speicherelement (Zeiger `new`) wird der Referenzzeiger eingetragen und das zukünftige Ende der einfach verketteten Liste durch `new->next=NULL` gekennzeichnet. Der Referenzzeiger dient später bei einem Lese- oder Schreibzugriff als Vergleich, um das richtige Speicherelement zu finden. Sollte kein Speicher zur Verfügung stehen, so wird die Funktion ohne Fehlermeldung verlassen. Das damit fehlende Element wird erst bei einem Lese- oder Schreibzugriff als Fehler gemeldet.

[3] des dazugehörigen Hostadapters

2. Das Hinzufügen des neuen Speicherelements bedeutet eine Manipulation der Liste, welche ohne Unterbrechung oder doppelten Zugriff durch einen anderen Prozeß erfolgen muß, da sonst die Konsistenz ihres Inhaltes gefährdet wäre. Daher wird davor der Zugriff auf die Liste gesperrt und am Ende wieder freigegeben.

3. Da die Liste bereits einen Eintrag enthält wird das Ende der Liste gesucht (bis einer der `next`-Zeiger eines Elementes auf `NULL` zeigt). Das neue Speicherelement wird dann an das Ende der Liste angehangen und es ergibt sich die neue Speicherblockliste wie unten rechts gezeigt.

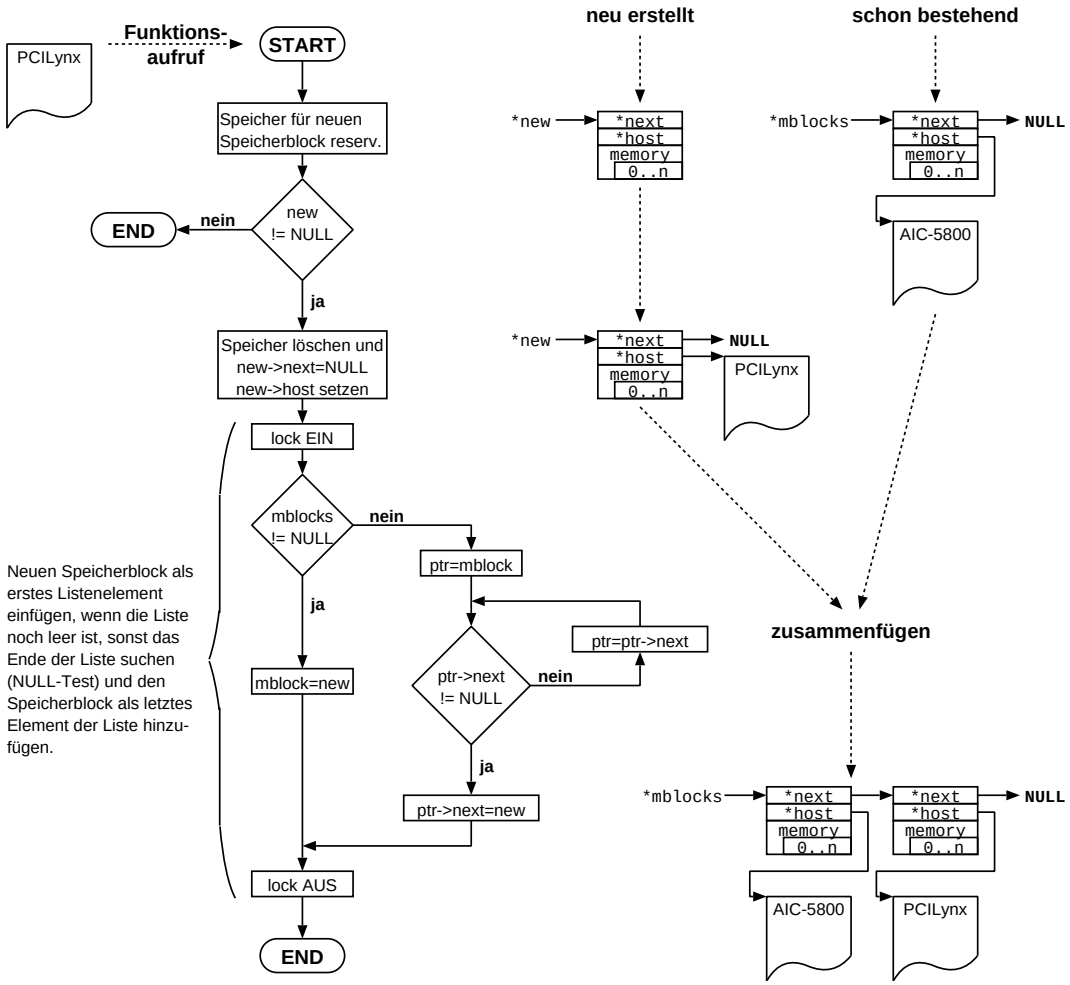

Abbildung 4.11.: PAP für das Hinzufügen eines Speicherblocks

Die gesamten C Quellen von `hl_sample.[ch]` sind im Anhang, Abschnitt D.2.1 auf Seite 110, abgedruckt. Die Quellen sind so gestaltet, daß sie sich mit meinen Änderungen, Ergänzungen und Anpassungen der Make- und Configfiles des Linuxkernel auf herkömmliche Weise als Kernelmodul übersetzen lassen. Ich habe sie als Teil des Linuxkernels in eine eigene Kernelrevisionein einfließen lassen. Diese Revision, `linux-2.2.9-1394-kdb-19991227`, liegt als Patch und `tar.gz`-Archiv vor.

4.3.3. Tcl/Tk-Anwendung — elias v0.0.3

Um einen Zugriff auf den durch obiges Kernelmodul bereitgestellten Speicher herzustellen, kann man von einem beliebigen IEEE 1394 Knoten aus mit Hilfe irgendeines Werkzeuges ein asynchrones Lese- oder Schreib-Request auslösen. Dieser Vorgang ist in den meisten Fällen sehr mühsam. Daher schrieb ich ein Benutzerprogramm, das unter Verwendung der Programmierbibliothek `libraw1394 v0.2` und dem High-Level Gerätetreiber `raw1394.[ch]` des Stacks die notwendigen Schreib- und Leseaktionen durchführt. Die Bibliothek und der zugehörige Gerätetreiber sind Teil des GNU/Linux IEEE 1394 Projects und erlauben einen direkten Zugriff auf die Funktionen des Stacks. Die Bibliothek besaß einige gravierende Fehler, speziell im Eventhandling. Diese korrigierte ich und überführte sie in die Version v0.2a. Dieser Stand ist durch die rasante Entwicklung des Projekts schon längst veraltet, wodurch mein Benutzerprogramm nicht mehr unter der neusten Bibliothek (v0.4) lauffähig ist.

Tcl/Tk ist eine weit verbreitete Scriptsprache von Sun zum einfachen C-ähnlichen Programmieren von Anwendungsprogrammen und besteht aus den zwei Teilen Tcl (*Tool Command Language*), ein zeilenweise arbeitender Scriptinterpreter, und Tk (*Tool Kit* für Windowsoberflächen), das viele verschiedene Widgets für die Gestaltung eigener Bedienoberflächen bereitstellt. Der Interpreter Tcl ist mit einem C Programmierinterface ausgestattet, über das man eigene Erweiterungen dem Interpreter bereitstellen kann [Fos97], [Oust95]. Diese Programmierschnittstelle habe ich für meine Anwendung genutzt, um eine Verbindung zwischen Bedieneraktionen an der Oberfläche und den notwendigen Aufrufen in der IEEE 1394 Bibliothek herzustellen.

In Abbildung 4.12 auf der nächsten Seite ist die Bedienoberfläche der Tcl/Tk Applikation `elias v0.0.3` abgebildet. Über sie kann der Benutzer im unteren Bereich

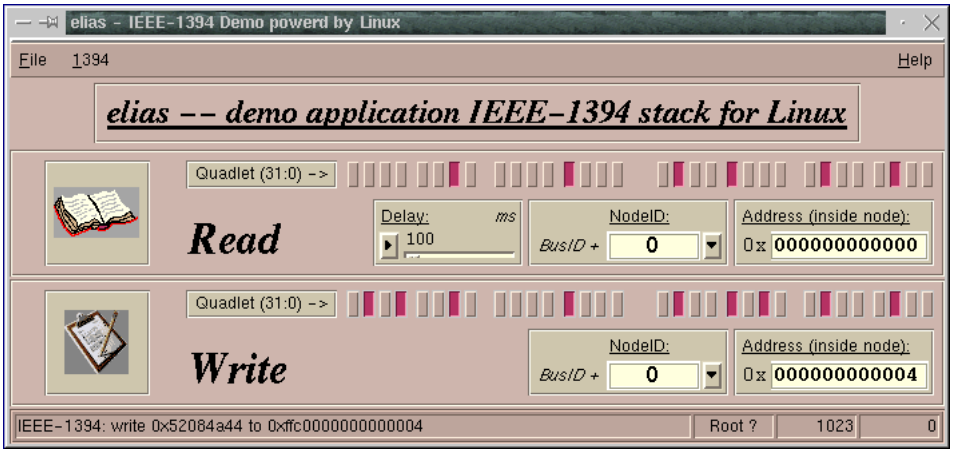

Abbildung 4.12.: Frontend des Benutzerprogramms

Schreib- und im oberen Bereich Leseoperationen auslösen. Die dafür notwendige Zieladresse wird im zugehörigen Eingabefeld durch Angabe der `node_ID` und der restlichen CSR Adresse bekanntgegeben. Mit dieser Oberfläche ist die Manipulation einer 32 Bit breiten Speicherzelle möglich. Dazu dienen die kleinen, zu je 1 Byte zusammengefaßten Taster. Für ein periodisches Auslesen einer Speicherzelle (polling) kann man den Zeitgeber im Feld „Read“ benutzen. Dessen Periode läßt sich über einen Schiebeschalter einstellen.

Diese Tcl/Tk Applikation besteht aus zwei Teilen. Der wichtigste Teil ist ein C Programm (`eliasApp.[ch]`), das den gesamten Funktionsumfang von Tcl und Tk in Form von hinzugelinkten Bibliotheken mit meinen Interpretererweiterungen vereint. Die so neu entstandenen Tcl Befehle und Variablen können im zweiten Teil der Applikation, dem eigentlichen Script `eliasMain.tcl`, genutzt werden. Den prinzipiellen Aufbau des C Programms zeigt Abbildung 4.13.

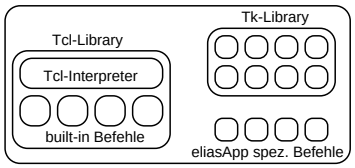

Abbildung 4.13.: Softwaremodell von eliasApp.[ch]

Der C Quellcode von `elias v0.0.3` ist im Anhang, Abschnitt D.2.2 auf Seite 118, abgedruckt. Das Tcl/Tk Script wurde mit Hilfe eines visuellen Werkzeugs,

Visual Tcl, hergestellt und hat einen beträchtlichen Umfang angenommen. Aus diesem Grund liegt es nicht in gedruckter Form bei, sondern kann nur der beiligenden Distribution `elias-0.0.3.tar.gz` entnommen werden. Dieses Packet setzt für die Compilierung die IEEE 1394 Bibliothek `libraw1394-0.2a.tar.gz` und Tcl/Tk 8.0 voraus. Die Konfiguration, Übersetzung und Installation ist unter Zuhilfenahme der beiden GNU Tools automake/autoconfig weitestgehend automatisiert, aber momentan nur unter Linux möglich.

4.3.4. Gesamtaufbau

Neben der Software ist auch die Hardware wichtiger Bestandteil des Demonstrators. Es wurden zwei handelsübliche Standard PCs der Pentiumklasse mit je einem durch den IEEE 1394 Stack von Linux unterstützten Hostadapter benutzt. Eine genaue Auflistung findet man im Anhang, Abschnitt D.1 auf Seite 109.

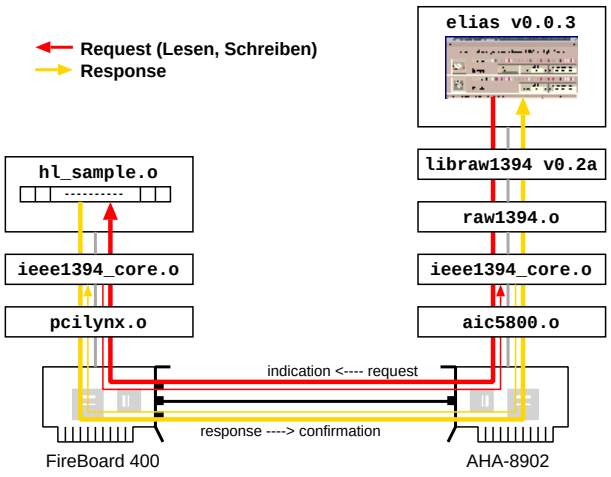

Abbildung 4.14.: Gesamtaufbau des Demonstrators

Der Gesamtaufbau gestaltet sich so, wie in Abbildung 4.14 dargestellt. Aktionen des Benutzers auf der rechten Seite werden durch die Tcl/Tk Application weiter über die Programmbibliothek und den IEEE 1394 Stack im Kernel an den Hostadapter weitergereicht. Dieser sendet das asynchrone Paket (Request) sobald er darf an den Adressat links. Der dortige Hostadapter empfängt das Paket und reicht es an den Stack im Kernel weiter. Dieser prüft das Paket auf logische Richtigkeit,

bestätigt den korrekten Empfang und übergibt es bei Erfolg dem von mir geschriebenen Speichermodul (Indication). Hier werden nun bei einer Schreiboperation die Paketdaten an die entsprechende Stelle geschrieben und bei einer Leseoperation die angeforderten Daten herausgelesen. In beiden Fällen wird je nach Operation mit oder ohne Dateninhalt der Erfolg oder das Scheitern als Response an den rechten Knoten zurückgeschickt. Diese Bestätigung signalisiert dann dem IEEE 1394 Stack rechts und der darüber liegenden Anwendung den Ausgang der zurückliegenden Aktion (Confirmation).

4.3.5. Lizenzbedingungen

Die beiden Programmteile des Demonstrators unterstehen unterschiedlichen Lizenzbedingungen. Das Kernelmodul `hl_sample.[ch]` untersteht der GPL, da es substantiell auf Quellcode aufbaut, der schon unter GPL steht (der Kernel selbst). Die GPL fordert, daß Programme, die mit Hilfe von GNU Code funktionieren, ebenfalls der GPL unterstehen.

Tcl/Tk untersteht einer gesonderten Lizenz von Sun Microsystems, Inc. und der Universität von Kalifornien, welche die Benutzung, Verbreitung und Erweiterung ausdrücklich erlaubt. Anders sieht es mit der benutzten Programmbibliothek `libraw1394 v0.2a` des GNU Projekts aus. Sie unterliegt ebenfalls der GPL. Trotzdem gilt für `elias v0.0.3` nicht das gleiche wie für das Kernelmodul, da diese Applikation lediglich eine Programmbibliotheken benutzt und daher eine etwas abgewandelte Lizenz, die *Lesser GPL* (*LGPL*), Anwendung findet. Diese erlaubt unter bestimmten Umständen das Linken mit kommerziellen Programmen, so wie in diesem Fall. Momentan liegt das Copyright von `elias v0.0.3` noch beim Urheber. Ich veröffentliche jedoch mit dieser Diplomarbeit den aktuellen Arbeitsstand, Version 0.0.3, und gebe diesen für Veränderungen frei. Diese müssen mir aber bekannt gegeben werden.

Um ein wenig mehr über die Lizenzpolitik von GNU zu erfahren, verweise ich auf den Artikel **GPL & Co.** im Linux Magazin 11/99. Eine interessante Internetseite im Zusammenhang mit Open Source Software kann man bei IBM unter http://www.software.ibm.com/developer/library/license.html finden.

5. Abschlußbetrachtung

5.1. Erkenntnisse der Arbeit und Ausblick

Am Ende meiner Arbeit steht eine voll funktionstüchtige Anwendung, die unter Nutzung des teilweise fertig gestellten *GNU/Linux IEEE 1394 Projects* und diversen kommerziellen Hardwarekomponenten die direkte Kommunikation zwischen verschiedenen Geräten demonstrieren kann. Diese Applikation ist als Arbeitstand zu bewerten, da sich durch die hohe Dynamik des zugrundeliegenden Linux Projekts die Programmierschnittstellen schnell ändern und die Fehlerhaftigkeit mit der Nutzung dieser Quellen steht und fällt. So ist es nicht verwunderlich, daß mein Ergebnis breits auf veralteten und inkompatiblen Programmbibliotheken basiert. Nach dem Abschluß dieser Arbeit wird es mich also noch viel Zeit kosten, um das bestehende Resultat anzupassen, auszubauen und zu verfeinern. Hier stelle ich mir vor allem eine bessere Abstraktion innerhalb der Tcl/Tk Anwendung vor. Ich plane an dieser Stelle eine Tcl-Erweiterung in Form einer Programmbibliothek, um den Zugriff auf IEEE 1394 Geräte für beliebige Tcl/Tk Applikationen zu ermöglichen. Das soll plattformunabhängig gestaltet werden, so daß diese Komponente überall dort benutzt werden kann, wo auch Tcl/Tk Anwendung findet.

— ♢ —

Die Einarbeitung und Mitarbeit am GNU/Linux Projekt hat aber auch viele interessante Erkenntnisse in Bezug auf den Umfang der notwendigen Softwarekomponenten zum Vorschein gebracht. Angesichts der über 10.000 Programmzeilen C Code und den damit verbundenen softwaretechnischen Voraussetzungen an das benutzte Betriebssystem, die notwendig sind, um ein Mikrorechnersystem mit der notwendigen Firmware (dem Protokollstack) zu erweitern, ist schnell klar, daß innerhalb

eines Kundenprojekts, bei dem IEEE 1394 nur unter anderem als Transportmedium eine Rolle spielt und nicht das Hauptproblem darstellt, dieser sehr umfangreich ausfallende Softwareteil nicht selbst entwickelt werden sollte. Man wird sich an einem solchen Punkt immer wieder nach kommerziell verfügbaren Lösungen umsehen müssen. Die Rechnung dazu ist ganz einfach. Die durchschnittliche Entwicklungszeit des GNU/Linux Projekts kann man mit etwa 2 Mannjahren beziffern. Ein um Funktion und Ausstattung noch bei weitem überlegener kommerzieller IEEE 1394 Stack ist dann mit seinen US$ 50.000,- viel rentabler. Der Versuch der Portierung des GNU/Linux Projekts kann sich auch nur in Grenzen lohnen. Sollte jedoch ein Lösungsweg unter Verwendung von Linux selbst möglich sein, so ist das die preiswerteste, aber bisher leider noch nicht die vollständigste Variante. Die Preise in dieser Branche werden hoffentlich durch die zunehmende Beliebtheit von IEEE 1394 in der Konsumgüterindustrie weiter fallen.

— ♢ —

Die Betrachtung von verschiedenen Chipsets hat gezeigt, daß je nach Problemlage und Funktionsumfang des Hostsystems Standardbausteine existieren. Diese können genauso ohne Umstände in embedded Systeme für Automatisierungsgeräte eingebaut werden, wie sie bereits in der Unterhaltungselektronik schon regen Einsatz finden. Der Hardwareaufbau gestaltet sich dank der umfangreichen Referenzdesignes und Applicationnotes seitens der Chiphersteller als relativ einfach. Man kann es mit dem Aufbau einer Ethernetanschaltung vergleichen. Die Weiterentwicklungen auf diesem Sektor werden in naher Zukunft eine noch höhere Miniaturisierung bei gleichzeitiger Kostenreduzierung mit sich bringen. So kündigte Texas Instruments bereits für das Jahr 2000 einen 1-Chipset mit PHY, LLC und PCI Hostinterface an, der in großen Mengen nur noch US$ 6,- kosten soll. An diesem Preisverfall sind ebenfals die riesigen Märkte der Unterhaltungs- und Computerindustrie maßgeblich beteiligt. Allein 1999 wurden ca. 7 Millionen Digitalkameras mit IEEE 1394 Anschluß gebaut und rund 8 Millionen PCs mit diesem High-Speed Interface ausgestattet. Allein Sony plant für das Jahr 2000 etwa 35 Millionen Playstation II mit IEEE 1394 Anschluß auszuliefern (Zahlen entstammen [Kro99a]). Das sind Stückzahlen, die selten so schnell durch das Auftragsvolumen anderer Branchen erreicht werden können. Angesichts dieser rapid steigenden Zahlen kann man den Standard IEEE 1394 getrost als etabliert betrachten.

— ♢ —

Für die Automatisierungstechnik stellt dieser rasant wachsende Zuspruch eine einmalige Chance dar. Noch nie gab es für ein „neues Feldbussystem“ einen solch großen Referenzmarkt, durch den Millionen teure Vorlaufentwicklungen eingespart und Fehlentwicklungen schon im Vorfeld entdeckt werden können. Das Vorhandensein mehrerer verläßlicher und stabiler Standards sichert die notwendige Interoperabilität und Langlebigkeit nachträglich ab. Da es dennoch aktuelle Hindernisse bei der Einführung von IEEE 1394 in industrielle Bereiche gibt, wird sich der Wandel von speziellen eigenständigen Lösungen hin zu einheitlichen Globalsystemen nur allmählich vollziehen. Erst durch den Abschluß der Erweiterungsstandards IEEE 1394b und IEEE 1394.1 und dem zügigen Vorankommen der diversen Industriearbeitsgruppen innerhalb der 1394 Trade Association, wird dieses Bussystem den Durchbruch in der Automatisierungstechnik schaffen. Meine bisherigen Untersuchungen zeigen, daß IEEE 1394 in Zukunft viele Voraussetzungen für den industriellen Einsatz bieten wird. Kein anderes Feldbussystem kann bis jetzt absolut störungsfrei durch den Einsatz von LWL Technik mit Übertragungsraten von 800 Mbps Prozeßdaten, Parameterdaten und Bildverarbeitungadaten parallel und in Echtzeit über ein und denselben Bus transportieren. Doch was bedeutet dieses Szenario für den Entwickler heute? Die Probleme mit IEEE 1394 liegen alle in der physikalischen Schicht begründet. Die aktuell realisierbare Definition erlaubt nur max. 4,5 m Kabellänge, max. 63 Geräte in einem Bus und ein Steckersystem, das auf keinen Fall industrietauglich ist. Das Kommunikationsmodell von IEEE 1394 ist jedoch ohne Zweifel der entscheidendere Teil der Standardisierung, der dieses System so interessant macht. Es wird sich aufgrund von Veränderungen auf der physikalischen Ebene kaum oder sehr wenig ändern und heute gemachte Vorlaufentwicklungen werden auch in Zukunft funktionstüchtig bleiben. Bis jetzt haben nur sehr wenige Unternehmen diese Tatsachen verstanden und ihre Vorlaufentwicklungen auf die Integration von IEEE 1394 in industrielle Umgebungen ausgerichtet. Die Zeit ist aber reif dafür.

— ♢ —

5.2. Verwendete Hilfsmittel

Die Hilfsmittel und Werkzeuge, die für die Erstellung und Bearbeitung dieser Diplomarbeit nötig waren, stammen alle aus dem Bereich der Open Sources Projekte. Die Softwareentwicklung wurde unter Linux mit einem einfachen Editor vorgenommen. Die verwendete Linuxdistribution ist die „Halloween Linux Volume III“ auf Basis einer „Red Hat 6.0 GPL“. Folgende Programme wurden primär benutzt.

Programmierung

- GNU C Compiler 2.91.66 19990314 (egcs)
- GNU BIN Utilities 2.9.1.0.23
- GNU make 3.77
- GNU diffutils 2.7
- GNU patch 2.5
- GNU automake 1.4a
- GNU autoconfig 2.13
- Tcl/Tk 8.0
- Visual Tcl 1.10
- Concurrent Versions System (CVS) 1.10.5

Dokumentation

- LaTeX 2_ε + KOMA-Script 2.5e
- longtable 1998/05/13
- varioref 1999/02/24
- hyperref Juni 1998
- pdfTeX 0.10
- GhostScript 5.10 mit GhostView 3.5.8
- Adobe Acrobat Reader 4.0 for Linux

Sonstiges

- VIM, Vi IMproved 5.3
- Midnight Commander 4.5.31
- GNU tar 1.12
- GNU zip 1.2.4

Literaturverzeichnis

[ACIM99] Apple, Compaq, Intel, Microsoft, ...: *1394 Open Host Controller Interface Specification* / Release 1.06 **10. August 1999**
Ref. auf Seite: 63

[And98] Anderson, Don: *FireWire® System Architecture : IEEE 1394* / MindShare Inc. — 2. Auflage — Addison Wesley **1998**
ISBN : 0-201-69470-0
FHJ : 54.21 98/24757
Ref. auf Seite: 19, 22, 29, 30, 30, 32, 32, 34, 35, 97, 98

[ARC97] Automation Research Corporation (ARC): *Marktstudie zum Thema Feldbusse* / http://www.arcweb.com/Public/studies/docs/dnet.htm im **Jan. 1997**
Ref. auf Seite: 103, 103

[Beck99] Beck, Michael ...: *Linux-Kernel-Programmierung : Algorithmen und Strukturen der Version 2.2* / 5. Auflage — Addison Wesley **1999**
ISBN : 3-8273-1476-3
FHJ : 54.54-606 99/07530
Ref. auf Seite: 78

[Card98] Card, Rémy ...: *The Linux Kernel Book : englische Übersetzung von Chris Skrimshire* / John Wilay & Sons Ltd. **1998**
ISBN : 0-471-98141-9
FHJ : 54.54-49 98/14566
Ref. auf Seite: 78

[CIM98] Compaq, Intel, Microsoft: *device BAY : Device Bay Interface Specification* / Revision 0.90 **6. November 1998**
Ref. auf Seite: 65

[Dank94] DANKMEIER, Wilfried: *Codierung : Fehlerbeseitigung und Verschlüsselung* / DuD Fachbeiträge — Vieweg **1994**
ISBN : 3-528-05399-2
FHJ : 54.62-34 94/04466
Ref. auf Seite: 45

[Dem97] DEMBOWSKI, Klaus: *Feuerdraht : FireWire und andere serielle Bussysteme* / c't Heft 2 **1997**, Seite 284–290
Ref. auf Seite: 19

[Fos97] FOSTER-JOHNSON, Eric: *Graphical Applications with Tcl and Tk* / 2. Auflage — M&T Books (IDG Books Worldwide, Inc.) **1997**
ISBN : 1-55851-569-0
FHJ : 54.53-847 99/07568
Ref. auf Seite: 85

[IEEE91] IEEE: *ANSI/IEEE 1212-1991 : IEEE Standard Control and Status Register (CSR) Architecture for Microcomputer Buses* / **1991**
Ref. auf Seite: 11, 19, 26

[IEEE95] IEEE: *IEEE 1394-1995 : IEEE Standard for a High Performance Serial Bus* / **1995**
Ref. auf Seite: 19, 19, 103

[ISO84] ISO: *ISO 7498 : OSI Reference Model* / **1984**
Ref. auf Seite: 23

[Kro97] KROLL, Joachim: *FireWire — der Multimedia-Bus* / Elektronik Heft 13 **1997**, Seite 46–54
Ref. auf Seite: 19, 29

[Kro98] KROLL, Joachim: *Der PC fürs Wohnzimmer* / Elektronik Heft 2 **1998**, Seite 46–50
Ref. auf Seite: 65

[Kro99] KROLL, Joachim: *Sprachstandard für Industrie-Equipment am 1394-Bus* / Elektronik Heft 17 **1999**, Seite 40–43
Ref. auf Seite: 49

[Kro99a] KROLL, Joachim: *IEEE 1394 / Firewire — Auch die Industrie zeigt Interesse* / Elektronik Heft 17 **1999**, Seite 24–28(30)
Ref. auf Seite: 90

[Lnz98] LINZ, Stephan: *Einführung eines Feldbussystems : Erarbeitung einer Konzeption für eine optimale Signalführung im Grundgerät einer Elektronenstrahlbelichtungsanlage unter Verwendung standardisierter Feldbussysteme.* / FH Jena, Leica Microsystems Lithography GmbH / **SS 1998**
Ref. auf Seite: 43, 45, 101, 103

[Ost97] PROF. DR. OSTRITZ, Werner: *Vorlesungsmitschriften „Digitale Nachrichtentechnik“ : Vorlesung nach Auszügen aus O. Georg: Telekommunikationstechnik; Springer 1996* / **SS 1997**
Ref. auf Seite: 23

[Oust95] OUSTERHOUT, John K.: *Tcl und Tk* / Addison Wesley **1995**
`ISBN : 3-89319-793-1`
`FHJ : 54.54-284 95/08979`
Ref. auf Seite: 85

[P95] IEEE: *P1394 : IEEE Standard for a High Performance Serial Bus* / proposal Draft 7.1v1 **5. August 1994**
Ref. auf Seite: 19, 22, 23, 25, 25, 29, 31, 32, 32, 97, 98

[P97.1] IEEE: *P1394.1 : Draft Standard for a High Performance Serial Bus Bridges* / proposal Draft 0.03 **18. Oktober 1997**
Ref. auf Seite: 41, 103

[P99a] IEEE: *P1394a : Draft Standard for a High Performance Serial Bus (Suplement)* / proposal Draft 4.0 **15. September 1999**
Ref. auf Seite: 40, 103

[P99b] IEEE: *P1394b : Draft Standard for a High Performance Serial Bus (Suplement)* / proposal Draft 0.92 **10. November 1999**
Ref. auf Seite: 40, 103

[Pei99] PEISKER, Peter: *High-Speed-Bus IEEE 1394: Reif für die Anwendung (Teil 1/2)* / Elektronik Heft 6 **1999**, Seite 42–46 und Heft 8 **1999**,

Seite 108–111
Ref. auf Seite: 19, 63

[prEN97] CENELEC TC 65CX: *prEN 50254 : High efficiency communication subsystem for small data packages* / final Draft **Juli 1997**
Ref. auf Seite: 103

[Rub98] Rubini, Alessandro: *Linux-Gerätetreiber : Deutsche Übersetzung von Matthias Kalle Dalheimer* / Oreilly **1998**
ISBN : 3-89721-122-X
Ref. auf Seite: 78

[T10] SCSI Trade Association: *T10 Technical Committee : SCSI-3 Standards Architecture* / http://www.t10.org/ am **8. November 1999**
Ref. auf Seite: 20, 48

[Tee99] Teener, Michael D. Johas: *Ein Führer durch die Papierflut der IEEE 1394-Standards* / Elektronik Heft 17 **1999**, Seite 44–50
Original: „1394 Standards and specifications Summary“ / Zayante, Inc.
Ref. auf Seite: 19, 48, 107

[TI97] Texas Instruments: *SLLS273 : (PCILynx) IEEE 1394-1995 BUS TO PCI INTERFACE* / **1997**
Ref. auf Seite: 66

[TI97a] Texas Instruments: *PCILynx : 1394 to PCI Bus Interface — Functional Specification* / Revision 1.2 **1997**
Ref. auf Seite: 67

[TI98] Texas Instruments: *SLLS306 : (PCILynx-2) IEEE 1394 LINK LAYER CONTROLLER* / **1998**
Ref. auf Seite: 66

[Urb97] Urban, Gregory A.: *Angefeuert zu neuen Anwendungen — „FireWire“ – Brücke zwischen Unterhaltungselektronik und PC* / Elektronik Heft 9 **1997**, Seite 92–98
Ref. auf Seite: 19

A. IEEE 1394

A.1. Steckverbindung

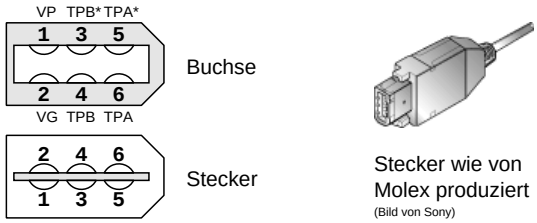

Abbildung A.1.: IEEE 1394 Steckverbinder — Buchse & Stecker

Kontakt	Signal	Bedeutung
1	VP	Betriebsspannung im Kabel (Spannungsbereich 8-40 V DC / 1,5 A)
2	VG	Masse im Kabel
3	TPB*	Twisted pair B — Differenzsignal STROBE
4	TPB	
5	TPA*	Twisted pair A — Differenzsignal DATA
6	TPA	

Tabelle A.1.: IEEE 1394 Steckverbinder — Signale

Die Kontakte 1 und 2 der Betriebsspannung sind länger als die der Datenleitungen, um beim Anschließen erst die Spannungen zuzuschalten ([And98] Seite 64ff. und [P95] 4.2.1–4.2.2 Seite 51ff.).

A.2. Kabeldaten

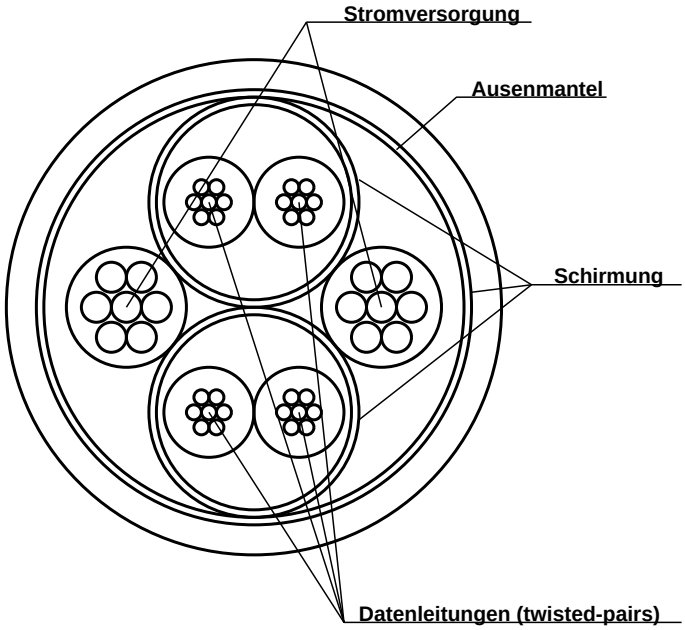

Abbildung A.2.: IEEE 1394 Kabelaufbau — Querschnitt

- maximale Kabellänge 4,5 m
- TPA/TPA* und TPB/TPB* sind im Kabel miteinander gekreuzt
- Impedanz der Twisted-pair Leitungen für Differenzsignale 110 Ω — für Stromversorgung 33±6 Ω
- Signallaufzeit $\leq$ 5.05 ns/m
- Signaldämpfung:

  $\leq$ 2.3 dB S100 (98,304 Mbps $\approx$ 100 MHz)

  $\leq$ 3.2 dB S200 (196,608 Mbps $\approx$ 200 MHz)

  $\leq$ 5.8 dB S400 (393,216 Mbps $\approx$ 400 MHz)
- Übersprechen TPA/TPB $\leq$ -26 dB in einem Bereich von 1 MHz bis 500 MHz

([And98] Seite 67ff. und [P95] 4.2.1.2 Seite 60ff.)

A.3. Berechnung der Übertragungsgeschwindigkeit

Diese Herleitung ist erst mit Verständnis des ersten Kapitels und u.U. mit Hilfe von Hintergrundinformationen aus dem Standard IEEE 1394 nachvollziehbar.

Vorgaben, Voraussetzungen

1. die Frequenz der Buszyklen beträgt 8 kHz $\rightarrow$ Zykluszeit $t_Z = 125\mu s$
2. der Cycle Master als synchronisierende Einheit zählt die Zyklen in einem Register, wofür 2^{12} Bit für einen Überlaufzähler existieren $(WB : 0 \ldots 3071)$
3. dieser Zähler läuft bei Abschluß eines Zyklus mit dem Zählerstand $Z = 3072$ über
4. die Übertragungsgeschwindigkeit ist ein Vielfaches des Basistaktes zur Basis 2

Herleitung

1. Basistakt für den Zähler:

$$f_B = \frac{Z}{t_Z} = \frac{3072}{125\mu s} = 24,576MHz \tag{A.1}$$

2. aus A.1 abgeleitet und umgestellt:

$$\begin{aligned} \frac{3072}{125\mu s} &= 3072 \cdot 8kHz = 24,576MHz \\ (2^{11} + 2^{10}) \cdot 2^3 kHz &= 24,576MHz \\ (2^1 + 2^0) \cdot 2^{13} kHz &= 24,576MHz \end{aligned}$$

$$2^{13} \cdot 3kHz = 24,576MHz \tag{A.2}$$

3. in A.2 Vielfaches zur Basis 2 einführen (*Geometrische Reihe*):

$$\begin{aligned} f_{\ddot{U}} = f(m = \{0, 1, \ldots\}) = 2^m \cdot 2^{13} \cdot 3kHz &= \{24,576MHz; 49,152MHz; \ldots\} \\ &\downarrow \quad wobei : m = n - 13 \\ f_{\ddot{U}} = f(n = \{13, 14, \ldots, 17\}) &= 2^n \cdot 3kHz \end{aligned} \tag{A.3}$$

B. IEEE 1394 vs. InterBus

B.1. Tabellarische Gegenüberstellung

	InterBus	**IEEE 1394**
Haupteigenschaft	Feldbus	Multimediabus
Masteranzahl	1	(multimasterfähig)
Teilnehmerzahl	256	63
Topologie	Ring (als Baum)	Baum
Leiterzahl, Übertragungsart	- 3×2 geschirmt - LWL	- 2×2 + 2×1 geschirmt - zukünftig auch UTP5 (wie Ethernet) und LWL
Leitungslänge	- 50 m zwischen Knoten - max. 13 km	- 4,5 m zwischen Knoten - max. 72 m - zukünftig bis zu 100 m zwischen Knoten

Tabelle B.1.: Vergleich IEEE 1394, InterBus (b. w.)

	InterBus	IEEE 1394
max. Übertragungsrate	- 0,5 Mbps - geplant 2 Mbps	- 100, 200 & 400 Mbps für Kabelvariante - 25 & 50 Mbps für Backplanevariante - zukünftig bis zu 3200 Mbps, aber nur 100 Mbps für UTP5 basierende Systeme
Reaktionszeiten	- 1,2...17 ms; last- bzw. strukturabhängig; siehe [Lnz98] 7.2.6 Seite 36	- garantierte 125 μs durch 8 kHz Buszyklus
Zugriffsverfahren	- Master-Slave - Summenrahmen	- Fairneßintervall
Anzahl Datenbytes	- bis zu 512 Byte (4096 Bit) Prozeß- und Parameterdaten gemischt	- isochronen Daten stehen 7/8 der Bandbreite zu (Prozeßdaten) - Paketgröße asynchroner Daten ist bandbreitenabhängig (Parameterdaten)

Tabelle B.1.: Vergleich IEEE 1394, InterBus (b. w.)

	InterBus	IEEE 1394
OSI Schicht 1	- asynchrone serielle Schnittstelle mit symmetrischem Signal nach RS-485 - NRZ-Coding - Taktsynchronisation durch Start-Stop-Bit - SUB-D 9 Stecker - Umsetzung zu LWL vorhanden	- synchrone serielle Schnittstelle mit standardeigenem symmetrischem Signal - NRZ-Coding - Taktrückgewinnung durch XOR-Verknüpfung aus DATA und STROBE - standardeigenes 6/4-poliges Steckersystem - zukünftig 8B10B-Coding mit Verwürfelung und für Raten > 400 Mbps neues Steckersystem - Steckersystem speziell für Industrieanwendung wird noch entwickelt
OSI Schicht 2	- 16 Bit CRC - Stopbitkontrolle - Timeouts - Request-Response-Verfahren für Parameterdaten	- 32 Bit CRC - Kontrolle von Transactioncodes - Request-Response-Verfahren für asynchrone Pakete

Tabelle B.1.: Vergleich IEEE 1394, InterBus (b. w.)

	InterBus	IEEE 1394
OSI Schicht 7	- über das *Application Layer Interface* stehen verschiedene Dienste für den objektorientierten Zugriff auf Prozeß- und Parameterdaten zur Verfügung; siehe [Lnz98] 7.4 Seite 42	- höhere Transportschichten werden nicht durch IEEE 1394 selbst definiert - das *Industrial & Instrumentation Control Protocol* (IICP), bisher einziger Ansatz, stellt nur eine Zwischenschicht dar — kein wirklicher Anwenderzugriff
Standards	[prEN97]	[ARC97] [IEEE95] [P99a] [P99b] [P97.1]
Weltmarktanteile	37,4 % seit mehreren Jahren [ARC97]	im Feldbusbereich so gut wie keine

Tabelle B.1.: Vergleich IEEE 1394, InterBus

C. Systemintegration

Alle in diesem Abschnitt genannten Internetreferenzen beziehen sich auf einen Verfügbarkeitszeitraum von Oktober bis November 1999.

C.1. Anbieter von IEEE 1394 Chipsets

- ***Cirrus Logic, Inc.***:
  spez. Singelchiplösung (LLC+PHY) für die Integration von IEEE 1394 in Disk Drive Controller Boards; bietet einen Datenpuffer aus SDRAM mit einem Durchsatz von 109,3 MBps
  http://www.cirrus.com/products/overviews/sh7760.html

- ***FUJIFILM Microdevices Co., Ltd.***:
  LLC mit generischem 8, 16 bzw. 32 Bit Hostinterface; 3-Port Kabel-PHY; S100. . .S400; unterstützt IEEE 1394a
  http://www.fujifilm.co.jp/ffm/

- ***Fujitsu Electronics***:
  Singelchip-Lösung (LLC+PHY) mit OHCI; S400; S800-Kabel-PHY in Entwicklung
  http://www.fujitsu-fme.com/products/multi/09.html

- ***IBM — International Business Machines Corporation***:
  LLC-to-PCI-Bridge; 3-Port Kabel-PHY; S200. . .S400; unterstützt IEEE 1394a
  http://www.chips.ibm.com/techlib/products/1394/prodbriefs.html

- ***NEC Electronics, Inc.***:
  Singelchip-Lösung (LLC+PHY) mit OHCI; solo LLC und 3-Port Kabel-PHY; S400; unterstützt IEEE 1394a;
  besonderes: **erste Realisierung eines Kabel-PHY nach 1394b** (lange Strecken, optisch, drahtlos, UTP-CAT5)
  http://www.necel.com/

- ***Oki Semiconductor***:
  LLC mit generischem Hostinterface; arbeitet optional mit einem ext. FIFO-Controller zusammen; S100...S400
  http://www.okisemi.com/communicator/public/fm/docs/Intro-1230.html

- ***OPTi, Inc.***:
  LLC mit OHCI (kompatibel zu Texas Instruments OHCI Chips); 3-Port Kabel-PHY; S100...S400; unterstützt IEEE 1394a
  besonders: *Arbeitsunterlagen für eine USB-1394-CombiController Add-On PCI-Karte*
  http://www.opti.com/html/usb1394.html

- ***Philips Semiconductors***:
  LLC spez. für A/V; 3-Port Kabel-PHY; S100...S400; unterstützt IEEE 1394a;
  besonderes: **SBP-2 Highlevel-Protocol-Chip SAA7356HL mit Programmierinterface zu existierenden SCSI Controller ASIC**
  http://www-us.semiconductors.philips.com/1394/
  http://www-us.semiconductors.philips.com/1394/products/

- ***Sony Electronics, Inc.***:
  LLC-to-PCI-Bridge; LLC spez. für A/V u. SBP-2 mit generischem 8 bzw. 16 Bit Hostinterface; 3-Port Kabel-PHY; S100...S200;
  geplant: LLC mit OHCI; ASICs für allg. Anwendungen und spez. für Printer, Scanner, ...
  http://www.sel.sony.com/semi/ieee1394wp.html

- ***Symbios Logic, Inc.***:
  LLC mit PCI o. OHCI; 3-Port Kabel-PHY; S100...S400
  http://www.symbios.com/

- ***Texas Instruments***:
  LLC mit PCI, OHCI o. generischem 8, 16 bzw. 32 Bit Hostinterface; 1...3-Port Kabel-PHY (auch für Backplanevariante); S100...S400; unterstützt IEEE 1394a
  http://www.ti.com/sc/docs/msp/1394/

- ***VIA Technologies, Inc.***:
  LLC mit OHCI und Device Bay-Interface; S100...S400
  http://www.viatech.com/products/prodviafire.htm

- ***YAMAHA Corporation***:
  spez. Highlevel-Chip (mLAN[1] Pakethändler) für den Verbundbetrieb mit den Chipsets von FUJIFILM
  http://www.yamaha.co.jp/tech/1394mLAN/Products/Products.html

C.2. Anbieter von VHDL-, AHDL-Cores

- ***EnThink, Inc.***:
  LLC mit PCI, OHCI (mit o. ohne PCI), generischem 32 Bit Hostinterface o. spez. A/V-Interface; Kabel-PHY mit 1...16 Ports; S100...S400; unterstützt IEEE 1394a; VerilogHDL; synthesefähig
  http://www.EnThink.com/products/index.html

- ***Innovative Semiconductors, Inc.***:
  LLC mit generischem 8, 16 bzw. 32 Bit Hostinterface, PCI o. spez. für A/V; Kabel-PHY (auch für Backplanevariante); S50, S100...S400; unterstützt IEEE 1394a; synthesefähige RTL für VHDL-Designs;
  besonderes: **LLC mit kompatiblen Hostinterface zu TI GPLynx**
  http://www.isi96.com/products/products.html

[1] mLAN steht für music LAN, ein Vernetzungskonzept für elektronische Musikinstrumente definiert von YAMAHA

- ***Phoenix Technologies, Ltd.***:
  LLC mit generischem 8, 16 bzw. 32 Bit o. user-spez. Hostinterface; Kabel-PHY mit 1...16 Ports; RTLs für VerilogHDL-Designs; unterstützt IEEE 1394a;
  http://www.phoenix.com/ipcores/1394-cphy.html
  http://www.phoenix.com/ipcores/1394-dclc.html
  http://www.phoenix.com/ipcores/1394-core.html

- ***Simple Silicon, Inc.***:
  LLC mit PCI o. ISA; PHY; S100...S400; VerilogHDL und AlteraHDL; simulations- u. synthesefähig
  http://www.simplesi.com/products.htm

- ***Zayante, Inc.***:
  LLC mit anwendungsunabhängigem Hostinterface — darauf aufbauend gibt es Highlevel-Cores für A/V, SBP-2, IP-over-1394 u. generischem Hostinterface (local bus); PHY mit 1...16 Ports; S100...S400; unterstützt IEEE 1394a; VerilogHDL o. VHDL; simulations- u. synthesefähig;
  besonders: *diese Firma wurde vom leitenden IEEE 1394 Entwickler von Apple, M.D. Johas Teener, gegründet und verfügt durch eine aktive Rolle innerhalb der IEEE 1394 Trade Association über ein sehr breites technisches Wissen in Bezug auf die fortschreitende Entwicklung [Tee99]*
  http://www.zayante.com/html/products/Silicon_Text.html

C.3. Anbieter von eigenständigen IEEE 1394 Stacks

- ***Intoto, Inc.***:
  „FireStack“ für versch. RTOS; Plattformen: PPC, ARM, StrongARM, MIPS, DSP, x86; Chipsets von: TI, IBM, NEC, Fujitsu, Philips, OHCI Chips; höhere Protokolle: IP-over-1394, SBP-2, DPP, A/VC, HAVi, mLAN, IICP
  besonders: *benutzt WindRiver für die Integration von IEEE 1394 unter Tornado™*
  http://www.intotoinc.com/1394/fire.htm

- ***Phoenix Technologies, Ltd.***:
  „FireAccess“ für versch. RTOS und WindowsCE; Chipsets von: TI, OHCI Chips; höhere Protokolle: IP-over-1394, SBP-2, DPP, A/VC, HAVi
  besonders: *benutzt QNX für die Integration von IEEE 1394*
  http://www.edtn.com/embeddedreview/1999/er99005.html

- ***Sederta, Inc.***:
  „SedNet“ für WinNT, VxWorks, QNX, LynxOS uvm.; Protokolle: IP-over-1394, A/VC
  http://www.sederta.com/product/api.htm

- ***Thesycon® Systemsoftware & Consulting GmbH***:
  „Versatile IEEE 1394 Device Driver“ für Win98, Win200 und WinCE;
  http://www.thesycon.de/

- ***Zayante, Inc.***:
  „TNF™“ für versch. RTOS; Plattformen: anwenderspez.; Chipsets von: TI, Fujitsu, OHCI Chips; höhere Protokolle: IP-over-1394, SBP-2, DPP, A/VC
  http://www.zayante.com/html/products/Software_Text.html

D. Demonstrator

D.1. Hardware

Der Aufbau des Demonstrators besteht aus den folgenden Kaufkomponnenten:

- 2 Standard-PC, Pentium-Klasse mit 32 MB Hauptspeicher
- 2 IEEE 1394 Anschaltbaugruppen (PCI Add-On):
  1. ***FireBord^{TM} 400***: von unibrain[1] (Grichenland); bestehend aus:
     - LLC-PCI-Bridge von Texas Instruments: TSB12LV21B (PCILynx2)
     - IEEE 1394a S400 PHY (3 Ports) von Texas Instruments: TSB41LV03
  2. ***AHA®-8920***: von adaptec, Inc.[2] (Kanada); bestehend aus:
     - LLC-PCI-Bridge von Adaptec: AIC-5800 („Pele“ Derivat)
     - S200 PHY (3 Ports) von IBM: 21S750PFC (nur 1 Port herausgeführt)

[1] http://www.unibrain.com/
[2] http://www.adaptec.com/1394/

D.2. Software

D.2.1. C Quellen von hl_sample.[ch]

D.2.1.1. hl_sample.h

```
/*
 * hl_sample.h - Linux 1394 High Level Sample Memory driver header file
 * 1999 Stephan Linz <linz@mazet.de>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

#ifndef __IEEE1394_HIGHLEVEL_HL_SAMPLE_H
#define __IEEE1394_HIGHLEVEL_HL_SAMPLE_H

#ifdef CONFIG_IEEE1394_HIGHLEVEL_SAMPLE_BASE
#  define HLS_MEM_START CONFIG_IEEE1394_HIGHLEVEL_SAMPLE_BASE
#else
#  define HLS_MEM_START 0x0
#endif

#ifdef CONFIG_IEEE1394_HIGHLEVEL_SAMPLE_QUATNUM
#  define HLS_QUAT_NUM CONFIG_IEEE1394_HIGHLEVEL_SAMPLE_QUATNUM
#else
#  define HLS_QUAT_NUM  (128 / sizeof(quadlet_t))
#endif

#define HLS_NAME           "hl_sample"

struct hls_mblock {
    struct hls_mblock *next;
    struct hpsb_host  *host;
    quadlet_t         memory[HLS_QUAT_NUM];
};

#endif /* __IEEE1394_HIGHLEVEL_HL_SAMPLE_H */
```

D.2.1.2. hl_sample.c

```
/*
 * hl_sample.c - Linux 1394 High Level Sample Memory driver
 * 1999 Stephan Linz <linz@mazet.de>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software Foundation,
 * Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/slab.h>

#include "../ieee1394_types.h"
#include "../ieee1394.h"
#include "../hosts.h"
#include "../highlevel.h"

#include "hl_sample.h"


/* print general informations */
#define HLS_PRINT(level, fmt, args...) printk(level HLS_NAME ": " fmt "\n" , ## args)
#define ERROR(fmt, args...) HLS_PRINT(KERN_ERR, fmt , ## args)
#define INFO(fmt, args...) HLS_PRINT(KERN_INFO, fmt , ## args)

/* print host specific informations */
/* FIXME get id in better way */
#define ERROR_H(fmt, args...) ERROR("%s%d: " fmt , host->template->name , \
    *(int *)host->hostdata , ## args)
#define INFO_H(fmt, args...) INFO("%s%d: " fmt , host->template->name , \
    *(int *)host->hostdata, ## args)


static struct hpsb_highlevel *hls_hook = NULL;

static struct hls_mblock *mblocks = NULL;
static int num_of_mblocks;
rwlock_t mblocks_lock = RW_LOCK_UNLOCKED;
```

```
void hls_host_added(struct hpsb_host *host);
void hls_host_removed(struct hpsb_host *host);
void hls_host_reseted(struct hpsb_host *host);

static struct hpsb_highlevel_ops hls_ops = {
    hls_host_added,
    hls_host_removed,
    hls_host_reseted
};

int hls_addr_read(struct hpsb_host *host, quadlet_t *buffer, u64 addr,
                  unsigned int length);
int hls_addr_write(struct hpsb_host *host, quadlet_t *data, u64 addr,
                   unsigned int length);
int hls_addr_lock(struct hpsb_host *host, quadlet_t *store, u64 addr,
                  quadlet_t data, quadlet_t arg, int ext_tcode);
int hls_addr_lock64(struct hpsb_host *host, quadlet_t *store, u64 addr,
                    quadlet_t data, quadlet_t arg, int ext_tcode);

static struct hpsb_address_ops hls_addr_ops = {
    hls_addr_read,
    hls_addr_write,
    NULL, /* no lock */
    NULL  /* no lock64 */
};


/*
 * helper functions:
 */
struct hls_mblock *lookup_hls_mblock(struct hpsb_host *host)
{
    struct hls_mblock *lup, *ptr;

    for (ptr = lup = mblocks;
            lup->next != NULL && lup->host != host;
            ptr = lup, lup = lup->next) ;
    if (lup->host != host) {
        return NULL;
    }

    return lup;
}

void reset_hls_mblock(struct hls_mblock *mblk)
{
    /* NOTE: misc stuff to reset an element place here */
    memset(&mblk->memory, 0, sizeof(mblk->memory));
}
```

```
void free_hls_mblocks(void)
{
    struct hls_mblock *old;

    write_lock_irq(&mblocks_lock);
    for (old = mblocks; old != NULL; old = mblocks) {
        mblocks = old->next;
        num_of_mblocks--;
        kfree(old);
    }
    write_unlock_irq(&mblocks_lock);
}


/*
 * driver's structure control specific functions:
 */
void hls_host_added(struct hpsb_host *host)
{
    struct hls_mblock *new, *ptr;

    new = (struct hls_mblock *)kmalloc(
            sizeof(struct hls_mblock), GFP_KERNEL);
    if (!new) {
        ERROR("no ressources for new memory block");
        return;
    }

    memset(new, 0, sizeof(struct hls_mblock));
    new->next = NULL;
    new->host = host;

    write_lock_irq(&mblocks_lock);
    if (!mblocks) {
        mblocks = new;
        num_of_mblocks++;
    } else {
        for (ptr = mblocks; ptr->next != NULL; ptr = ptr->next) ;
        ptr->next = new;
        num_of_mblocks++;
    }
    write_unlock_irq(&mblocks_lock);

    INFO_H("added, hold %d memory block(s) each 0x%02x byte",
            num_of_mblocks, sizeof(new->memory));
}

void hls_host_removed(struct hpsb_host *host)
{
    struct hls_mblock *old, *ptr;
```

```
    write_lock_irq(&mblocks_lock);
    if (!mblocks) {
        ERROR(__FUNCTION__ "(): run without have a memory block list");
        write_unlock_irq(&mblocks_lock);
        return;
    } else {
        for (ptr = old = mblocks;
                old->next != NULL && old->host != host;
                ptr = old, old = old->next) ;
        if (old->host == host) {
            if (old == ptr)
                mblocks = old->next;
            else
                ptr->next = old->next;
            num_of_mblocks--;
            kfree(old);
            INFO_H("removed, hold %d memory block(s)",
                    num_of_mblocks);
        } else {
            ERROR_H("can't remove, it haven't had a memory block");
        }
    }
    write_unlock_irq(&mblocks_lock);
}

void hls_host_reseted(struct hpsb_host *host)
{
    struct hls_mblock *rst;

    write_lock_irq(&mblocks_look);
    if (!mblocks) {
        ERROR(__FUNCTION__ "(): run without have a memory block list");
        write_unlock_irq(&mblocks_lock);
        return;
    } else {
        if ((rst = lookup_hls_mblock(host))) {
            reset_hls_mblock(rst);
            INFO_H("reseted, hold %d memory block(s)",
                    num_of_mblocks);
        } else {
            ERROR_H("can't reset, it haven't had a memory block");
        }
    }
    write_unlock_irq(&mblocks_look);
}


/*
 * address access specific functions:
 */
int hls_addr_read(struct hpsb_host *host, quadlet_t *buffer, u64 addr,
```

```
                unsigned int length)
{
    struct hls_mblock *mblk;
    int i, j, retval = RCODE_TYPE_ERROR;

    write_lock_irq(&mblocks_lock);
    if (!mblocks) {
        ERROR(__FUNCTION__ "(): run without have a memory block list");
        write_unlock_irq(&mblocks_lock);
        return retval;
    } else {
        if ((mblk = lookup_hls_mblock(host))) {
            if (addr & 0x3 || addr < HLS_MEM_START ||
                addr + length > HLS_MEM_START + 4 * HLS_QUAT_NUM) {
                ERROR_H(__FUNCTION__ "(): worng address format");
                retval = RCODE_ADDRESS_ERROR;
            } else {
                /* FIXME - Block transfer size which are not a
                 * multiple of 4 are not handled correctly */
                for (i = (addr - HLS_MEM_START) / 4, j = 0;
                     j < length / 4; i++, j++) {
                    buffer[j] = mblk->memory[i];
                }
                retval = RCODE_COMPLETE;
                INFO_H("read on offset %d / %d quadlets",
                        i-j, j);
            }
        } else {
            ERROR_H("read failed, haven't had a memory block");
        }
    }
    write_unlock_irq(&mblocks_lock);

    return retval;
}

int hls_addr_write(struct hpsb_host *host, quadlet_t *data, u64 addr,
                   unsigned int length)
{
    struct hls_mblock *mblk;
    int i, j, retval = RCODE_TYPE_ERROR;

    write_lock_irq(&mblocks_lock);
    if (!mblocks) {
        ERROR(__FUNCTION__ "(): run without have a memory block list");
        write_unlock_irq(&mblocks_lock);
        return retval;
    } else {
        if ((mblk = lookup_hls_mblock(host))) {
            if (addr & 0x3 || addr < HLS_MEM_START ||
                addr + length > HLS_MEM_START + 4 * HLS_QUAT_NUM) {
```

```
                ERROR_H(__FUNCTION__ "(): worng address format");
                retval = RCODE_ADDRESS_ERROR;
            } else {
                /* FIXME - Block transfer size which are not a
                 * multiple of 4 are not handled correctly */
                for (i = (addr - HLS_MEM_START) / 4, j = 0;
                     j < length / 4; i++, j++) {
                    mblk->memory[i] = data[j];
                }
                retval = RCODE_COMPLETE;
                INFO_H("write on offset %d / %d quadlets",
                        i-j, j);
            }
        } else {
            ERROR_H("write failed, haven't had a memory block");
        }
    }
    write_unlock_irq(&mblocks_lock);

    return retval;
}

int hls_addr_lock(struct hpsb_host *host, quadlet_t *store, u64 addr,
                  quadlet_t data, quadlet_t arg, int ext_tcode)
{
    return RCODE_TYPE_ERROR;
}

int hls_addr_lock64(struct hpsb_host *host, quadlet_t *store, u64 addr,
                    quadlet_t data, quadlet_t arg, int ext_tcode)
{
    return RCODE_TYPE_ERROR;
}


/*
 * ----------------------------------------
 */
int cleanup_hls(void)
{
    hpsb_unregister_highlevel(hls_hook);
    free_hls_mblocks(); /* safety been safety */

    INFO("IEEE-1394 High Level Sample Memory driver removed");
    return 0;
}

int init_hls(void)
{
    hls_hook = hpsb_register_highlevel(HLS_NAME, &hls_ops);
```

```
    if (!hls_hook) {
        ERROR("error while general ops registration");
        return -ENOMEM;
    }

    if (!(hpsb_register_addrspace(hls_hook, &hls_addr_ops,
            HLS_MEM_START, HLS_MEM_START + 4 * HLS_QUAT_NUM))) {
        hpsb_unregister_highlevel(hls_hook);
        ERROR("error while address ops registration");
        return -ENOMEM;
    }

    INFO("IEEE-1394 High Level Sample Memory driver loaded (0x%02x,0x%02x)",
            HLS_MEM_START, HLS_QUAT_NUM);
    return 0;
}

#ifdef MODULE

/* EXPORT_NO_SYMBOLS; */

MODULE_AUTHOR("Stephan Linz <linz@mazet.de>");
MODULE_DESCRIPTION("Linux IEEE-1394 High Level Sample Memory module");
MODULE_SUPPORTED_DEVICE("hl_sample");

void cleanup_module(void)
{
    cleanup_hls();
}

int init_module(void)
{
    return (init_hls());
}

#endif /* MODULE */
```

D.2.2. C und Tcl/Tk Quellen von elias v0.0.3

D.2.2.1. eliasApp.h

```
/* eliasApp.h
 * Tcl/Tk application & command-procedures for IEEE-1394 demo
 *
 * Copyright 1999 Stephan Linz <linz@mazet.de, Stephan.Linz@gmx.de>
 */


#ifdef HAVE_CONFIG_H
#  include"config.h"
#endif

#include<sys/types.h>


typedef u_int8_t                byte_t;
typedef u_int16_t               doublet_t;
//typedef u_int32_t               quadlet_t;
//typedef u_int64_t               octlet_t;


typedef enum {offline, online}  eliasStat_t;


#define ELIAS_NEWTARGET_RDADDR  0
#define ELIAS_NEWTARGET_RDNODE  1
#define ELIAS_NEWTARGET_WRADDR  2
#define ELIAS_NEWTARGET_WRNODE  3


#define ELIAS_GLOBALSTRINGSIZE  1024


#define ELIAS_DEBUG_PRINT(fmt, args...) fprintf(stderr, __FILE__ "(" \
        __FUNCTION__ ", %d): \n  `--->\t" fmt "\n", __LINE__, ## args);


#define ELIAS_REGIST_TCLCMD(intpr, tclname, cname, cdptr, delfuncptr) \
    if (Tcl_CreateCommand(intpr, tclname, cname, \
            (ClientData)cdptr, (Tcl_CmdDeleteProc *)delfuncptr) == TCL_OK) { \
        return TCL_ERROR; }

#define ELIAS_REGIST_TCLVAR(intpr, tclname, cname, flags) \
    if (Tcl_LinkVar(intpr, tclname, (char *)&cname, flags) != TCL_OK) { \
        return TCL_ERROR; }

#define ELIAS_REGIST_TCLTRC(intpr, tclname, flags, cfuncname, cdptr) \
    if (Tcl_TraceVar(intpr, tclname, flags, cfuncname, \
```

```
            (ClientData)cdptr) != TCL_OK) { \
        return TCL_ERROR; }

#define ELIAS_REGIST_TCLTRC2(intpr, tclname, tclidx, flags, cfuncname, cdptr) \
    if (Tcl_TraceVar2(intpr, tclname, tclidx, flags, cfuncname, \
            (ClientData)cdptr) != TCL_OK) { \
        return TCL_ERROR; }


#define ELIAS_APPERROR_TCLRES(intpr, errmessages...) \
    Tcl_AppendResult(intpr, __FILE__, "(", __FUNCTION__, "): ", \
            ## errmessages, (char *)NULL);

#define ELIAS_APPERROR_TCLTRC(errmessages...) \
    (__FILE__ "(" __FUNCTION__ "): " ## errmessages);


/* ---------------------------------------- vim-modeline (don't remove it):
 * vi: set et ai sw=4 ts=4:
 */
```

D.2.2.2. eliasApp.c

```
/* eliasApp.c
 * Tcl/Tk application & command-procedures for IEEE-1394 demo
 *
 * Copyright 1999 Stephan Linz <linz@mazet.de> <Stephan.Linz@gmx.de>
 */

#include"eliasApp.h"

#ifdef STDC_HEADERS
#  include<stdio.h>
#  include<stdlib.h>

#  include<signal.h>
#  include<sys/stat.h>
#  include<fcntl.h>
#  include<linux/ioctl.h>

#  ifdef HAVE_TK_H
#    include<tk.h>

#    ifdef HAVE_RAW1394_H
#      include<raw1394.h>


/* importend vars needed to build read/write accesses over IEEE-1394 */
static eliasStat_t       eliasStatus = offline;
static octlet_t          eliasU64VarReadAddr, eliasU64VarWriteAddr;
static quadlet_t         eliasU32VarReadValue, eliasU32VarWriteValue;

/* vars for internal organization */
static Tcl_Interp       *eliasInterp;
static Tk_TimerToken     eliasTkTimerToken;
static RAW1394config     eliasRaw1394Config;
static RAW1394port       eliasRaw1394Port;

/* this variables have to link to tcl vars */
static int               eliasIntVarTimerInterval = 100;
static int               eliasIntVarTimerGoing = 0;
static int               eliasIntVarMainInitialized = 0;
static int               eliasIntVarBusID = 0x3ff;
static int               eliasIntVarNodeID = 0;
static int               eliasIntVarReadNodeID = 0;
static int               eliasIntVarWriteNodeID = 0;
static char             *eliasStringVarReadAddr = NULL;
static char             *eliasStringVarWriteAddr = NULL;


/*
 * ---------------------------- HELPER -----------------------------
```

```
 */

/* update read/write addresses */
inline octlet_t eliasAddrUpd(octlet_t *value, doublet_t *bid, int *nid)
{
    octlet_t work = *value;

    work &= 0x0000ffffffffffffULL;
    work |= ((octlet_t)(*bid & 0x3ff) << 54);
    work |= ((octlet_t)(*nid & 0x3f) << 48);

    return work;
}

void eliasInitAddr1394(void)
{
    eliasU64VarReadAddr = eliasAddrUpd(&eliasU64VarReadAddr,
            (doublet_t *)&eliasIntVarBusID, &eliasIntVarReadNodeID);
    eliasU64VarWriteAddr = eliasAddrUpd(&eliasU64VarWriteAddr,
            (doublet_t *)&eliasIntVarBusID, &eliasIntVarWriteNodeID);
}

inline byte_t eliasGetNumberOfNodes(void)
{
    if (eliasStatus == offline) {
        return 20;
    } else {
        return (byte_t)raw1394GetNodeCount(eliasRaw1394Port);
    }
}

inline byte_t eliasGetSelfNodeID(void)
{
    if (eliasStatus == offline) {
        return 0;
    } else {
        return (byte_t)raw1394GetPortPhyID(eliasRaw1394Port);
    }
}

inline doublet_t eliasGetSelfBusID(void)
{
    if (eliasStatus == offline) {
        return 0x3ff;
    } else {
         return (doublet_t)raw1394GetBusNumber(eliasRaw1394Port);
    }
}

int eliasRead1394(octlet_t *addr, quadlet_t *quadlet)
{
```

```
    RAW1394quadread *qr;

    if (eliasStatus == offline) {

        *quadlet = (quadlet_t)random();

        ELIAS_DEBUG_PRINT("IEEE-1394: dummy read 0x%08x from 0x%016Lx",
                *quadlet, *addr);

    } else {

        if (!(qr = (RAW1394quadread *)Tcl_Alloc(sizeof(RAW1394quadread)))) {
            ELIAS_DEBUG_PRINT("%s", "can't allocate memory");
            return -1;
        }

        qr->destinationOffset = (eliasU64VarReadAddr & 0x0000ffffffffffffULL);
        qr->generation = -1; /* If -1 filled in by the driver */
        qr->destinationID =
            (nodeid_t)((eliasU64VarReadAddr & 0x003f000000000000ULL) >> 48);
        qr->quadletData = 0xdeadbead; /* Returned by the driver */
        qr->requestStatus = 0xdeadfeed; /* Returned by the driver */
        qr->responseCode = 0xdeadfeed; /* Returned by the driver */

        ELIAS_DEBUG_PRINT("DestNodeID: %u", qr->destinationID);
        ELIAS_DEBUG_PRINT("DestOffset: 0x%016Lx", qr->destinationOffset);

        if (raw1394Quadread(eliasRaw1394Port, qr)) {
            Tcl_SetVar(eliasInterp, "eliasBusState",
                    "IEEE-1394: Quadlet Read Transaction Failed", 0);
            ELIAS_DEBUG_PRINT("%s",
                    "IEEE-1394: Quadlet Read Transaction Failed");
            return -1;
        }

        ELIAS_DEBUG_PRINT("QuadData:   0x%08x", qr->quadletData);
        *quadlet = qr->quadletData;

        return 0;

    }

    return 0;
}

int eliasWrite1394(octlet_t *addr, quadlet_t *quadlet)
{
    RAW1394quadwrite *qw;

    if (eliasStatus == offline) {
```

```
            ELIAS_DEBUG_PRINT("IEEE-1394: dummy write 0x%08x to 0x%016Lx",
                    *quadlet, *addr);

        } else {

            if (!(qw = (RAW1394quadread *)Tcl_Alloc(sizeof(RAW1394quadread)))) {
                ELIAS_DEBUG_PRINT("%s", "can't allocate memory");
                return -1;
            }

            qw->destinationOffset = (eliasU64VarWriteAddr & 0x0000ffffffffffffULL);
            qw->generation = -1; /* If -1 filled in by the driver */
            qw->destinationID =
                (nodeid_t)((eliasU64VarWriteAddr & 0x003f000000000000ULL) >> 48);
            qw->quadletData = *quadlet;
            qw->requestStatus = 0xdeadfeed; /* Returned by the driver */
            qw->responseCode = 0xdeadfeed; /* Returned by the driver */

            ELIAS_DEBUG_PRINT("DestNodeID: %u", qw->destinationID);
            ELIAS_DEBUG_PRINT("DestOffset: 0x%016Lx", qw->destinationOffset);

            if (raw1394Quadwrite(eliasRaw1394Port, qw)) {
                Tcl_SetVar(eliasInterp, "eliasBusState",
                        "IEEE-1394: Quadlet Write Transaction Failed", 0);
                ELIAS_DEBUG_PRINT("%s",
                        "IEEE-1394: Quadlet Write Transaction Failed");
                return -1;
            }

            ELIAS_DEBUG_PRINT("QuadData:   0x%08x", qw->quadletData);

            return 0;

        }

        return 0;
    }

    int eliasEvent1394(void *portptr)
    {
        RAW1394port port = (RAW1394port)portptr;

        ELIAS_DEBUG_PRINT("event registred (code: 0x%016Lx)", port->event.event);

        switch (port->event.event) {

            case EVT_BUS_RESET_STARTED:
                Tcl_SetVar(eliasInterp, "eliasBusState",
                        "IEEE-1394: Bus Reset Started", 0);
                ELIAS_DEBUG_PRINT("%s", "IEEE-1394: Bus Reset Started");
                break;
```

```
        case EVT_BUS_RESET_COMPLETED:
            Tcl_SetVar(eliasInterp, "eliasBusState",
                    "IEEE-1394: Bus Reset Completed", 0);
            ELIAS_DEBUG_PRINT("%s", "IEEE-1394: Bus Reset Completed");

            eliasIntVarBusID = (int)eliasGetSelfBusID();
            eliasIntVarNodeID = (int)eliasGetSelfNodeID();

            Tcl_UpdateLinkedVar(eliasInterp, "eliasAppBusID");
            Tcl_UpdateLinkedVar(eliasInterp, "eliasAppNodeID");

            eliasInitAddr1394();

            break;

        default:
            Tcl_SetVar(eliasInterp, "eliasBusState",
                    "IEEE-1394: Event unknown and remain untreated", 0);
            ELIAS_DEBUG_PRINT("%s", "IEEE-1394: Event remain untreated");

    }

    return 0;
}

void eliasInit1394(void)
{
    eliasRaw1394Config = raw1394NewConfig();
    eliasRaw1394Config->event_handler = eliasEvent1394;
    raw1394SetCardNumber(eliasRaw1394Config, 0); /* unnecessary, 0 is default */

    eliasRaw1394Port = raw1394OpenPort(eliasRaw1394Config);
    if (eliasRaw1394Port) {
        eliasStatus = online;
        ELIAS_DEBUG_PRINT("%s", "could open port -- online");
    } else {
        eliasStatus = offline;
        ELIAS_DEBUG_PRINT("%s", "couldn't open port -- bee offline");
    };

    eliasIntVarBusID = (int)eliasGetSelfBusID();
    eliasIntVarNodeID = (int)eliasGetSelfNodeID();

    eliasInitAddr1394();
}


/*
 * ------------------------ TIME-CALLBACKS ------------------------
 */
```

```
void eliasTimeProc(ClientData clientData)
{
    Tcl_Interp *interp = (Tcl_Interp *)clientData;
    static int eliasTimerLevel = 0;
    unsigned int i;
    char *helpstr;

    ELIAS_DEBUG_PRINT("+++ TIMEREVENT: %d", eliasTimerLevel++);

    if (eliasRead1394(&eliasU64VarReadAddr, &eliasU32VarReadValue)) {
        ELIAS_DEBUG_PRINT("%s", "error while 1394 read");
        return;
    }

    if ((helpstr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE)) != NULL) {

        for (i = 0; i < 32; i++) {
            sprintf(helpstr, "%d", i);
            if (eliasU32VarReadValue & (0x1 << i)) {
                Tcl_SetVar2(interp, "eliasReadValue", helpstr, "1", 0);
            } else {
                Tcl_SetVar2(interp, "eliasReadValue", helpstr, "0", 0);
            }
        }

    } else {
        ELIAS_DEBUG_PRINT("%s", "can't allocate memory");
    }
    Tcl_Free(helpstr);

    eliasTkTimerToken = Tk_CreateTimerHandler(eliasIntVarTimerInterval,
            eliasTimeProc, clientData);
}


/*
 * ---------------------- TRACE-PROCEDURES ----------------------
 */

/* trace write acces of several vars */
char *eliasProcVarWrite(ClientData clientData, Tcl_Interp *interp,
        char *name1, char *name2, int flags)
{
    char *value;
    value = Tcl_GetVar2(interp, name1, name2, flags & TCL_GLOBAL_ONLY);

    if (value != NULL) {
        if (name2 == NULL) {
            fprintf(stderr, "you write %s with: %s\n", name1, value);
        } else {
```

```
            fprintf(stderr, "you write %s(%s) with: %s\n", name1, name2, value);
        }
    }

    return NULL;
}

/* trace write acces of eliasWriteValue~ vars */
char *eliasProcWriteValueWrite(ClientData clientData, Tcl_Interp *interp,
        char *name1, char *name2, int flags)
{
    unsigned int i, bit, val;
    char *helpstr, *charval;

    /* reset result data */
    Tcl_ResetResult(interp);

    if ((charval = Tcl_GetVar2(interp, name1, name2,
                    flags & TCL_GLOBAL_ONLY)) != NULL) {

        if (name2 != NULL) {

            if (Tcl_GetInt(interp, name2, &bit) != TCL_OK) {
                Tcl_ResetResult(interp);
                return ELIAS_APPERROR_TCLTRC("expected integer arg");
            }

            if (Tcl_GetInt(interp, charval, &val) != TCL_OK) {
                Tcl_ResetResult(interp);
                return ELIAS_APPERROR_TCLTRC("expected integer arg");
            }

            switch (val & 0x1) {
                case 1:
                    eliasU32VarWriteValue |= (0x1 << bit);
                    break;
                case 0:
                    eliasU32VarWriteValue &= ~(0x1 << bit);
                    break;
            }

            if ((helpstr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE)) != NULL) {

                sprintf(helpstr, "IEEE-1394: write 0x%08x to 0x%016Lx",
                        eliasU32VarWriteValue, eliasU64VarWriteAddr);
                Tcl_SetVar(interp, "eliasBusState", helpstr, 0);

                Tcl_Free(helpstr);

            } else {
                return ELIAS_APPERROR_TCLTRC("can't allocate memory");
```

```
            }

            ELIAS_DEBUG_PRINT("%s", "calling 1394 write");
            if (eliasWrite1394(&eliasU64VarWriteAddr, &eliasU32VarWriteValue)) {
                ELIAS_DEBUG_PRINT("%s", "error while 1394 write");
            }

        } else {

            return ELIAS_APPERROR_TCLTRC("expacted array to trace");
        }

    } else {

        return ELIAS_APPERROR_TCLTRC("can't get value of traced array");
    }

    return NULL;
}


/*
 * ---------------------- COMMAND-PROCEDURES ----------------------
 */

/* generate IEEE-1394 Bus Reset */
int eliasProcBusReset(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[])
{
    if (eliasStatus == offline) {

        ELIAS_DEBUG_PRINT("%s", "IEEE-1394: dummy bus reset");

    } else {

        ELIAS_DEBUG_PRINT("%s", "IEEE-1394: force bus reset");
        if (raw1394ForceBusReset(eliasRaw1394Port)) {
            Tcl_SetVar(eliasInterp, "eliasBusState",
                    "IEEE-1394: Force Bus Reset Action Failed", 0);
            ELIAS_DEBUG_PRINT("%s",
                    "IEEE-1394: Force Bus Reset Action Failed");
            return -1;
        }

    }

    return TCL_OK;
}

/* response the list request of available node ids */
int eliasProcFreshNodeList(ClientData clientData, Tcl_Interp *interp,
```

```
        int argc, char *argv[])
{
    unsigned int i;
    char *helpstr;
    char *resultval;

    /* reset result data */
    Tcl_ResetResult(interp);

    if (argc != 2) {
        ELIAS_APPERROR_TCLRES(interp, "wrong # args");
        return TCL_ERROR;
    }

    if ((helpstr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE)) != NULL) {

        for (i = 0; i < eliasGetNumberOfNodes(); i++) {
            sprintf(helpstr, "%u", i);

            if (Tcl_VarEval(interp, argv[1], " insert end ",
                        helpstr, (char *)NULL) != TCL_OK) {
                ELIAS_APPERROR_TCLRES(interp, "can't write to list ", argv[1]);
                return TCL_ERROR;
            }

        }

    } else {
        ELIAS_APPERROR_TCLRES(interp, "can't allocate memory");
    }
    Tcl_Free(helpstr);

    return TCL_OK;
}

/* regist new target event */
int eliasProcNewTarget(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[])
{
    int cmd;
    octlet_t value;
    char *helpstr;

    /* reset result data */
    Tcl_ResetResult(interp);

    if (argc != 2) {
        ELIAS_APPERROR_TCLRES(interp, "wrong # args");
        return TCL_ERROR;
    }
```

```
if (Tcl_GetInt(interp, argv[1], &cmd) != TCL_OK) {
    Tcl_ResetResult(interp);
    ELIAS_APPERROR_TCLRES(interp, "expected integer arg");
    return TCL_ERROR;
}

switch (cmd) {
    case ELIAS_NEWTARGET_RDADDR:
        if ((helpstr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE)) != NULL) {

            if (sscanf(eliasStringVarReadAddr, "%Lx", &value)) {

                eliasU64VarReadAddr = eliasAddrUpd(&value,
                        (doublet_t *)&eliasIntVarBusID,
                        &eliasIntVarReadNodeID);
                ELIAS_DEBUG_PRINT("set new rdaddr 0x%016Lx",
                        eliasU64VarReadAddr);

                sprintf(helpstr, "Set new read address 0x%012Lx. "
                        "(IEEE-1394: 0x%016Lx)",
                        (eliasU64VarReadAddr & 0x0000ffffffffffffULL),
                        eliasU64VarReadAddr);
                Tcl_SetVar(interp, "eliasBusState", helpstr, 0);

            } else {
                Tcl_SetVar(interp, "eliasBusState",
                        "ERROR: Wrong hex value.", 0);
            }

            /* refresh entry field to correct form */
            sprintf(helpstr, "%012Lx",
                    (eliasU64VarReadAddr & 0x0000ffffffffffffULL));
            Tcl_SetVar(interp, "eliasAppReadAddressString", helpstr, 0);

            Tcl_Free(helpstr);

        } else {
            ELIAS_APPERROR_TCLRES(interp, "can't allocate memory");
            return TCL_ERROR;
        }

        break;

    case ELIAS_NEWTARGET_RDNODE:
        if ((helpstr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE)) != NULL) {

            eliasU64VarReadAddr = eliasAddrUpd(&eliasU64VarReadAddr,
                    (doublet_t *)&eliasIntVarBusID, &eliasIntVarReadNodeID);
            ELIAS_DEBUG_PRINT("set new rdaddr 0x%016Lx",
                    eliasU64VarReadAddr);
```

```
                sprintf(helpstr, "Set new read node id %u. "
                        "(IEEE-1394: 0x%016Lx)",
                        eliasIntVarReadNodeID, eliasU64VarReadAddr);
                Tcl_SetVar(interp, "eliasBusState", helpstr, 0);

                Tcl_Free(helpstr);

            } else {
                ELIAS_APPERROR_TCLRES(interp, "can't allocate memory");
                return TCL_ERROR;
            }

            break;

        case ELIAS_NEWTARGET_WRADDR:
            if ((helpstr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE)) != NULL) {

                if (sscanf(eliasStringVarWriteAddr, "%Lx", &value)) {

                    eliasU64VarWriteAddr = eliasAddrUpd(&value,
                            (doublet_t *)&eliasIntVarBusID,
                            &eliasIntVarWriteNodeID);
                    ELIAS_DEBUG_PRINT("set new wraddr 0x%016Lx",
                            eliasU64VarWriteAddr);

                    sprintf(helpstr, "Set new write address 0x%012Lx. "
                            "(IEEE-1394: 0x%016Lx)",
                            (eliasU64VarWriteAddr & 0x0000ffffffffffffULL),
                            eliasU64VarWriteAddr);
                    Tcl_SetVar(interp, "eliasBusState", helpstr, 0);

                    ELIAS_DEBUG_PRINT("%s", "calling 1394 write");
                    eliasWrite1394(&eliasU64VarWriteAddr,
                            &eliasU32VarWriteValue);
#if 0
                    if (eliasWrite1394(&eliasU64VarWriteAddr,
                                &eliasU32VarWriteValue)) {
                        ELIAS_APPERROR_TCLRES(interp, "error while 1394 write");
                        return TCL_ERROR;
                    }
#endif

                } else {
                    Tcl_SetVar(interp, "eliasBusState",
                            "ERROR: Wrong hex value.", 0);
                }

                /* refresh entry field to correct form */
                sprintf(helpstr, "%012Lx",
                        (eliasU64VarWriteAddr & 0x0000ffffffffffffULL));
                Tcl_SetVar(interp, "eliasAppWriteAddressString", helpstr, 0);
```

```
                Tcl_Free(helpstr);

            } else {
                ELIAS_APPERROR_TCLRES(interp, "can't allocate memory");
                return TCL_ERROR;
            }

            break;

        case ELIAS_NEWTARGET_WRNODE:
            if ((helpstr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE)) != NULL) {

                eliasU64VarWriteAddr = eliasAddrUpd(&eliasU64VarWriteAddr,
                        (doublet_t *)&eliasIntVarBusID,
                        &eliasIntVarWriteNodeID);
                ELIAS_DEBUG_PRINT("set new wraddr 0x%016Lx",
                        eliasU64VarWriteAddr);

                sprintf(helpstr, "Set new write node id %u. "
                        "(IEEE-1394: 0x%016Lx)",
                        eliasIntVarWriteNodeID, eliasU64VarWriteAddr);
                Tcl_SetVar(interp, "eliasBusState", helpstr, 0);

                ELIAS_DEBUG_PRINT("%s", "calling 1394 write");
                eliasWrite1394(&eliasU64VarWriteAddr, &eliasU32VarWriteValue);
#if 0
                if (eliasWrite1394(&eliasU64VarWriteAddr,
                            &eliasU32VarWriteValue)) {
                    ELIAS_APPERROR_TCLRES(interp, "error while 1394 write");
                    return TCL_ERROR;
                }
#endif

                Tcl_Free(helpstr);

            } else {
                ELIAS_APPERROR_TCLRES(interp, "can't allocate memory");
                return TCL_ERROR;
            }

            break;

        default:
            ELIAS_APPERROR_TCLRES(interp, "wrong arg range");
            return TCL_ERROR;
    };

    return TCL_OK;
}
```

```
/* start/stop the timer-event procedure */
int eliasProcTimerPlay(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[])
{
    eliasTkTimerToken = Tk_CreateTimerHandler(0, eliasTimeProc,
            (ClientData)interp);
    return TCL_OK;
}

int eliasProcTimerStop(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[])
{
    Tk_DeleteTimerHandler(eliasTkTimerToken);
    return TCL_OK;
}


/*
 * ---------------------- TCL-EXIT-CALLBACKS ----------------------
 */

void eliasExitProc(ClientData clientData)
{
    if (eliasRaw1394Port) {
        raw1394ClosePort(eliasRaw1394Port);
    }

    if (eliasRaw1394Config) {
        raw1394FreeConfig(eliasRaw1394Config);
    }

    ELIAS_DEBUG_PRINT("%s", "free all IEEE-1394 ressources");
}


/*
 * ---------------------- TCL/TK-CONFIGSTUFF ----------------------
 */

/* init application specific things */
int Tcl_AppInit(Tcl_Interp *interp)
{
    unsigned int i;
    char *helpstr;
    int   status = TCL_OK;

    /* init Tcl specific things */
    if (Tcl_Init(interp) != TCL_OK) {
        return TCL_ERROR;
    }
```

```
/* init Tk specific things and prepare it for load */
if (Tk_Init(interp) != TCL_OK) {
    return TCL_ERROR;
}
Tcl_StaticPackage(interp, "Tk", Tk_Init, Tk_SafeInit);


/* FIXME: insert your code here... */
eliasInterp = interp;
Tcl_CreateExitHandler(eliasExitProc, (ClientData)NULL);

/* register C functions as Tcl commands */
ELIAS_REGIST_TCLCMD(interp, "eliasAppBusReset", eliasProcBusReset,
        NULL, NULL);
ELIAS_REGIST_TCLCMD(interp, "eliasAppNewTarget", eliasProcNewTarget,
        NULL, NULL);
ELIAS_REGIST_TCLCMD(interp, "eliasAppFreshNodeList", eliasProcFreshNodeList,
        NULL, NULL);
ELIAS_REGIST_TCLCMD(interp, "eliasAppTimerPlay", eliasProcTimerPlay,
        NULL, NULL);
ELIAS_REGIST_TCLCMD(interp, "eliasAppTimerStop", eliasProcTimerStop,
        NULL, NULL);

/* regist C vars as Tcl vars */
eliasStringVarReadAddr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE);
strcpy(eliasStringVarReadAddr, "000000000000");
ELIAS_REGIST_TCLVAR(interp, "eliasAppReadAddressString",
        eliasStringVarReadAddr, TCL_LINK_STRING);
eliasStringVarWriteAddr = Tcl_Alloc(ELIAS_GLOBALSTRINGSIZE);
strcpy(eliasStringVarWriteAddr, "000000000000");
ELIAS_REGIST_TCLVAR(interp, "eliasAppWriteAddressString",
        eliasStringVarWriteAddr, TCL_LINK_STRING);
ELIAS_REGIST_TCLVAR(interp, "eliasAppBusID", eliasIntVarBusID,
        TCL_LINK_INT);
ELIAS_REGIST_TCLVAR(interp, "eliasAppNodeID", eliasIntVarNodeID,
        TCL_LINK_INT);
ELIAS_REGIST_TCLVAR(interp, "eliasAppReadNodeID", eliasIntVarReadNodeID,
        TCL_LINK_INT);
ELIAS_REGIST_TCLVAR(interp, "eliasAppWriteNodeID", eliasIntVarWriteNodeID,
        TCL_LINK_INT);
ELIAS_REGIST_TCLVAR(interp, "eliasAppTimerInterval",
        eliasIntVarTimerInterval, TCL_LINK_INT);
ELIAS_REGIST_TCLVAR(interp, "eliasAppTimerGoing",
        eliasIntVarTimerGoing, TCL_LINK_INT);
ELIAS_REGIST_TCLVAR(interp, "eliasAppMainInitialized",
        eliasIntVarMainInitialized, TCL_LINK_INT);

/* regist Tcl var trace functions */
ELIAS_REGIST_TCLTRC2(interp, "eliasWriteValue", (char *)NULL,
        TCL_TRACE_WRITES, eliasProcWriteValueWrite, NULL);
```

```
    /* init what?? */
    eliasInit1394();


    /* specify user-specific startup file to invoke if the application
     * is run interactively */
    Tcl_SetVar(interp, "tcl_RcFileName", "~/.eliasrc", TCL_GLOBAL_ONLY);

    return TCL_OK;
}


int main(int argc, char **argv)
{
    /* only call Tk's main -- Tk_Main() -- and set my owne init
     * function -- Tk_AppInit() */
    Tk_Main(argc, argv, Tcl_AppInit);
    return 0;
}


#    endif /* HAVE_RAW1394_H */

#  endif /* HAVE_TK_H */

#endif /* STDC_HEADER */
/* ---------------------------------------- vim-modeline (don't remove it):
 * vi: set et ai sw=4 ts=4:
 */
```

D.2.2.3. eliasMain.tcl

Diese Quelle stellt die eigentliche Bedienoberfläche dar. Weil sie jedoch durch ein visuelles Entwicklungswerkzeug (Visuel Tcl v1.10) erstellt wurde und eine nicht gerade unerhebliche Anzahl an Programmzeilen mit sich bringt, wird der Quelltext nicht abgedruckt, sondern nur der elektronischen Form dieser Diplomarbeit beigelegt.

D.2.2.4. elias.n

Die hier abgebildete Quelle ist eine groff/troff Vorlage. Sie dient zur Generierung von sogenannten Man-Pages unter Unix. Die Werkzeuge groff/troff werden dann als Text-Interpreter benutzt und stellen die Man-Page in einem gut lesbaren ASCII-Schriftbild dar.

```
.TH elias n 07-Okt-1999 IEEE1394 "IEEE-1394 Tcl/Tk Samples"
.SH NAME
elias \- graphical interface for remote manipulating IEEE-1394 memory
.SH SYNOPSIS
eliasApp eliasMain.tcl
.RI [ unknown ]
.SH DESCRIPTION
.B elias
presents a simple graphical front-end for read and write access to remote
IEEE-1394 node (user-specific CSR memory array).
.PP
elias is a bundel of Tcl/Tk application and script.
.PP
There is no further man page available till now.
.SH SEE ALSO
tclsh(1), wish(1)
.SH AUTHOR
Stephan Linz <linz@mazet.de, Stephan.Linz@gmx.de>
```

Erklärung

Ich erkläre, daß ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, den 01. 02. 2000