



Universität des Saarlandes

L1394 Bibliothek

Entwurf und Implementierung einer
FireWire Bibliothek unter Linux

Fortgeschrittenenpraktikum

Lehrstuhl für Computergraphik
Prof. Dr. Philipp Slusallek
Fakultät für Informatik
Universität des Saarlandes

Michael Replinger
replix@graphics.cs.uni-sb.de

Betreuer:
Andreas Pomi

Inhaltsverzeichnis

1	Einleitung	1
2	Der IEEE-1394 Standard	3
2.1	Die ISO/IEC 13213 Spezifikation	3
2.1.1	Node Architektur und Adressierung	4
2.1.2	Control&Status Register und Configuration Rom	4
2.1.3	Basistransaktionen	6
2.1.4	Broadcast Mechanismen	7
2.2	IEEE-1394 Protokoll Layer	7
2.2.1	Der Transaction Layer	7
2.2.2	Der Link Layer	8
2.2.3	Der Physical Layer	8
2.2.4	Der Bus Management Layer	9
2.3	Erweiterungen zum IEEE-1394 Standard	10
2.3.1	1394a	10
2.3.2	1394b	10
3	FireWire unter Linux	11
3.1	Linux FireWire Treiber	11
3.2	Raw1394 Bibliothek	12
4	Die L1394 Bibliothek	13
4.1	Grobentwurf der Bibliothek	13
4.1.1	Anwendungsfallmodellierung	13
4.1.2	3-Schichten-Architektur	14
4.2	Feinentwurf und Implementierung	15
4.2.1	Darstellung eines FireWire Node	15
4.2.2	Interface-Schicht	16
4.2.3	Anwendungslogik-Schicht	19
4.2.4	Datenhaltungs-Schicht	20

5	Beispiel Applikationen	23
5.1	QVcrPanel	23
5.2	QCamControl	25

1. Einleitung

Im Rahmen der Projekte *Network-Integrated Multimedia-Environment for Linux* (NMM) und *Virtual Studio Lab*, kommen mehrere Geräte basierend auf der IEEE-1394 Spezifikation unter dem freien Betriebssystem Linux zum Einsatz. Dieser Standard definiert einen seriellen Hochgeschwindigkeitsbus der den speziellen Anforderungen des Multimedia-Bereichs angepasst ist und erlaubt, über entsprechende Steckkarten, eine direkte Übertragung der digitalen Daten in alle gängigen Computersysteme. Da zum aktuellen Zeitpunkt keine einfache und einheitliche Möglichkeit zum Ansteuern entsprechender Geräte unter Linux existiert, entstand das in dieser Ausarbeitung beschriebene Projekt. Ziel dieses Projekts ist der Entwurf und die Implementierung einer objektorientierten Highlevel Bibliothek zum Steuern von Geräten basierend auf der IEEE-1394 Spezifikation. Die Bibliothek ist für x86 kompatible Systeme unter Linux¹ ausgelegt und erhält den Namen L1394.

Bei den verwendeten Geräten konnte, durch eine inzwischen weite Verbreitung des Standards, weitgehend auf teure Spezial-Hardware verzichtet werden. Firmen wie Sony und Apple liefern, unter der Bezeichnung FireWire(Sony) und i-Link(Apple), ihre aktuellen Produkte serienmäßig mit entsprechenden Anschlüssen aus. Im Folgenden werden die Bezeichnungen FireWire und IEEE-1394 Standard synonym verwendet.

Innerhalb dieses Projekts kommen mehrere Sony Digitalkameras vom Typ DFW-V500 und DFW-VL500, sowie zwei Videorecorder und ein SONY Camcorder zum Einsatz, an denen die Bibliothek getestet werden konnte. Der Anschluss an den Computer erfolgt über mehrere 1394-OHCI² kompatible FireWire Karten unterschiedlicher Hersteller und einer PCI-Lynx Karte, deren Treiberentwicklung unter Linux allerdings eingestellt wurde.

Zu Beginn eines jeden Projekts sollte zunächst eine Anforderungsanalyse stattfinden, in der ein Grundverständnis der Problemwelt und der projektspezifischen Anforderungen erfolgt. Um diesem Anspruch gerecht zu werden, wird in Kapitel 2 ein Überblick über den Aufbau und die Arbeitsweise des IEEE-1394 Standard gegeben. Kapitel 3 setzt sich mit den unter Linux vorhandenen Ressourcen auseinander. In Kapitel 4 wird die Planung, der Aufbau und die Implementierung der L1394-Bibliothek beschrieben und diskutiert. Zum Abschluss dieser Arbeit wird in Kapitel 5 die Anwendung der Bibliothek an Hand zweier Programme demonstriert.

¹Eine Portierung auf andere UNIX-Systeme sollte jedoch wenig Probleme bereiten.

²Bis auf einige ältere Modellen, sollten alle aktuellen FireWire Karten auf dem 1394-Open Host Controller Interface (OHCI) von IBM basieren.

2. Der IEEE-1394 Standard

In diesem Kapitel wird ein Überblick über den Aufbau und die Arbeitsweise des IEEE-1394 Standard gegeben. Der unter der offiziellen Bezeichnung '*IEEE-1394.1995 Standard for a High Performance Serial Bus*' laufende Standard, zeichnet sich durch folgende Eigenschaften aus, die sich besonders für den Einsatz in Multimedia-Umgebungen eignen:

- Bis zu 400Mbps Datentransfer - Diese Transferrate erlaubt etwa das gleichzeitige Senden von sechzehn DV-Datenströmen.
- Echtes Hot-Plugging und Plug'n'Play - Im laufenden Betrieb können Geräte entfernt oder angeschlossen werden.
- Automatische Buskonfiguration und Optimierung - Bei einer Änderung der Buskonfiguration organisieren sich die angeschlossenen Geräte selbst. Der Standard sieht zusätzlich einige automatische Optimierungsmöglichkeiten vor.
- Stromversorgung durch Verbindungskabel - Der IEEE-1394 Standard sieht Verbindungskabel einer maximalen Länge von 4.5 Metern mit und ohne Stromversorgung vor.
- Reservieren von Bandbreite - Geräte müssen sich für die zu übertragenen Daten Bandbreite reservieren und erhalten hierdurch eine garantierte Bandbreite für die Übertragung der Daten.

Zu Beginn wird ein Überblick über die ISO/IEC 13213 Spezifikation gegeben, die als Grundlage für diesen Standard dient. Anschließend wird die Realisierung der durch diesen Standard gesetzten Anforderungen durch das Schichtenmodell der FireWire Spezifikation erläutert. Zum Abschluss dieses Kapitels wird ein kurzer Überblick über die wichtigsten Erweiterungen dieses Standards gegeben.

2.1 Die ISO/IEC 13213 Spezifikation

Der unter der offiziellen Bezeichnung '*Information technology-Microprozessor Systems-Control and Status Register (CSR)*' laufende Standard definiert den grundlegenden Aufbau eines Mikroprozessor Busses und dessen Arbeitsweise. Die IEEE-1394 (FireWire Bus), IEEE 896(Futurbus+) sowie die IEEE 1596(Scalable Coherent Interface) Spezifikationen basieren auf diesem Standard und beteiligten sich nach [1] an dessen Entwurf.

Neben der Möglichkeit, auf dieser Spezifikation basierende Bussysteme durch geringen Aufwand zusammen zuschalten (*Bridging*), gehörte eine Vereinfachung der systemübergreifenden Zusammenarbeit zwischen den einzelnen Geräten zu den Hauptzielen dieses Standards. Zusätzlich sollte die benötigte Software zur Unterstützung der unterschiedlichen Bussysteme auf ein Minimum reduziert werden. Für die Realisierung der genannten Ziele definiert, der im Folgenden als CSR-Architektur bezeichnete Standard, ein grundlegendes Design, ermöglicht aber den darauf aufbauenden Standards spezifische Erweiterungen. Neben der Aufteilung eines Gerätes zur physikalischen und logischen Repräsentation an einem Bus, definiert der Standard folgende grundlegenden Features:

- Node Architektur
- Node Adressierung und Node Adressraum
- Control und Status Register
- Configuration Rom
- Basistransaktionen
- Broadcast Mechanismen

Der restliche Abschnitt beschreibt die oben angesprochenen Features und bezieht sich hierbei, falls nicht explizit angegeben, auf nicht Bus-Spezifische Implementierungen.

2.1.1 Node Architektur und Adressierung

Ein Gerät wird nach der CSR-Architektur in *Nodes*, *Units* und *Module* aufgeteilt, wie in Abbildung 2.1 zu sehen ist. Das Modul repräsentiert ein physikalisches Gerät, was sich aus mehreren Nodes zusammensetzen kann. Ein Node stellt hierbei eine adressierbare und vom Bus unabhängig konfigurierbare Einheit dar. Jeder Node verfügt über ein Control und Status Register, das mit anderen Nodes innerhalb eines Moduls geteilt werden kann. Units stellen interne, von einem Node benötigte I/O-Einheiten dar, wie zum Beispiel Prozessoren oder Festplatten.

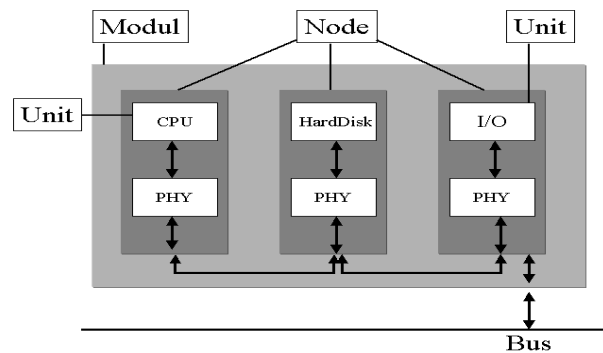


Abbildung 2.1: Node-Architektur der ISO/IEC-13213 Spezifikation

Für die Adressierung der Geräte stellt die CSR-Architektur sowohl ein 32, als auch ein 64 Bit Modell zur Verfügung. Da der FireWire Standard auf dem 64-Bit Adressierungsmodell basiert, wird lediglich dieses Modell beschrieben. Die 64 Bit Adresse wird in mehrere Segmente aufgeteilt, wie in Abbildung 2.2 zu sehen ist. Die ersten 16 Bit (most significant Bits) bestimmen die *node_ID*, die sich wiederum im FireWire Standard aus der *bus_ID*, den ersten 10 Bit, und der *physical_ID*, den restlichen 6 Bit, zusammensetzt. Somit können bis zu 1023 Busse mit bis zu 63 Nodes pro Bus angesprochen werden. Die restlichen 48 Bits der Adresse bestimmen einen 256 TeraByte großen Adressraum, der jedem Node zu Verfügung steht. Für die Verwendung dieses Adressraum definiert die CSR-Architektur das *Control & Status Register*, sowie das *Configuration Rom* die im folgenden Abschnitt beschrieben werden.

2.1.2 Control&Status Register und Configuration Rom

Das Control und Status Register definiert ein Satz von grundlegenden Registern und deren Bedeutung, die zu einer einfacheren Zusammenarbeit der einzelnen Bussysteme benötigt werden. Tabelle 2.1 beschreibt die Core-Register der CSR-Architektur, die von jedem Node implementiert werden müssen. Die

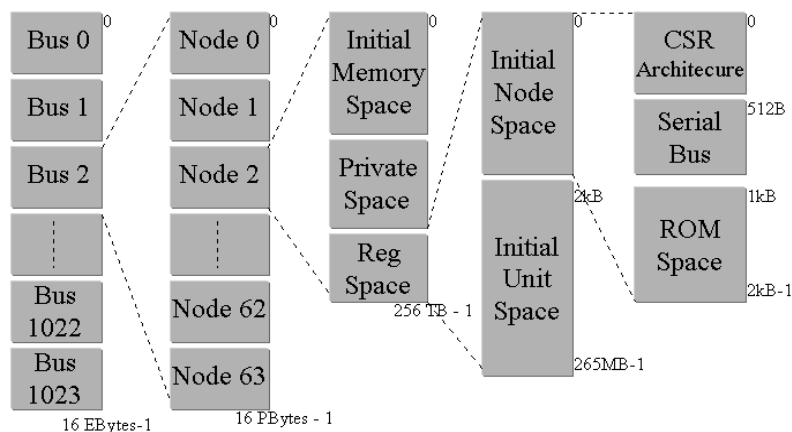


Abbildung 2.2: Aufteilung des 64 Bit Adressierungsmodell

Register-Offset	Register-Name	Beschreibung
0x000	STATE_CLEAR	Verwaltet Status- und Kontrollinformationen
0x004	STATE_SET	Setzt das STATE_CLEAR Register
0x008	NODE_IDS	Speichert die 16-Bit node_ID
0x00C	RESET_START	Versetzt den Node in seinen ursprünglichen Zustand
...
0x018	SPLIT_TIMEOUT_HI	Definiert Timeout für Transaktionen
0x01C	SPLIT_TIMEOUT_LOW	Definiert Timeout für Transaktionen

Tabelle 2.1: Core-Register der CSR-Architektur

Startadresse für diese Register beginnt bei dem Offset $0xFFFF\ F000\ 0000$. In [1] ist eine ausführliche Auflistung und Beschreibung aller Register nachzulesen.

Neben den vordefinierten Registern, erlaubt die CSR-Architektur auf dem Adressbereich $0x200-0x3FC$, bezüglich des CSR-Register Offsets, das definieren von Bus-Spezifischen Registern. In Tabelle 2.2 sind diese Register für den FireWire Standard aufgelistet und beschrieben. Zu den wichtigsten Registern gehören die `BANDWIDTH_AVAILABLE` und `CHANNELS_AVAILABLE` Register, die für das Reservieren von Bandbreite auf dem Bus zuständig sind. Die genaue Vorgehensweise wird im Abschnitt 2.2 beschrieben.

Das *Configuration Rom* beschreibt Einträge, die zur Initialisierung eines Nodes benötigt werden und gibt Aufschluss über dessen Funktionalität. Zwei Config-ROM Formate sind durch den Standard definiert, das minimale und generelle ROM Format. Das minimale ROM Format speichert lediglich die 24-Bit `Vendor_ID` des Herstellers. Das generelle ROM Format setzt sich aus zwei Blöcken zusammen, dem `Bus_Info_Block` und dem `CSR-Directory`. Hierbei spezifiziert der `Bus_Info_Block` neben der 64 Bit `GUID` (global unique identifier) die implementierten Dienste des Nodes. Eine ausführliche Auflistung der Bedeutung des generellen ROM-Formats findet sich in [1]. Das `CSR-Directory` beschreibt eine Verzeichnis-Struktur, welche das speichern von komplexeren Datenstrukturen erlaubt. Diese Verzeichnis-Struktur kann zusätzlich von Herstellern entsprechender Hardware erweitert werden.

Register-Offset	Register-Name	Beschreibung
0x200	CYCLE_TIME	Die verfügbare Bandbreite wird in Intervalle bestimmter Länge aufgeteilt, die durch dieses Register bestimmt wird.
0x204	BUS_TIME	Erweiterung des CYCLE_TIME Registers
0x208	POWER_FAIL_IMMINENT	Benachrichtigt den Bus über Spannungsprobleme
0x20C	POWER_SOURCE	Versetzt den Node in seinen ursprünglichen Zustand
0x218	BUSY_TIMEOUT	Setzt Zeitdauer, nachdem ein Node Antworten muss
0x214-0x217	Reserviert	Reserviert
0x218	FAIRNESS_BUDGET	Bestimmt die Anzahl der Transaktionen pro Intervall
0x21C	BUS_MANAGER_ID	Speicher die ID des aktuellen Bus-Managers
0x220	BANDWIDTH_AVAILABLE	Verwaltet die verfügbare Bandbreite
0x224	CHANNELS_AVAILABLE	Verwaltet verfügbaren Channel 0..31
0x228	CHANNELS_AVAILABLE	Verwaltet verfügbaren Channel 32..61
0x22C	MAINT_CONTROL	Ermöglicht Diagnose durch verursachen von Fehlern
0x230	MAINT_UTILITY	Für Diagnose
0x234-0x3FC	Reserviert	Reserviert

Tabelle 2.2: Bus-Spezifische Register der CSR-Architektur

2.1.3 Basistransaktionen

Die CSR-Architektur sieht zwei unterschiedliche Modi zur Datenübertragung vor, *asynchron* und *isochron*. Der Grund hierfür sind die unterschiedlichen Anforderungen an die zu sendenden Daten.

Für Anwendungen, die eine garantierte Bandbreite benötigen, jedoch auf eine Bestätigung der übertragenen Daten verzichten können, ist der isochrone Modus vorgesehen. Die isochrone Übertragung findet zwischen mindestens zwei Nodes statt. Der so genannte *isochrone Talker* sendet die Daten, die einer oder mehrere *isochrone Listener* empfangen. Um einen isochronen Datentransfer zu initiieren wird lediglich eine Basistransaktion, die *isochrone Request* Transaktion, benötigt. Diese setzt sich aus zwei so genannten *Subactions* zusammen. Aufgabe der ersten Subaction ist es, für den isochronen Talker Bandbreite und eine Channel Nummer auf dem Bus zu reservieren. Nach erfolgreicher Reservierung steht diesem Node über die Channel Nummer innerhalb eines fest definierten Intervalls ein Zeitfenster zum Senden seiner Daten zu Verfügung. Die zweite Subaction initiiert auf dem isochronen Listener eine *'link isochronous Control'* Anfrage. Hierbei wird der isochrone Listener auf die Channel Nummer des isochronen Talker konfiguriert. Falls der isochrone Listener bereits Daten sendet, wird lediglich diese Subaction ausgeführt.

Für Anwendungen, die eine Bestätigung der gesendeten Daten benötigen, ist der asynchrone Modus vorgesehen. Diese Daten werden periodisch übertragen und erhalten eine Rückmeldung über den Verlauf der Übertragung. Wenn innerhalb eines festgelegten Zeitraums keine Bestätigung erfolgt, der durch das SPLIT_TIME Register festgelegt ist, wird die Übertragung als fehlerhaft angesehen und wiederholt. Der asynchrone Datentransfer findet normalerweise¹ zwischen zwei Nodes statt, dem *Requester* und dem *Responder*. Der Requester initiiert auf dem Bus einen Datentransfer, der von dem Responder empfangen und beantwortet wird. Hierbei sind drei verschiedene Basistransaktionen vorgesehen:

READ: Die Read Transaktion veranlasst den Responder Node ein Quadlet² aus seinem Adressbereich an den Requester Node zu senden.

WRITE: Die Write Transaktion überschreibt ein Quadlet aus dem Adressbereich des Responder Nodes mit den gesendeten Daten des Requester Nodes.

LOCK: Diese Transaktion repräsentiert eine so genannte atomare Transaktion. Hierbei garantiert der Responder Node, dass während des Zugriffs auf seinen Adressbereich durch den Requester Node kein weiterer Node Zugriff auf diesen Bereich erhält.

¹Für den asynchronen Datentransfer ist auch ein Broadcast Mechanismus vorgesehen (siehe 2.1.4)

²Der FireWire Standard bezeichnet 4 Byte als ein Quadlet.

Die Basistransaktionen werden, analog zur isochronen Basistransaktion, in Subactions zerlegt, wobei sich jede der beschriebenen Transaktionen aus zwei Subactions zusammensetzt. Die *request* Subaction, vom Requester Node initiiert, leitet eine neue Transaktion ein. Hierbei wird der Responder Node veranlasst, eine bestimmte Operation auszuführen. Diese Operation hängt von den oben beschriebenen Transaktionstypen ab und wird durch die *response* Subaction bestätigt, die ebenfalls die Transaktion beendet.

2.1.4 Broadcast Mechanismen

Zum Abschluss dieses Abschnitts werden noch zwei optionale Mechanismen zum Broadcasten von Nachrichten und Ereignissen beschrieben, die in der CSR-Architektur vorgesehen sind.

- **Message Broadcast** - Message Broadcasting bietet jedem Node die Möglichkeit asynchrone Transaktionen an alle Nodes eines Busses zu senden. Hierzu wird die Nachricht an den Node 63 adressiert, der als Broadcast Adresse reserviert ist. Um einen Ansturm von Rückmeldungen zu vermeiden, beschränkt sich das Message Broadcasting auf Write- und Lock-Transaktionen. Eine Bestätigung durch die Requester Nodes entfällt ebenso.
- **Interrupt Broadcast** - Der auch unter der Bezeichnung *Nodecast* bekannte Mechanismus bietet die Möglichkeit Interrupts innerhalb einer Unit an alle Nodes zu senden. Hierzu müssen die Register `INTERRUPT_TARGET` und `INTERRUPT_MASK` implementiert werden. Insgesamt können bis zu 32 Interrupt Events definiert werden, wobei jedes Bit im `INTERRUPT_TARGET` Register ein Interrupt Event definiert. Um einen bestimmten Interrupt Event an alle Units innerhalb eines Nodes zu senden, wird das Bit für das entsprechende Interrupt Event im `Interrupt_Target` Register gesetzt und anschließend mit dem `Interrupt_Mask` Register, durch Anwendung der logischen AND Operation, gesetzt oder ignoriert.

2.2 IEEE-1394 Protokoll Layer

Dieser Abschnitt beschreibt die Realisierung der in den vorherigen Abschnitten beschriebenen Dienste durch das Schichtenmodell der FireWire Spezifikation. Insgesamt sind drei verschiedene Schichten für die Datenübertragung definiert, sowie eine Schicht für das Busmanagement. Abbildung 2.3 verdeutlicht den Aufbau des Schichtenmodells, der in den folgenden Abschnitten beschrieben wird.

2.2.1 Der Transaction Layer

Der Transaction Layer bildet die Schnittstelle zwischen der Applikation und der Link Layer. Er ist für den asynchronen Datentransfer zuständig und stellt für die Applikation die bereits beschriebenen Transaktionstypen `READ`, `WRITE` und `LOCK` bereit. Eine Transaktion wird in ihre Subactions zerlegt und anschließend an den entsprechenden Dienst innerhalb der Layer zur Weiterverarbeitung geleitet. Für jede Subaction ist ein eigener Dienst zuständig, die im folgenden kurz beschrieben werden:

Request Service: Der Request Service ist für den Aufbau von Transaktionen auf dem Bus zuständig und initiiert die request subaction.

Indication Service: Der Indication Service benachrichtigt den Responder über eine Anfrage und schließt die request subaction ab.

Response Service: - Der Response Service ist für die Beantwortung einer Anfrage zuständig und initiiert die response subaction.

Confirmation Service: Der Confirmation Service benachrichtigt den Requester, dass die Antwort eingetroffen ist und schließt die response subaction ab.

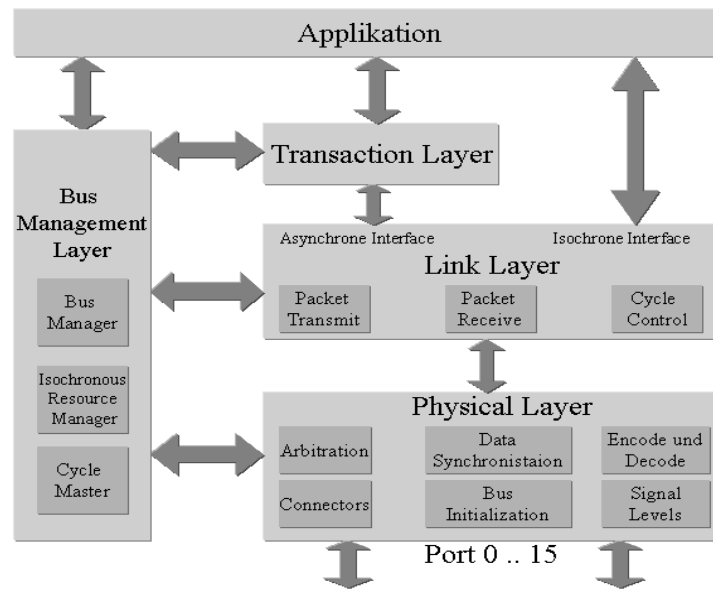


Abbildung 2.3: FireWire Layer

Da in den Transfermodellen der CSR Architektur kein Fehlercode für die übertragenen Daten vorgesehen ist, erweitert der FireWire Standard das Modell zur asynchronen Datenübertragung. Hierzu wird jedes empfangene Paket mit einem 1-byte großem *acknowledge*(ACK) Paket beantwortet, dessen Wert den Erfolg bzw. Fehler der Transaktion beschreibt. Ausnahmen bilden Broadcast Pakete, um einen Ansturm von Rückmeldungen zu vermeiden, sowie *acknowledge* Pakete selbst.

2.2.2 Der Link Layer

Der Link Layer ist für das Weiterleiten eintreffender Pakete an die entsprechende Schicht verantwortlich. Bei isochroner Übertragung bildet er die Schnittstelle zwischen der Physical Layer und der Applikation. Eintreffende isochrone Daten werden in dieser Schicht dekodiert und überprüft, ob die Daten für diesen Node bestimmt sind. Falls dies zutrifft, werden die Daten an die Applikation weitergeleitet, andernfalls werden sie ignoriert. Werden isochrone Daten gesendet, zerlegt diese Schicht die Daten in Pakete, kodiert sie mit der Channel Nummer und leitet sie an den Physical Layer zum Senden weiter. Die Größe der Pakete hängt von der maximalen Geschwindigkeit des Nodes ab. Bei einer Geschwindigkeit von $x \cdot 100 \text{ MB/s}$ ergibt sich eine Paketgröße von $x \cdot 512$ Bytes.

Für die asynchrone Übertragung bildet der Link Layer die Schnittstelle zwischen der Physical Layer und der Transaction Layer. Daten von der Transaction Layer werden in asynchrone Pakete zerlegt, mit einem CRC-Code versehen und an den Physical Layer zum Senden weitergereicht. Bei eintreffenden asynchronen Paketen von der Physical Layer werden die Pakete zunächst auf Übertragungsfehler überprüft. Falls die asynchronen Pakete korrekt eingetroffen sind, werden sie in Transaktionen übersetzt und an den Transaction Layer weitergereicht. Falls die Nachricht fehlerhaft ist, wird eine Nachricht an den Requester Node gesendet, die Übertragung zu wiederholen.

2.2.3 Der Physical Layer

Der Physical Layer bildet die Schnittstelle zwischen dem Bus und der Link Layer. Diese Schicht definiert die elektrische und mechanische Schnittstelle zum Übertragen von Paketen auf dem seriellen Bus, sowie den Aufbau der zu verwendenden Kabel. Das Kabel wird über so genannte Ports mit der Layer verbunden,

wobei der Physical Layer bis zu sechzehn Ports verwalten kann. Hierdurch wird die Konstruktion einer Netzwerk-Topologie ermöglicht, in der das Bilden jeder nicht-zyklischen Baumstruktur erlaubt ist. Die zu verwendenden Verbindungskabel bestehen je aus zwei Twisted Pair Kabeln (TPA/TPA* und TPB/TPB*). Insgesamt sieht der Standard zwei unterschiedliche Kabel vor, mit und ohne Stromversorgung. Der Physical Layer ist zusätzlich für zwei unterschiedliche Umgebungen ausgelegt. Neben der für den Betrieb mit Computern üblichen *Backplane Umgebung* (backplane environment), sieht die Spezifikation eine so genannte *Kabel Umgebung* (cable environment) vor. Hierdurch wird der Aufbau von echten Peer-to-Peer Verbindungen zwischen FireWire Geräten ermöglicht, die einen Betrieb ohne Computer ermöglichen.

Zu den wichtigsten Aufgaben der Physical Layer gehört die Bus-Konfiguration und die Bus-Verhandlung, die im Folgenden beschrieben werden.

Die Bus-Konfiguration setzt sich aus der Bus-Initialisierung, der Baum-Initialisierung und der Selbst-Identifizierung (Self-ID) zusammen. Wenn sich die Bus-Konfiguration, durch Hinzufügen oder Entfernen eines Gerätes, ändert, verursacht der Node, dessen Port-Konfiguration sich geändert hat, einen Busreset. Alle angeschlossenen Nodes werden in einen speziellen Zustand versetzt und alle Topologie-Informationen gelöscht. Zu diesem Zeitpunkt wissen die Nodes lediglich, ob sie ein Zweig (mehr als einen Nachbar), ein Blatt (genau einen Nachbar) oder isoliert (keine Nachbar) innerhalb der Baumkonfiguration sind. Diese Informationen können direkt durch die Port-Konfiguration ausgelesen werden. Anschließend wird diese Konfiguration durch die Baum-Initialisierung in eine Baumstruktur übersetzt. Jeder Blatt-Node sendet jetzt ein *Parent-Notify* Signal an seinen einzigen Nachbar-Node. Der Nachbar-Node vermerkt an diesem Port, das der daran angeschlossene Node sein Child-Node ist. Anschließend senden alle Nodes mit einem unbenannten Port das Parent-Notify Signal an diesen Node. Der Node, dessen Ports alle ein Parent-Notify Signal erhalten wird der neue Root-Node. Das dieser Ablauf deterministisch ist, folgt direkt aus der Tatsache, dass jede nicht zyklische Bus-Konfiguration als Baumstruktur dargestellt werden kann. Zum Abschluss initialisiert der Root-Node den Self-ID Prozess, der jedem Node eine eindeutige ID am Bus zuweist. Der Root-Node erhält hierbei die höchste ID. Die komplette Neu-Organisation des Busses benötigt maximal $340\mu s$.

Bevor ein Node eine Anfrage über den Bus senden bzw. beantworten kann, muss er Eigentümer des Busses werden³. Hierzu erfolgt eine Verhandlung zwischen allen Nodes, die eine Transaktion innerhalb eines Intervalls ausführen möchten. Zu Beginn eines neuen Intervalls, das durch ein cycle-Start Paket eingeleitet wird, werden zunächst die isochronen Daten gesendet. Der Node, der dem Root-Node am nächsten ist, gewinnt die Verhandlung und darf seine Daten als Erster senden. Besitzen mehrere Nodes den selben Abstand, erfolgt die Vergabe des Busses von der niedrigsten zur höchsten Port Nummer des Root-Nodes. Nachdem jeder Node seine isochronen Daten gesendet hat, die insgesamt maximal 80% des Zeitintervalls einnehmen dürfen, erfolgt das Senden der asynchronen Daten. Zwischen den Nodes erfolgt, analog zum isochronen Fall, eine Verhandlung, die die Reihenfolge der Buseigentümer bestimmt. Nachdem alle Nodes ihre Daten gesendet haben, fällt der Bus in einen Leerlauf (idle-Zustand), bis ein neues Intervall beginnt. Zusätzlich sieht der Standard Möglichkeiten vor, das Zeitintervall zu verzögern, um einen asynchronen Request zu beenden. Weiterhin ist die Beantwortung eines Requests innerhalb des selben Intervalls möglich. Diese Sonderfälle werden hier nicht weiter besprochen und können in [1] nachgelesen werden.

2.2.4 Der Bus Management Layer

Der Bus Management Layer ist für die Verwaltung des Busses zuständig und regelt die Aktivitäten der Nodes. Insgesamt sind in dieser Layer drei Dienste definiert:

Cycle Master: Aufgabe des Cycle Master ist es den Bus in Zeitintervalle von $125\mu s$ aufzuteilen, wobei jedes neue Intervall mit einem Cycle Start Paket eingeleitet wird.

³Ausnahme ist das Senden des Acknowledge Byte, für das ein Node nicht Eigentümer des Busses sein muss.

Isochrone Resource Manager(IRM): Jeder Node der isochrone Daten sendet, muss die Fähigkeit zum isochronen Resource Manager besitzen. Der IRM verwaltet die verfügbare Bandbreite und Channels des Busses.

Bus Manager: Der Bus Manager verwaltet neben der benötigten und vorhanden Spannungsversorgung die Speed-Map und Topologie-Map. Basierend auf diesen beiden Daten kann der Bus Manager die Bus-Konfiguration Optimieren, indem Geräte mit langsamer Geschwindigkeit, wenn möglich, nicht als Parent-Node verwendet werden.

Diese Dienste können auf unterschiedlichen Nodes laufen, lediglich der Root-Node muss die Fähigkeit zum Cycle Master besitzen. Wenn der Root-Node die beiden anderen Dienste nicht anbietet, wird bei den restlichen Nodes, in absteigender node_ID Reihenfolge, überprüft, ob sie diese Dienste anbieten können. Der erste Node, der über die entsprechende Fähigkeit besitzt, stellt diesen Dienst den anderen Nodes bereit, indem er seine node_ID broadcastet. Die Verwaltung des Busses wird durch das Vorhandensein dieser Dienste bestimmt. Man unterscheidet hierbei folgende Möglichkeiten:

- Bus wird vollständig verwaltet - Mindestens ein Node am Bus besitzt die Fähigkeit zum Busmanager und Isochronem Resource Manager.
- Bus wird teilweise verwaltet - Mindestens ein Node am Bus besitzt die Fähigkeit zum Isochronen Resource Manager.
- Bus wird nicht verwaltet - Keiner der Nodes am Bus besitzt die Fähigkeit zum Busmanager oder isochronen Resource Manager.

2.3 Erweiterungen zum IEEE-1394 Standard

Zum Abschluss dieses Kapitels soll ein kurzer Überblick über die Erweiterungen des FireWire Standards gegeben werden.

2.3.1 1394a

Die Erweiterung des FireWire Standards beziehen sich im wesentlichen auf Optimierungen der Busverhandlungen und der Datenübertragung. Bei allen Erweiterungen wurde besonderen Wert auf die Kompatibilität zu der 1394-1995 Spezifikation gelegt, so dass Geräte nach der älteren und neueren Spezifikation an einem gemeinsamen Bus betrieben werden können. An dieser Stelle werden nur die wichtigsten Erweiterungen beschrieben. Eine ausführliche Auflistung der Erweiterungen sind in [1] nachzulesen.

Connection Debouncing: Der Anschluss eines Nodes an einen Port kann einen Ansturm von Bus-Resets verursachen. Dies könnte den isochronen Datentransfer beeinträchtigen. Um einen solchen Ansturm zu vermeiden, sieht die IEEE-1394a Erweiterung einen Timeout von 340ms zwischen der Bestätigung eines Anschluss an einem Port vor.

Asynchrone Streams: Analog zu dem beschriebenen isochronen Datentransfer wurde die Spezifikation um asynchrone Streams erweitert. Hierbei handelt es sich um die Möglichkeit isochrone Pakete während des asynchronen fairness Intervalls zu übertragen. Diese Daten werden jedoch mit derselben Priorität wie alle asynchronen Daten übertragen. Die Motivation dieser Erweiterung liegt in der Möglichkeit IP über FireWire anzubieten.

2.3.2 1394b

Die wichtigste Erweiterung des 1394b Standards ist die Erhöhung der Datentransferrate auf bis zu 3.2 GBit/s. Bei einer Übertragungsrate größer als 800 Mbit/s werden hierfür allerdings optische Kabel benötigt. Für die hohe Übertragungsrate musste jedoch die Physical Layer überarbeitet werden. Der Bus selbst ist jedoch Abwärtskompatibel, so dass ältere Geräte auch an diesem Bus verwendet werden können. Eine ausführliche Beschreibung der Erweiterung, speziell für den Physical Layer, ist in der IEEE-1394b Spezifikation nachzulesen.

3. FireWire unter Linux

In diesem Kapitel wird ein Überblick über die wichtigsten, unter Linux verfügbaren, Ressourcen gegeben. Neben dem FireWire-Treiber, der über die Raw1394 Bibliothek angesprochen wird, existieren noch zwei Bibliotheken, die *DC-1394* und die *DV* Bibliothek.

Die DV-Bibliothek wird für das Dekodieren eines DV-Datenstroms verwendet, das aktuelle Videorecorder und Camcorder mit FireWire-Anschluss zum speichern ihrer Daten verwenden. Die DC-1394 Bibliothek erlaubt einen vereinfachten Zugriff auf Kameras, welche über das DCC-Protokoll gesteuert werden. Kameras, die auf dem AV/C Standard basieren, werden jedoch nicht unterstützt. Weiterhin basiert sie auf keinem objektorientiertem Design, womit sie für die beiden Projekte eher ungeeignet ist. Unter [7] findet sich ein Link-Sammlung zu den verfügbaren Ressourcen.

Der aktuelle Treiber und die Raw1394 Bibliothek werden in den folgenden beiden Abschnitten ausführlicher beschrieben.

3.1 Linux FireWire Treiber

Dieser Abschnitt gibt einen Überblick über den aktuellen Stand (Januar 2001) des Linux FireWire Treibers.

Der Treiber setzt sich aus mehreren Modulen zusammen, deren Aufbau in Abbildung 3.1 zu sehen ist. Die aktuellen Treiber bestehen aus drei low-level Modulen für die FireWire Karten von Adaptec *AIC-5800 PCI-IEEE1394* (*aic5800.o*), Texas Instruments *PCILynx* (*pcilynx.o*) und für das *1394 Open Host Controller Interface* (*ohci1394.o*). Diese Module können kombiniert eingesetzt werden und jeweils bis zu vier Karten verwalten.

Darauf Aufbauend sitzt das *ieee1394* Modul. Dieses Modul bildet die Schnittstelle zwischen allen low- und high-level Modulen und ermöglicht einen einheitlichen Zugriff auf die unterschiedlichen low-level Module der Karten. Zu seinen Aufgaben gehört es Daten, über das entsprechende low-level Modul, zu senden bzw. registrierte high-level Module über eintreffende Events zu benachrichtigen. Zum jetzigen Zeitpunkt gibt es drei high-level Module, *raw1394*, *sbp2* und *video1394*, die als Schnittstelle zum Userspace dienen.

Das *raw1394* Modul bildet die Schnittstelle zwischen der Applikation und dem Bus. Neben den Standard-Funktionen eines Character-Device, `READ`, `WRITE` und `OPEN`, verfügt das Modul zusätzlich Kenntnis über alle installierten Hostadapter. Das Modul bindet sich an das Device `/dev/raw1394` welches von der Raw1394 Bibliothek (siehe Abschnitt 3.2) für den Zugriff auf den Treiber verwendet wird.

Das *sbp2* Modul dient der Ansteuerung von Geräten, die das *Serial Bus Protocol 2* verwenden, das normalerweise von Massenspeichern wie Festplatten verwendet wird.

Das *video1394* Modul ist für das Abbilden isochroner Daten in den Speicher zuständig. Es ermöglicht der FireWire Karte einen direkten Zugriff auf den User-Space (DMA), was überflüssiges kopieren der Daten vermeidet. Die aktuelle Version erlaubt allerdings noch keinen Empfang von DV-Daten.

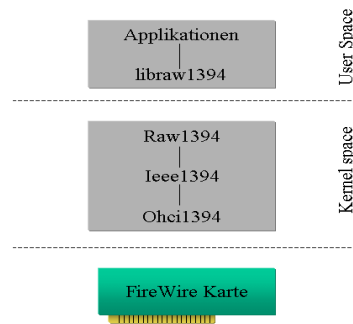


Abbildung 3.1: Aufbau des FireWire Treibers nach [3]

3.2 Raw1394 Bibliothek

Die Raw1394 Bibliothek ist eine C-Bibliothek, die für den Zugriff auf den aktuellen Treiber unter Linux entwickelt wurde. Der Zugriff auf den Treiber erfolgt über das Character-Device `/dev/raw1394`. Für den Zugriff auf das Device stellt die Bibliothek Funktionen für die Basistransaktionen `READ`, `WRITE` und `LOCK` bereit. Zusätzlich unterstützt sie das Senden und Empfangen isochroner Daten. Für Geräte, die über das *Function Control Protocol* (FCP) angesprochen werden, stellt die Bibliothek eine eigene `SEND` Funktion bereit.

Um über eintreffende Events benachrichtigt zu werden, muss eine Funktion für das entsprechende Ereignis registriert werden. Neben einem Busreset gehören auch eintreffende Daten zu den Events, für die entsprechende Handler angemeldet werden müssen. Durch einen so genannten raw1394-Handle, der beim Registrieren übergeben werden muss, kann das Event wieder eindeutig einer Karte bzw. Transaktion zugeordnet werden. Die Raw1394 Bibliothek sieht folgende Event-Handler vor:

Busrest-Handler: Der Busreset-Handler ist für die Verarbeitung eintreffender Busresets zuständig. Für jede Karte muß ein eigener raw1394-Handle vorgesehen werden.

Tag-Handler: Der Tag-Handler wird bei eintreffenden Antworten auf asynchrone Transaktionen aufgerufen und eine Referenz auf die Daten übergeben. Die Bibliothek implementiert bereits einen Tag-Handler, der von den Funktionen `READ`, `WRITE` und `LOCK` verwendet wird.

FCP-Handler: Eintreffende FCP-Daten werden an den registrierten FCP-Handler weitergeleitet. Diese Funktion muß für die Kommunikation mit AV/C Geräten implementiert werden.

Iso-Handler: Der ISO-Handler ermöglicht den Empfang isochroner Daten. Für jeden Channel, über den Daten empfangen werden, muß ein eigener raw1394-Handle initialisiert werden. Der Funktion wird bei einem Aufruf eine Referenz auf die Daten übergeben.

4. Die L1394 Bibliothek

Das vorherige Kapitel gab einen groben Überblick über die zum aktuellen Zeitpunkt verfügbaren Ressourcen unter Linux und zeigte die Notwendigkeit einer Bibliothek zum Steuern gängiger FireWire Geräte. In diesem Kapitel wird der Entwurf und die Implementierung der L1394 Bibliothek beschrieben und diskutiert. Abschnitt 4.1 bezieht sich auf den Grobentwurf der Bibliothek und Abschnitt 4.2 auf den Feinentwurf sowie die Implementierung.

Die Bibliothek basiert auf den vorhandenen Treibern und verwendet die Raw1394 Bibliothek für den Zugriff auf den FireWire Bus. Zusätzlich wird die Standard-Template-Library (STL) sowie die PThread-Bibliothek verwendet.

Für Klassennamen wird die Notation '*Namespace*::*Klassenname*' verwendet. Instanzen einer Klasse werden als '*Namespace*' '*Klassenname*' Objekt oder einfach als '*Klassenname*' Objekt bezeichnet.

4.1 Grobentwurf der Bibliothek

Dieser Abschnitt beschreibt und diskutiert den Grobentwurf der L1394 Bibliothek. Zu Beginn werden die von der Bibliothek zu bewältigenden Anwendungsfälle aufgelistet. Anschließend erfolgt eine grobe Gliederung der Bibliothek durch eine Softwarearchitektur.

4.1.1 Anwendungsfallmodellierung

Wie bereits erwähnt ist es Ziel dieser Bibliothek eine objektorientierte Highlevel Bibliothek zum Steuern von FireWire Geräten unter Linux zu entwerfen. Dem Anwendungsentwickler soll hierbei ein einheitliches Interface für den Zugriff auf die unterschiedlichen Geräte zur Verfügung stehen. Die Geräte selbst präsentieren sich dem Programmierer als Interface-Objekte, die ihm Zugriff auf deren Funktionalität ermöglichen. Folgende Anwendungsfälle sollen hierbei von der Bibliothek realisiert werden.

- Automatische Erkennung und Darstellung der Nodes - Die angeschlossenen Nodes an einem Bus sollen automatisch erkannt und Instanzen der entsprechenden Klassen erstellt werden. Hierbei soll jeder Node durch ein Software Objekt dargestellt werden, über das der Zugriff auf das Gerät erfolgt. Diese Objekte werden *L1394 Node* Objekte genannt.
- Bereitstellen von Geräte-Schnittstellen - Neben dem Node Objekt, über das der Zugriff auf den entsprechenden FireWire Node erfolgt, soll für jedes Gerät eine Schnittstelle mit Funktionen für die wichtigsten Features bereitgestellt werden. Diese Objekte werden als *L1394 Device* Objekte bezeichnet.
- Automatische Rekonfiguration nach einem Busreset - Nach einem Busreset, sollen die internen Parameter der Software Objekte, sowie die Listen der zur Verfügung stehenden Geräte aktualisiert werden.



Abbildung 4.1: 3-Schichten-Architektur

- Einfaches Event-Handling - Für Applikationen soll ein einfacher Mechanismus für eintreffende Events bereitstehen, um über Änderungen der Bibliothek informiert zu werden.
- Transparenz bezüglich der installierten Karten - Der Zugriff auf die angeschlossenen Geräte soll Karten übergreifend erfolgen.
- Einfache Methoden zur Erweiterungen durch Module - Zusätzlich soll die Bibliothek so flexibel wie möglich gehalten werden. Hierzu beschränkt sich die Bibliothek lediglich auf die Steuerung der Geräte. Erweiterte Methoden, zum Beispiel das darstellen eines Kamerabildes, werden in eigene Bibliotheken, so genannte Module, ausgelagert.

4.1.2 3-Schichten-Architektur

Dieser Abschnitt beschreibt die Unterteilung der Bibliothek durch eine Software-Architektur.

Für den grundlegenden Aufbau dieser Bibliothek wird ein vereinfachtes Modell der so genannten *3-Schichten-Architektur* verwendet. Hierbei handelt es sich nach [5] um ein grundlegendes Konzept aus der objektorientierten Softwareentwicklung für den Entwurf von Applikationen. Diese Architektur unterteilt eine Applikation in die drei Schichten Datenhaltungs-Schicht, Anwendungslogik-Schicht und externe Schnittstellen-Schicht. Abbildung 4.1 verdeutlicht diese Gliederung. Die Schichtenarchitektur dient einer besseren Komplexitätsbeherrschung und soll den Überblick über diese Bibliothek vereinfachen. Bei dem Entwurf und der Implementierung des Schichtenmodells wurde versucht, die Schichten so auszulegen, dass sich Änderungen möglichst nur Schichtenlokal auswirken¹.

Die externe Schnittstellen-Schicht bietet dem Anwender Zugriff auf eine Applikation. Die Klassen innerhalb dieser Schicht werden als *Schnittstellenklassen* (boundary classes) bezeichnet. Hauptaufgabe der Schnittstellenklassen ist es, einen Zugriff auf die Applikation in anwendungsinterne Ereignisse zu übersetzen, die von der Anwendungslogik-Schicht verarbeitet werden können. Umgekehrt müssen Daten der Anwendungslogik-Schicht in, für den Benutzer verwertbare, Information überführt werden. Übertragen auf die Bibliothek bedeutet dies, dass diese Schicht alle gerätespezifischen Schnittstellenklassen verwaltet. Die Schnittstellenklassen selbst müssen einen Steuerbefehl, je nach Gerätetyp in entsprechende Basistransaktionen bzw. Kommandos zerlegen und an die Anwendungslogik-Schicht übergeben. Weiterhin soll der Programmierer in geeigneter Form über den Erfolg der Operation informiert werden. Zusätzlich müssen geeignete Methoden vorgesehen werden, um auf Änderungen der Bus-Konfiguration reagieren zu können. Der Zugriff auf die Geräteschnittstellen selbst soll über eine Hauptschnittstellenklasse erfolgen.

Die Anwendungslogik-Schicht fokussiert auf die Funktionalität einer Applikation und deren korrekten Ablauf. Der korrekte Ablauf wird durch Regeln sichergestellt, deren Einhaltung durch so genannte *Kontrollklassen* (control classes) garantiert wird. Die Kontrollklassen bilden den Kern der Anwendungslogik-Schicht. Für die Bibliothek ergibt sich hieraus, dass alle Klassen, die den Zugriff auf den FireWire Bus

¹Im Rückblick auf die Arbeit stellte sich besonders dieser Aspekt mehrfach als hilfreich heraus

realisieren und für die konsistente Haltung der Objekte verantwortlich sind, Teil dieser Schicht sind. Die hierfür notwendigen Regeln werden in Abschnitt 4.2.3 formuliert.

Die Datenhaltungs-Schicht bildet die unterste Schicht dieses Modells. Sie ist für die Verwaltung und persistente Speicherung der Anwendungsobjekte verantwortlich. Innerhalb einer Anwendung sollte lediglich die Anwendungslogik-Schicht direkten Zugriff auf die Datenhaltung besitzen. Die Klassen dieser Schicht werden als *Entitätsklassen* (entity classes) bezeichnet. Bezüglich der Bibliothek gehören die Klassen zur Speicherung der Node Objekte zu dieser Schicht. Zusätzlich werden die Klassen zur Verwaltung Node spezifischer Informationen, wie das CSR-Directory und die Fähigkeiten eines Nodes, in diese Schicht aufgenommen.

4.2 Feinentwurf und Implementierung

Dieser Abschnitt beschreibt und diskutiert den Feinentwurf und die Implementierung der im letzten Abschnitt beschriebenen 3-Schichten-Architektur. Zuvor wird die Darstellung der FireWire Nodes innerhalb der Bibliothek beschrieben.

4.2.1 Darstellung eines FireWire Node

Wie bereits erwähnt, wird jeder FireWire Node durch ein entsprechendes Software Objekt, dem so genannten Node Objekt, dargestellt. Basisklasse aller Node Objekte bildet die Klasse `L1394::Node`. Die Klasse verwaltet die internen Parameter eines Nodes, wie `node_ID`, `bus_ID` und einen Referenz-Counter. Für die Schnittstellen-Schicht stehen Funktionen für die Basistransaktionen `READ`, `WRITE` und `LOCK` bereit.

Da sich die Buskonfiguration im laufenden Betrieb ändern kann, muß dies in geeigneter Weise von der Bibliothek berücksichtigt werden. Wird ein FireWire Node vom Bus getrennt, soll das entsprechende Node Objekt noch für einen bestimmten Zeitraum von der Bibliothek verwaltet werden. Wird der FireWire Node wieder angeschlossen, soll dieses Objekt wieder verwendet werden. Damit eine Anwendung keinen Zugriff auf ein ungültiges Node Objekt erhält, werden einem Node Objekt die Zustände `L1394_ENABLED`, `L1394_DISABLED` und, für interne Zwecke, `L1394_BUSRESET` zugeordnet. Ein Node Objekt ist im Zustand `L1394_ENABLED`, wenn der durch ihn verwaltete FireWire Node über eine FireWire Karte erreichbar ist, sonst befindet er sich in dem Zustand `L1394_DISABLED`. In diesem Zustand erhält das Node Objekt keinen Zugriff auf den Bus und die Funktionen für die Basistransaktionen liefern einen Fehler zurück. Somit soll sichergestellt werden, dass eine Anwendung, die eine Referenz auf ein solches Objekt besitzt keine Daten mehr senden kann. Damit eine Anwendung über einen Zustandswechsel benachrichtigt wird, sieht die Interface-Schicht entsprechende Methoden vor, die in Abschnitt 4.2.2 beschrieben werden.

Für den internen Ablauf der Bibliothek wird der Zustand `L1394_BUSRESET` benötigt. Wenn die Bibliothek einen Busreset an einem Bus registriert, werden die entsprechenden Node Objekte in den Zustand `L1394_BUSRESET` gesetzt. Nachdem die Konfiguration aktualisiert wurde, werden die Node Objekte wieder in den Zustand `L1394_ENABLED` oder `L1394_DISABLED` gesetzt. Wie diese Konfiguration erfolgt, wird in Abschnitt 4.2.3 beschrieben. Für eine Anwendung stellt sich ein Node Objekt lediglich in den beiden Zuständen `L1394_ENABLED` und `L1394_DISABLED` dar.

Die Bibliothek unterscheidet zwischen folgenden vier Node Objekten, deren Klassendiagramm in Abbildung 4.2 zu sehen ist.

L1394 Card: Eine Instanz der Klasse `L1394::Card` repräsentiert eine FireWire Karte. Jedes Card Objekt verwaltet und aktualisiert die Node Objekte für die an der FireWire Karte angeschlossenen Geräte. Die Node Objekte erhalten über ihr Card Objekt Zugriff auf den Bus. Zusätzlich stellt die Klasse Funktionen zum Suchen bestimmter Node Objekte für die Interface-Schicht bereit.

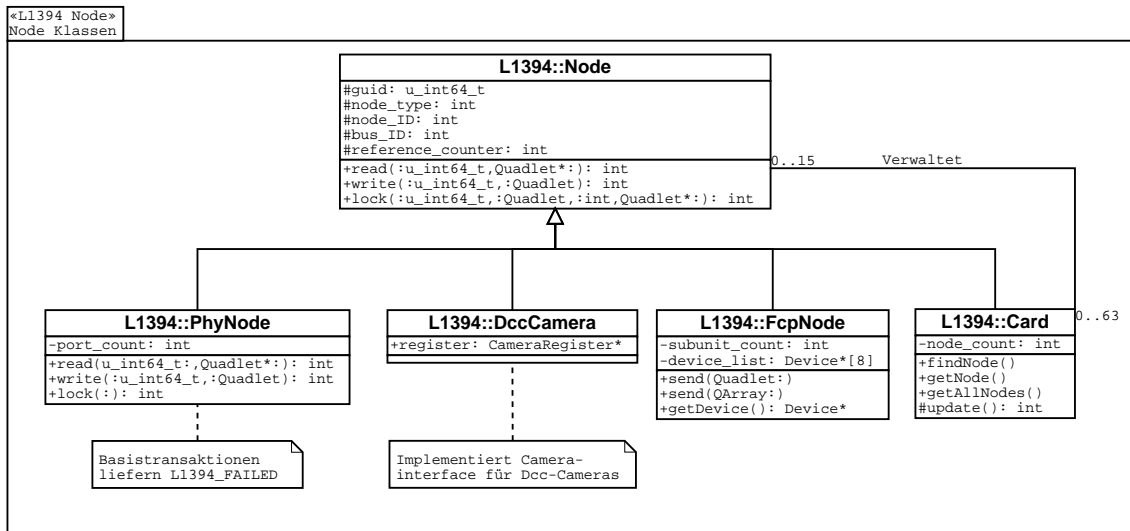


Abbildung 4.2: Klassen Hierarchie der L1394 Nodes

L1394 DccCamera: Eine Instanz dieser Klasse verwaltet eine Kamera, basierend auf der “*1394-based Digital Camera Specification*” (Version 1.20) und implementiert das Camera-Interface.

L1394 FcpNode: Objekte dieser Klasse repräsentieren Nodes, basierend auf der “*1394-general AVC Specification*”. Für die Kommunikation mit diesen Geräten wird das *Function Control Protocol* verwendet. Hierfür stellt die Klasse die Funktion SEND bereit. Geräte, die auf dieser Spezifikation basieren, verwalten bis zu acht so genannter Subunits. Eine Subunit realisiert einen bestimmten Gerätetyp innerhalb des Nodes, auf den die Anwendung durch ein entsprechendes Interface Zugriff erhält.

L1394 PhyNode: Eine Instanz dieser Klasse verwaltet Geräte, die lediglich eine Physical Layer besitzen. Hierbei handelt es sich normalerweise um Repeater. Die Funktionen für die Basistransaktionen READ, WRITE und LOCK liefern immer einen Fehler (L1394_FAILED) zurück und besitzen die GUID 0.

4.2.2 Interface-Schicht

Die Interface Schicht setzt sich aus den für den Programmierer sichtbaren Klassen zusammen. Hierzu gehören neben den Interface-Klassen der Geräte, die Klasse L1394::Session sowie die Klasse L1394::EventHandle, deren Zusammenhang in Abbildung 4.3 zu sehen ist.

Der Zugriff auf die Bibliothek selbst erfolgt über eine globale Instanz der Klasse L1394::Session. Als Hauptschnittstellenklasse stellt sie Funktionen zum Suchen der unterschiedlichen Geräte bereit und ist für die Initialisierung aller benötigten Komponenten zuständig. Aus diesem Grund wurde für die Klasse das Entwurfsmuster *Einzelstück* (Singleton) verwendet. Somit wird sichergestellt, dass nur ein Instanz dieser Klasse existiert und keine mehrfache Initialisierung der Bibliothek möglich ist. Die statische Funktion GetSession() erstellt das Session Objekt und liefert eine Referenz auf das Objekt zurück. Das Session Objekt instantiiert für jede Karte ein Card Objekt und richtet einen Reset-Handler durch die Raw1394 Bibliothek ein. Anschließend wird der Update-Vorgang² bei den Card Objekten aufgerufen, der die restlichen Node Objekte erstellt.

²Diese Funktion ist auch für die Aktualisierung der Node Objekte nach einem Bus-Reset zuständig.

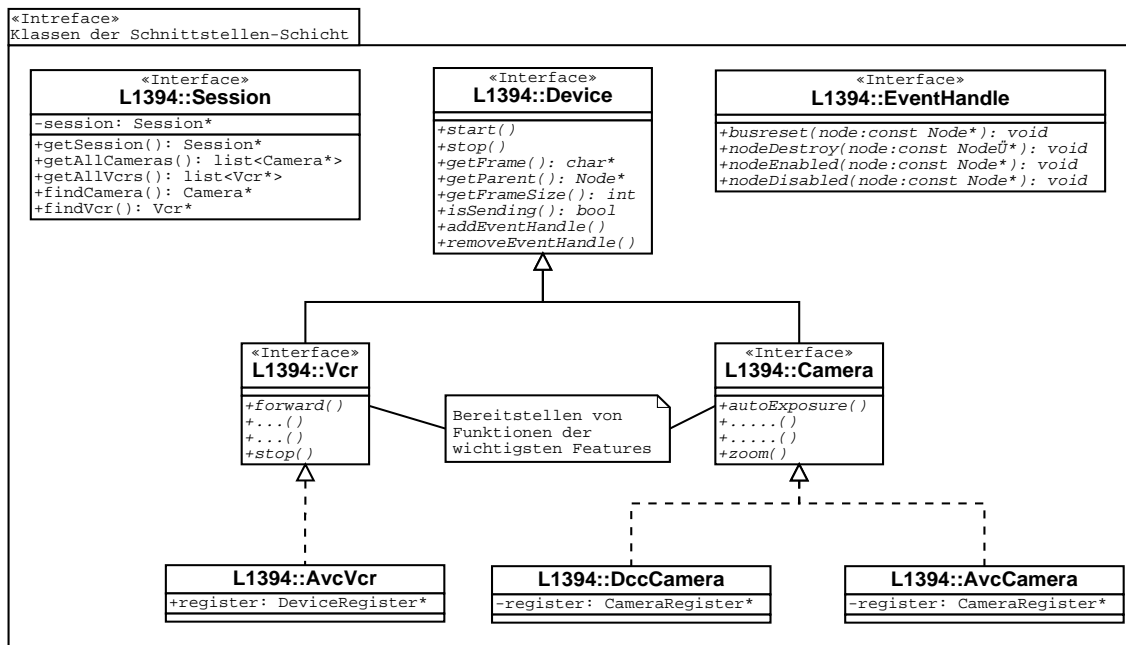


Abbildung 4.3: Klassen der Interface-Schicht

Die Bibliothek stellt für jedes Gerät ein entsprechendes Interface bereit, das als *L1394 Device* bezeichnet wird. Die abstrakte Klasse `L1394::Device` ist Basisklasse für jedes `L1394 Device` und definiert grundlegende Funktionen für das Event-Handling. Das Device Objekt erhält über ein `Node` Objekt, den so genannten *Parent Node*, Zugriff auf den Bus.

In der aktuellen Version der Bibliothek sind zwei Klassen von der Klasse `L1394::Device` abgeleitet:

L1394::Camera: Die abstrakte Klasse `L1394::Camera` definiert das Interface für Kameras und wird durch die Klassen `L1394::DccCamera` und `L1394::AvcCamera` implementiert.

L1394::Vcr: Die abstrakte Klasse `L1394::Vcr` definiert das Interface für Videorecorder und wird durch die Klasse `L1394::AvcVcr` implementiert.

Diese beiden Interface-Klassen definieren Funktionen für den Zugriff auf die Features eines Gerätes. Da ein Feature auf unterschiedliche Weise ausgeführt werden kann, verfügt jedes `L1394 Device` über ein zusätzliches Objekt, das als *L1394 Register* bezeichnet wird. Dieses Objekt stellt Funktionen bereit, wie ein Feature ausgeführt werden kann. Abbildung 4.4 zeigt die entsprechenden Klassen. Eine Referenz auf das entsprechende Register Objekt wird zurückgegeben, sobald die Funktion eines Features aufgerufen wird. Über diese Referenz kann nun bestimmt werden, wie das Feature ausgeführt werden soll.

Ein Beispiel soll dieses Idee verdeutlichen.

```

#include <l1394_session.h>

using namespace L1394;

int main(int argc, char* argc[]) {
    //Initialisieren der Bibliothek;
    Session *session = GetSession();

    //Suche eine Kamera (AV/C oder DCC)
}

```

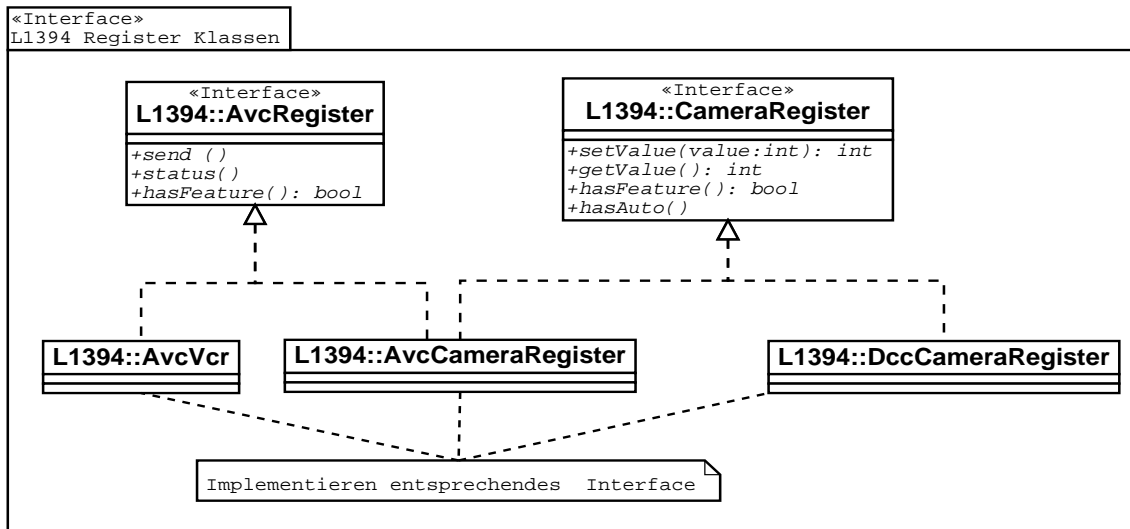


Abbildung 4.4: L1394 Register Klassen

```

Camera *camera = session->findCamera();

//Wenn Pointer = 0 wurde keine Kamera gefunden.
if(camera == NULL) {
    cout << "Es wurde keine Kamera gefunden" << endl;
    exit(0); }

// Überprüfe ob dieses Feature von der Kamera unterstützt wird.
if( camera->zoom()->hasFeature() )
    //Falls das Feature unterstützt wird, soll der Zoom Wert auf
    //sein Maximum gesetzt werden.
    camera->zoom()->setValue( camera->zoom()->getMaxValue() );

//Überprüfe, ob der Wert richtig gesetzt wurde.
    if( camera->zoom()->getValue() == camera->zoom()->getMaxValue() );
    cout << "Wert wurde korrekt gesetzt" << endl;
else
    cout << "Wert wurde nicht korrekt gesetzt" << endl;

//Lediglich das Session Objekt muss gelöscht werden.
delete session;

return 0;
}

```

Über eine Instanz der Klasse L1394::EventHandle wird der Programmierer über den Zustandswechsel eines Device Objekts informiert. Das Device Objekt besitzt den selben Zustand wie sein Parent-Node. Die Klasse L1394::EventHandle stellt für jeden Zustandswechsel eine Funktion bereit, wie im Klassendiagramm 4.3 zu sehen ist. Durch Ableiten der Klasse L1394::EventHandle können diese Funktionen überladen werden. Das EventHandle Objekt kann nun bei beliebig vielen Device und Node Objekten registriert werden. Ändert sich der Zustand eines L1394 Device, wird die entsprechende Funktion bei registrierten EventHandle Objekten aufgerufen. Hierfür ist die Klasse L1394::Event zuständig, deren Aufbau und Arbeitsweise in Abschnitt 4.2.4 besprochen wird. In der aktuellen Implementierung muss ein EventHandle Objekt sich

bei dem entsprechenden Device wieder abmelden, bevor es gelöscht wird. In einer späteren Version ist jedoch vorgesehen, dass diese Aufgabe von der Basisklasse übernommen wird. In Kapitel 5 finden sich hierzu zwei Beispiele.

4.2.3 Anwendungslogik-Schicht

Die Anwendungslogik-Schicht setzt sich aus den in Abschnitt 4.2.1 beschriebenen Node Objekten und der Klasse L1394::NodeFactory, die für das Erstellen der Node Objekte zuständig ist, zusammen. Neben dem Zugriff auf den FireWire Bus, der durch die Node Objekte erfolgt, gehört der korrekte Ablauf der Bibliothek zu den Hauptaufgaben dieser Schicht. Für den korrekten Ablauf der Bibliothek müssen lediglich die Daten der Node Objekte für den Zugriff auf den Bus nach einem Busreset aktualisiert werden. Folgende Regeln, für deren Einhaltung die Instanzen der Klasse L1394::Card und L1394::Node zuständig sind, sollen diesen Vorgang vereinfachen und sicherstellen, dass während des Update-Vorgangs keine ungültige Übertragung erfolgt.

- Jeder FireWire Node wird durch genau ein Node Objekt dargestellt, damit die Node Informationen eindeutig sind.
- Jedes Node Objekt, im Zustand L1394_ENABLED, erhält durch genau ein Card Objekt Zugriff auf den Bus, auch wenn der FireWire Node an mehrere Karten angeschlossen ist. Diese Regel soll sicherstellen, dass die Daten nur von einem Card Objekt gesendet werden.
- Ein Node Objekt im Zustand L1394_DISABLED erhält keinen Zugriff auf den Bus, da der entsprechende FireWire Node nicht angeschlossen ist.
- Während sich ein Node Objekte im Zustand L1394_BUSRESET befindet, werden alle Funktionen für die Interface-Schicht blockiert. Diese Regel soll sicherstellen, dass eine Anwendung keine falschen Informationen erhält.

Registriert die Bibliothek einen Busreset, wird der Update-Vorgang bei dem entsprechenden Card Objekt eingeleitet. Dieser Vorgang erfolgt in drei Schritten:

1.Zustandswechsel: Zuerst werden die betroffenen Node Objekte durch das Card Objekt in den Zustand L1394_BUSRESET versetzt. In diesem Zeitraum blockieren die Node Objekte ihre Funktionen und unterbrechen laufende Transaktionen. Das Card Objekt verwirft alle Topologie-Informationen und die Liste der verwalteten Node Objekte wird aufgelöst.

Aktualisierung: Das Card Objekt aktualisiert zuerst seine eigenen internen Parameter, wie node_ID und die Anzahl der angeschlossenen Nodes. Anschließend wird die GUID jedes Nodes bestimmt und bei der Datenhaltungs-Schicht angefragt, ob ein entsprechendes Objekt für diesen Node bereits existiert. Falls ein solches Objekt existiert, liefert diese Schicht eine Referenz auf das Objekt zurück und inkrementiert dessen Referenz-Counter. Die dafür zuständigen Klassen innerhalb der Datenhaltungs-Schicht garantieren dem Card Objekt die Eindeutigkeit der Nodes. Wie das erreicht wird, wird in Abschnitt 4.2.4 erläutert. Das Card Objekt aktualisiert die internen Datenstrukturen des Node Objekts und speichert die Referenz des Objekts in seiner Liste. Falls kein Node Objekt existiert, wird die NodeFactory beauftragt ein solches Objekt zu erstellen. Das Card Objekt übergibt das Node Objekt an die Datenhaltungs-Schicht und merkt sich ebenfalls eine Referenz auf das Objekt.

2.Zustandswechsel: Zum Abschluss werden die Node Objekte wieder von dem Card Objekt in den Zustand L1394_ENABLED bzw. L1394_DISABLED überführt und unterbrochene Transaktionen wiederholt.

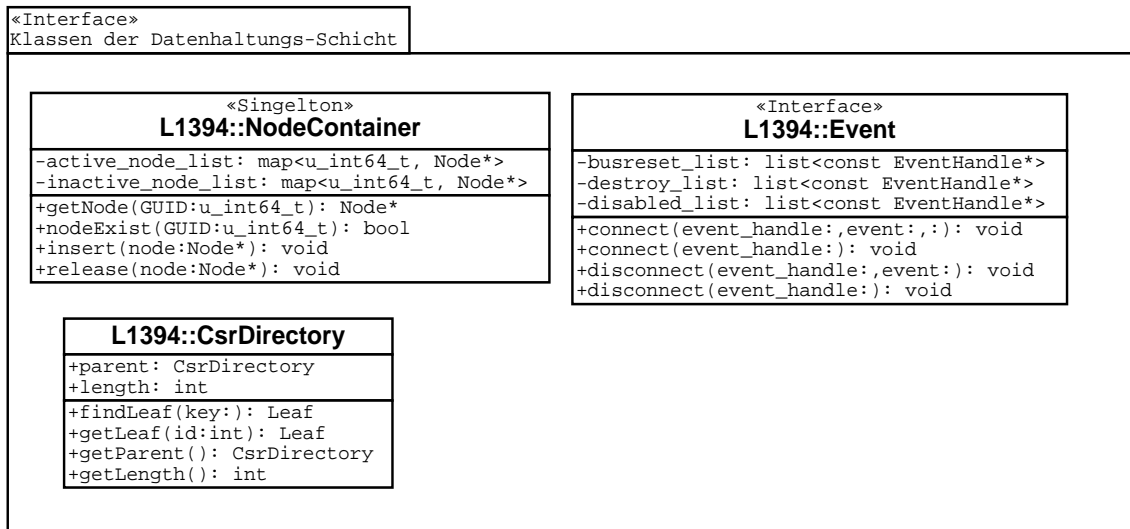


Abbildung 4.5: Klassen der Datenhaltungs-Schicht

An dieser Stelle soll noch kurz die Instantiierung der Node und Device Objekte besprochen werden. Das Erstellen aller Node und Device erfolgt über eine Instanz der Klasse `L1394::NodeFactory`. Das Erzeugende Muster *abstrakte Fabrik* (abstract factory) und *Singelton* diene als Vorlage für diese Klasse. Eine abstrakte Fabrik bietet eine Schnittstelle zum Erzeugen verwandter Objekte, ohne ihre konkreten Klassen zu benennen. Das Card Objekt braucht somit selbst den Typ des zu erstellenden Nodes nicht zu kennen und kann die Verwaltung beliebiger Node Objekte übernehmen. Analog dazu verwaltet ein Fcp-Node lediglich Device Objekte und überlässt die konkrete Instantiierung seiner Objekte der NodeFactory. Hierdurch erhält der Programmierer die Möglichkeit, durch Ableiten von der Klasse `L1394::NodeFactory`, die entsprechenden Funktionen zu überladen und seine eigenen Node bzw. Device Objekte von der Bibliothek verwalten und erstellen zu lassen. Die Klasse wird zusätzlich als Singelton implementiert um zu garantieren, dass für identische FireWire Geräte Objekte gleichen Typs erstellt werden.

Damit eine neue NodeFactory von der Bibliothek verwendet wird, muss diese *vor* der Initialisierung der Bibliothek erstellt werden.

4.2.4 Datenhaltungs-Schicht

Die Datenhaltungs-Schicht setzt sich aus den unterschiedlichen Klassen zur Datenhaltung zusammen. Hierzu gehören neben der Klasse `L1394::NodeContainer`, die Klassen `L1394::CsrDirectory` und `L1394::Event`, deren Entwurf und Implementierung in diesem Abschnitt beschrieben und diskutiert wird. Abbildung 4.5 zeigt die Schnittstellen dieser Klassen.

Zu den Hauptaufgaben dieser Schicht gehört die Verwaltung der Objekte aus der Anwendungslogik. Realisiert wird dies durch die Klasse `NodeContainer`, die zu den wichtigsten Klassen dieser Schicht gehört. Der `NodeContainer` basiert auf einer Kombination der Entwurfsmuster *Container* und *Singelton*, dessen globale Instanz allen Card Objekten zu Verfügung steht. Die Entscheidung der Kombination zweier Entwurfsmuster ergab sich aus den speziellen Anforderungen an die zu verwaltenden Objekte. Neben den rudimentären Funktionen zum Suchen, Einfügen und Vergeben von Objekten, die zu den charakteristischen Aufgaben eines Containers gehören, muss ein Card Objekt sich auf die Eindeutigkeit der Ergebnisse des `NodeContainer` verlassen um eine Verletzung seiner Regeln auszuschließen. Hierzu wurde diese Klasse zusätzlich als Singelton entworfen um eine mehrfache Instantiierung zu verhindern. Somit wird den Card Objekten garantiert, dass sie alle auf dem selben `NodeContainer` operieren und keine zusätzlichen Instanzen existieren können, die ebenfalls Node Objekte verwalten. Diese Eigenschaft wäre auch durch

eine globale Variable innerhalb der Card Objekte realisierbar gewesen. Der Grund gegen diese Methode liegt lediglich in einem übersichtlicherem Design, der die besondere Eigenschaft der Klasse direkt aus der Wahl des Entwurfsmuster erkennen lässt. Für die Implementierung der Klasse selbst werden zwei *Pair Associative Container* der STL verwendet, welche die Node Objekt als Key-Value Paare verwalten. Diese Datenstruktur garantiert das Einfügen und Finden von Objekten in logarithmischer Zeit und erlaubt somit einen schnellen Zugriff für die Card Objekte. Innerhalb der Klasse besitzt immer nur eine Funktion des Klassen-Interface Zugriff auf diese Datenstrukturen, der durch Mutex-Variablen garantiert wird. Somit wird einem Card Objekt zusätzlich der exklusive Zugriff auf die Daten ermöglicht. Bei der Vergabe von Objekten vermerkt der NodeContainer in dem Objekt selbst, durch inkrementieren des Referenz-Counter, wie oft dieses an Card Objekte vergeben wurde. Card Objekte müssen nicht mehr benötigte Node Objekte wieder freigeben, wodurch der Referenz-Counter wieder decrementiert wird. Wird ein Node Objekt nicht mehr benötigt, bewahrt es der NodeContainer für eine gewisse Zeit noch in der Datenstruktur für inaktive Node Objekte auf, bis es endgültig gelöscht wird.

Die Klasse `L1394::CsrDirectory` ist für das Abbilden des Control&Status Register eines FireWire Nodes als Software Objekt zuständig. Diese Objekte werden jedoch nur bei einem Zugriff seitens des Programmiers instantiiert. Die aktuelle Implementierung der Klasse `CsrDirectory` erstellt daraufhin eine komplette, dem Csr-Directory des FireWire Node entsprechende Verzeichnisstruktur mit den darin gespeicherten Informationen und übergibt dem Programmierer eine Referenz auf das Root-Directory. Unter Berücksichtigung der Tatsache, dass einem Control&Status Register theoretisch bis zu 256MB Adressraum zu Verfügung steht, kann diese Implementierung zu schwerwiegenden Problemen führen und bedarf auf Dauer einer Überarbeitung. Zur Rechtfertigung dieser Implementierung sollte allerdings gesagt werden, dass die während des Entwurfs und Implementierung der Bibliothek verwendeten Geräte weniger als 10KByte für das gesamte Csr-Directory inklusive der gespeicherten Informationen verwenden. Lediglich einige digitale Photo-Kameras verwenden mehrere MByte, zum Speichern der Photos, des CSR-Directories.

Zum Abschluss dieses Abschnitts wird noch die Klasse `L1394::Event` besprochen. Jedes Node Objekt verfügt über eine Event Objekt, das die angemeldeten EventHandle Objekte verwaltet. Das Anmelden eines EventHandle Objekt erfolgt über ein Device-Objekt. Das Device Objekt reicht den Aufruf an seinen Parent-Node weiter, der es bei seinem Event Objekt anmeldet. Folgende Events werden von der Bibliothek verarbeitet.

- `BUSRESET` - Nach einem eintreffenden Busreset aktualisiert das Card Objekt die von ihm verwalteten Node Objekte. Im Anschluss wird das Ereignis dem Event Objekt gemeldet, das die angemeldeten EventHandle Objekte, durch Aufruf der Funktion `busreset()`, über das Ereignis informiert. Dieses Event wird nur von Card Objekten verarbeitet.
- `NODEDISABLED` - Wenn ein Node vom FireWire Bus getrennt wird und von keinem Card Objekt mehr verwaltet wird, wird das Objekt in den Zustand `L1394_DISABLED` versetzt. Das Objekt selbst bemerkt dies, indem der Referenz-Counter auf 0 gesetzt wird. Es führt die Zustandsänderung aus und benachrichtigt sein Event Objekt über die Zustandsänderung, das seinerseits die angemeldeten EventHandle Objekte, durch Aufruf der Funktion `nodeDisabled()`, informiert.
- `NODEENABLED` - Analog zu dem Ereignis `NODEDISABLED`, existiert das Ereignis `NODEENABLED`. Dieses Ereignis tritt ein, wenn ein Node an den Bus angeschlossen wird, dessen Objekt, im Zustand `L1394_DISABLED`, noch existiert. Somit wird das Node Objekt wieder von einem Card Objekt verwaltet, was das inkrementieren des Referenz-Counter bedeutet. Das Node Objekt versetzt sich wieder in den Zustand `L1394_ENABLED` und veranlasst sein Event Objekt die angemeldeten EventHandle Objekte über das Ereignis zu informieren.
- `NODEDESTROY` - Bevor ein Node Objekt gelöscht wird, werden die die angemeldeten EventHandle Objekte durch das Event Objekt benachrichtigt. Das Objekt selbst wird erst gelöscht, wenn bei den EventHandle Objekten die Funktion `nodeDestroy()` abgearbeitet wurde.

5. Beispiel Applikationen

Dieses Kapitel beschreibt die Implementierung zweier Anwendungsprogramme, basierend auf der L1394 Bibliothek. Anhand dieser Beispiele soll ein Eindruck über die Möglichkeiten der Bibliothek vermittelt werden.

Bei den Programmen handelt es sich um graphische Oberflächen zum Steuern eines Videorecorders bzw. einer Kamera.

Für die Darstellung der graphischen Oberflächen wird die Qt-Bibliothek der Firma Troll-Tech in der aktuellen Version 2.2 verwendet. Der angegebene Source-Code betrachtet nur die Teile, in der die Bibliothek verwendet wird. Die meisten Funktionen implementieren hierbei Callback-Funktionen die auf Eingaben der Oberflächen reagieren.

Unter [7] findet sich neben der API ein ausführliches Tutorial für diese Bibliothek, sowie der vollständige Source-Code der Programme.

5.1 QVcrPanel

Das Programm QVcrPanel stellt eine einfaches GUI zum steuern eines Videorecorders dar. Es stellt Tasten für die wichtigsten Steuer-Befehle bereit und zeigt zusätzlich den aktuellen Timecode an.

Die Klasse QVcrPanelView bestimmt das Aussehen des Kontrollpanels, das in Abbildung 5.1 zu sehen ist und implementiert die Funktionen für den Zugriff auf den Videorecorder. Diese Klasse ist zusätzlich von der Klasse L1394::EventHandle abgeleitet um über Ereignisse informiert zu werden.

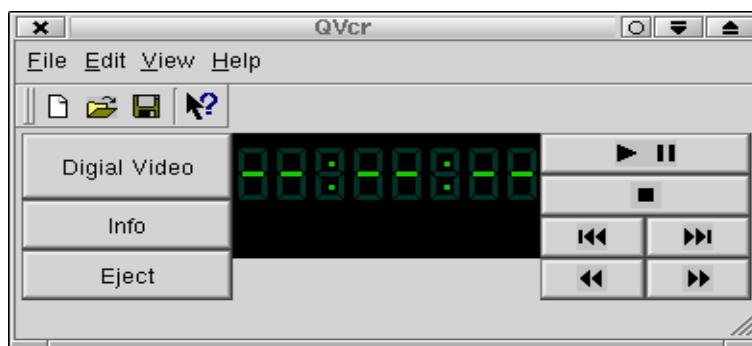


Abbildung 5.1: Programm QVcrPanel

```

QVcrPanelView::QVcrPanelView(QWidget *parent)
: QWidget(parent)
{
    session = GetSession();           //Initialisieren der Bibliothek
    vcr = session->findVcr();         //Suche Videorecorder
    if (vcr == NULL)                 //Wenn kein Videorecorder gefunden wurde
        setEnabled(false);           //soll das Panel auf keine Eingabe reagieren
    else {
        vcr->addEventHandle(this);    //sonst melde diese Instanz für Device-Spezifische
    session->addEventHandle(this);    //Events und Busreset-Events an.
    }
    initLayout();                    //Initialisiere das Layout (Qt-Spezifisch)
    initConnect();                   //Verbinde Button-Ereignisse mit Funktionen (Qt-Spezifisch)
}

```

Die folgenden Funktionen werden bei einem Maus-Klick auf den entsprechenden Button aufgerufen.

```

void QVcrPanelView::fastForwardPressed()
{ vcr->fastForward()->send(); } //Schneller Vorlauf

void QVcrPanelView::rewindPressed()
{ vcr->rewind()->send(); } //Schneller Rücklauf

void QVcrPanelView::playPressed(){ //Implementiert Start und Pause
    if (!playing) {
        vcr->play(); //Starte Videorecorder
        playing = true;
        timer->start( 500, false ); // Der Timer ruft nach 0.5 sec die Funktion
    } else { // setTimecode() auf
        vcr->pause(); //Halte Videorecorder an
        playing = false;
        timer->stop();
    }
}

```

Die folgende Funktion aktualisiert den Timecode des Displays

```

void QVcrPanelView::setTimecode()
{
    Timecode time_code = vcr->getTimeCode();

    if (time_code.hour() == 0xff) { //Wenn Timecode ungültig setze
        play_timer[0]->display('-'); //Display auf --
        play_timer[1]->display('-');
    } else { //sonst auf den entsprechenden Wert
        play_timer[0]->display((int)time_code.hour() / 16);
        play_timer[1]->display((int)time_code.hour() % 10);
    }
    play_timer[2]->display(':');

    if (time_code.minute() == 0xff){ //Wenn Timecode ungültig setze
        play_timer[3]->display('-'); //Display auf --
    }
}

```

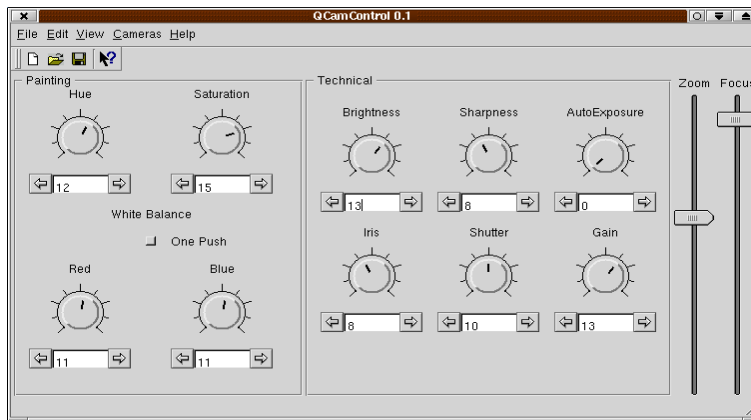


Abbildung 5.2: Programm QCamControl

```

    play_timer[4]->display('-') ;
} else {
    //sonst auf den entsprechenden Wert
    play_timer[3]->display((int)time_code.minute()/16) ;
    play_timer[4]->display((int)time_code.minute()%16) ;
}
play_timer[5]->display(':') ;
if (time_code.second() == 0xff){ //Wenn Timecode ungültig setze
    play_timer[6]->display('-') ; //Display auf --
    play_timer[7]->display('-') ;
} else {
    //sonst auf den entsprechenden Wert
    play_timer[6]->display((int)time_code.second()/16) ;
    play_timer[7]->display((int)time_code.second()%16) ;
}
}

//Diese Funktion bestimmt das Verhalten der Applikation bei einem Busreset
void QVcrPanelView::busreset(Node*){
    if (vcr->getState() == L1394_Enabled) //Überprüfe den Zustand des Vcr-Objekts.
        return;
    else {
        //Wenn das Gerät über den Bus nicht erreichbar ist,
        timer->stop();
        vcr = session->findVcr(); //suche einen neuen Videorecorder.
        if (vcr == NULL) //Wenn keiner angeschlossen ist,
            setDisabled(true); //schalte das Display ab
        else
            setEnabled(true);}
}

```

5.2 QCamControl

Bei dem Programm QCamcontrol handelt es sich um eine graphische Oberfläche zum Steuern von Kameras am FireWire Bus. Das Programm verfügt über Drehregler und Slider für die wichtigsten Features einer Kamera, wie in Abbildung 5.2 zu sehen ist. Die Regler für nicht unterstützte Features werden abgestellt und reagieren auf keine Eingabe. Über das Menü Camera, kann jede angeschlossene Kamera ausgewählt werden. Zusätzlich können, durch Copy(CTRL-C) und Paste(CTRL-V), die Einstellungen der Kameras synchronisiert werden.

Die Klasse QCamControlView bestimmt den Aufbau der Oberfläche und implementiert die Funktionen zum Steuern der Kamera.

```

QCamControlView::QCamControlView(QWidget *parent) : QWidget(parent) {
    session = GetSession(); //initialisieren der Bibliothek
    camera_list = session->getAllCameras(); //suche alle Kameras
    if (camera_list.size() == 0);
        camera = NULL;
    else camera = *(camera_list.begin());
    session->addEventHandle(this); //registriere dieses Objekt für Busresets.
    initPaint(); //Qt-spezifisch
    initTechnical(); //Qt-spezifisch
    initSlider(); //Qt-spezifisch
    toplevel->activate(); //Qt-spezifisch

    if (camera == NULL) {
        cout << "no cam connected" << endl;
        setDisabled(true);
    } else {
        update();
        buttonConnect();
    }
}
}

```

Der Aufbau der Funktionen zum Ändern eines Wertes wird an einem Beispiel beschrieben. Die restlichen Funktionen ergeben sich analog.

```

void QCamControlView::changeSharpness(int i){
    if (i!= camera->sharpness()->getValue())
        //Wenn i != aktueller Wert, setze neuen Wert. Der Aufruf setValue(i) setzt
        //den neuen Wert liest ihn anschließend aus gibt liefert ihn zurück. Hierdurch
        //kann kontrolliert werden, ob der Vorgang erfolgreich war.
        technical_button[1]->setValue(camera->sharpness()->setValue(i));}

```

Bei einem eintreffenden Busreset wird die Liste der angeschlossenen Kameras aktualisiert.

```

void QCamControl::busreset(const Node*) {
    camera_list = session->getAllCameras(); //Aktualisiere die Kameraliste.
    if (camera_list.size() == 0) //Wenn keine Kamera angeschlossen ist,
        setEnabled(false); //soll das Programm auf keine Eingabe reagieren
    else
        setEnabled(true);
    if (camera->getState() != L1394_ENABELD)//Wenn die aktuelle Kamera nicht mehr
        camera = *(camera_list.begin()); //angeschlossen ist, setze neue Kamera
    update(); //und aktualisiert die Oberfläche.
}

```

Literaturverzeichnis

- [1] Don Anderson: FireWire System Architecture Second Edition, Addison Wesley Longman Inc., 1998.
- [2] Alessandro Rubini: Linux-Gerätetreiber, O'Reilly & Associates Inc, 1998.
- [3] Soeren Andersen: An Overview of the GNU/Linux IEEE-1394 Subsystem, December 1999.
- [4] Erich Gamma, Richard Helm, Ralph Johnson und John M. Vlissied: Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley 1994.
- [5] Ivar Jacobson, Grady Booch, James Rumbaugh: The Unified Software Development Process, Addison Wesley, 1999.
- [6] AV/C Digital Interface Command Set, General Specification, 1394 Trade Association, 1998.
- [7] Homepage der L1394 Bibliothek, http://graphics.cs.uni-sb.de/~replix/11394_home
- [8] Homepage der 1394 Trade Association, <http://www.1394ta.com>

