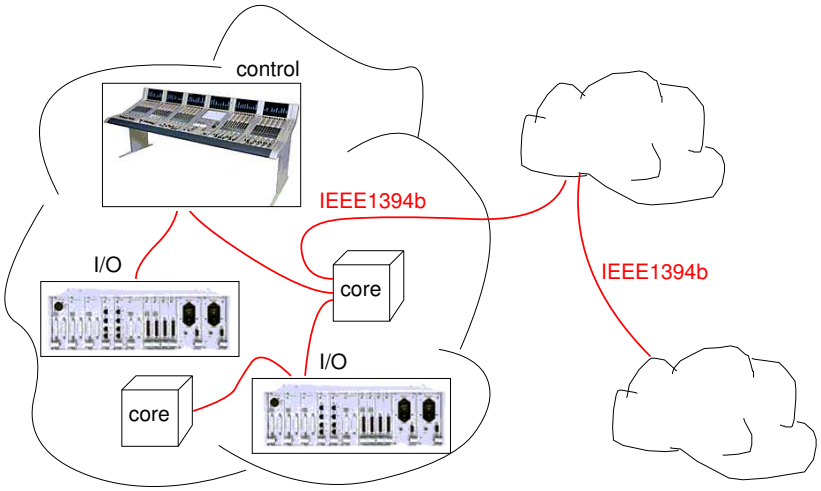
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Prof. L. Thiele
Sommersemester 2003

Semesterarbeit SA-2003-31

Realisierung eines professionellen Audio-Netzwerkes mit IEEE-1394



Verfasser:
Jonas Biveroni

Betreuer:
Adrian Riedo
Philipp Blum

Vorwort

Diese Arbeit entstand im Auftrag von STUDER Professional Audio AG in Regensdorf. Betreut wurde ich extern von Adrian Riedo (adrian.riedo@studer.ch) und Attila Karamustafaoglu (attila.karamustafaoglu@studer.ch), meine Ansprechperson an der ETH war Philipp Blum (blum@tik.ee.ethz.ch). Ich möchte allen diesen Personen für die Unterstützung danken, die sie mir entgegengebracht haben. Obwohl die Arbeit nicht ganz den gewünschten Verlauf nahm, konnte ich viel lernen dabei und das Arbeitsklima bei STUDER in Regensdorf sagte mir sehr zu.

Zürich, 6. Juli 2003

Jonas Biveroni

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze
Prof. L. Thiele
Sommersemester 2003

SEMESTERARBEIT

für
Herrn Jonas Biveroni

Realisierung eines professionellen Audio-Netzwerkes mit IEEE-1394

Betreuer: Adrian Riedo, STUDER AG
Stellvertreter: Philipp Blum, ETZ G82

Ausgabe: 4. April 2003
Abgabe: 9. Juli 2003

Einleitung

Studer Professional Audio AG ist einer der führenden Hersteller von Audiosystemen für den professionellen Einsatz in Rundfunk, Produktion, Postproduktion und Aufnahme. Für die kommende Generation von Geräten spielt die Vernetzung eine zentrale Rolle. Ziel ist, eine Vielzahl von Geräten wie Mischpulte, Audioserver, Stageboxes, etc. mittels eines Standardnetzwerks zu verbinden. Dafür hat sich der IEEE-1394 Standard insbesondere in der "b"Variante als sehr geeignet erwiesen. Es können damit Netzwerkknoten mit hohen Übertragungsraten bis 3.2GB/s über weite Distanzen bis meherere km verbunden werden. Eine Trennung von asynchronen und isochronen Kanälen im IEEE-1394 Protokoll erlaubt eine garantierte Übertragung der sensitiven Multimediadaten ohne Beeinflussung durch die Last der zu übertragenden Steuerdaten.

Aufgabenstellung

Die aktuelle Semesterarbeit hat zum Ziel, ein Demonstrationsnetzwerk unter Verwendung des genannten Standards aufzubauen. Dabei soll folgender- massen vorgegangen werden:

1. Studium Dokumentation der IEEE-1394a/b Spezifikation und Einarbeitung in die Entwicklungsumgebung und Evaluationsplattform des DM-1000 der Firma BridgeCo.

2. Aufbau und Inbetriebnahme einer mehrkanaligen Audiostrecke via IEEE-1394a mit Hilfe zweier Evaluations-Kits von BridgeCo.

3. Definition eines simplen Konfigurations-Protokolls zur Verwaltung von DM-1000 Knoten auf dem IEEE-1394 Netzwerk (Aktivieren/Deaktivieren der Audio Quellen/Senken auf den einzelnen Knoten) und Implementation in Form einer Testapplikation.

4. Ausarbeitung des Schemas eines IEEE-1394b Knotens (DM-1000 inkl. RAM, Flash, PHY, etc) basierend auf bestehenden Arbeiten und mit Hilfe der Betreuer bei STUDER. Das Layout und die Herstellung des PCB's wird von der Firma STUDER übernommen.

5. Testen und evtl. Anpassen der Testapplikation auf der Zielhardware.

6. Verfassen des Berichtes.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Einführung

Die vorliegende Arbeit hat zum Ziel, eine Audio-Teststrecke über IEEE 1394 aufzubauen. STUDER Professional Audio AG stellte mir diese Aufgabe. Die Firma ist bekannt für Audiogeräte, wie sie im Studiobereich verwendet werden. In der Audioproduktion sind zum Teil sehr grosse Anlagen mit vielen Kanälen notwendig. Hier ist die Vernetzung der einzelnen Komponenten eine grosse Herausforderung, da viele Kanäle mit hohen Übertragungsraten flexibel verwaltet werden müssen. Heute setzt STUDER ein proprietäres Verbindungsprotokoll ein, welches eine sehr hohe Bandbreite bietet. Bis zu 96 Audiokanäle ($96kHz$, $24bit$) können durch ein gewöhnliches CAT5 LAN-Kabel übertragen werden. Es besteht aber der Wunsch, ein Standardnetzwerk zu verwenden. Dabei hat sich IEEE 1394b als viel versprechend herausgestellt, unter anderem, da hohe Datenraten über lange Distanzen erreichbar sind. Im folgenden Bericht wird häufig der Begriff "FireWire" als Synonym für "IEEE 1394" gebraucht.

Zu Beginn der Arbeit studierte ich Unterlagen über die FireWire Spezifikationen und über die von BridgeCo zur Verfügung gestellten Evaluation Boards mit dem DM1000 Chip. Dazu kam die Einrichtung und das Kennenlernen der Entwicklungsumgebung, mit deren Hilfe ich Beispielprogramme entwickelte und diese auf den Evaluation Boards ausführte und ausprobierte. Aus einem ersten Versuch, eine Firmware für eine Audioteststrecke zu schreiben, entstand *myHardwareStreaming.c*. Ein anderer Ansatz führte zu *myArmStreaming.c*. Kapitel 6 und die Zusammenfassung am Schluss dieses Berichts geben Informationen über diese Testapplikationen. Gegen Ende des Semesters begann ich mit dem Verfassen des vorliegenden Schlussberichts und der Präsentation. Parallel dazu entwickelte ich den RS232-Kontroller (Kapitel 7).

Das erste Kapitel befasst sich mit dem Standard IEEE 1394 und gibt einen Überblick über die wichtigsten Merkmale und Daten des Busses. Im folgenden Kapitel ist die Hardware und das Softwarepaket des DM1000 Interface Processor ICs von BridgeCo beschrieben. Dieser Chip wurde von STUDER ausgewählt für die Entwicklung von Audiogeräten mit IEEE 1394 Netzwerkknoten. Er bietet neben komplettem Link Layer Core, Packet-Handler und ARM Prozessor zahlreiche Audioschnittstellen. Um mit dem DM1000 erste Versuche machen zu können, wurden zwei Evaluation Boards eingesetzt, welche in Kapitel 3 beschrieben sind. Im näch-

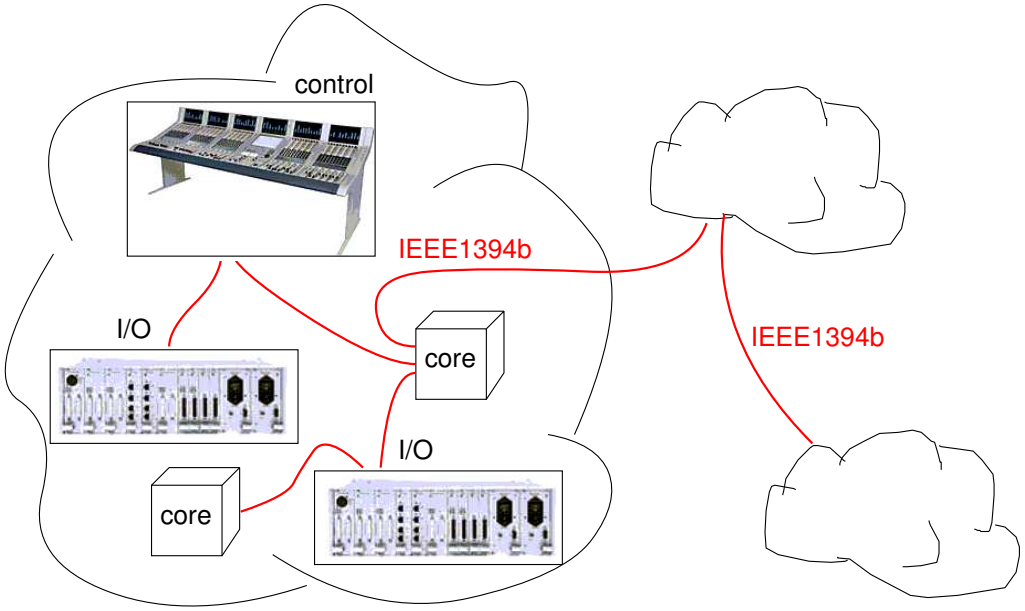


Abbildung 1: Illustration eines professionellen Audio-Netzwerks, wie es von STUDER geplant ist. Es soll vollständig mit IEEE 1394b aufgebaut werden. Abgebildet sind die Komponenten Kontrolleinheit ("Mischpult"), Core (DSP Cluster) für die Datenverarbeitung und I/O Frame mit analogen und digitalen Schnittstellen.

sten Kapitel wird ein Kommunikationsmodell für Netzwerke in einem professionellen Audiostudio vorgestellt. Kapitel 5 und 6 erläutern den Umgang mit den verwendeten Entwicklungswerkzeugen und dem DM1000 Softwarepaket sowie meine eigenen Programme, die im Laufe der Semesterarbeit entstanden. Im letzten Kapitel schliesslich wird ein System vorgestellt, das ich entwickelt habe, um zwei Netzwerkknoten über eine serielle Schnittstelle konfigurieren zu können.

Kapitel 1

IEEE 1394 Serial Bus

1.1 Geschichte

1986
In diesem Jahr begannen Ingenieure von Apple Computers mit der Weiterentwicklung des seriellen Busses IEEE 1394, einer bis dahin noch nicht ausgereiften Implementation. Das Ziel war ein schneller, kostengünstiger und einfacher Bus, der die bestehenden und in Aussicht gestellten Systeme an Übertragungsgeschwindigkeit und Flexibilität übertreffen sollte. Um Multimediadaten effizient transportieren zu können, wurde eine Technik entwickelt, die es erlaubte, Daten in Echtzeit zu übertragen.
1990
Apple begann, den bis anhin proprietären Bus zu standardisieren unter dem Namen *FireWire*. Eine IEEE Arbeitsgruppe wurde gebildet mit Delegierten der Firmen Apple, Texas Instruments und anderen.
1993
In einer öffentlichen Videovorführung demonstrierte Apple zum ersten Mal die Übertragung von Multimedia- und Steuerdaten in Echtzeit durch den IEEE 1394 / FireWire Bus.
1994
Die *1394 Trade Association* (1394 TA) wurde gegründet, um die Entwicklung von Systemen voranzutreiben, die via IEEE 1394 verbunden waren.
1995
Bald wurden die ersten Chipsets erhältlich, und mit ihnen auch erste Geräte im Bereich Unterhaltungselektronik. Der 1394 Standard wurde verbessert, und 1996 präsentierten Microsoft, Texas Instruments und PAVO joined forces in einem gemeinsamen Projekt die Übertragung von Audiodaten von einer Festplatte über 1394 zu einem audiophilen Endgerät.
2001
Dem alten FireWire Standard IEEE 1394a mit seiner nominellen Datenrate von 400MBit/s wurde USB 2.0 mit 480MBit/s gegenübergestellt.

2003
Im neuen Standard IEEE 1394b sind Übertragungsraten von bis zu 1600MBit/s vorgesehen. Ausserdem wird die Verwendung von längeren Kabeln als zuvor ermöglicht. Apple bietet bereits Geräte an mit einer eigenen, 800MBit/s schnellen Implementierung von IEEE 1394b, genannt FireWire 800.

1.2 Aufbau von IEEE 1394 Netzen

1.2.1 Verkabelung

Übliche FireWire-Kabel sind dünne Kupferleitungen mit 6-poligen Steckern, die ursprünglich für Nintendos GameBoy als kindersichere Stecker entwickelt wurden. Neben zwei verdrillten Adernpaaren für die Datenübertragung enthält das Kabel zwei Drähte für die Stromversorgung der angeschlossenen Geräte.

Der neue Standard 1394b nutzt ebenfalls zwei verdrillte Adernpaare, verwendet aber eine andere Signalkodierung und andere Pegel, den so genannten *Beta-Mode*. Hier werden die Daten mit dem 8B/10B Verfahren kodiert. Dieser Code ist gleichspannungsfrei, weshalb Beta-Netzwerkknoten einfach mittels Kondensatoren oder Transformatoren elektrisch isoliert werden können. Ausserdem kann der Takt zuverlässig mit einer PLL rekonstruiert werden - es sind also keine separaten *strobe*-Leitungen notwendig wie beim alten IEEE 1394a. Dies ist auch der Grund, weshalb Beta-Nodes im Gegensatz zu den "alten" über zwei Adernpaare im Vollduplexbetrieb kommunizieren können.

Tabelle 1.1 gibt einen Überblick über die maximalen Kabellängen zwischen Beta-Netzwerkknoten je nach Übertragungsrate. Die längsten Kabel für 1394a messen 4.5m (Quelle: [1]).

Medium	S100[a]	S200	S400	S800	S1600
Shielded Twisted Pair	-	-	4.5m	4.5m	4.5m
CAT-5 UTP	100m	-	-	-	-
Plastic Optic Fiber	50m	50m	-	-	-
Hard Polymer Clad Fiber	100m	100m	-	-	-
Multimode Fiber	-	-	100m	100m	100m

[a] S100: Datenrate von 100MBit/s, etc.
[b] - nicht spezifiziert

Tabelle 1.1: Maximale Kabellängen für Verbindungen zwischen Beta-Mode Knoten, wie sie im Standard IEEE 1394b spezifiziert sind.

1.2.2 Topologie

Der IEEE 1394 Standard erlaubt Topologien mit *daisy chain*-, Baum-, Sternstruktur oder auch Kombinationen davon. In 1394b sind sogar Ringverbindungen möglich. Diese werden automatisch detektiert und vom Bus Manager aufgelöst. Zwischen zwei Geräten dürfen nicht mehr als 16 *hops* auftreten. Bis zu 63 Geräte können innerhalb eines Busses verbunden werden, und 1023 solcher Busse können ein Netzwerk bilden.

Während der Systeminitialisierung, zum Beispiel nach einem Bus-Reset, organisiert sich der Bus von selber in einer virtuellen baumartigen Hierarchie. Ein Gerät bildet dabei die Wurzel, welche die *Arbitration Requests* entgegennimmt und bestimmt, welcher Teilnehmer ein Datenpaket an den Bus legen darf.
Im IEEE 1394b Standard ist ein neues Arbitrierungsverfahren namens *BOSS*[1] implementiert. Dabei übernimmt jeweils der Daten sendende Knoten, der BOSS, vorübergehend die Funktion der Wurzel, das heisst, alle *Arbitration Requests* der anderen Teilnehmer werden von ihm verwaltet (Abbildung 1.1). Da die Beta-Nodes im Vollduplexverfahren kommunizieren, fliessen die Daten und die Requests gleichzeitig, weshalb diese Arbitrierung mit einem äusserst geringen zeitlichen *overhead* vonstatten geht.

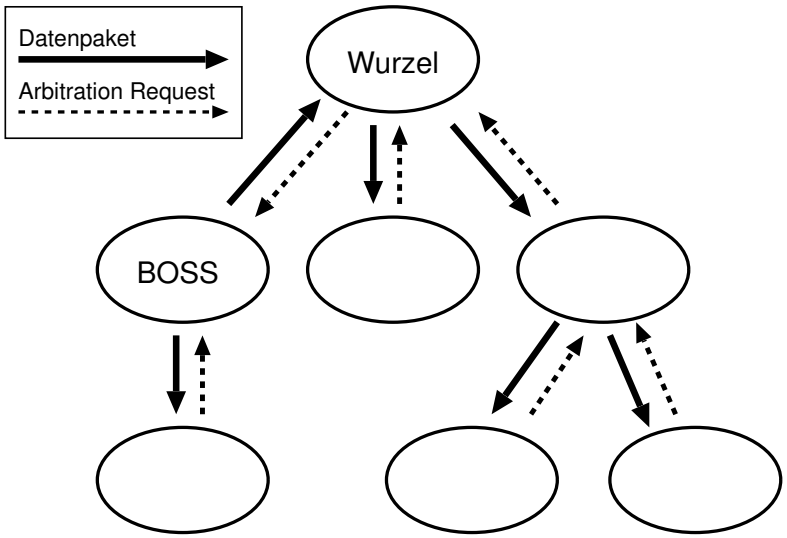


Abbildung 1.1: Arbitrierung und Datenfluss in einem vollduplexfähigen 1394b Bus. Der BOSS legt ein Datenpaket an den Bus und empfängt gleichzeitig die Arbitrierungsanfragen der anderen Knoten.

[1] Bus Owner/ Supervisor/Selector

1.2.3 Datenübertragungsraten

Der Standard IEEE 1394a sieht drei Brutto-Übertragungsgeschwindigkeiten vor:

- 98304 kBit/s, aufgerundet zu 100 MBit/s, auch Betriebsart S100 genannt
- 196608 kBit/s, aufgerundet zu 200 MBit/s, auch Betriebsart S200 genannt
- 393216 kBit/s, aufgerundet zu 400 MBit/s, auch Betriebsart S400 genannt

Die Spezifikation von 1394b vervierfacht diese Werte auf gerundete 800 und 1600 MBit/s, vorgesehen ist sogar die Erweiterung auf 3200 MBit/s.

1.3 Isochrone Übertragung

Isochrone Datenströme werden übertragen mit vorhersagbarer maximaler Verzögerung und garantierter Bandbreite. Mehrere Audio- oder Videokanäle können durch einen der 63 verfügbaren isochronen Kanäle transportiert werden, wobei die Bandbreite den limitierenden Faktor darstellt.
Die Übertragungen sind unterteilt in Zeitsegmente von 125μs Dauer. Jeder dieser Zyklen beginnt, wenn der *bus cycle master*, ein dazu bestimmter Knoten, ein spezielles asynchrones Paket, das CSP[2], verschickt. Darin enthalten ist der Wert eines Taktzählers, welcher nun von allen angeschlossenen Geräten verwendet wird, um jeweils die lokale "Uhr" zu aktualisieren. So wird eine gemeinsame Zeitbasis gewonnen.

1.4 Asynchrone Übertragung

Asynchrone Transfers können jederzeit stattfinden, wenn der Bus frei ist. Das Protokoll garantiert jedoch mindestens 20% der Zeit für diese Art der Datenübertragung. Durch die request- und response- Meldungen entstehen Lücken und Verzögerungen in der Datenübertragung, weshalb asynchrone Transfers eher für Steuerzwecke prädestiniert sind.

1.5 IEEE 1394 Protocol Stack

1.5.1 Physical Layer

Die physikalische Schicht bestimmt, wie die Daten elektrisch oder optisch übertragen werden, sowie die Beschaffenheit der Anschlüsse und Kabel. Hier wird auch der Konfigurationsprozess der Knoten beim Anlegen der Betriebsspannung geregelt und die Arbitrierung des Busses.

[2] Cycle Start Packet

1.5.2 Link Layer

- Asynchrone Übertragungen:
  Der Link Layer bekommt *transaction requests* vom Transaction Layer und formt daraus Pakete, die über die physikalische Schicht versendet werden. Wenn ein Paket empfangen wird, leitet der Link Layer den darin enthaltenen Request zum Transaction Layer weiter.

- Isochrone Übertragungen:
  Ein Software-Treiber übergibt die Daten direkt dem Link Layer, welcher sie als Pakete weiterreicht. Trifft ein empfangenes Paket ein, wird es dem Software-Treiber weitergegeben, sofern die Kanalnummer übereinstimmt mit einem der als Empfänger registrierten Kanäle. Andernfalls wird es verworfen.

1.5.3 Transaction Layer

Der Transaction Layer nimmt asynchrone *transfers* vor, die von einer Applikation veranlasst wurden. Diese Transfers werden unterteilt in mehrere *transactions*. Es gibt *read, write* und *lock* Transactions, welche alle ein *request* und ein *response* beinhalten.

1.5.4 Bus Management Layer

In dieser Schicht sind Dienste implementiert, die gewährleisten, dass die Knoten des Busses effizient funktionieren. Hier werden sie identifiziert, die Topologie des Busses wird ermittelt und die Ressourcen, inklusive isochrone Kanäle und Bandbreite, werden zugeteilt.

Kapitel 2

DM1000 Interface Processor

2.1 Beschreibung des DM1000 Bausteins

Der DM1000 Chip ist ein integriertes System mit den Hauptbestandteilen, die in den folgenden Abschnitten beschrieben sind. Er enthält einen voll-duplex fähigen Link Layer Controller, der sowohl den älteren Standard IEEE 1394a als auch 1394b unterstützt. Der integrierte ARM 946 Prozessorkern übernimmt die Steuerung und ermöglicht es dem Benutzer, eigene Applikationen zur Datenverarbeitung zu entwickeln. Die grosse Anzahl Portpins umfasst unter anderem Schnittstellen zu externen Speicherbausteinen und zum 1394a/b PHY-Chip sowie Audio und Video Datenleitungen und konfigurierbare General Purpose Ein- und Ausgänge (GPIOs). Diverse flexible Module wie zwei eingebaute UARTs[1], ein SPI[2] und SCI[3] ermöglichen es, den Aufwand an externen Bauteilen gering zu halten.
Der Baustein wird mit $3.3V$ und $1.8V$ (I/O bzw. core Spannung) versorgt. Er wird in ein PQFP128 Gehäuse gepackt. Weiterführende Informationen folgen in den untenstehenden Abschnitten oder sind dem Datenblatt [2] zu entnehmen.

2.1.1 Übersicht der Architektur

Abbildung 2.1 zeigt den inneren Aufbau des DM1000 Interface Processors.

2.1.2 AV-Ports

Die AV-Ports 1 und 2 können mit verschiedenen Audio- und Videoformaten umgehen. Beide sind individuell konfigurierbar für eine von 9 Betriebsarten (Tabelle 2.1). Jeder Port umfasst 8 Datenleitungen (AVxDATA[0:7]), einen Clock Ein- oder Ausgang (AVxCLK) und zwei Steuerleitungen (AVxCTRL[1:0]).

[1] Universal Asynchronous Receiver and Transmitter
[2] Serial Peripheral Interface
[3] Serial Communication Interface

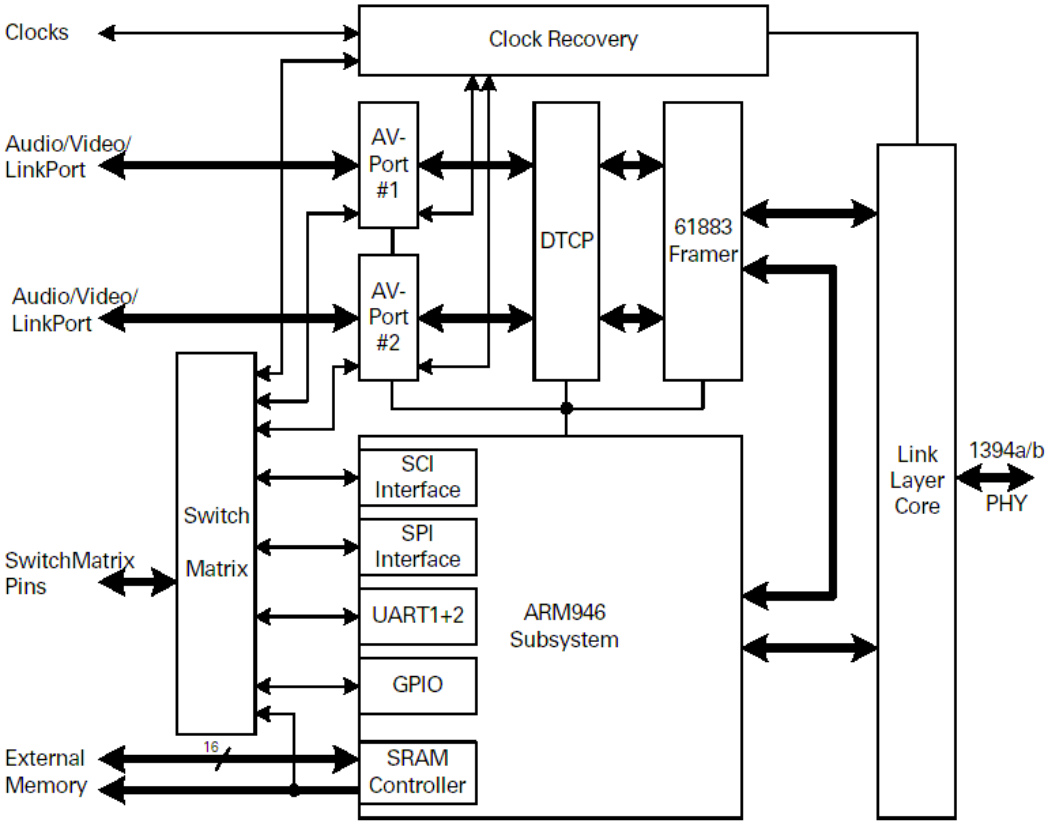


Abbildung 2.1: Blockschaltbild des DM1000 Interface Processor Bausteins von BridgeCo. Dieser Chip stellt Schnittstellen zwischen FireWire (Link Layer, rechte Seite der Abbildung) und digitalen Audio- oder Videosignalen (links) zur Verfügung.

Betriebsart	Typ	Bemerkungen
1	Stereo Audio	Bis zu 8 I^2S Audio Signale (entsprechend 16 Audiokanälen pro AV-Port)
2	I^8S Audio	Bis zu 8 I^8S Audio Signale (entsprechend 64 Audiokanälen pro AV-Port)
3	IEC60958 Audio	Bis zu 8 S/PDIF Audio Signale
4	LinkPort 4-bit	4-bit Schnittstelle zu *Analog Devices* 210xx DSPs
5	LinkPort 8-bit	8-bit Schnittstelle zu *Analog Devices* 211xx DSPs
6	MPEG-2 TS	Parallele Schnittstelle für komprimierte Videodaten
7	DV	Parallele Schnittstelle für komprimierte Videodaten
8	Video 8-bit	Parallele Schnittstelle für unkomprimierte Videodaten
9	Video 10-bit	Parallele Schnittstelle für unkomprimierte Videodaten

Tabelle 2.1: Betriebsarten, für welche die Audio und Video Ports des DM1000 konfigurierbar sind.

2.1.3 Switch-Matrix

Die Switch-Matrix ist ein konfigurierbarer Multiplexer. Den Pins SWMATRIX[0:8] können durch Softwarebefehle gruppenweise verschiedene Funktionen zugeordnet werden. So besteht zum Beispiel die Möglichkeit, den Pins SWMATRIX[5:8] eine der drei folgenden internen Signalgruppen zuzuordnen:

- NCS, SCK, RxD, TxD der SPI Schnittstelle
- SCL, SDA des I^2C Interfaces und AV1DATA[8:9] für 10-bit Videodaten
- GPIO[0:3], vier General Purpose I/Os (Abschnitt 2.1.4)

Eine Liste aller Konfigurationen findet man im DM1000 Datenblatt [2].

2.1.4 General-Purpose I/O (GPIO)

Der DM1000 stellt 20 GPIOs zur Verfügung. Vier davon sind über die Switch-Matrix nach aussen geführt; zusätzlich können AV-Port Pins als GPIOs benutzt werden, falls sie nicht für die Übertragung von Audio oder Video Daten gebraucht werden.
Jeder GPIO kann einzeln in einem Taktzyklus gelesen oder geschrieben werden; ausserdem kann auf alle gleichzeitig zugegriffen werden mittels des data registers und des direction registers.

2.1.5 ARM946 Core

Abbildung 2.2 zeigt eine schematische Darstellung des ARM946 Subsystems. Wichtige Eckdaten des integrierten RISC[4] Controllers:

- Taktfrequenz bis zu 100MHz
- Je ein 4kB grosser Daten und Instruktions-Cache Block
- 16kB *tightly coupled* (TC) Programmspeicher und 8kB TC-Datenspeicher; diese Blöcke ermöglichen Speicherzugriffe in einem einzigen Taktzyklus
- Ein zusätzlicher 80kB grosser SRAM Block wird geteilt benutzt vom DMA Controller und vom ARM

2.1.6 Clock Recovery

Der Clock Recovery Block hat die Aufgabe, den Takt des DM1000 und somit des Netzwerkknotens mit der ausgewählten Referenzquelle zu synchronisieren (PLL Block) und die von den Applikationen benötigten Clocks zu generieren (Clock Generator Block). Die einzelnen Funktionsblöcke sind in Abbildung 2.3 dargestellt.
In der Tabelle 2.2 sind die möglichen Referenzsignale für den PLL aufgelistet.

[4] Reduced Instruction Set Computer

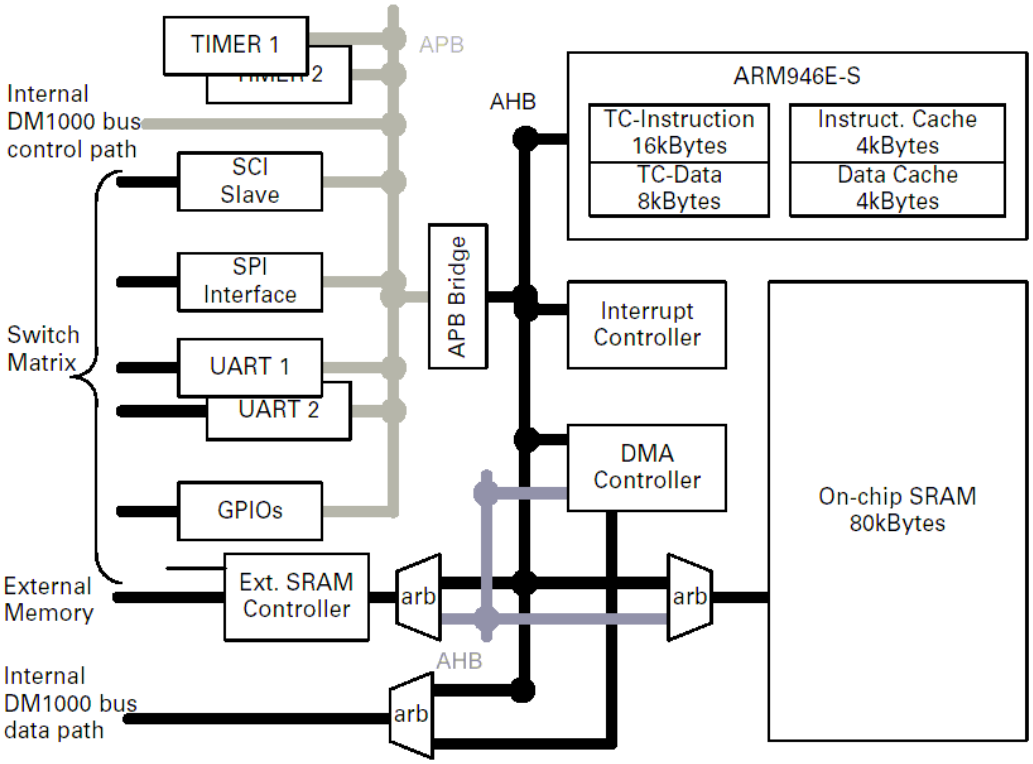


Abbildung 2.2: Blockdiagramm des im DM1000 integrierten ARM946 Subsystems.

Bezeichnung	Bemerkungen
SytMatch	Clock wird aus einem AV-gebundenen isochronen Kanal gewonnen
CSP	Clock wird aus den Cycle Start Packets (CSP) gewonnen
SYNC_I/O	Clock wird aus dem Signal am SYNC_I/O Pin gewonnen
AV-Input	Clock wird aus einem der AV-Eingangspins gewonnen

Tabelle 2.2: Mögliche PLL Referenzquellen des Clock Recovery Block.

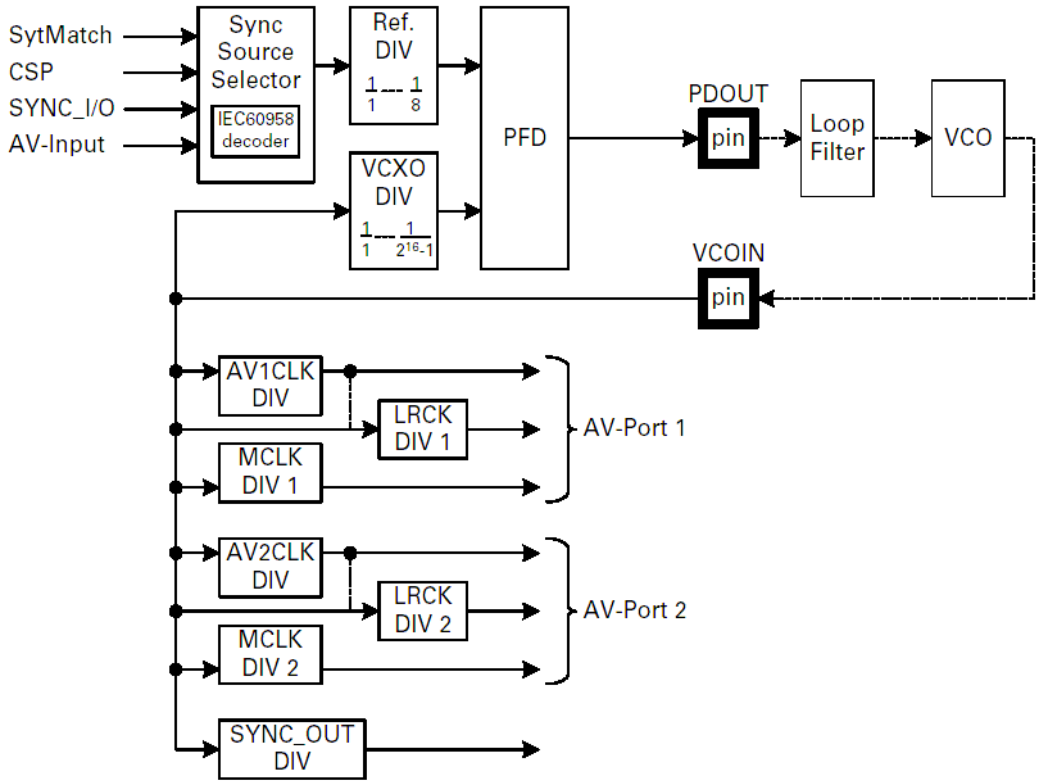


Abbildung 2.3: Clock Recovery Block: Diese Einheit ist zuständig für die Synchronisierung des DM1000 und die Generierung von Clocks für die AV-Ports.

2.1.7 61883 Framer/Deframer

Dieser Funktionsblock des DM1000 übernimmt das Ein- und Auspacken der isochronen IEEE 1394-Pakete. Sobald der Block konfiguriert ist, läuft die Paketbearbeitung ohne weiteres Eingreifen der Software, das heisst, die Ressourcen des Prozessors können für andere Zwecke genutzt werden.
Eine Besonderheit des 61883 Framer/Deframers ist seine Fähigkeit, aus mehreren (Audio-) Kanälen sogenannte Clusters zu bilden, beziehungsweise, diese wieder aufzuspalten. So können mehrere Audiostreams auf einen isochronen Kanal gelegt werden, wodurch es erst möglich wird, die Bandbreite des IEEE 1394 Busses auszunutzen.

2.2 Software für den DM1000

2.2.1 Übersicht

Das 1394 Kernel & OS Programmpaket von BridgeCo ist ein Set von Softwaremodulen, die auf dem DM1000 Interface Processor laufen. Der Benutzer kann eigene Applikationen schreiben, die auf den API[5]-Funktionen aufbauen. Für alle Hardwaremodule sind bereits Treiber vorhanden. Umfangreiche Funktionen unterstützen das IEEE 1394 Bus Management und andere Dienste. Ein Betriebssystem basierend auf ThreadX ist auch inbegriffen. In den folgenden Abschnitten werden ausgewählte Funktionsgruppen erläutert. Weiterführende Beschreibungen finden sich im 1394 Kernel & OS Programmers Manual [3]. In der HTML Ausgabe [4] sind alle API Funktionen und Datenstrukturen detailliert aufgelistet. Abbildung 2.4 zeigt die Architektur der Software schematisch.

2.2.2 Interne Isochrone Verbindungen

Es gibt zwei verschiedene Arten von internen isochronen streams:

- Hardware Isochronous Streaming Internal Links: Verbindungen zwischen dem IEEE 1394 Bus und einem AV-Port

- Software Isochronous Streaming Internal Links: Verbindungen zwischen dem IEEE 1394 Bus und dem ARM, sowie zwischen einem AV-Port und dem ARM

Im folgenden sind als Beispiele ein paar Softwaredienste aufgeführt, die gebraucht werden, um Streams aufzusetzen:
isoLinkOpen(), isoLinkSetup(), isoLinkStart(), isoLinkStop(), isoLinkClose(), isoStreamRead(), isoStreamWrite().

[5]Application Programming Interface

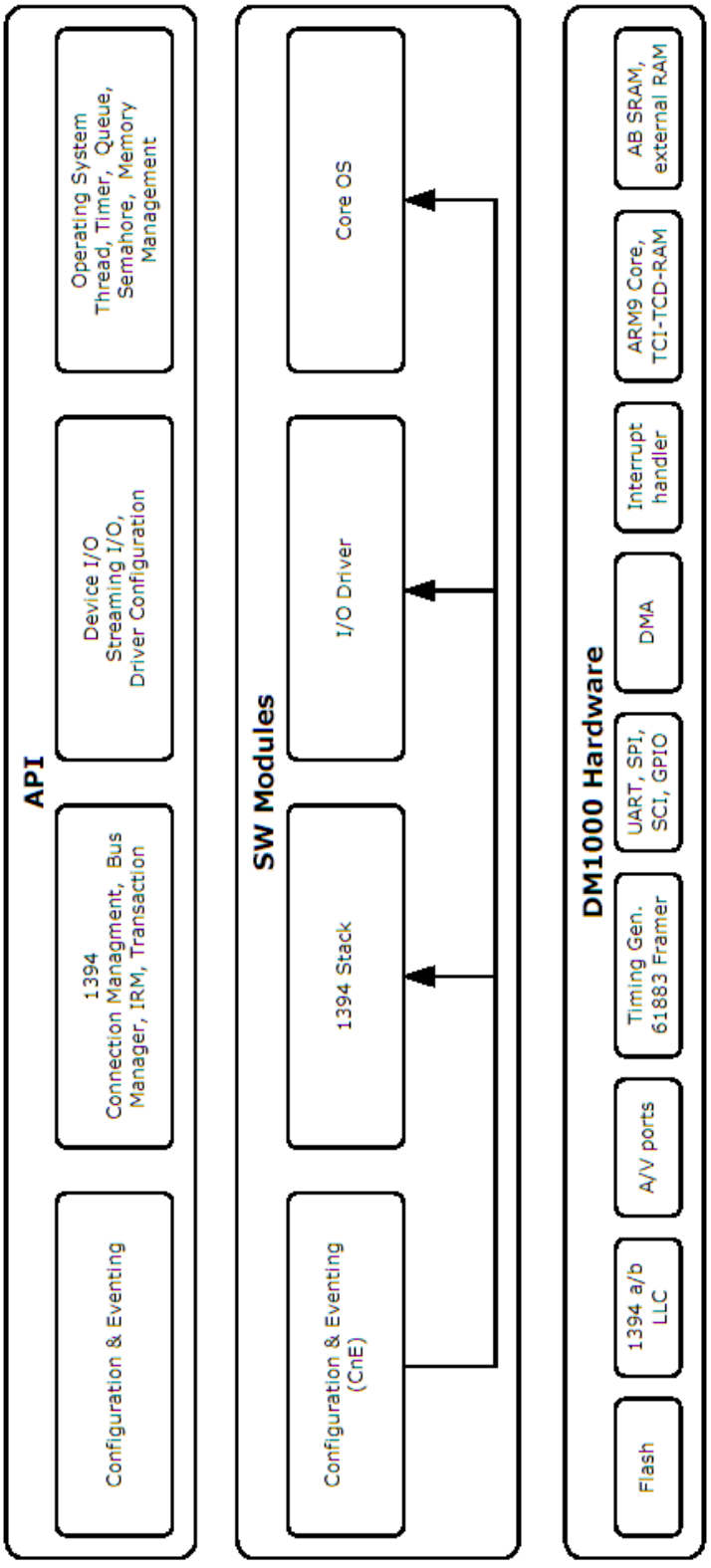


Abbildung 2.4: 1394 Kernel & OS Software-Modell für den DM1000 von BridgeCo.

2.2.3 Asynchrone Transaktionen

Die zur Verfügung gestellten Funktionen erlauben es dem Benutzer, von den nach IEEE 1394 standardisierten Asynchronen Transaktionsarten Gebrauch zu machen. Es gibt read, write und lock Operationen, um auf die asynchronen Streams, 1394 speed maps und Konfigurationsregister zuzugreifen.

2.2.4 IEEE 1394 Bus Management

Ein weiterer integraler Bestandteil des 1394 Kernel & OS (KNOS) Softwarepakets ist der serielle Bus Manager. Er ist getreu den IEEE 1394a und b Spezifikationen implementiert. Insbesondere ist die Steuerung der Netzwerkknoten (node control), das Management der Isochronen Ressourcen (isochronous resource manager, IRM) und des Busses (bus manager, BM), sowie die Konfigurationsregister (configuration status register, CSR) implementiert.
Diese Funktionen laufen normalerweise im Hintergrund ab, ohne dass sich der Benutzer bei der Entwicklung von Applikationen darum kümmern muss. Trotzdem kann auf das Bus Management Einfluss genommen werden über die API Funktionen.

2.2.5 Connection Management

Diese Funktionen sind Connection Management Procedures (CMP) nach dem Standard IEC 61883-1 [5]. Speziell werden mit ihnen auch die DM1000-internen AV streams aufgesetzt. Indem auf iPCR[6] oder oPCR[7] eines AV Geräts zugegriffen wird, können *point-to-point*- oder *broadcast*-Verbindungen aufgebaut und unterbrochen werden. Die Allozierung und Freigabe von Busressourcen wird vom CMP stack vergenommen. Ebenso die Wiederherstellung von Verbindungen nach einem Bus Reset.

2.2.6 Configuration and Eventing

Das Configuration and Eventing Modul (CnE) bietet eine kleine Datenbank auf dem Chip. Der Benutzer kann über die Funktionen des CnE API Parameter und Konfigurationsdaten (Attributes) ablegen, ohne direkt auf Hardwareregister zugreifen zu müssen. Dadurch werden beispielsweise Änderungen der Betriebsart, die zahlreiche Hardwareblöcke betreffen, stark vereinfacht.
Weiter ist es möglich, verschiedene Konfigurationen auf dem Flash-Speicher und dem RAM abzulegen. So können feste Werkeinstellungen und default-Werte auf Flash und häufig wechselnde Parameter im RAM gespeichert werden.

Eventing: Eine Applikation kann sich in eine Liste von event subscribers eintragen, um bei Änderung eines Attributs benachrichtigt zu werden. Sobald sich

[6] Input Plug Control Register
[7] Output Plug Control Register

ein Attribut ändert, wird eine control function aufgerufen. Falls es für das erfolgte Ereignis subscribers gibt, sendet die control function diesen allen einen event.

2.2.7 Betriebssystem-Dienste

In den folgenden Abschnitten sind die wichtigsten Aufgaben des Betriebssystems kurz ausgeführt.

2.2.7.1 Programmausführung

Betriebssystem-Dienste kontrollieren den Ablauf der laufenden Applikationen. Dabei gibt es vier Arten der Programmausführung:

- Initialisierung: Erster Schritt nach einem Prozessor-Reset
- Abarbeiten der Threads: Nach der Initialisierung startet das KNOS den thread scheduling loop
- Interrupt Service Routine (ISR): Sobald ein Interrupt detektiert wird, startet die ISR
- Application Timers: Ähnlich wie die ISR, mit dem Unterschied, dass der Applikation die Ursache des Interrupts verborgen bleibt. Diese Timer werden häufig eingesetzt als watchdogs, in periodischen oder time-out Diensten

2.2.7.2 Memory Management

Der Memory Manager unterhält vier verschiedene Bereiche: DRAM (heap, meist auf einem externen SRAM), SRAM-AHB (80kB on-chip SRAM), TCMI (16kB tightly coupled instruction memory) und TCMD (8kB tightly coupled data memory). Es gibt folgende Funktionsgruppen:

- Memory Pool Initialisierung
- Speicherallokation und Speicherdeallokation innerhalb eines Memory Pools
- Berechnen des freien Speicherplatzes in einem bestimmten Bereich

2.2.7.3 Ausführen der Threads

Die Threads können untereinander mittels Message Queues kommunizieren.
Abbildung 2.5 zeigt die Zustandsübergänge, die ein Thread durchlaufen kann. Es sind fünf verschiedene Zustände:

- *ready*: Der Thread wird vom Thread Scheduler kontrolliert.
- *executing*: Der Thread wird gerade ausgeführt.
- *suspended*: Der Thread kann momentan nicht ausgeführt werden.

- *completed*: Der Thread wurde fertig ausgeführt und kann nicht wieder laufen.
- *terminated*: Der Thread wurde vorzeitig beendet und kann nicht wieder laufen.

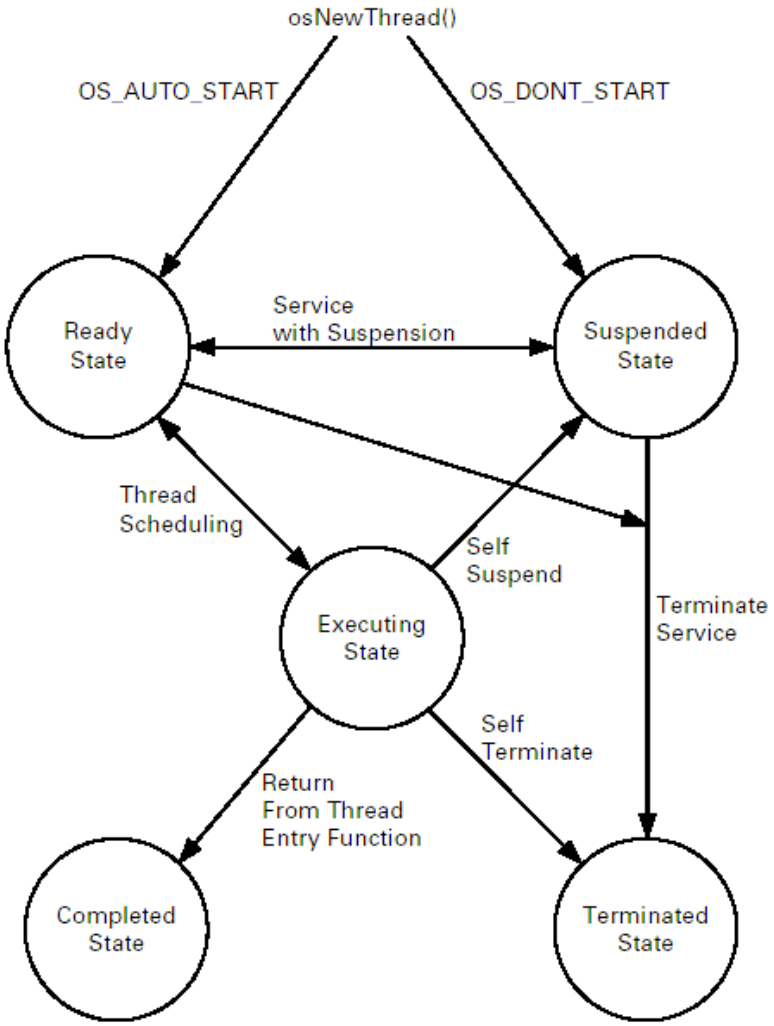


Abbildung 2.5: Zustände und Zustandsübergänge von Threads im DM1000-Betriebssystem.

Kapitel 3

DM1000 Evaluation Board

Mit dem *EVM AUDIO1 Board* stellt die Firma BridgeCo eine Plattform zur Verfügung, die es ermöglicht, Applikationen für den DM1000 zu entwickeln. Neben der IEEE 1394a Schnittstelle bietet das Evaluation Board zahlreiche Audio Schnittstellen, analoge und digitale. Über einen RS-232 Leitungstreiber kann das Board mit einer COM-Schnittstelle des Hostcomputers kommunizieren. Auf dem selben Weg werden firmware-images (*.bcd Dateien) auf die Zielarchitektur heruntergeladen.

Das Evaluation Board ist nicht dazu geeignet, den DM1000 auszulasten, oder anders gesagt, so viele Audio-Verbindungen zu betreiben, wie im Betrieb in einem Studio (z.B. bei einem DSP core Netzwerkknoten) gebraucht werden. Es bietet für diesen Zweck zu wenige digitale Audio Schnittstellen. So sind zum Beispiel keine I^8S fähige Pins auf Steckverbinder geführt.

3.1 Hardwarebeschreibung

3.1.1 Blockschaltbild

Abbildung 3.1 zeigt schematisch den Aufbau der Evaluationsplattform. In den folgenden Abschnitten werden die Interfaces näher erläutert. Weitere Informationen finden sich im *Hardware/Software Manual* [6].

3.1.2 IEEE 1394a Schnittstelle

Auf dem Board sind drei 6-polige FireWire Buchsen angebracht. Unterstützt werden die Modi S100, S200 und S400 entsprechend den Übertragungsraten von 100, 200 beziehungsweise 400 MBit/s. Der PHY-Chip ist ein *TSB41AB3* von Texas Instruments. Die Schnittstelle ist nicht elektrisch isoliert aufgebaut. Die Schaltung kann direkt über den IEEE 1394 Bus mit Spannung versorgt werden oder alternativ über ein externes Netzteil.

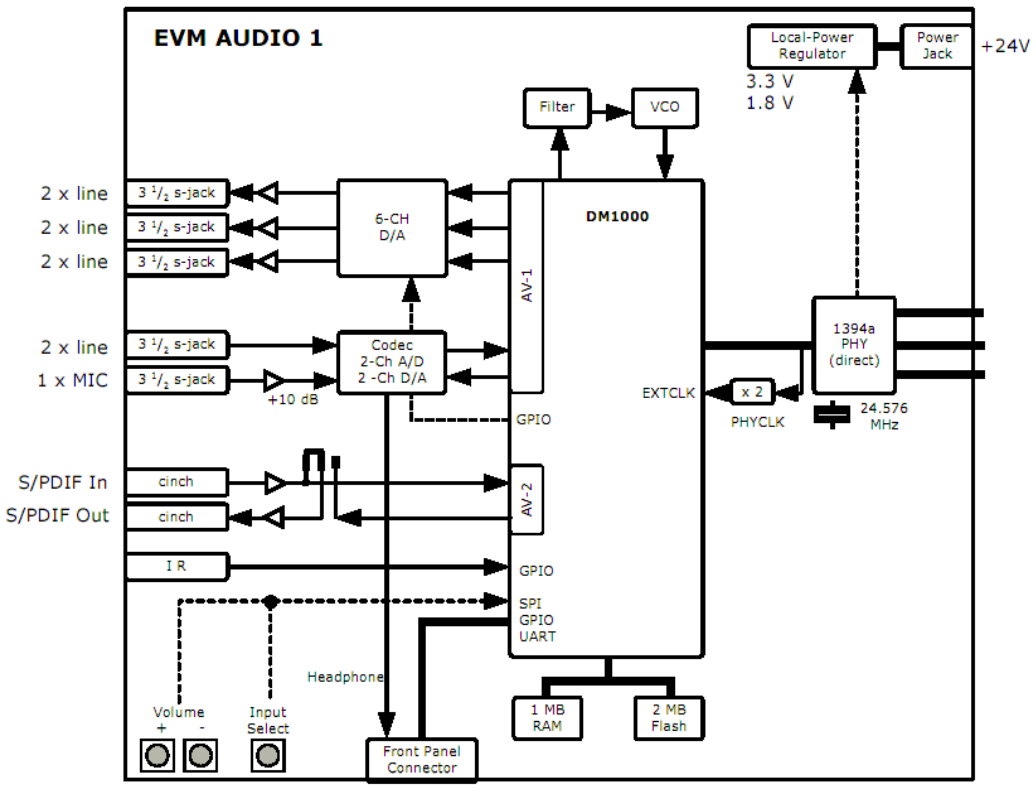


Abbildung 3.1: Blockschaltbild des EVM AUDIO1 Evaluation Boards von BridgeCo. Zentral der DM1000 Interface Processor, rechts der 1394a PHY Chip, links analoge und digitale Audio Ein- und Ausgänge.

3.1.3 Audio Schnittstellen

3.1.3.1 Eingänge

- Stereo Line Eingang
- Mono Mikrophon Eingang
- Digitaler S/PDIF Eingang

Anmerkung: Die beiden analogen Eingangssignale (Line und Mic) werden im Codec-Baustein gemischt, falls sie parallel benutzt werden.

3.1.3.2 Ausgänge

- Drei Stereo Line Ausgänge
- Stereo Kopfhörer Ausgang
- Digitaler S/PDIF Ausgang (DC-entkoppelt)

Anmerkung: Mittels eines Jumpers kann der S/PDIF Ausgang wahlweise mit dem DM1000 (AV2Data1 Pin) oder mit dem S/PDIF Eingang verbunden werden.

3.1.4 Weitere Anschlüsse auf dem Board

An einem 20-poligen Pfostenstecker (Front Panel Connector) kann auf weitere Signale zugegriffen werden. Tabelle 3.1 zeigt die Pinbelegung des Steckers.

Pin	Funktion
1	GND
2	GND
3	SPI RxD
4	SPI TxD
5	SPI Clock
6	SPI Chip Select
7	GND
8	Headphone Out Left
9	Headphone Out Right
10	$3.3V$ Supply
11	DM1000 GPIO (AV2Data2)
12	DM1000 GPIO (AV2Data3)
13	GND
14	Reset
15	$3.3V$ Supply
16	Infrared Receiver
17	DM1000 GPIO (AV2Data4)
18	DM1000 GPIO (AV2Data5)
19	DM1000 UART1: RxD
20	DM1000 UART1: TxD

Tabelle 3.1: Pinbelegung des User-Steckers auf dem EVM AUDIO1 Evaluation Board. Hier sind unter anderem die Signale des SPI und der UART bequem zugänglich.

Kapitel 4

Übertragung von Audiodaten mittels IEEE 1394

4.1 Kommunikationsmodell

Das hier vorgestellte Kommunikationsmodell für eine professionelle Audio-Studioumgebung nach [7] zeigt, wie der 1394 Protokoll-Stack erweitert wird. In Abbildung 4.1 sind die zusätzlichen Blöcke CMP[1] und AV/C[2] erkennbar.

4.1.1 Connection Management Procedures

Die CMP sind definiert in IEC 61883-1 [5]. Durch dieses Set von Funktionen werden virtuelle *input-plugs* und *output-plugs* von Sendern und Empfängern an isochrone Kanäle des seriellen Busses gebunden. So werden die Multimediadaten im System geroutet.

4.1.2 Das AV/C Modell

AV/C ist eine Gruppe von Protokollspezifikationen, welche die Grundlage der Steuerung eines AV-Systems definieren. Man unterscheidet vier Hauptbestandteile: das Modell *component*, die Datenfluss- Abstraktion *plug*, *commands* und *descriptors*.

- **Components:**
  Das Modell der Components hat eine hierarchische Struktur, deren oberste Stufe die eigentlichen Geräte, auch Module genannt, bilden. Beispiele dafür sind Synthesizer oder Mixer. Diese enthalten jeweils mindestens einen Knoten des 1394 Busses. Jeder Knoten bildet eine *unit*, welche wiederum mehrere *sub-units* mit besonderen Funktionen enthalten kann. Die Units sind individuell adressierbar.

[1] Connection Management Procedures
[2] Audio Video Control

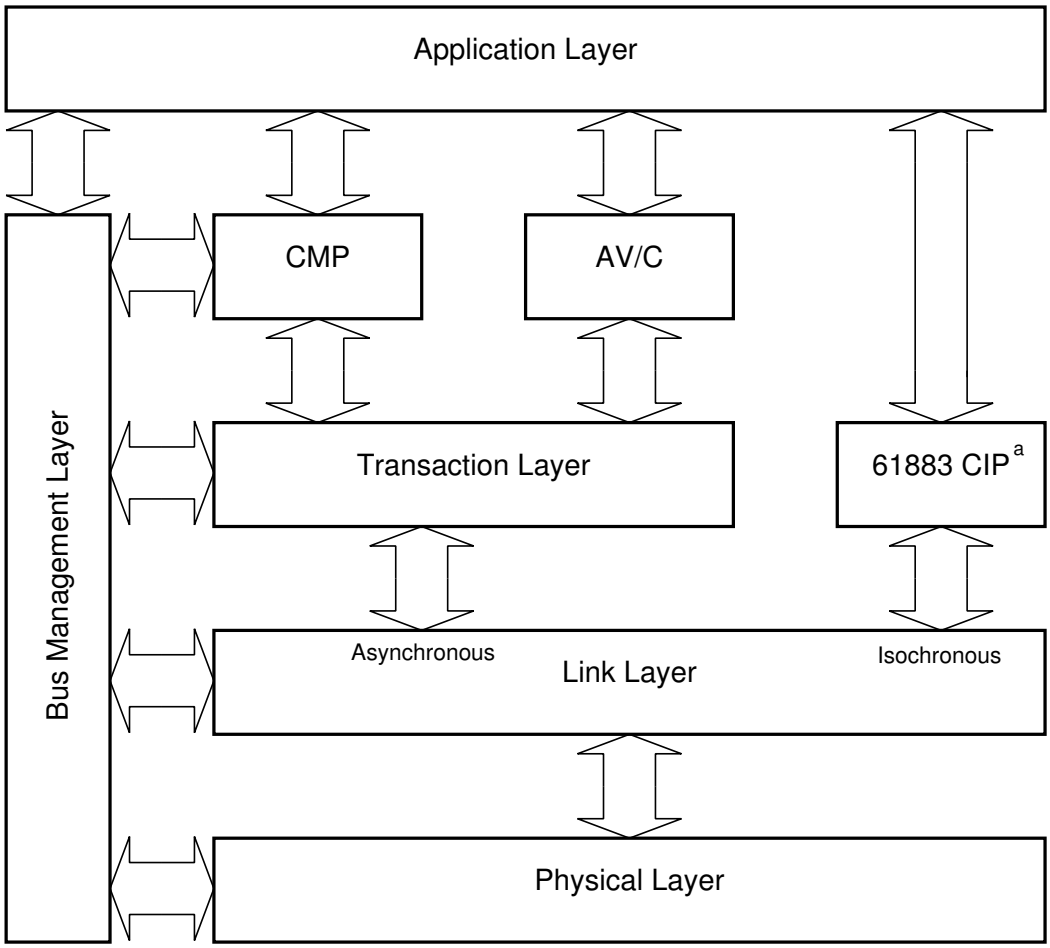


Abbildung 4.1: Kommunikationsmodell für eine professionelle Audio-Studioumgebung nach Digital Harmony [7].

- **Plugs:**
  Die Plug Abstraktion wurde entwickelt, um es Applikationen zu ermöglichen, auf einfache Weise den Fluss der Multimediadaten zwischen Units und Subunits zu steuern. So hat eine AV Unit Input- und Output Plugs für den 1394 Bus wie auch für andere Interfaces, zum Beispiel I^8S. AV Subunits besitzen Source und Destination Plugs, worüber Datenströme zu und von anderen Subunits innerhalb der gleichen Unit geroutet werden können.

- **Commands:**
  Mit den AV Commands kann eine Applikation einerseits die Funktion und den Zustand der Units und Subunits abfragen und modifizieren und andererseits Verbindungen zwischen deren Plugs aufbauen und abbrechen. Die Kommandos werden in asynchronen Paketen getreu dem *Function Control Protocol* (FCP) transportiert [5].

- **Descriptors:**
  Ein Descriptor ist eine Datenstruktur, die zu einer Unit oder Subunit gehört. Sie enthält statische und dynamische Informationen über die Funktion und den Status eines Geräts.

4.2 Synchronisation der Netzwerkknoten auf Samples genau

Im Aufbau einer professionellen Studioumgebung nach [7] ist ein Gerät, genannt *Conductor*, vorgesehen, das folgende Merkmale hat:

- **Media routing:** Der Conductor erlaubt dem Benutzer des Systems, die Datenflüsse zu steuern, ähnlich wie mit einer traditionell verwendeten *patch bay* (Gerät mit zahlreichen Ein- und Ausgangsbuchsen, die mit kurzen *patch*-Kabeln verbunden werden können).

- **Sync management:** Mit Hilfe des Conductors kann der Benutzer eine Synchronisationskonfiguration einstellen, also beispielsweise einen *Synchronization Master* bestimmen.

- **Latency management:** Verzögerungen auf Sampleebene, die zum Beispiel in Prozessiereinheiten auftreten, können vom Benutzer ermittelt und entsprechende Latenzen können anderswo eingefügt werden, damit keine Verschiebungen auftreten.

Der Digital Harmony Studio Conductor [7] ist typischerweise in Software implementiert auf einer Computer Workstation.

Kapitel 5

Software-Entwicklung für den DM1000

Dieses Kapitel behandelt die Entwicklung von Software für den DM1000, im Falle meiner Semesterarbeit speziell für das Evaluation Board von BridgeCo.

5.1 Vom C-Code zur Applikation

5.1.1 Code-Entwicklung

Wie von BridgeCo vergeschlagen, wurde mit der ARM Development Suite (ADS) v1.2 gearbeitet. Sie umfasst einen C/C++ Compiler und Linker (Metrowerks Code Warrior) sowie einen ARM-Debugger (AXD). Abbildung 5.1 zeigt ein Metrowerks-Projekt als Beispiel. Es empfiehlt sich, bei der Entwicklung eines eigenen Software-Projekts von einem bestehenden, funktionsfähigen Muster auszugehen und dieses dann Schritt für Schritt zu ergänzen und anzupassen.

In der Datei *file.scatter* wird dem Linker mitgeteilt, wo die verschiedenen Programmteile physikalisch abgelegt werden. Dies kann in einem internen Speicher des DM1000 sein oder im externen SRAM oder Flash auf der Zielhardware.

5.1.2 Download auf die Zielhardware

Nach dem *make project* Vorgang steht das Programm als Objectfile zur Verfügung (*file.axf*). Diese Datei muss mit einem Programm von BridgeCo (bcd_generate.exe) in ein *.bcd*-File umgewandelt werden, welches wiederum über die serielle Schnittstelle auf das Board geladen wird. Dies erfolgt mit der Funktion *send file* aus einem Terminal-Programm (z. B. *hyperterminal*). Als Protokoll für die Übertragung ist *1K Xmodem* zu wählen.

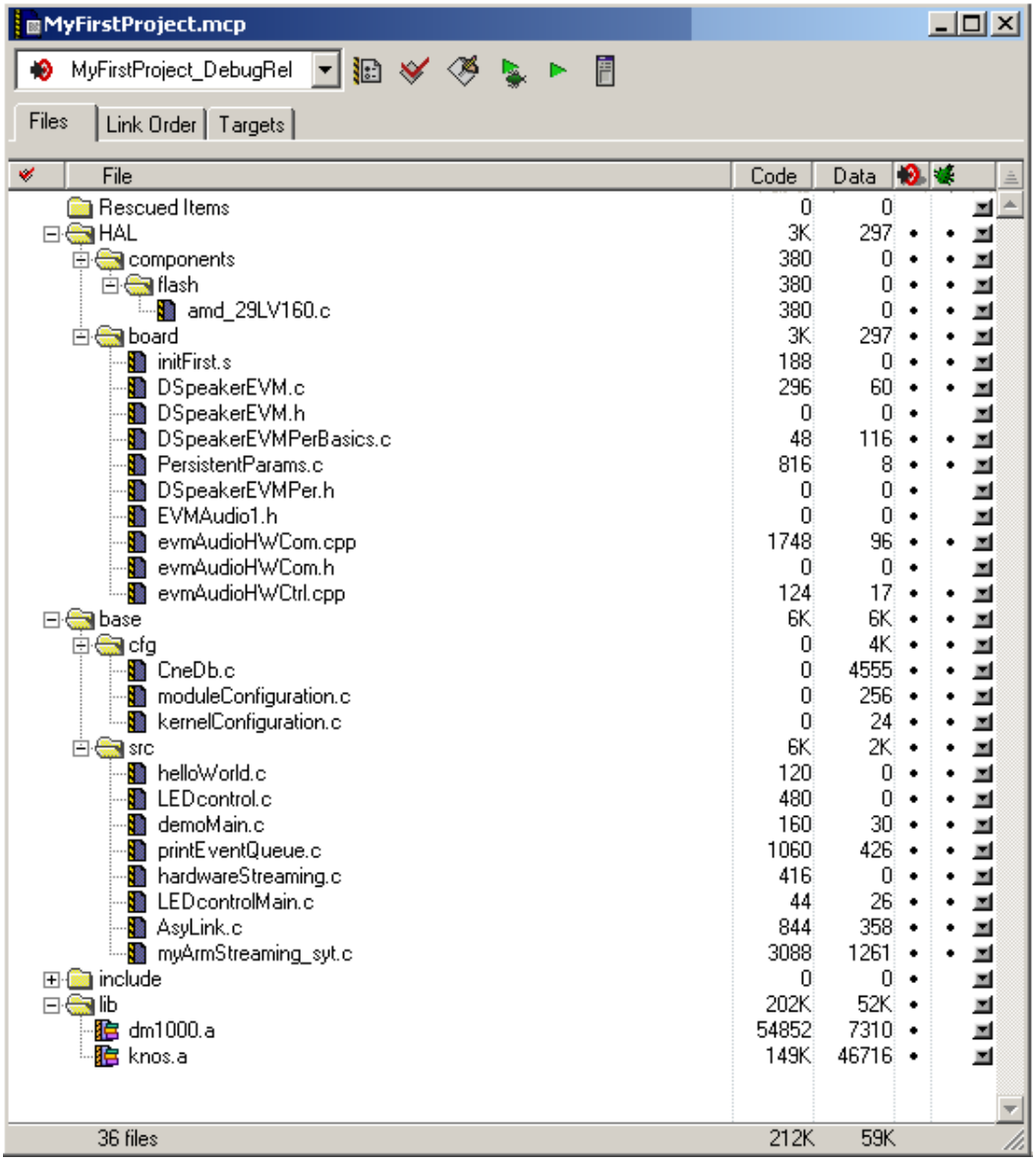


Abbildung 5.1: Beispiel eines Code Warrior-Softwareprojekts.

5.1.3 Debugging

Der AXD erlaubt das Debugging von Code direkt auf der Zielhardware, die über FireWire mit dem PC verbunden wird. Dazu sind einige Installationsmassnahmen notwendig, die in den Unterlagen von BridgeCo beschrieben sind. Diese Debugging-Methode hat sich für meine Zwecke nicht als geeignet erwiesen, da sie die schrittweise Programmausführung nur auf Assemblerebene ermöglicht. Um dies Nutzen zu können, wären tiefere Kenntnisse über die Kernel & OS-Funktionen nötig, die aber nur als Bibliotheken zur Verfügung stehen.
Statt dessen hat es sich als zweckmässig erwiesen, mittels der *printf* Funktion im C-Code Fehler- und Statusmeldungen über die serielle Schnittstelle an den PC zu schicken.

5.2 Gebrauch des 1394 Kernel & OS Software-Pakets

Im Code der Applikation muss die Headerdatei *1394kernel.h* eingefügt werden. Beim Kompilieren werden die Bibliotheken *knos.a* und *dm1000.a* mit eingebunden. Eine Applikation kann ein oder mehrere Softwaremodule umfassen. Ein Modul enthält eine Initialisierungsfunktion, welche zu Beginn der Laufzeit vom Betriebssystem ausgeführt wird. Diese Funktion nimmt die notwendigen Initialisierungen vor, zum Beispiel Speicherallokation oder Start eines Threads. Ausserdem muss das Modul registriert werden. Dafür ist die API-Funktion *registerSystemModule()* vorgesehen. Schliesslich muss das Modul in der Datei *moduleConfiguration.c* eingetragen werden. Unten ist einfaches Beispiel dazu abgedruckt. Weitere Informationen sind in [3] und [4] zu finden.

```
...
OS_Thread helloWorldThread;

// module initialisation function
void myApplicationInit(void) {

     unsigned int err;
     unsigned long entryInput;

     entryInput = 0; // parameter for the thread

     err = osNewThread(&myApplicationThread, "myApplicationThread",
          myApplicationMain, entryInput, OS_AUTO_START);

     if (err == OS_SUCCESS) {
          // everything is ok
     }
     else {
```

```
          // not ok
     }
}

void myApplicationMain(unsigned long entryInput) {

     // application code

}

// module registration
registerSystemModule(myApplicationInit, OS_MOD_USER, OS_IN_USE,
     myApplicationModule);
```

Kapitel 6

Testapplikationen

In diesem Kapitel sind die wichtigsten Programme beschrieben, die ich implementiert habe im Laufe der Semesterarbeit. Das Ziel war, für die Evaluation Boards eine Firmware zu entwickeln, die es erlaubt, eine mehrkanalige Audiostrecke mit zwei oder auch mehreren IEEE 1394 Netzwerkknoten aufzubauen. Dazu versuchte ich auf zwei Arten, im DM1000 interne isochrone Links zwischen den AV Schnittstellentreibern und FireWire aufzusetzen. Einerseits in der Weise, dass die Datenpakete direkt zwischen den AV-Treibern und dem (De)Framer weitergereicht werden, ohne dass die Software auf dem ARM damit belastet wird. Diese Methode wird *hardware-streaming* genannt. Andererseits wurde von der Möglichkeit Gebrauch gemacht, die Datenpakete durch den ARM-Prozessor zu schleusen, wo sie von einem Paketverarbeitungsthread verarbeitet werden (*arm-streaming*). Die zwei Implementierungen sind in den Abschnitten *myHardwareStreaming* und *myArmStreaming* beschrieben, siehe auch Abbildung 6.1.

Beide Implementierungen (*myHardwareStreaming.c* und *myArmStreaming.c*) sind brauchbar sowohl für eine Teststrecke, wie sie in der Aufgabenstellung meiner Arbeit beschrieben ist, als auch für eine Applikation, die STUDER für die Vernetzung eines Studios vorsieht. Im Falle des Audiostudios müssen sehr viele Kanäle zur Verfügung stehen und die Möglichkeit, Datenpakete im DM1000 zu verarbeiten, ist von grossem Nutzen. Der Grund dafür ist, dass die Audiokanäle im Betrieb unter Last geroutet werden (*patching*), wobei die Signale kurz aus- und eingeblendet werden müssen. Dies ist möglich mit ARM-Streaming. Andererseits kann mit der Methode des Hardware Streamings vermutlich[1] eine grössere Anzahl Kanäle verwaltet werden, ohne dass der Prozessor überlastet wird.

Mit beiden Implementierungsvarianten gelang es mir, eine Audioverbindung in Stereo herzustellen zwischen zwei Evaluation Boards. Versuche, eine zweite Verbindung aufzubauen, schlugen fehl. Gegen Ende der Semesterarbeit kam ich zum

[1] Diese Vermutung stützt sich auf Überlegungen, welche die DM1000 Hardware in Betracht ziehen. Es gibt keine Bestätigung von BridgeCo oder durch eigene Tests.

Schluss, dass das Problem wahrscheinlich mit den *Connection Management*-Funktionen zu lösen sei. Diese erst stellen Verbindungen zwischen den IEEE 1394 Netzwerkknoten her und stellen sicher, dass die isochronen Kanäle korrekt verwaltet werden. Leider reichte mir dann die Zeit nicht mehr, Versuche in dieser Richtung anzustellen.

Im Abschnitt *Shell-Kommandos* wird gezeigt, wie die Funktionen so in die DM1000-Software eingebaut werden, dass sie mit Kommandos über ein serielles Terminal gesteuert werden können. In *LEDcontrol* ist ein kleines Programm beschrieben, welches zeigt, wie man die Kommunikationsschnittstellen des DM1000 (in diesem Beispiel SPI) ansteuert. Es wurde als eines der ersten Programme geschrieben und diente als Einstieg in die Programmierpraxis mit dem DM1000 Kernel & OS. Für die Audioteststrecke ist es unerheblich. Der letzte Abschnitt (*AsyLink*) zeigt einen unfertigen Versuch, asynchrone Datentransfers über FireWire auszuführen. Solche Transaktionen eignen sich sehr gut, um Steuerbefehle neben den Audiodaten zu übertragen. Zum Beispiel könnten alle Netzwerkknoten von einer Stelle aus konfiguriert und das Routing der Audiokanäle über das Netz vorgenommen werden.

6.1 Shell-Kommandos

Hier wird am Beispiel des untenstehenden Code-Fragments gezeigt, wie eine Funktion über ein Shell-Kommando aufgerufen wird.
Während der Initialisierung des Softwaremoduls wird die Funktion *shellAddLinkCommands()* ausgeführt. Dabei fügt das Betriebssystem setupLink, closeLink, startLink und closeLink in die Liste der zulässigen Befehle ein. *CSetupLink* heisst die C-Funktion, die aufgerufen wird, sobald "setupLink" empfangen wird. Das dritte Argument von *osShellAddCommand* ist ein String, der Hinweise zum Gebrauch des Befehls macht.

```
void shellAddLinkCommands(void)
{
  osShellAddCommand( "setupLink", CSetupLink, "src dest
    samplefreq iso-channel");
  osShellAddCommand( "closeLink", CCloseLink, "closeLink
     (link name)");
  osShellAddCommand( "startLink", CStartLink, "startLink
     (link name)");
  osShellAddCommand( "stopLink", CStopLink, "stopLink (
    link name)");
}
```

Argumente werden den Funktionen als Strings übergeben. Das Array *argv[]* enthält Pointer auf die Zeichenketten und *argc* gibt die Anzahl der Argumente an.

```
void CSetupLink(char *argv[],int argc){
```

```
    // function code
}
```

6.2 LEDcontrol

LEDcontrol.c ist ein kurzes Beispielprogramm, welches das SPI des DM1000 ansteuert. Auf den Evaluation Boards geht die Datenleitung des SPI auf ein Schieberegister mit parallelen Ausgängen. Daran sind 5 LEDs angeschlossen. Diese können also mit dem seriellen Bitstrom aktiviert und deaktiviert werden. Der Quellcode ist im Anhang B.1 abgedruckt.

6.3 myHardwareStreaming

Mit *myHardwareStreaming.c* können maximal acht interne isochrone Verbindungen im DM1000 (siehe Abbildung 6.1) aufgebaut werden. Dazu stehen die Kommandos setupLink, startLink, stopLink und closeLink zur Verfügung. Der C-Code ist im Anhang B.2 aufgeführt.

- **setupLink** eröffnet eine Verbindung mit *isoLinkOpen()*, welche mit den übergebenen Parametern konfiguriert wird. Der erste, source, gibt die Quelle der Audiodaten an. Mögliche Werte sind entweder 1 bis 16, entsprechend den Audio-Interfaces des DM1000, auf dem Evaluation Board zum Beispiel 6 für den Line Eingang, oder der Wert 1394, der für das FireWire Interface steht. An zweiter Stelle steht der Parameter destination, welcher die Datensenke angibt, mit der gleichen Werteauswahl wie source. Weiter muss die Samplingfrequenz angegeben werden. Sie kann $44.1kHz$ oder $48kHz$ betragen. Das letzte Argument ist die Nummmer des gewählten isochronen Kanals (0..63).
- **startLink** startet das Streaming über die angegebene Verbindung.
- **stopLink** stoppt das Streaming über die angegebene Verbindung.
- **closeLink** schliesst die Verbindung und gibt die allozierten Ressourcen frei.

6.4 myArmStreaming

MyArmStreaming.c ist speziell dafür vorgesehen, zwei interne isochrone Links aufzusetzen, wobei die Datenpakete durch den ARM geschleust werden (Abbildung 6.1). Dort können sie im Datenverarbeitungsthread mit der Funktion *armStreamDataProcess()* verarbeitet werden (siehe Anhang B.3). Die Verbindungen haben in dieser Implementierung feste Quellen und Senken: Link 1 steht für ein Streaming vom FireWire Interface zum Line Ausgang 1, Link 2 vom Line Eingang zu FireWire.
Die Bedienung erfolgt über die gleichen Kommandos wie bei *myHardwareStreaming.c*. Als Argument ist bei allen Befehlen jeweils 1 oder 2, entsprechend Link

1 oder 2, anzugeben. Für setupLink sind noch die Parameter Samplingfrequenz in *kHz* (44.1 oder 48) sowie die Nummer des isochronen Kanals notwendig.

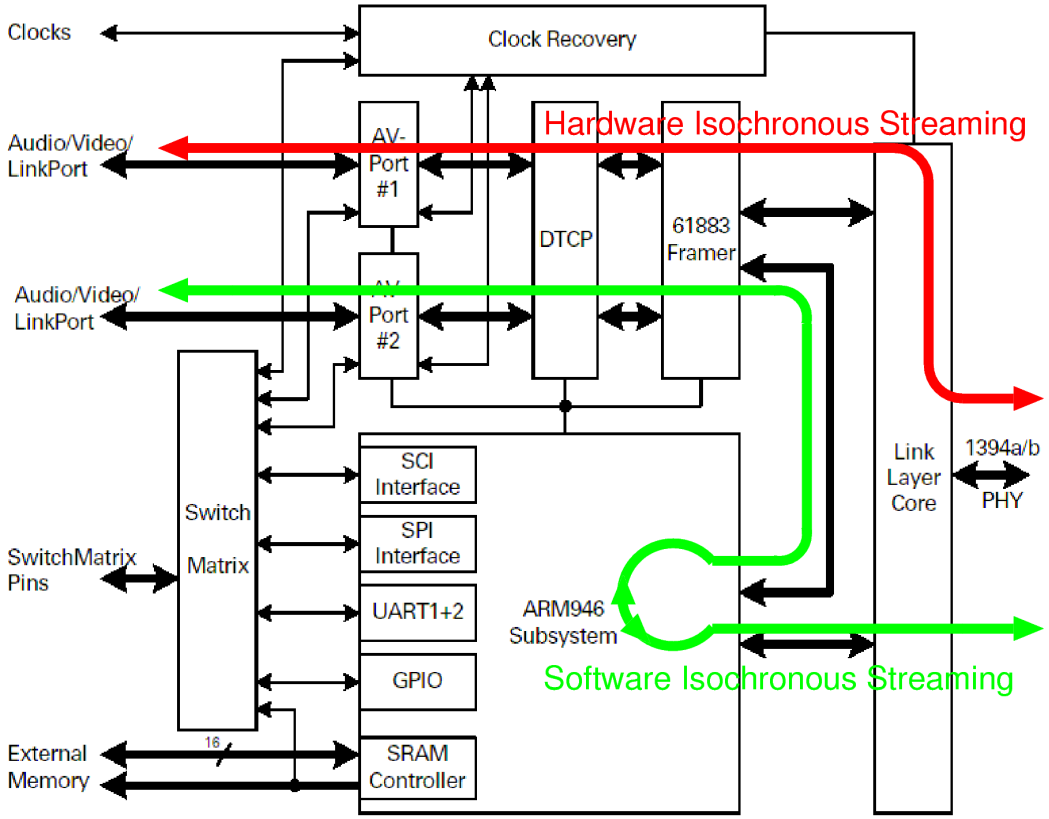


Abbildung 6.1: Illustration des Hardware Streamings (oben) bzw. Software (ARM-)Streamings (unten): Datenpaketfluss innerhalb des DM1000 zwischen AV-Interfaces und IEEE 1394.

6.5 AsyLink

Das unfertige Programm *AsyLink.c* (Anhang B.4) zeigt die ersten Schritte, welche erforderlich sind, um asynchrone Transfers zu ermöglichen. So ist zum Beispiel das Eröffnen einer Message-Queue und die Allozierung eines Block Pools implementiert. Bei der Ausführung der Funktion *bmRegisterAddressRange()* erfolgte jeweils ein Systemabsturz, ohne dass eine Ursache dafür ersichtlich war. So blieb das Programm in unvollendetem Zustand.

Kapitel 7

RS232-Kontrollerboard

Das RS232-Kontrollerboard ermöglicht es, über zwei serielle Schnittstellen Kommandos abzuschicken, ohne dazu einen PC mit Terminal-Programm und zwei COM-Ports zu benötigen. Beispielsweise können direkt zwei Evaluation-Boards von BridgeCo angeschlossen und mit dem Kontrollerboard gesteuert werden. Per Tastendruck sind bis zu acht Kommandos abrufbar, die im Flash-Memory des Microcontrollers gespeichert sind.

7.1 Hardware

Die Schaltung wurde auf einer Lochstreifen-Experimentierplatine aufgebaut. Versorgt wird sie nominell mit 5V, ist aber dafür ausgelegt, auch mit 3.3V zu funktionieren. So kann sie beispielsweise vom BridgeCo-Evaluation Board gespiesen werden. Der zentrale Baustein ist ein 8-bit Microcontroller von Microchip (PIC16F876). Er enhält einen Flash Programmspeicher, der über einen seriellen Port "in circuit" programmierbar ist. Von den zahlreichen integrierten Peripheriemodulen wird vor allem die UART [1] genutzt.
Mit zwei OR-Gattern kann eines der beiden RS232-Interfaces aktiviert werden. Der Baustein MAX3235E von *maxim* wandelt die CMOS-Logikpegel in RS232-konforme Spannungen um. Die LEDs dienen als Statusanzeigen (send/receive). Im Anhang A.1 ist der Schaltplan zu finden.

7.2 Firmware

Der Sourcecode, geschrieben in Assembler, ist im Anhang B.5 abgedruckt. Die Funktionen sind in den folgenden Abschnitten kurz beschrieben.

[1] Universal Asynchronous Receiver and Transmitter

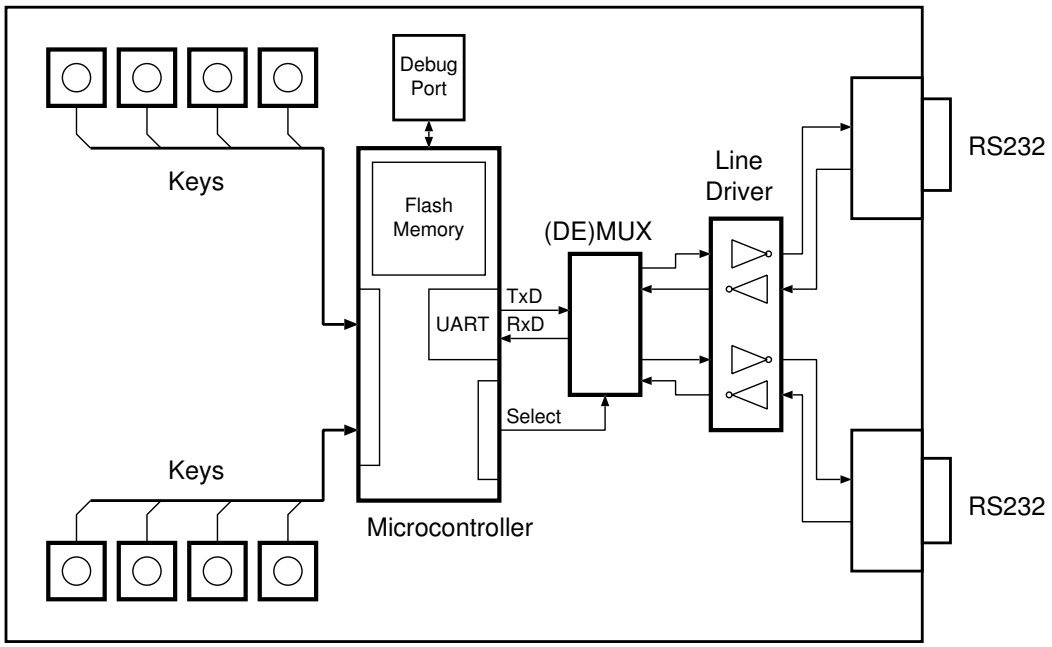


Abbildung 7.1: Blockschaltbild des RS232-Kontrollerboards.

7.2.1 main

Nach einem Reset werden die Instruktionen unter *main* ausgeführt. Die *Special Function Registers* werden initialisiert und damit die Hardwaremodule des Microcontrollers.

7.2.2 poll

Die Schlaufe *poll* wird nach der Initialisierung ohne Abbruch ausgeführt. Dabei wird zyklisch der Zustand aller Tasten abgefragt. Ist eine Taste gedrückt, wird das zugehörige Kommando (cmd0..7) ausgeführt.

7.2.3 cmd0..7

Hier wird zuerst die Funktion *getmsg* aufgerufen, welche die gewünschte Zeichenkette (z. B. für cmd0: MSG0) in den Buffer schreibt. Dann wird das Transmitter-Interface ausgewählt und *sendmsg* ausgeführt. Nach einer Verzögerung kehrt der Programmablauf zur *poll*- Schleife zurück.

7.2.4 getmsg

Diese Funktion holt das Zeichen an der Stelle table_offset aus dem Speicherbereich *table*, schreibt es an die Stelle Buffer[table_offset] und inkrementiert den Index ta-

ble_offset, um danach das nächste Byte zu holen. Die Schleife wird abgebrochen, sobald das Zeichen den Wert 0 aufweist.

7.2.5 sendmsg

Byteweise wird die Zeichenkette Buffer über die asynchrone Schnittstelle gesendet, bis ein Zeichen den Wert 0 hat. Das Senden eines Buchstabens wird ausgelöst, indem er ins Register *TXREG* geschrieben wird. Eine Verzögerung von $10ms$ wurde eingefügt, damit der Eingangsbuffer des Empfängers (in diesem Fall das Evaluation Board) sicher nicht überfüllt wird.

7.2.6 delay und bigdelay

Diese Verzögerungsfunktionen fragen den Stand des Timers *TMR0* ab. Erreichbare Delays mit dieser Konfiguration sind $277\mu s$ bis $71ms$ mit *delay* und $277\mu s$ bis ca. $18s$ mit *bigdelay*.

7.2.7 table

Hier wird der Wert table_offset direkt zum Programmzähler addiert. Die darauf ausgeführte Instruktion ist ein *return*, wobei der zugehörige Buchstabe der Zeichenkette im Arbeitsregister W abgelegt ist.

Zusammenfassung und Ausblick

Zusammenfassung

In der Zeit meiner Semesterarbeit hatte ich die Möglichkeit, wertvolle Erfahrungen zu machen mit einer relativ neuen, viel versprechenden Netzwerktechnologie und, auf der anderen Seite, mit der Industrie und der Zusammenarbeit zwischen den zwei Firmen STUDER Professional Audio AG und BridgeCo.

Die Einarbeitung in die Materie, die das Studium des Standards IEEE 1394 unter besonderer Berücksichtigung von Anwendungen im Audiobereich sowie das Kennenlernen der Hardware und Entwicklungssoftware umfasste, nahm mehr Zeit in Anspruch als vermutet. Trotzdem gelang es mir, einen Teil der gesteckten Ziele zu erreichen. Mit eigenen Programmen konnte ich eine einkanalige Audiostrecke zwischen zwei Evaluation Boards aufbauen und damit zwei Arten von isochronen Datenübertragungen im DM1000 demonstrieren. Zudem erlaubt eine eigens entwickelte Hardware, die zwei Boards bequem zu konfigurieren.
Im Laufe der Semesterarbeit wurde ein Punkt der Aufgabenstellung, die Ausarbeitung des Schemas eines Netzwerkknotens für STUDER, zurückgestellt, denn es war vorauszusehen, dass die Herstellung des PCBs nicht bis zum Ende meiner Arbeit möglich sein würde. Es gelang mir auch nicht mehr, eine mehrkanalige Audiostrecke aufzubauen mit selbst geschriebener Software. Ein Grund dafür war, dass die Zeit knapp wurde, ein anderer, dass es sich als schwierig herausstellte, von BridgeCo die benötigten Informationen zu bekommen. So wurde beispielsweise ein Treffen zwischen einem Programmierer von BridgeCo und mir vom Management kurzfristig gekündigt. Stattdessen ergab sich ein Meeting der höheren Etagen, womit mir nicht geholfen war. Ich denke, eine engere Zusammenarbeit der beiden Firmen wäre notwendig und hätte meine Arbeit stark vereinfacht.

Ausblick

Zu Beginn der Arbeit war ich überzeugt, dass es im Hinblick auf zukünftige Applikationen von STUDER am besten sei, Software zu entwickeln, die möglichst wenig Rechenleistung des Prozessors und Speicher braucht. So versuchte ich, die Audioverbindungen mit elementaren API-Funktionen herzustellen, ohne die bestehende Implementierung des AV/C Stacks und der Interfaces zu verwenden.

Für zukünftige Entwicklungen wäre es notwendig, zusammen mit BridgeCo zu evaluieren, welcher Weg einzuschlagen ist. Ausserdem ist nicht klar geworden, ob der DM1000 den Anforderungen von STUDER gerecht wird. Einerseits wurden keine Tests unter den von STUDER erwarteten Betriebsbedingungen durchgeführt, andererseits lassen die Zahlen von BridgeCo keine genauen Abschätzungen zu.

Literaturverzeichnis

[1] Johannes Schuster Harald Bögeholz. Firewire prescht vor. *c't*, 2003.

[2] BridgeCo AG. *Data Sheet DM1000 IEEE 1394a/b Interface Processor*, 2002.

[3] BridgeCo AG. *Programmer's Manual: DM1000 1394 Kernel & OS*, 2002.

[4] BridgeCo AG. *1394 Kernel & OS API Reference(HTML)*, 2002.

[5] IEC 61883-1. Consumer audio/video equipment - digital interface - part 1: General, 1 2003.

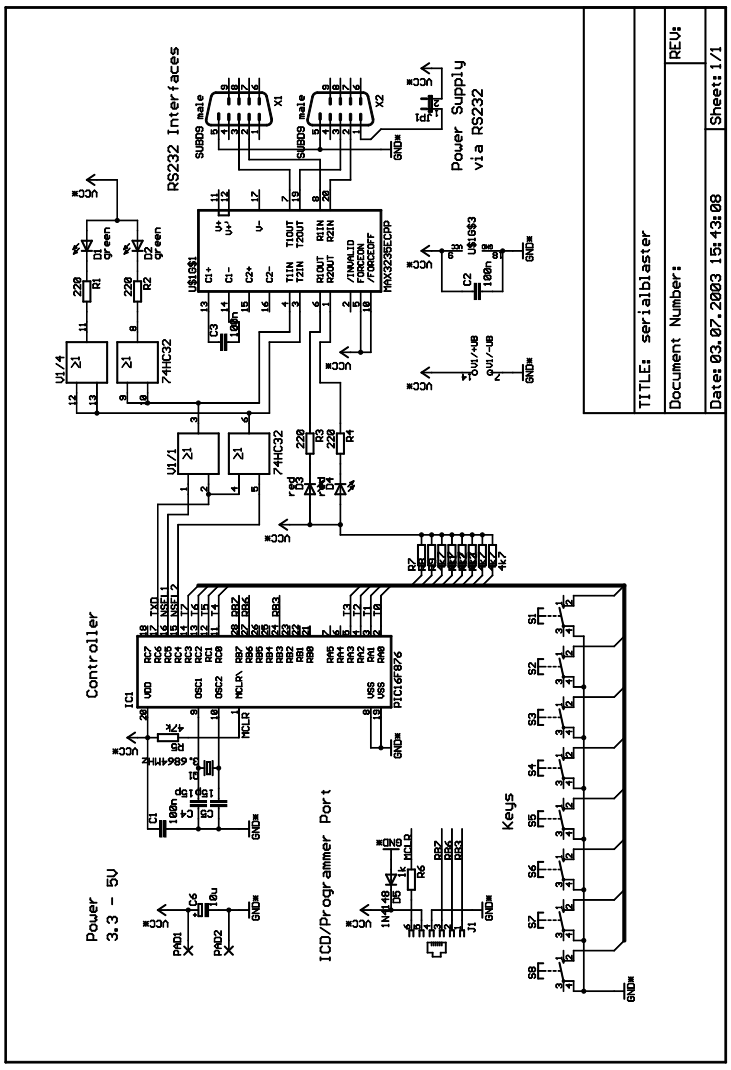
[6] BridgeCo AG. *Hardware/Software Manual: DM1000 EVM Audio1 Board*, 2002.

[7] Bob Moses Rob Laubscher and Richard Foss. A 1394-based architecture for professional audio production. Technical report, Audio Engineering Society (AES), 2000.

Anhang A

Hardware

A.1 Schema des RS232-Kontrollers



A.2 Bauteileliste

Menge	Wert	Beschreibung	Bauteilnummer
1	PIC16F876	Microcontroller	IC1
1	74HC32	Logikbaustein	V1
1	MAX3235E	RS232 Treiber	U$1
1	$3.6864MHz$	Quarz	Q1
2	grün	LED $3mm$	D1, D2
2	rot	LED $3mm$	D3, D4
1	1N4148	Diode	D5
4	220Ω	Widerstand	R1, R2, R3, R4
1	$47k\Omega$	Widerstand	R5
1	$1k\Omega$	Widerstand	R6
8	$4.7k\Omega$	Widerstand	R7, R8, R9, R10, R11, R12, R13, R14
3	$100nF$	Kondensator	C1, C2, C3
2	$15pF$	Kondensator	C4, C5
1	$10\mu F$	Elko	C6
8		Taster	S1, S2, S3, S4, S5, S6, S7, S8
1		Jumper	JP1
1	6-polig	Modularbuchse	J1
2	9-polig	SubD Stecker	X1, X2

Tabelle A.1: Bauteileliste des RS232-Kontrollers.

Anhang B

Software

B.1 LEDcontrol.c

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/*  LEDcontrol.c                                                      */
/*                                                                    */
/*--------------------------------------------------------------------*/
/* purpose: LEDcontrol() starts the thread "SPIThread" which opens the  */
/*          device spi and sends the argument through the spi.        */
/*          In LEDcontrol(), the string argv[0] is converted to an un-  */
/*          signed long integer.                                      */
/*                                                                    */
/* usage:   shell command LEDcontrol bbbbbbbb (8 bits) where every    */
/*          b is dedicated to one LED on the evaluation board (ref    */
/*          hardware decription). b=0: bright, b=1: dark              */
/*                                                                    */
/* author:  jonas biveroni                                            */
/*--------------------------------------------------------------------*/

#include <stdio.h>
#include <string.h>

#include "1394mem.h"
#include "1394isolink.h"
#include "1394kernel.h"


void spi_thread_entry(unsigned long argument) {

    int spiHandle;
    int err;
    unsigned long* buffer;

    buffer = &argument;

    //open SPI device (1: write only)
    spiHandle = open("/dev/spi", 1);

    if (spiHandle < 0)
        printf("error opening SPI device \n");

    //send buffer
    err = write(spiHandle, buffer, 4);
```

```
    if (err == -1)
        printf("error writing buffer \n");

    printf("wrote %d bytes to SPI \n", err);

    //close device
    close(spiHandle);

    return;

} //spi_thread_entry



void LEDcontrol(char* argv[], int argc) {

    OS_Thread spi_thread;
    STATUS status;
    unsigned long entryInput = 0; //parameter for spi_thread_entry()
    unsigned long byte = 0;
    unsigned long one = 1;
    int i = 0;

    for (i=0;i<8;i++) {  //build a byte from input string
        if (*argv[0] == '1')
            byte = byte + (one << (7-i));
        if (*argv[0] == '\0') break;
        argv[0]++;
        }
    byte = byte + (byte << 8);
    byte = byte + (byte << 16);

    entryInput = byte;
    printf("sending entryInput = %X \n", (int)entryInput);

    //create new thread with entry function spi_thread_entry()
    status = osNewThread( &spi_thread, "SPIThread", &spi_thread_entry, entryInput, OS_AUTO_START);

    if (status != OS_SUCCESS) {
      printf("error from osNewThread(), error Code: %d\n", status);
      return;
      }

    // wait 5 secs before stopping
    //osThreadSleep(5000);
    printf("SPIThread is going to stop now \n");

    // kill spi_thread
    osFreeThread(&spi_thread);

    return;

} //LEDcontrol



/************************************************/
/* LED module initialization and registration   */
/************************************************/
```

```
//LED module initialization function
void LEDcontrolMain(void) {

  osShellAddCommand("LED", LEDcontrol, "usage: LED 10011100");

}

//LED module registration
registerSystemModule( LEDcontrolMain, OS_MOD_USER, OS_IN_USE, LEDModule );
```

B.2 myHardwareStreaming.c

```
/*-------------------------------------------------------------------------*/
/*                                                                         */
/*  myHardwareStreaming.c                                                  */
/*                                                                         */
/*-------------------------------------------------------------------------*/
/* purpose:  setup until 8 isochronous internal links ("hardware        */
/*           streaming")    between AV-interface and 1394 iso-channel   */
/* usage:    call the functions CSetupLink(), CStartLink(), CStopLink() */
/*           and CCloseLink() through the following shell commands.     */
/*                                                                         */
/* setupLink source(1..16ę1394) destination(1..16ę1394) 44.1ę48         */
/*           iso-channel(0..63)                                          */
/*           source/destination(1..16): AV-interfaces                    */
/*                                                                         */
/* startLink link number                                                  */
/*                                                                         */
/* stopLink  link number                                                  */
/*                                                                         */
/* closeLink link number                                                  */
/*                                                                         */
/*                                                                         */
/* author:   jonas biveroni   16/05/2003                                  */
/*-------------------------------------------------------------------------*/

#include "string.h"
#include "1394Kernel.h"

#define MAX_ISO_LINKS 8
#define MAX_DESCR_LENGTH 20

typedef struct{
  ISOConfig *isoConfig;
  char descr[MAX_DESCR_LENGTH];
}ISOLink;

ISOLink *isoLinks[MAX_ISO_LINKS];



void CCloseLink(char *argv[],int argc){

  int index;
  int connNumber;
  int status;

    if (argc < 1){
          printf("breakLink [ConnectionNumber | Description]");
        return;
    }

  index=0;


  if ((connNumber=atoi(argv[0]))!=0){          // if argv[1] is a number

    if (isoLinks[connNumber]!=0){
      if ((status=isoLinkClose(isoLinks[connNumber]->isoConfig))!=ISO_SUCCESS){
        printf("Link %s could not be closed. Error: %d\r\n",isoLinks[connNumber]->descr,status);
        return;
      }else{
```

```
        printf("Link %s successfully closed!\r\n",isoLinks[connNumber]->descr);
      }
      osFree(isoLinks[connNumber]);
      isoLinks[connNumber]=0;
      return;
    }
  }

  // try to take argv[1] as a description

  for (index=1;index<MAX_ISO_LINKS;index++){
    if (strcmp(argv[0],isoLinks[index]->descr)==0){
      if ((status=isoLinkClose(isoLinks[index]->isoConfig))!=ISO_SUCCESS){
        printf("Link %s could not be closed. Error: %d\r\n",isoLinks[index]->descr,status);
        return;
      }else{
        printf("Link %s successfully closed!\r\n",isoLinks[index]->descr);
      }
      osFree(isoLinks[index]);
      isoLinks[index]=0;
      return;
    }
  }

  printf("Connection not found\r\n");
}



void CSetupLink(char *argv[],int argc){

    ISOConfig *isoConfig;
    int status;
    int src,dest;
    int index;
    int source, destination;


    if (argc < 1){
        return;
    }

    //choose the right version! (depending on version of libraries *.a)
    //isoConfig=isoLinkOpen(OS_HEAP);
    isoConfig=isoLinkOpen();


    if (atoi(argv[0]) != 0){        //if argv[0] is a number
        if (argc < 4){
            printf("Not enough argments!\r\n");
            printf("Use: setupLink src(1..16,1394) dest(1..16,1394) 44.1/48 iso-channel\r\n");
            return;
        }
        source = atoi(argv[0]);
        destination = atoi(argv[1]);
        if (source != 1394){
            if (source < 1 || source > 16){
                printf("wrong source number\r\n");
                return;
            }
        }
        if (destination != 1394){
```

```
            if (destination < 1 || destination > 16){
                printf("wrong destination number\r\n");
                return;
            }
        }
        if (source == 1394){
            src = LINK1394;
        }
        else{
            src = AV;
            isoConfig->interfaces[0] = source;
        }
        if (destination == 1394){
            dest = LINK1394;
        }
        else{
            dest = AV;
            isoConfig->interfaces[0] = destination;
        }

        if (strcmp(argv[2],"44.1") == 0){        //sampling frequency = 44.1k
            isoConfig->fdf = FREQ_44_1;
        }else if (strcmp(argv[2],"48") == 0){    //sampling frequency = 48k
            isoConfig->fdf = FREQ_48;
        }else{
            printf("wrong sampling frequency\r\n");
            return;
        }


        isoConfig->isoChannel = atoi(argv[3]);
        if ((isoConfig->isoChannel < 0) || (isoConfig->isoChannel > 63)){
            printf("invalid iso channel number\r\n");
            return;
        }


        isoConfig->fmt = FMT_AUDIO;

        status = isoLinkSetup(src, dest, isoConfig);    //setup the link
        if (status == ISO_SUCCESS){
            for (index=1; index < MAX_ISO_LINKS; index++){
                if (isoLinks[index] == 0){        //free place found
                    isoLinks[index] = (ISOLink*)osMalloc(sizeof(ISOLink), OS_HEAP);
                    isoLinks[index]->isoConfig = isoConfig;
                    sprintf(isoLinks[index]->descr,"Link%d",index);//copy string "Link%d" to ->descr

                    printf("Link successfully setup\r\n");
                    printf("Link number: %d\r\n", index);
                    //printf("Link description: %s\r\n", isoLinks[index]->descr);
                    printf("Link %d -> %d, interfaces[0]= %d\r\n",src,dest,isoConfig->interfaces[0]);
                    return;
                }
            }
        }else{
            printf("avDriver error: %d\r\n", status);
        }
    }
}


void CStartLink(char *argv[],int argc){
```

```
  int index;
  int connNumber;
  int status;

    if (argc < 1){
          printf("startLink [ConnectionNumber | Description]\r\n");
        return;
    }

  index=0;


  if ((connNumber=atoi(argv[0]))!=0){          // if argv[1] is a number

    if (isoLinks[connNumber]!=0){
      if ((status=isoLinkStart(isoLinks[connNumber]->isoConfig))!=ISO_SUCCESS){
        printf("Link %s could not be started. Error: %d\r\n",isoLinks[connNumber]->descr,status);
        return;
      }else{
        printf("Link %s successfully started!\r\n",isoLinks[connNumber]->descr);
        return;
      }
      return;
    }
  }

  // try to take argv[1] as a description

  for (index=1;index<MAX_ISO_LINKS;index++){
    if (strcmp(argv[0],isoLinks[index]->descr)==0){
      if ((status=isoLinkStart(isoLinks[index]->isoConfig))!=ISO_SUCCESS){
        printf("Link %s could not be started. Error: %d\r\n",isoLinks[index]->descr,status);
        return;
      }else{
        printf("Link %s successfully started!\r\n",isoLinks[index]->descr);
        return;
      }
    }
  }

  printf("Connection not found\r\n");

}


void CStopLink(char *argv[],int argc){

  int index;
  int connNumber;
  int status;

    if (argc < 1){
          printf("stopLink [ConnectionNumber | Description]\r\n");
        return;
    }

  index=0;


  if ((connNumber=atoi(argv[0]))!=0){          // if argv[1] is a number
```

```
    if (isoLinks[connNumber]!=0){
      if ((status=isoLinkStop(isoLinks[connNumber]->isoConfig))!=ISO_SUCCESS){
        printf("Link %s could not be stoped. Error: %d\r\n",isoLinks[connNumber]->descr,status);
        return;
      }else{
        printf("Link %s successfully stoped!\r\n",isoLinks[connNumber]->descr);
        return;
      }
      return;
    }
  }

  // try to take argv[1] as a description

  for (index=1;index<MAX_ISO_LINKS;index++){
    if (strcmp(argv[0],isoLinks[index]->descr)==0){
      if ((status=isoLinkStop(isoLinks[index]->isoConfig))!=ISO_SUCCESS){
        printf("Link %s could not be stoped. Error: %d\r\n",isoLinks[index]->descr,status);
        return;
      }else{
        printf("Link %s successfully stoped!\r\n",isoLinks[index]->descr);
        return;
      }
    }
  }

  printf("Connection not found\r\n");

}


void shellAddLinkCommands(void)
{
  osShellAddCommand( "setupLink", CSetupLink, "src dest samplefreq iso-channel");
  osShellAddCommand( "closeLink", CCloseLink, "closeLink (link name)");
  osShellAddCommand( "startLink", CStartLink, "startLink (link name)");
  osShellAddCommand( "stopLink", CStopLink, "stopLink (link name)");

}


void isoLinkShellInit(){
    int index;

    for (index=0;index < MAX_ISO_LINKS;index++)
      isoLinks[index]=0;

    shellAddLinkCommands();
}

registerSystemModule( isoLinkShellInit,OS_MOD_AVDRIVER , OS_IN_USE, isoLinkShellCommands );
```

B.3 myArmStreaming.c

```
/*--------------------------------------------------------------------------*/
/*                                                                          */
/*  myArmStreaming.c                                                        */
/*                                                                          */
/*--------------------------------------------------------------------------*/
/* purpose: setup 2 isochronous streams through the ARM                     */
/*          Link 1: LINK1394->ARM->AV interface 1 (= LineOut 1);            */
/*          Link 2: AV interface 6 (= LineIn) ->ARM->LINK1394               */
/*                                                                          */
/* usage:   setupLink  1|2, 44.1/48, isoChannel                             */
/*          closeLink  1|2                                                  */
/*          startLink  1|2                                                  */
/*          stopLink   1|2                                                  */
/*                                                                          */
/* author:  jonas biveroni   06/06/2003                                     */
/*--------------------------------------------------------------------------*/

#include "string.h"
#include <stdlib.h>
#include <stdio.h>
#include "1394mem.h"
#include "1394isolink.h"


ISOConfig *streamFromARMtoAV;
ISOConfig *streamFromLinktoARM;
ISOConfig *streamFromARMtoLink;
ISOConfig *streamFromAVtoARM;
IsoStreamBlockPool* pool = NULL;
static OS_Thread dataProcessThread1, dataProcessThread2;
int status;


void armStreamDataProcess1(unsigned long argument)
{

IsoStreamBlock* packet;

    while(1)
    {
        packet = isoStreamRead(streamFromLinktoARM);


        // Process data here

        isoStreamWrite(streamFromARMtoAV, packet);
    }
}

void armStreamDataProcess2(unsigned long argument)
{

IsoStreamBlock* packet;

    while(1)
    {
        packet = isoStreamRead(streamFromAVtoARM);


        // Process data here
```

```
        isoStreamWrite(streamFromARMtoLink, packet);
    }
}

/*********************************************************************
*
* setupLink [1:1394->AV | 2:AV->1394], sample frequency, isoChannel
*
*********************************************************************/


void CSetupLink(char* argv[], int argc){

    unsigned int numBlocks, blockSize, queueSize;

    blockSize = 100;    // 100 bytes for each block
    numBlocks = 8;      // allocate eight blocks to read data in
    queueSize = 8;      // each queue element corresponds to a block of data

    if (argc < 3) {
            printf("usage: setupLink [1:1394->AV|2:AV->1394],sample frequency,isoChannel\r\n");
            return;
    }

    if (atoi(argv[0]) != 1 && atoi(argv[0]) != 2) {
        printf("first argument invalid\r\n");
        return;
    }

    if (atoi(argv[0]) == 1) {        //setup Link 1 from 1394 to AV

            // open input link
        streamFromLinktoARM = isoLinkOpen();

        // create data block used for reading data
        streamFromLinktoARM->pStreamPool=isoLinkAllocStreamBlockPool(blockSize,numBlocks,OS_AHB_NO_CACHE);
        if( streamFromLinktoARM->pStreamPool == 0 ) {
            printf("osBlockPoolCreate failed!\r\n");
            isoLinkClose(streamFromLinktoARM);
            return;
        }

        // set link related values
        streamFromLinktoARM->isoChannel     = atoi(argv[2]);// stream from this channel to ARM
        streamFromLinktoARM->memType        = OS_TCMD;      // read queue is allocated from this memory
        streamFromLinktoARM->queueElements  = queueSize;

        streamFromLinktoARM->syncMode        = CSP;         // syncMode CSP
        streamFromLinktoARM->fmt            = FMT_AUDIO;
           if (strcmp(argv[1],"44.1") == 0) {
            streamFromLinktoARM->fdf = FREQ_44_1; }         // audio frequency 44.1
        else
            streamFromLinktoARM->fdf = FREQ_48;             // audio frequency 48

        // open output link
        streamFromARMtoAV = isoLinkOpen();

        streamFromARMtoAV->isoChannel       = atoi(argv[2])+1;// stream out of ARM on this channel
        streamFromARMtoAV->interfaces[0]    = 1;            // audio is played out on AV1 interface 1
        streamFromARMtoAV->fmt              = FMT_AUDIO;    // audio stream
```

```
    if (strcmp(argv[1],"44.1") == 0) {
        streamFromARMtoAV->fdf = FREQ_44_1; }           // audio frequency 44.1
    else
        streamFromARMtoAV->fdf = FREQ_48;               // audio frequency 48
    streamFromARMtoAV->memType          = OS_TCMD;      // memory type
    streamFromARMtoAV->queueElements    = queueSize;
    streamFromARMtoAV->syncMode            = CSP;       // syncMode CSP

    // setup output stream
    status = isoLinkSetup(ARM, AV, streamFromARMtoAV);

    if (status != 0){
        printf("Test failed, error from isoLinkSetup(streamFromARMtoAV),error Code: %d",status);
        return;
    }

    // setup input stream
    status = isoLinkSetup(LINK1394, ARM, streamFromLinktoARM);

    if (status != 0){
        printf("Test failed, error from isoLinkSetup(streamFromARMtoAV),error Code: %d",status);
        pool = streamFromLinktoARM->pStreamPool;
        isoLinkClose(streamFromARMtoAV);
        isoLinkFreeStreamBlockPool(pool);
        return;
    }

    // create the worker thread
    if ((status = osNewThread(&dataProcessThread1, "DataProcessing", &armStreamDataProcess1,
                              NULL, OS_AUTO_START)) != OS_SUCCESS){
        pool = streamFromLinktoARM->pStreamPool;
        isoLinkClose(streamFromARMtoAV);
        isoLinkClose(streamFromLinktoARM);
        isoLinkFreeStreamBlockPool(pool);
        printf("Test failed, error from osNewThread(), error Code: %d\n", status);
        return;
    }

}    //setup Link 1


if (atoi(argv[0]) == 2) {        //setup Link 2 from AV to 1394

        // open input link
    streamFromAVtoARM = isoLinkOpen();

    // create data block used for reading data
    streamFromAVtoARM->pStreamPool=isoLinkAllocStreamBlockPool(blockSize,numBlocks,OS_AHB_NO_CACHE);
    if( streamFromAVtoARM->pStreamPool == 0 ) {
        printf("osBlockPoolCreate failed!\r\n");
        isoLinkClose(streamFromAVtoARM);
        return;
    }

    // set link related values
    streamFromAVtoARM->isoChannel     = atoi(argv[2])+1;// stream from AV to ARM on this channel
    streamFromAVtoARM->memType        = OS_TCMD;        // read queue is allocated from this memory
    streamFromAVtoARM->queueElements  = queueSize;
    streamFromAVtoARM->interfaces[0]  = 6;              // audio is recorded from AV1 interface 6
    streamFromAVtoARM->fmt            = FMT_AUDIO;      // audio stream
    if (strcmp(argv[1],"44.1") == 0) {
        streamFromAVtoARM->fdf = FREQ_44_1; }           // audio frequency 44.1
```

```
        else
            streamFromAVtoARM->fdf = FREQ_48;               // audio frequency 48

        streamFromAVtoARM->syncMode            = CSP        // syncMode CSP

        // open output link
        streamFromARMtoLink = isoLinkOpen();

        streamFromARMtoLink->isoChannel       = atoi(argv[2]);// stream out of ARM to this 1394 channel
        streamFromARMtoLink->memType          = OS_TCMD;
        streamFromARMtoLink->queueElements    = queueSize;

        streamFromARMtoLink->syncMode          = CSP;         // syncMode CSP
        streamFromARMtoLink->fmt            = FMT_AUDIO;      // audio stream
        if (strcmp(argv[1],"44.1") == 0) {
            streamFromARMtoLink->fdf = FREQ_44_1; }           // audio frequency 44.1
        else
            streamFromARMtoLink->fdf = FREQ_48;               // audio frequency 48

        // setup output stream
        status = isoLinkSetup(ARM, LINK1394, streamFromARMtoLink);

        if (status != 0){
            printf("Test failed, error from isoLinkSetup(streamFromARMtoLink),error Code: %d",status);
            return;
        }

        // setup input stream
        status = isoLinkSetup(AV, ARM, streamFromAVtoARM);

        if (status != 0){
            printf("Test failed, error from isoLinkSetup(streamFromAVtoARM),error Code: %d",status);
            pool = streamFromAVtoARM->pStreamPool;
            isoLinkClose(streamFromARMtoLink);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // create the worker thread
        if ((status = osNewThread(&dataProcessThread2, "DataProcessing", &armStreamDataProcess2,
                                  NULL, OS_AUTO_START)) != OS_SUCCESS){
            pool = streamFromAVtoARM->pStreamPool;
            isoLinkClose(streamFromARMtoLink);
            isoLinkClose(streamFromAVtoARM);
            isoLinkFreeStreamBlockPool(pool);
            printf("Test failed, error from osNewThread(), error Code: %d\n", status);
            return;
        }

    }    //setup Link 2

}    //CSetupLink


/*********************************************************************
*
* startLink  1:1394->AV | 2:AV->1394
*
*********************************************************************/
```

```
void CStartLink(char* argv[], int argc) {

    if (argc != 1) {
        printf("usage: startLink 1 | 2\r\n");
        return;
    }

    if (atoi(argv[0]) != 1 && atoi(argv[0]) != 2) {
        printf("argument invalid\r\n");
        return;
    }


    if (atoi(argv[0]) == 1) {    //Link 1: 1394->ARM->AV

        // start input stream
        status = isoLinkStart(streamFromLinktoARM);

        if (status != 0){
            printf("Test failed, error from isoLinkStart(streamFromLinktoARM),error Code: %d",status);
            pool = streamFromLinktoARM->pStreamPool;
            osFreeThread(&dataProcessThread1);
            isoLinkClose(streamFromARMtoAV);
            isoLinkClose(streamFromLinktoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // start output stream
        status = isoLinkStart(streamFromARMtoAV);

        if (status != 0){
            printf("Test failed, error from isoLinkStart(streamFromARMtoAV),error Code: %d",status);
            pool = streamFromLinktoARM->pStreamPool;
            osFreeThread(&dataProcessThread1);
            isoLinkClose(streamFromARMtoAV);
            isoLinkClose(streamFromLinktoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

    }    //start Link 1

    if (atoi(argv[0]) == 2) {    //Link 2: AV->ARM->1394

        // start input stream
        status = isoLinkStart(streamFromAVtoARM);

        if (status != 0){
            printf("Test failed, error from isoLinkStart(streamFromAVtoARM),error Code: %d",status);
            pool = streamFromAVtoARM->pStreamPool;
            osFreeThread(&dataProcessThread2);
            isoLinkClose(streamFromARMtoLink);
            isoLinkClose(streamFromAVtoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // start output stream
        status = isoLinkStart(streamFromARMtoLink);

        if (status != 0){
```

```
            printf("Test failed, error from isoLinkStart(streamFromARMtoLink),error Code: %d",status);
            pool = streamFromAVtoARM->pStreamPool;
            osFreeThread(&dataProcessThread2);
            isoLinkClose(streamFromARMtoLink);
            isoLinkClose(streamFromAVtoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

    }        //start Link 2

}    //CStartLink


/*********************************************************************
*
* stopLink  1:1394->AV | 2:AV->1394
*
*********************************************************************/

void CStopLink(char* argv[], int argc) {

    if (argc != 1) {
        printf("usage: stopLink 1 | 2\r\n");
        return;
    }

    if (atoi(argv[0]) != 1 && atoi(argv[0]) != 2) {
        printf("argument invalid\r\n");
        return;
    }

    if (atoi(argv[0]) == 1) {    //Link 1: 1394->ARM->AV

        // stop output stream
        status = isoLinkStop(streamFromARMtoAV);

        if(status != ISO_SUCCESS) {
            printf("Error from isoLinkStop(streamFromARMtoAV), error Code: %d", status);
            pool = streamFromLinktoARM->pStreamPool;
            osFreeThread(&dataProcessThread1);
            isoLinkClose(streamFromARMtoAV);
            isoLinkClose(streamFromLinktoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // stop intput stream
        status = isoLinkStop(streamFromLinktoARM);

        if(status != ISO_SUCCESS) {
            printf("Error from isoLinkStop(streamFromLinktoARM), error Code: %d", status);
            pool = streamFromLinktoARM->pStreamPool;
            osFreeThread(&dataProcessThread1);
            isoLinkClose(streamFromARMtoAV);
            isoLinkClose(streamFromLinktoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

    }    //stop Link 1
```

```
    if (atoi(argv[0]) ==2) {    //Link 2: AV->ARM->1394

         // stop output stream
        status = isoLinkStop(streamFromARMtoLink);

        if(status != ISO_SUCCESS) {
            printf("Error from isoLinkStop(streamFromARMtoLink), error Code: %d", status);
            pool = streamFromAVtoARM->pStreamPool;
            osFreeThread(&dataProcessThread2);
            isoLinkClose(streamFromARMtoLink);
            isoLinkClose(streamFromAVtoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // stop input stream
        status = isoLinkStop(streamFromAVtoARM);

        if(status != ISO_SUCCESS) {
            printf("Error from isoLinkStop(streamFromAVtoARM), error Code: %d", status);
            pool = streamFromAVtoARM->pStreamPool;
            osFreeThread(&dataProcessThread2);
            isoLinkClose(streamFromARMtoLink);
            isoLinkClose(streamFromAVtoARM);
            isoLinkFreeStreamBlockPool(pool);
            return;
           }
       }    //stop Link 2

}    //CStopLink



/*************************************************************************
*
* closeLink  1:1394->AV | 2:AV->1394
*
*************************************************************************/

void CCloseLink(char* argv[], int argc) {

    if (argc != 1) {
        printf("usage: closeLink 1 | 2\r\n");
        return;
    }

    if (atoi(argv[0]) != 1 && atoi(argv[0]) != 2) {
        printf("argument invalid\r\n");
        return;
    }


    if (atoi(argv[0]) == 1) {    //Link 1: 1394->ARM->AV

        // kill data processing thread
        osFreeThread(&dataProcessThread1);

        // save the block pool location before closing the link
        pool = streamFromLinktoARM->pStreamPool;

        // close output stream
        status = isoLinkClose(streamFromARMtoAV);
```

```
        if (status != 0) {
            printf("Test failed, error from isoLinkClose(streamFromARMtoAV),error Code: %d",status);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // close input stream
        status = isoLinkClose(streamFromLinktoARM);
        if (status != 0) {
            printf("Test failed, error from isoLinkClose(streamFromLinktoARM),error Code: %d",status);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // release memory block
        isoLinkFreeStreamBlockPool(pool);

        return;

    }    //close Link 1

    if (atoi(argv[0]) == 2) {    //Link 2: AV->ARM->1394

        // kill data processing thread
        osFreeThread(&dataProcessThread2);

        // save the block pool location before closing the link
        pool = streamFromAVtoARM->pStreamPool;

        // close output stream
        status = isoLinkClose(streamFromARMtoLink);
        if (status != 0) {
            printf("Test failed, error from isoLinkClose(streamFromARMtoLink),error Code: %d",status);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // close input stream
        status = isoLinkClose(streamFromAVtoARM);
        if (status != 0) {
            printf("Test failed, error from isoLinkClose(streamFromAVtoARM),error Code: %d",status);
            isoLinkFreeStreamBlockPool(pool);
            return;
        }

        // release memory block
        isoLinkFreeStreamBlockPool(pool);

        return;

    }    //close Link 2

}    //CCloseLink



/************************************************************
*
* add shell commands, register module
*
************************************************************/
```

```
void shellAddLinkCommands(void)
{
  osShellAddCommand( "setupLink", CSetupLink, "setupLink (1|2, 44.1/48, isoChannel)");
  osShellAddCommand( "closeLink", CCloseLink, "closeLink (1|2)");
  osShellAddCommand( "startLink", CStartLink, "startLink (1|2)");
  osShellAddCommand( "stopLink", CStopLink, "stopLink (1|2)");
}


void isoLinkShellInit(){

    shellAddLinkCommands();
}

registerSystemModule( isoLinkShellInit,OS_MOD_AVDRIVER , OS_IN_USE, isoLinkShellCommands );
```

B.4 AsyLink.c

```
/*------------------------------------------------------------------------*/
/*                                                                        */
/*  AsyLink.c (under construction)                                        */
/*                                                                        */
/*------------------------------------------------------------------------*/
/* purpose:    starts a thread which receives asynchronous requests.     */
/*             needed elements: message queue, block pool.               */
/*                                                                        */
/* problem: bmRegisterAddressRange causes the operating system to crash.*/
/*            reason unknown                                              */
/*                                                                        */
/* author:     jonas biveroni                                             */
/*------------------------------------------------------------------------*/

#include "1394kernel.h"

OS_Queue *asyRequestQueue;
EventMessage *message;
OS_BlockPool *blockPool;
BmQuadletWriteRequest *writeRequest;

unsigned short addr1Hi, addr2Hi; //high order bits of lower and upper memory range
                                 //(see bmRegisterAddressRange)
unsigned int addr1Lo, addr2Lo;   //low order bits of lower and upper memory range
                                 //(see bmRegisterAddressRange)

unsigned int *regAddressRangeHandle;

OS_Thread asyReceiveThread;


void asyLinkInit( void ) {

int status;
int bmStatus;

    //allocate memory for queue control block
    asyRequestQueue = (OS_Queue *)osMalloc(sizeof(OS_Queue), OS_HEAP);
    if (asyRequestQueue == NULL) {
        printf("Mem alloc error while creating eventMessage in asyLinkInit module\n");
        return;
    }

    //allocate memory for 4 EventMessages
    message = (EventMessage *)osMalloc( 4 * sizeof(EventMessage), OS_HEAP);
    if (message == NULL) {
        printf("Mem alloc error while creating message in asyLinkInit module\n");
        osFree((void*)asyRequestQueue);
        return;
    }

    //create asyRequestQueue
    status = osQueueCreate(asyRequestQueue, "asyRequestQueue", OS_4_ULONG, message,
                           4 * sizeof(EventMessage));
    if (status != OS_SUCCESS) {
        printf("memory error creating message in asyLinkInit module\n");
        osFree((void*)asyRequestQueue);
        osFree(message);
        return;
    }
```

```
    //allocate memory for block pool control
    blockPool = (OS_BlockPool *)osMalloc(sizeof(OS_BlockPool), OS_HEAP);
    if (blockPool == NULL) {
        printf("Malloc error while creating blockPool in AsyLinkInit module\n");
        return;
    }

    //allocate memory for block pool
    writeRequest = (BmQuadletWriteRequest *)osMalloc(4 * sizeof(BmQuadletWriteRequest), OS_HEAP);
    if (writeRequest == NULL) {
        printf("Malloc error while creating block pool in AsyLink module init\n");
        osFree((void*)blockPool);
        return;
    }

    //create block pool
    status = osBlockPoolCreate(blockPool, "blockPool", sizeof(BmQuadletWriteRequest), writeRequest,
                               4 * sizeof(BmQuadletWriteRequest));
    if (status != OS_SUCCESS) {
        printf("Error creating block pool in asyLink module init\n");
        osFree((void*)blockPool);
        osFree(writeRequest);
        return;
    }

    waitUs(100000);

    //get address range of block pool
    /*addr1Lo = (unsigned int)writeRequest;
    addr1Hi = (unsigned short)(((unsigned long)writeRequest) >> 32);
    addr2Lo = (unsigned int)(addr1Lo + 4 * sizeof(BmQuadletWriteRequest));
    addr2Hi = (unsigned short)
              (((unsigned long)writeRequest + 4 * sizeof(BmQuadletWriteRequest)) >> 32);*/

    addr1Hi = 0;
    addr2Hi = 0;
    addr1Lo = 0;
    addr2Lo = 10;

    printf("Address Range: 0x%X 0x%X --> 0x%X 0x%X\n", addr1Hi, addr1Lo, addr2Hi, addr2Lo);

    waitUs(100000);

    //register address range to bus manager
    bmStatus = bmRegisterAddressRange(addr1Hi, addr2Hi, addr1Lo, addr2Lo, blockPool,
                                      asyRequestQueue, regAddressRangeHandle);

    waitUs(100000);

    if (bmStatus == BM_STATUS_1394_SUCCESS)
        printf("Successfully registered address range.\n");
    else printf("Address range registration failed\n");

    waitUs(100000);

    return;

}


registerSystemModule( asyLinkInit, OS_MOD_USER, OS_IN_USE, asyLink );
```

B.5 serial.asm

```
;*********************************************************************
;                                                                    *
;    Filename:     serial.asm                                       *
;    Date:          22.06.2003                                       *
;    File Version:  1                                                *
;                                                                    *
;    Author:       Jonas Biveroni                                   *
;    Company:      ETH Zurich                                       *
;                                                                    *
;                                                                    *
;*********************************************************************
;                                                                    *
;    Files required:   none                                          *
;                                                                    *
;                                                                    *
;                                                                    *
;*********************************************************************
;                                                                    *
;    Notes: firmware for RS232-controllerboard                       *
;                                                                    *
;                                                                    *
;                                                                    *
;                                                                    *
;*********************************************************************


list      p=16f876            ; list directive to define processor
#include <p16f876.inc>        ; processor specific variable definitions


;***** VARIABLE DEFINITIONS ***********************************************
w_temp          EQU     0x70
status_temp     EQU     0x71
table_offset EQU 0x20
counter EQU 0x21

;***** CONSTANTS *********************************************************

#define BUFFER 0x22
#define NSel1 PORTC,5 ;select transmitter1 (active low)
#define NSel2 PORTC,4 ;select transmitter2 (active low)
#define Key0 PORTA,0 ;key0
#define Key1 PORTA,1 ;key1
#define Key2 PORTA,2 ;key2
#define Key3 PORTA,3 ;key3
#define Key4 PORTC,0 ;key4
#define Key5 PORTC,1 ;key5
#define Key6 PORTC,2 ;key6
#define Key7 PORTC,3 ;key7


;*********************************************************************
ORG     0x000             ; processor reset vector
nop   ; important for ICD
clrf    PCLATH            ; ensure page bits are cleared
   goto    main              ; go to beginning of program


ORG     0x004             ; interrupt vector location
movwf   w_temp            ; save off current W register contents
```

```
movf STATUS,w          ; move status register into W register
movwf status_temp       ; save off contents of STATUS register

; isr code can go here or be located as a call subroutine elsewhere

movf    status_temp,w     ; retrieve copy of STATUS register
movwf STATUS            ; restore pre-isr STATUS register contents
swapf   w_temp,f
swapf   w_temp,w          ; restore pre-isr W register contents
retfie                    ; return from interrupt

;***** MACROS *******************************************************

bank_0 macro ;bank_0: tmro,pcl,status,fsr,porta,b,eedata,
bcf STATUS,5 ;  eeadr,pclath,intcon, +general reg.
bcf STATUS,6
endm

bank_1 macro ;bank_1: option,pcl,status,fsr,trisa,b,eecon1,
bsf STATUS,5 ;  (eecon2),pclath,intcon, +mapped general reg.
bcf STATUS,6
endm


;*********************************************************************

main
clrf STATUS
movlw BUFFER
movwf FSR ;FSR = BUFFER
movlw b'00000000'
movwf PORTA
movlw b'00110000'
movwf PORTC ;set NSel1 and NSel2
clrf INTCON ;interrupts disabled

bank_1
movlw b'11010111'
movwf OPTION_REG ;pullups disabled, (), tmr0 internally clocked, (), prescaler
;assigned to tmr0, prescaler rate = 1:256
movlw b'00000110'
movwf ADCON1 ;set portA pins digital i/o
movlw b'11111111'
movwf TRISA ;port a: all inputs
movlw b'10001111'
movwf TRISC
movlw 1
movwf SPBRG ;baudrate: 115200 @ 3.6864MHz
movlw b'00100100'
movwf TXSTA ;async. transmitter setup and enabled

bank_0
bsf RCSTA,SPEN ;enable serial port

poll btfss Key0 ;keys polling loop
call cmd0 ;call cmd0 if key0 pressed (=low)
btfss Key1
call cmd1
btfss Key2
call cmd2
btfss Key3
call cmd3
```

```
btfss Key4
call cmd4
btfss Key5
call cmd5
btfss Key6
call cmd6
btfss Key7
call cmd7
goto poll ;end polling loop

;*****************************************************************
;***** FUNCTIONS *************************************************
;*****************************************************************


;***** cmd0..7 ***************************************************
;
; command executed when key pressed.
; arguments: none
;
;*****************************************************************

cmd0 movlw (MSG0 & 0xFF)-4 ;subtract 4 from LSBs of label MSG0
;because of the first four instructions in table
call getmsg ;write MSG1 to buffer
bcf NSel2 ;use transmitter2
call sendmsg ;send message
bsf NSel2
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return

;*****************************************************************
cmd1 movlw (MSG1 & 0xFF)-4
call getmsg ;write MSG1 to buffer
bcf NSel2 ;use transmitter2
call sendmsg ;send message
bsf NSel2
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return

;*****************************************************************
cmd2 movlw (MSG2 & 0xFF)-4
call getmsg ;write MSG2 to buffer
bcf NSel2 ;use transmitter2
call sendmsg ;send message
bsf NSel2
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return

;*****************************************************************
cmd3 movlw (MSG3 & 0xFF)-4
call getmsg ;write MSG3 to buffer
bcf NSel2 ;use transmitter2
```

```
call sendmsg ;send message
bsf NSel2
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return

;*********************************************************************
cmd4 movlw (MSG4 & 0xFF)-4
call getmsg ;write MSG4 to buffer
bcf NSel1 ;use transmitter1
call sendmsg ;send message
bsf NSel1
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return

;*********************************************************************
cmd5 movlw (MSG5 & 0xFF)-4
call getmsg ;write MSG5 to buffer
bcf NSel1 ;use transmitter1
call sendmsg ;send message
bsf NSel1
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return

;*********************************************************************
cmd6 movlw (MSG6 & 0xFF)-4
call getmsg ;write MSG6 to buffer
bcf NSel1 ;use transmitter1
call sendmsg ;send message
bsf NSel1
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return

;*********************************************************************
cmd7 movlw (MSG7 & 0xFF)-4
call getmsg ;write MSG7 to buffer
bcf NSel1 ;use transmitter1
call sendmsg ;send message
bsf NSel1
movlw 0x5
movwf counter
movlw 0x0
call bigdelay ;delay (calculated): ca. 350ms
return


;***** getmsg ***********************************************************
;
; writes message from table to buffer.
; arguments: table_offset in w.
; end of message: 0x0 as last character.
```

```
;
;*********************************************************************

getmsg movwf table_offset
getmsg_loop call table ;returns character at table[table_offset]
movwf INDF ;move character to buffer[n]
xorlw 0 ;w = character xor 0
btfsc STATUS,Z
goto getmsg_exit ;exit, if character = 0
incf FSR ;increment buffer
incf table_offset
goto getmsg_loop

getmsg_exit movlw BUFFER
movwf FSR ;FSR = BUFFER
return

;***** sendmsg ***************************************************
;
; transmits message in buffer over async. interface
; arguments: message stored in buffer.
; end of message: 0x0 as last character.
;
;*********************************************************************

sendmsg btfss PIR1,TXIF
goto sendmsg ;wait until tx-interruptflag is set
movf INDF,W ;get buffer[n]
btfsc STATUS,Z
goto sendmsg_exit ;exit, if character = 0
movwf TXREG ;move buffer[n] to transmit register
incf FSR
movlw 0xDC
call delay ;delay 10ms
goto sendmsg

sendmsg_exit btfss PIR1,TXIF
goto sendmsg_exit ;wait until transmission completed
movlw BUFFER
movwf FSR ;FSR = BUFFER
return

;***** delay *****************************************************
;
; delay: about 4/fosc * 256 * (256 - w) seconds (range @ 3.6864MHz:
;   277us .. 71ms)
; arguments: w
;
;*********************************************************************

delay movwf TMR0
bcf INTCON,T0IF ;clear interrupt flag
delay_loop btfss INTCON,T0IF
goto delay_loop ;wait until interrupt flag is set
return


;***** bigdelay **************************************************
;
; delay: about [(4/fosc * 256 * (256 - w)) * counter] seconds
;   (range @ 3.6864MHz: 277us .. 18s
; arguments: w, counter
```

```
;
;*****************************************************************

bigdelay movwf w_temp
bigdelay_loop movf w_temp,w
call delay
decfsz counter
goto bigdelay_loop
return

;***** table ************************************************
;
; returns character[table_offset] in w.
; arguments: table_offset
;
;*****************************************************************
ORG 0x100 ;table starts at 0x100
table movlw 0x1
movwf PCLATH ;set upper bits of program counter
movf table_offset,W
addwf PCL ,F ;Jump to char pointed to in W reg

MSG0 dt  "play 96 10",0xD,0x0 ;0xD : "enter" (sent by terminal)
MSG1 dt  "record 96 20",0xD,0x0
MSG2 dt  "play stop",0xD,0x0
MSG3 dt  "record stop",0xD,0x0
MSG4 dt  "play 96 20",0xD,0x0
MSG5 dt  "record 96 10",0xD,0x0
MSG6 dt  "play stop",0xD,0x0
MSG7 dt  "record stop",0xD,0x0
table_end dt  0


IF ( (table & 0x0FF) >= (table_end & 0x0FF) )
MESSG   "Warning - User Definded: Table crosses page boundry in computed jump"
ENDIF


END                       ; directive 'end of program'
```