Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

DIPLOMARBEIT

Robuste Funkübertragung für mobile Videoapplikationen

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

O. Univ. Prof. Dipl.-Ing. Dr.techn. Dietmar Dietrich
und
Univ.Ass. Dipl.-Ing. Dr.techn. Stefan Mahlknecht
als verantwortlich mitwirkendem Assistenten am
Institut für Computertechnik
Institutsnummer: 384

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Georg Kohlweiß
Matr.Nr. 9926105
Stuwerstraße 31/34, 1020 Wien

24.01.2007 _____

Abstract

Nowadays there exists a huge variety of applications using divers wireless data transmission techniques. Due to mutual exclusion of technological capabilities, national and international restrictions for radio systems, every application needs to choose its communication technique carefully and with a valuable tradeoff.
This thesis works out a solution for an application with enhanced real-time demands beside the need of enough bitrate for transmission of a video stream. The main focus is to be set on a robust wireless transfer of temporal critical data from several nodes to a dedicated central node, which itself transmits own critical data and an additional video stream. Video data has a lower priority and offers its bitrate to more critical data.
Used wireless technology was developed especially for rough conditions from a company named Nanotron Technologies. A more efficient use of the available data transfer rate can be achieved with MPEG-4-videocompression. To gain the required mobility, a MPEG-4-chip from INTIME Corporation is integrated in the development board for implementation of this complex algorithm. As a core component the BF537 microcontroller of Analog Devices is used. It enables the correct data transfer in a timely manner by the use of its great amount of features and interfaces.
In this theses the procedure of putting the chosen components to operation and the emerging application system is discussed. Therefore the development of driver software is needed that can be used for generating the application software.
Used radio transceiver supports the exact control of any transmit and receive operation, the delays are really short. Furthermore many additional features are supported to save application implementation costs. Therefore this wireless technology is most suitable for the application discussed in this theses.
The initialization of the MPEG-4-chip was pretty much laborious, but however it finally achieved quite good compression results. But due to lacking fault tolerance this chip is not the best choice for MPEG-4-decompression.

Zusammenfassung

Heutzutage gibt es eine breite Palette an Anwendungen mit verschiedendsten kabellosen Datenübertragungsmethoden. Durch sich gegenseitig ausschließende technologische Grenzen und national bzw. international vorgegebene Parameter für Funkübertragungen, müssen vorhandene Techniken für jede Anwendung sorgfältig überprüft und mit viel Kompromißbereitschaft ausgewählt werden.
Diese Arbeit beschäftigt sich mit einer Lösung speziell für Anwendungen, die erhöhte Echtzeitanforderungen an die Funkübertragung stellt, und nebenbei auch eine für Videoübertragungen ausreichende Bitrate bereitstellt. Das Hauptaugenmerk liegt dabei auf dem robusten Transfer von zeitkritischen Daten mehrerer Knoten an einen ausgewählten Zentralknoten, welcher selbst zusätzlich zu eigenen kritischen Daten auch noch einen Videodatenstrom aussendet. Die Videodaten haben dabei eine niedrigere Priorität und stellen ihre Bitrate bei Bedarf den zeitkritischen Daten zur Verfügung.
Als Funktechnologie wird ein speziell für rauhe Umgebungen entwickeltes, robustes System der Firma Nanotron Technologies eingesetzt. Durch MPEG-4-Videokomprimierung kann die zur Verfügung stehende Übertragungsrate effizient genutzt werden. Um die erforderliche Mobilität zu erreichen, wird für die rechenintensive Komprimierung am Entwicklungsboard ein MPEG-4-Chip der Firma INTIME integriert. Kernkomponente des Systems ist jedoch der Mikrocontroller BF537 der Firma Analog Devices. Er ermöglicht den korrekten und zeitgerechten Datenfluss durch eine Vielzahl an Funktionen und Schnittstellen.
In dieser Arbeit werden die ausgewählten Komponenten in Betrieb genommen und zu einem funktionierenden System zusammengesetzt. Notwendig dafür ist die Entwicklung von Treibersoftware, die in der zu erstellenden Anwendungssoftware verwendet wird.
Der eingesetzte Funkchip ermöglicht eine genaue Steuerung jeder Funkaktivität, die Reaktions- und Latenzzeiten sind sehr gering. Darüberhinaus unterstützt er viele benötigte Zusatzfunktionen, die nicht mehr in der Anwendungssoftware implementiert werden müssen. Diese Technologie eignet sich somit hervorragend für die zu entwickelnde Applikation.
Die Inbetriebnahme des MPEG-4-Chips gestaltete sich überaus schwierig, letztendlich zeigte er jedoch sehr gute Kompressionsergebnisse. Für die MPEG-4-Dekodierung ist er aber durch mangelnde Fehlertoleranz nur begrenzt einsetzbar.

Danksagung

Wesentlich zum erfolgreichen Abschluss dieser Arbeit hat meine Freundin Monika durch ihre mentale Unterstützung beigetragen, meine Motivation war dadurch stets gut genährt.

Weiters möchte ich mich bei meinem Betreuer, Dr.techn. Stefan Mahlknecht, bei Dr.techn. Gregor Novak und Dipl.-Ing. Martin Waitz für den fachlichen und technischen Support bedanken.

Inhaltsverzeichnis

Abkürzungen

μC	Mikrocontroller
ACK	Acknowledge
ADPCM	Adaptive Differential Pulse-Code Modulation
ARQ	Automatic Repeat Request
CAN	Control Area Network
CRC	Cyclic Redundancy Check
CSMA	Carrier Sense Multiple Access
CSMA/CA	CSMA / Collision Avoidance
CSS	Chirp Spread Spectrum
DIN	Deutsches Institut für Normung
DMA	Direct Memory Access
DSSS	Direct Sequence Spread Spectrum
DV	Data Valid
DVD	Digital Video Disc, Digital Versatile Disk
DVB	Digital Video Broadcasting
DVB-S	DVB - Satellite
DVB-T	DVB - Terrestrial
EAV	End of Active Video
EDR	Enhanced Data Rate
FCC	Federal Communications Commission
FDMA	Frequency Division Multiple Access
FEC	Forward Error Correction
FIFO	First In First Out
GFSK	Gaussian Frequency Shift Keying
GPIO	General Purpose Input Output
HS	Horizontal Synchronization
ICT	Institut für Computertechnik (an der Technische Universität Wien)
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers

IFS InterFrame Spacing
IO Input Output
IPTV Internet Protocol Television
IRQ Interrupt Request
ISM Industrial Scientific Medical
ISO International Organization for Standardization
ISR Interrupt Service Routine
I^2C Inter-Integrated Circuit
ITU International Telecommunication Union
ITU-R ITU - Radiocommunication Sector
JPEG Joint Photographic Experts Group
JTAG Joint Test Action Group
LAN Local Area Network
LBI Local Bus Interface
LSB Least Significant Bit
MAC Media Access Control
MDMA Memory DMA
MIMO Multiple Input Multiple Output
MPEG Moving Picture Experts Group
MPEG-2 TS MPEG-2 Transport Stream
MSB Most Significant Bit
NTSC National Television Systems Committee
OFDM Orthogonal Frequency Division Multiplex
OSD On Screen Display
OSI Open Systems Interconnection
PAL Phase Alternating Line
PAN Personal Area Network
PC Personal Computer
PPI Parallel Peripheral Interface
RAM Random Access Memory
RSSI Receive Signal Strength Indicator
SAV Start of Active Video
SDRAM Synchronous Dynamic RAM
SIG Bluetooth Special Interest Group
SPI Serial Peripheral Interface
TDMA Time Division Multiple Access
UART Universal Asynchronous Receiver Transmitter
UMTS Universal Mobile Telecommunications System

UWB	Ultra Wide Band
VBI	Vertical Blanking Interval
VS	Vertical Synchronization
WLAN	Wireless LAN
WPAN	Wireless PAN

Kapitel 1

Einführung

Kabellose Datenübertragungstechniken sind heutzutage wegen des großen Vorteils der Mobilität schon sehr weit verbreitet. Es existiert eine Vielzahl an Möglichkeiten, eine Information von A nach B über das uns umgebende Medium Luft zu transportieren.
Die technische Entwicklung von drahtlosen Datenübertragungsmethoden trägt wesentlich zur Akzeptanz vieler Anwendungen bei. Durch ein gewisses Maß an Mobilität werden Applikationen oft überhaupt erst interessant. Weil kein Installationsaufwand notwendig ist, wird auch ein Kabelgewirr vermieden und man ist nebenbei prinzipiell auch nicht ortsgebunden. Natürlich gibt es nicht nur Vorteile. Durch das Benutzen des frei zugänglichen Übertragungsmediums müssen viele Fragen erst gelöst werden bzw. haben in diesem Zusammenhang eine wesentlich höhere Relevanz. Zum Beispiel die der Sicherheit oder die der relativ leicht auftretenden Störungen. Weiters teilen sich unzählige Anwendungen dasselbe Medium, es kann nicht einfach durch ein zusätzliches Kabel eine höhere Verfügbarkeit erzielt werden. Auch regionale gesetzliche Regulierungen schränken letztendlich das Medium auf einen kleinen benützbaren Bereich ein.

Verglichen mit drahtgebundenen Datenübertragungen ist aufgrund der genannten Voraussetzungen die zur Verfügung stehende Bandbreite und damit die Datenübertragungsrate sehr gering. Wesentliche Fortschritte wurden in der Vergangenheit durch die Entwicklung von Techniken erreicht, die eine effiziente Nutzung geringer Bandbreiten ermöglichen. Besonders anschaulich ist hierbei die Entwicklung des WLAN-Standards (IEEE 802.11). Eine höhere Datenübertragungsrate vermindert im Allgemeinen aber wiederum die Störsicherheit und die Reichweite. Vernünftige Kompromisse sind demnach applikationsabhängig. Es gibt keine Technik, die für alle erdenklichen Anwendungen optimal geeignet ist. Die Palette der Übertragungsmethoden ist reichhaltig und bis dato noch keineswegs komplett. Eine Auswahl für eine bestimmte Anwendung ist meistens nicht ganz trivial.

1.1 Applikationsanforderungen

Diese Diplomarbeit beschäftigt sich mit der Realisierung von Anforderungen, die bei speziellen Anwendungen einen Knackpunkt für die Auswahl der zu verwendenden Funktechnologie

darstellen.
Folgende Anforderungen sind dabei zu erfüllen:

- Server-Client-Architektur des Funknetzes mit mehreren Clients
- Verwendung einer digitalen Funktechnologie im 2,4GHz-ISM-Band
- handliche und tragbare Knoten mit Batteriebetrieb
- Übertragung von Echtzeitdaten mit einer möglichst kurzen Verzögerung, in jedem Fall kleiner als 30ms
- Übertragung eines Videos mit ansprechender Qualität vom Server zu einem Client mit einer Verzögerung von maximal 500ms
- der Server führt Video-Live-Streaming und Steuerdatenverarbeitung durch
- ein Video-Client empfängt den Video-Live-Stream und verarbeitet nebenbei auch Steuerdaten
- mehrere Steuer-Clients erledigen nur die Steuerdatenverarbeitung

1.2 Anforderungsanalyse

Zunächst scheinen diese Anforderungen nicht sehr anspruchsvoll zu sein. Bei genauerer Analyse diverser Funktechnologien stellt sich jedoch heraus, dass der Punkt mit den garantierten Verzögerungen kleiner 30ms nicht immer erreichbar ist. Evaluiert man handelsübliche Funkmodule mit WLAN oder Bluetooth-Technik, so beobachtet man zumindest zeitweise höhere Übertragungsverzögerungen (siehe [Wai02]).

Die Anforderung der Übertragung eines Videos verlangt auch eine gewisse Mindestdatenrate. Für ein halbwegs ansprechendes Video im Format 320x240 und 25 Bildern/s würde man unkomprimiert eine Datenrate von ca. 30MBit/s benötigen. Keine derzeit übliche digitale Funktechnik ist in der Lage, diese Anforderung zu erfüllen (zukünftig eventuell UWB).
Der Ausweg liegt in der Videokompression. Durch geeignete Kompressionsalgorithmen wie beispielsweise MPEG-4, können Datenraten von etwa 500kBit/s für das geforderte Video in akzeptabler Qualität erreicht werden. Diese Datenrate ist für die meisten aktuellen Funktechnologien, die auf höhere Datenraten ausgelegt sind, keine besondere Herausforderung mehr.

Die Erfüllung der genannten Auflagen ist durch die erhöhten Verzögerungsanforderungen mit standardmäßigen WLAN oder Bluetooth-Komponenten nicht durchführbar. Als eine besonders geeignete Alternative stellt sich die neu entwickelte Funktechnik der deutschen Firma *Nanotron Technologies* heraus. Eine Evaluierung des angebotenen Funkmoduls meinerseits ergab sehr kurze und konstante Latenzzeiten, also geringen Jitter und eine mögliche Brutto-Datenübertragungsrate von 2MBit/s. Zugunsten einer besseren Störsicherheit als beispielsweise bei WLAN wird auf eine noch höhere Datenrate verzichtet. Diese Übertragungsrate ist jedoch für die gegebene Anwendung durchaus ausreichend.

Eine Echtzeit-Videokompression stellt für eine mobile Applikation mit Batteriebetrieb eine recht schwierige Aufgabe dar. Mit modernen und leistungsfähigen Personalcomputern ist dies recht einfach zu bewerkstelligen. Mit batteriebetriebenen und handlichen Geräten ist es jedoch bedingt durch die nötige hohe Rechenleistung nur schwer durchführbar. Die südkoreanische Firma *INTIME* hat einen eigenständigen Chip entwickelt, der die Komprimierung eines Videos in Echtzeit ermöglicht. Der Chip benötigt lediglich einen externen SDRAM-Baustein und kann somit auch in einem mobilen System eingesetzt werden.

Für eine mobile Applikation wird natürlich auch eine zentrale Steuereinheit benötigt, welche die Kommunikation zwischen allen beteiligten Komponenten übernimmt. Dies wird üblicherweise von einem Mikrocontrollersystem erledigt. Heutzutage sind solche Systeme schon sehr komplex und mit umfangreichen Features ausgestattet. Die Firma *Analog Devices* bietet eine reichhaltige Palette von Mikrocontrollern an, die sich für solche Anwendungen sehr gut eignen. Um eventuelle zukünftige Anforderungen einfacher erfüllen zu können, fiel die Auswahl auf den Typ BF-537, der fast keinen Wunsch offen lässt.

1.3 Überblick

Die Abbildung 1.1 zeigt das Schema des Gesamtsystems aus rein funktioneller Sicht.
Die Server- und Client-Wolken repräsentieren die äußere Schnittstelle zum System. Auf der Serverseite gibt es einerseits ein aufzunehmendes analoges Videosignal und andererseits eine Schnittstelle für die Steuerdaten mit erhöhten Echtzeitanforderungen.
Es gibt zwei unterschiedliche Typen von Clients. Einer davon hat die Aufgabe, das übertragene Video wieder analog auszugeben und nebenbei Steuerungsaufgaben über die genannte Schnittstelle weiterzuleiten. Der andere Typ muss sich um das Video nicht kümmern, die Verarbeitung der Steuerungdaten erfolgt jedoch gleich wie beim ersten Typ.

Die vorliegende Arbeit beschäftigt sich nun mit der Realisierung der genannten speziellen Applikation mithilfe der angeführten Kernkomponenten. Dazu ist in den Kapiteln 2 und 3 die nötige theoretische Basis festgehalten. Die für die bestmögliche Erfüllung der Applikationsanforderungen ausgewählten Komponenten und deren Zusammenwirken zeigt Kapitel 4. Kapitel 5 beschäftigt sich anschließend mit der hardwareseitigen, das Kapitel 7 mit der softwareseitigen Realisierung unter Verwendung der Komponenten-Treiber, die in Kapitel 6 behandelt werden.
Den Abschluß bildet das Kapitel 8 mit den erreichten Ergebnissen und Verbesserungsvorschlägen.

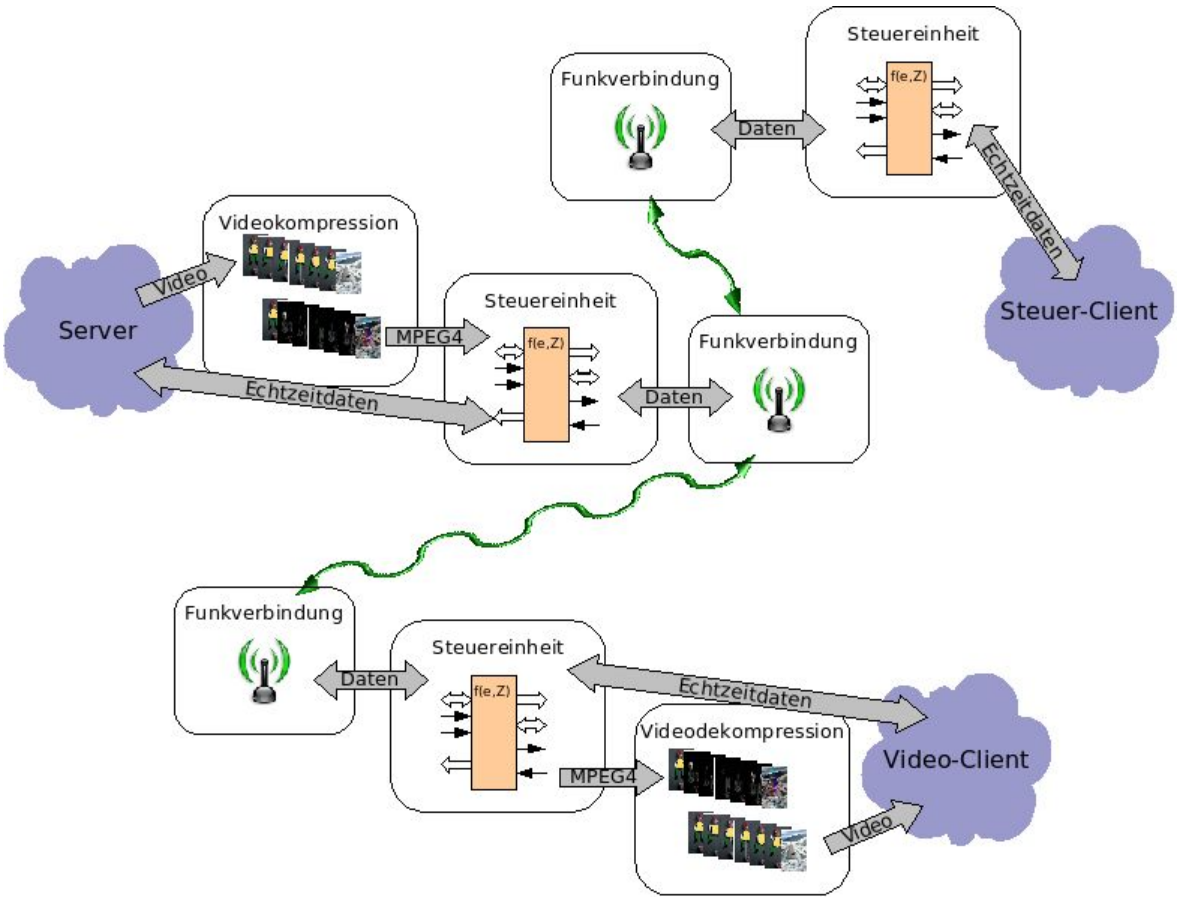

Abbildung 1.1: Funktionelle Systemübersicht

Kapitel 2

Drahtlose Echtzeit-Datenübertragung

Immer mehr Anwendungen verlangen eine Mobilität, wie sie nur durch drahtlose Daten- und Signalübertragungen erreicht werden kann. Ein sehr gutes Beispiel für nahezu grenzenlose Mobilität ist das Mobiltelefon, umgangssprachlich als Handy bezeichnet.
Jede Anwendung hat ganz spezielle Anforderungen bezüglich Mobilität und Datenvolumen. Aufgrund der Tatsache, dass keine optimale Technologie existiert, die alle erdenklichen Anforderungen bestens zu erfüllen in der Lage ist, haben sich viele spezielle Lösungen entwickelt. Viele laufende Forschungen sind auf diesem Gebiet angesiedelt und die Entwicklung neuer Methoden wird wohl noch länger immer bessere Ergebnisse erzielen.

2.1 Anforderungen

Allgemein soll eine digitale Funktechnologie natürlich eine hohe Bitrate bei geringer belegter Frequenzbandbreite übertragen und zusätzlich noch störungsunempfindlich sein. Auch die Latenzzeit soll meist so gering wie möglich sein. Dass es sich dabei mehr um ein Wunschkonzert als um eine technische Lösung handelt, mag auch dem Laien einleuchten. Tatsächlich ist bei der Auswahl der zu verwendenden Funktechnologie immer ein Kompromiss zwischen allen applikationsrelevanten Kennzahlen einzugehen.

Im Folgenden sind einige wichtige Anforderungen an eine digitale Funktechnologie aufgelistet:

- hohe Datenübertragungsrate
- hohe Reichweite Indoor und Outdoor
- geringe Latenzzeit
- geringer Stromverbrauch
- geringe Störungsempfindlichkeit
- gute Echtzeitfähigkeit
- viele Adressierungsarten

2.2 Echtzeitfähigkeit

Eine für diese Arbeit wichtige Anforderung ist die garantierte Übertragung eines bestimmten Datenpaketes innerhalb einer definierten Zeitspanne. Da bestimmte Deadlines einzuhalten sind, wird das System als Echtzeitsystem und im konkreten Fall als "festes Echtzeitsystem" bezeichnet (siehe [WB05, Seite 321]). Liefert ein System ein Ergebnis erst nach Ablauf einer "festen Deadline", hat das Ergebnis keine Relevanz mehr.
Viele Funktechnologien an sich würden zwar eine garantierte Übertragung innerhalb von etwa 30ms durchaus unterstützen, viele verfügbare Module als Implementierung sind dafür aber oft nicht geeignet. Also muss eine Technologie gefunden werden, deren implementierende Module zu normalen Betriebsbedingungen immer in der Lage sind, diese Deadline einzuhalten.

2.3 Bluetooth

Dieser Kurzstreckenfunkstandard wurde von der Bluetooth Special Interest Group (SIG) dafür entwickelt, Geräte über kurze Distanzen drahtlos miteinander zu verbinden. Ein dabei entstehendes Netzwerk wird Wireless Personal Area Network (WPAN) genannt. Speziell relativ einfache, mobile und ressourcenbegrenzte Geräte sollen dadurch unterstützt werden. Von der SIG wurden einige Protokolle entworfen, über die der Datenaustausch zwischen den Geräten durchgeführt wird.
Bluetooth-Geräte bieten Ihre Dienste als sogenannte Profile an, die die konkrete Anwendung bezüglich des Datenaustauschs bereits festlegen. Dadurch werden auch Aufgaben höherer Schichten des OSI-Referenzmodells durch Bluetooth zumindest schon definiert.

Bei Bluetooth kommt ein Frequenzsprungverfahren bei der Wahl der Trägerfrequenz zum Einsatz. Dabei werden die im verwendeten 2,4GHz-ISM-Band zur Verfügung stehenden, 1MHz breiten Kanäle in einem von der Masteradresse abhängigen Muster ständig durchgewechselt. Eine solche Trägerfrequenz wird dabei innerhalb eines Durchgangs nur 625μs lang benützt. Durch die ständigen Sprünge ist Bluetooth relativ unempfindlich gegenüber schmalbandigen Störungen innerhalb des ISM-Bandes. Zur Modulation wird ein einfaches Gaußsches Frequenz-umtastverfahren (GFSK) eingesetzt.

Die Version 1.2 des Bluetooth-Standards unterstützt eine maximale Bruttobitrate von 1MBit/s. Diese Datenrate ist mit einzurechnenden Abstrichen bei Störungen und durch den zusätzlichen Bedarf des Bluetooth-Protokollstacks für die zu bedienende Anwendung etwas zu gering. Der neuere Standard 2.0 würde mittels EDR (Enhanced Data Rate) und 3MBit/s die Anforderungen schon etwas leichter erfüllen können.
Jedoch ist Bluetooth durch den langsamen Verbindungsaufbau und den Overhead, der einen einfachen Datenaustausch ermöglichen soll, eigentlich nicht für Echtzeitsysteme geeignet, wo eine möglichst schnelle Reaktion gefordert ist (siehe dazu auch [Wai02, Kapitel 7 und 8]).

2.4 WLAN

Mit WLAN werden die von der IEEE-Arbeitsgruppe 802.11 spezifizierten drahtlosen Netzwerke, in Anlehnung an das leitungsgebundene Ethernet, bezeichnet. Entwickelt wurde und

wird diese Technologie immer noch mit dem Ziel, das Ethernet mobil zu machen.
Das Hauptaugenmerk liegt also bei WLAN auf der Übertragungsgeschwindigkeit. Der schon etwas ältere 802.11b-Standard erlaubte schon maximal 11MBit/s Bruttodatenrate. 802.11a unterstützte zur gleichen Zeit schon 54MBit/s, dieser Standard operiert aber im 5GHz-Band, das im Gegensatz zum 2,4GHz-Band nicht weltweit freigegeben ist. Mittlerweile ist der Standard 802.11g schon sehr weit verbreitet, er ermöglicht im weltweit lizenzfreien ISM-Band eine Brutto-Übertragungsrate von 54MBit/s. Der Nachfolgestandard 802.11n unter Verwendung einer MIMO-Technik soll sogar bis zu maximal 600MBit/s übertragen können.

Der derzeit weit verbreitete 802.11g-Standard setzt das Modulationsverfahren OFDM (Orthogonal Frequency Division Multiplex) ein (siehe dazu [Ste05, Seite 36]). Dieses Verfahren optimiert die Bandbreitenausnutzung durch den Einsatz mehrerer Subträger, die separat je nach auftretenden Umgebungseffekten amplituden- und phasenmoduliert werden. Aus dieser effizienten Nutzung folgt aber, dass die Übertragung durch schmalbandige Störer viel leichter beeinflusst oder sogar unterbrochen werden kann.
Für die zu entwickelnde Anwendung würden sich zumindest teilweise zu hohe Latenzzeiten im gestörten Umfeld ergeben (siehe [Wai02, Seite 33]).

2.5 ZigBee

Mehrere Funkhardware-Hersteller haben die ZigBee-Allianz gegründet, um eine Möglichkeit der Ad-hoc-Vernetzung im Bereich der Heimautomatisierung zu schaffen. Wichtig dafür sind nicht hohe Datenraten, sondern vielmehr eine hohe Batterielebensdauer von vielen kleinen Sensoren und Aktoren als eigenständige Funkteilnehmer. Ähnlich wie bei Bluetooth werden auch höhere Schichten des OSI-Referenzmodells mit bestimmten Anwendungsprofilen definiert.
ZigBee arbeitet auch im 2.4GHz-ISM-Band und verwendet die DSSS-Spreiztechnik (Direct Sequence Spread Spectrum, siehe [Ste05, Seite 63]). Durch Spreizverfahren wird zwar keine hohe Datenübertragungsrate, dafür aber eine höhere Störsicherheit erreicht. Die maximal erreichbare Datenrate wird mit 250 kBit/s angegeben.

2.6 nanoNET

Eine Funktechnik für einen etwas anderen Anwendungsbereich entwickelt die deutsche Firma *Nanotron Technologies*. Das Ziel dieser Technik ist, auch bei schwierigen Umgebungsbedingungen einen guten Datendurchsatz zu erreichen. Für Industrieanwendungen ist das eine wesentliche Forderung, da dort Störungen oftmals zu den normalen Betriebsbedingungen gehören. Erkauft wird diese Robustheit mit der begrenzten Bruttoübertragungsrate von maximal 2MBit/s und der Belegung des gesamten 2,4GHz-ISM-Bandes. Es steht also nur ein Kanal für Datenübertragungen zur Verfügung. Neuere Funkchip-Versionen dieser Technologie unterstützen jedoch schon drei nicht überlappende Kanäle (siehe [NAL]).

Das verwendete Übertragungsverfahren CSS (Chirp Spread Spectrum) wurde eigens dafür entwickelt und patentiert. Dabei wird eine Frequenzmodulation eingesetzt, die zwei Symbole

benützt. Das erste ist der sogenannte Upchirp, der durch ein monotones Durchstimmen von einer niedrigeren zu einer höheren Frequenz gebildet wird. Der Downchirp repräsentiert die Veränderung von einer höheren zu einer niedrigeren Frequenz hin. Die Symboldauer ist mindestens 1μs, die Symbolrate somit maximal 1MHz.
Wird nun beispielsweise das Upchirp-Symbol einer logischen Eins und der Downchirp einer logischen Null zugeordnet, ist eine Datenübertragungsrate von maximal 1MBit/s zu erreichen. Um auf 2MBit/s zu gelangen, werden nun zwei zusätzliche Symbole benützt, da die Symboldauer zur Bitratensteigerung nicht weiter verringert werden kann. Das dabei eingesetzte dritte Symbol ist ein überlagerter Up- und Downchirp. Als viertes Symbol dient einfach die Abwesenheit aller anderen Symbole. Mit diesen vier Symbolen können innerhalb einer Symboldauer von 1μs gleich zwei Bits übertragen werden.
Mit der so konfigurierbaren Datenrate wird auch die Störungsempfindlichkeit beeinflusst. Zwei unterschiedliche Symbole sind am Empfänger leichter eindeutig zu detektieren als vier, wobei auch noch erschwerend hinzukommt, dass das überlagerte Symbol ja direkt aus den ersten zwei gebildet wird. Bei der Verwendung des 2MBit/s-Modus ist somit unter denselben Bedingungen mit einer höheren Symbolfehlerrate zu rechnen.

Der Hersteller verspricht folgende Fähigkeiten seines Produkts nanoNET TRX:

- maximal 2MBit/s
- Reichweite Outdoor bis zu 900m (@8dBm → 6,31mW)
- Reichweite Indoor bis zu 60m (@8dBm → 6,31mW)
- integrierte Real-Time-Clock
- stromsparender Betrieb, maximal 70mA Verbrauch beim Senden, 30mA im Empfangsmodus, 1,5μA im Standbymodus
- integrierter MAC-Controller ermöglicht CSMA/CA, TDMA, FDMA (bei nanoLOC TRX, siehe [NAL])
- Unicast-, Multicast- und Broadcastaddressierung
- 128-Bit Encryption implementiert
- FEC, 2-stufiger CRC, ARQ

2.6.1 Spezielle Funktionen

Die wichtigsten Features, wovon einige für die zu bearbeitende Anwendung besonders wichtig sind, werden nun kurz beschrieben.

- **Time Division Multiple Access (TDMA)**
  Das ist ein Mediumzugriffsverfahren, das für jede beteiligte Sendestation einen bestimmten Zeitschlitz bereitstellt. Nur eine Station darf innerhalb dieser Zeitspanne das Medium belegen. Somit kann es nicht zu Kollisionen kommen, wenn sich alle Knoten daran

halten. Allerdings ist für ein optimales Ausnutzen der verfügbaren Übertragungsrate dabei einiges an Aufwand nötig. Alle Stationen müssen über dieselbe Zeitbasis verfügen und eine dynamische Zeitschlitzbelegung erfordert einen relativ hohen Managementaufwand.
Wegen der Determiniertheit wird dieser Modus jedoch für Echtzeitsysteme gerne verwendet. nanoNET bringt dafür auch wichtige Voraussetzungen mit.

- **Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA)**
  Bei diesem Zugriffsverfahren werden gelegentliche Kollisionen in Kauf genommen. Eine dynamische und relativ effiziente Nutzung des Mediums kann dann erreicht werden, wenn nicht zu viele Teilnehmer große Datenmengen gleichzeitig übertragen wollen. Kollisionen werden bei CSMA prinzipiell durch Abhören des Mediums vermieden, indem der Sendevorgang erst nach dem Freiwerden des Trägers gestartet wird. Die Erweiterung CA bedeutet, dass nicht gleich danach gesendet, sondern zusätzlich eine zufällige Zeit abgewartet wird. Ist der Träger dann immer noch frei, kann das Medium belegt werden.

- **Virtual Carrier Sensing**
  Dieses Feature gehört zur CSMA/CA-Fähigkeit des nanoNET TRX Transceivers. Das Abhören des Trägers wird dabei sozusagen virtuell durchgeführt. Durch den im Header aller am Medium bisher transportierten Pakete enthaltenen Pakettyp und der Paketlängeninformation kann auf den Zeitpunkt geschlossen werden, zu dem der Träger wieder zur Verfügung stehen müsste.
  Diese Funktion kann optional aktiviert werden, ist aber bei Parallelbetrieb mit anderen Funktechnologien im selben Band natürlich kein Ersatz für eine physikalische Trägerprüfung.

- **Physical Carrier Sensing**
  Physical Carrier Sensing ist auch Teil der CSMA/CA-Fähigkeit. Es kann entweder der Receive Signal Strength Indicator (RSSI), der Bitdetektor oder beides zur Prüfung des Trägers aktiviert werden. Der Bitdetektor erkennt das Auftreten von Chirp-Signalen (Upchirp, Downchirp oder überlager Up- und Downchirp) und ist somit auch an die nanoNET-Technologie gebunden. Der RSSI hingegen überprüft die Empfangsfeldstärke eines am Träger vorhandenen Signals und kann somit unabhängig von der gerade sendenden Funktechnik das Medium als belegt erkennen.

- **Automatic Repeat Request (ARQ)**
  Viele Funkmodule unterstützen diese Funktion schon standardmäßig. Dabei wartet der Sender eine maximale Zeitspanne auf eine Empfangsbestätigung (ACK-Paket). Kommt diese beim Sender nicht an, startet er ganz automatisch einen erneuten Sendeversuch. Dabei kann eine maximale Anzahl an Sendeversuchen voreingestellt werden. Sinnvoll und recht einfach einsetzbar ist dieser Vorgang natürlich nur bei Unicast-Messages, also bei Paketen, die genau einen Empfänger adressieren.

- **Forward Error Correction (FEC)**
  Diese Funktion wird eingesetzt, wenn eine höhere Störungunempfindlichkeit gefordert ist und man dafür Einbußen bei der Übertragungsrate in Kauf nimmt. Hierbei wird einfach Redundanz bei der Datenübertragung eingeführt, um die mit Fehlern versetzten Daten im Empfänger nicht nur detektieren, sondern bei nicht allzu starken Verfälschungen auch korrigieren zu können.

- **Cyclic Redundancy Check (CRC)**
  Das ist ein Verfahren, mit dem eine Erkennung von veränderten Daten möglich ist. Aus den zu beobachtenden Daten wird dabei ein Prüfwert errechnet, der sich bei veränderten Daten mit ändert. Bei gezielter Manipulation kann jedoch trotzdem der gleiche Prüfwert erreicht werden. Dennoch ist dies ein sehr wirkungsvolles Verfahren bei zufällig auftretenden Veränderungen, wie beispielsweise bei Fehlern, die am Übertragungsweg auftreten.
  nanoNET setzt das CRC-Verfahren gleich zweimal ein. Das erste Mal immer zur Überprüfung des Paketheaders und das zweite Mal optional zur Sicherung des gesamten Datenpakets. Zeigt die Überprüfung ein fehlerhaftes Paket an, kann bei Verwendung von ARQ durch das Auslassen des Bestätigungspakets gleich das erneute Senden ausgelöst werden.

- **Broadcast-Adressierung**
  Das ist vorallem bei Funktechnologien eine beliebte und auch recht einfache Möglichkeit, mehrere Empfänger gleichzeitig zu bedienen. Bei einer Broadcast-Adressierung können beliebige Empfänger dasselbe Datenpaket empfangen und bei Multicast-Messages eine bestimmte vordefinierte Gruppe von Empfängern. Beide Varianten werden von nanoNET unterstützt.

- **Encryption**
  Um eine sichere Datenverbindung zu erreichen müssen noch zusätzliche Verfahren eingesetzt werden. nanoNET bietet dafür schon einen Mechanismus in Form einer 128-Bit Verschlüsselung an. Dafür wird allerdings wieder eine gemeinsame Zeitbasis benötigt, da die Verschlüsselung nicht nur von der 48-Bit Knotenadresse und einem 128-Bit Schlüssel abhängt, sondern auch vom 32-Bit Wert in der RTC. Zum Austausch von Zeitinformationen bietet nanoNET allerdings auch schon Möglichkeiten an.

2.6.2 Paketformate

Es gibt unterschiedliche Pakettypen, die für bestimmte Zwecke eingesetzt werden. Zwei dieser Typen, die in Abbildung 2.1 zu sehen sind, werden für die zu entwickelnde Anwendung benötigt.

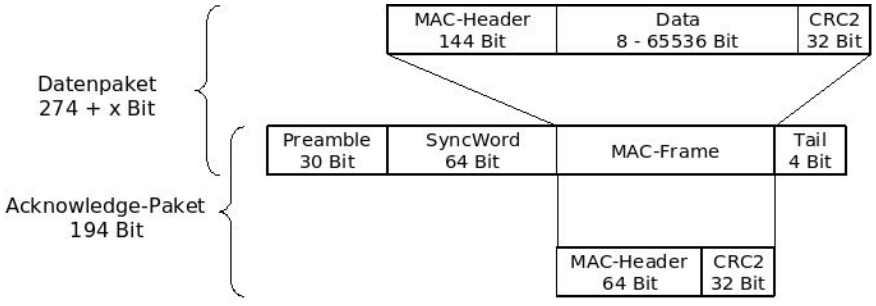

Abbildung 2.1: Daten- und Acknowledge-Pakettyp und deren Größe

Für die Übertragung eines Datenpakets wird sowohl im Broadcast- als auch im Unicast-Modus der Datenpakettyp eingesetzt. Er benötigt 274 Bits plus die zu sendenden Datenbits,

wenn die 32Bit-Variante des zweiten CRC eingesetzt wird. Ein allfälliges Bestätigungspaket im ARQ-Modus ist auch unter Verwendung von 32 CRC-Bits immer 194Bit groß.

Genauere Informationen sind in [NTRb, Chapter 3] zu finden, wo auch noch weitere Pakettypen definiert sind.

2.6.3 SPI-Protokoll

Die Kommunikation mit dem Funkchip nanoNET TRX Transceiver läuft über SPI (Serial Peripheral Interface), eine schnelle serielle Schnittstelle. Sie wird häufig dann zur Anbindung von peripheren Geräten an einen Mikrocontroller verwendet, wenn höhere Bitraten erforderlich sind.
Es gibt immer genau einen Master und prinzipiell beliebig viele Slaves. SPI ist als Ringbuffer organisiert, zur Datenübertragung werden zwei Leitungen benötigt. MOSI (Master Output Slave Input) ist der Ausgang des Masters, der mit dem Eingang aller Slaves verbunden wird. MISO (Master Input Slave Output) ist der Eingang des Masters, der mit den Ausgängen aller Slaves verbunden wird. Zusätzlich gibt es noch eine Taktleitung SCK und pro Slave eine Slave-Select-Leitung SSEL. Wird nur ein Slave benötigt, kann die vierte Leitung je nach angeschlossenem Gerät oft auch eingespart werden.

Die Datenübertragung wird stets vom Master durch das Taktsignal gesteuert. Mit jedem Takt wird je ein Bit vom Master zum Slave über MOSI und ein Bit vom gerade angesprochenen Slave (aktives SSEL) zum Master übertragen. Es gibt also immer einen gleich großen Datenfluss in beide Richtungen, auch wenn teilweise die Daten einer Seite irrelevant sind. Um eine Kommunikation über diese Schnittstelle nun zu ermöglichen ist noch ein vom jeweiligen angeschlossenen Gerät abhängiges Protokoll nötig.

Eine genaue Spezifikation des zu verwendenden Protokolls für den nanoNET TRX Transceiver ist in [NAS] nachzulesen. Hier sei nur eine kurze Zusammenfassung der für diese Arbeit relevanten Aspekte festgehalten.
Zunächst ist das Timing der Signale gemäß [NAS, Abbildung 2] zu konfigurieren und eine maximal zulässige Taktfrequenz von 16MHz auf SCK einzuhalten.

Abbildung 2.2 zeigt die Paketstruktur des zu verwendenden SPI-Protokolls.

Alle Datentransfers werden in Paketen zu je 3 bis 130 Bytes durchgeführt, davon sind die ersten beiden Bytes der Paketheader und der Rest die zu übertragenden Daten. Das MSB des ersten Bytes gibt die Richtung des Datentransfers an. Dabei zeigt 1 einen Schreibtransfer an, und 0 einen Lesetransfer. Die restlichen Bits des ersten Bytes enthalten die binär codierte Anzahl an zu transferierenden Datenbytes. Der Wert 0 (alle sieben Bits 0) spezifiziert dabei die maximale Datenbyteanzahl von 128. Das zweite Header-Byte eines Transfers gibt die Registeradresse des Funkchips an von der gelesen oder auf die geschrieben wird. Werden innerhalb des Transfers mehrere Datenbytes gelesen oder geschrieben, beginnt der Vorgang an der angegebenen Adresse mit dem ersten Datenbyte und jedes weitere Datenbyte korrespondiert mit einer jeweils um eins erhöhten Registeradresse. Somit wird ein zusammenhängender Registerbereich von der spezifizierten Basisadresse aus gelesen oder geschrieben.

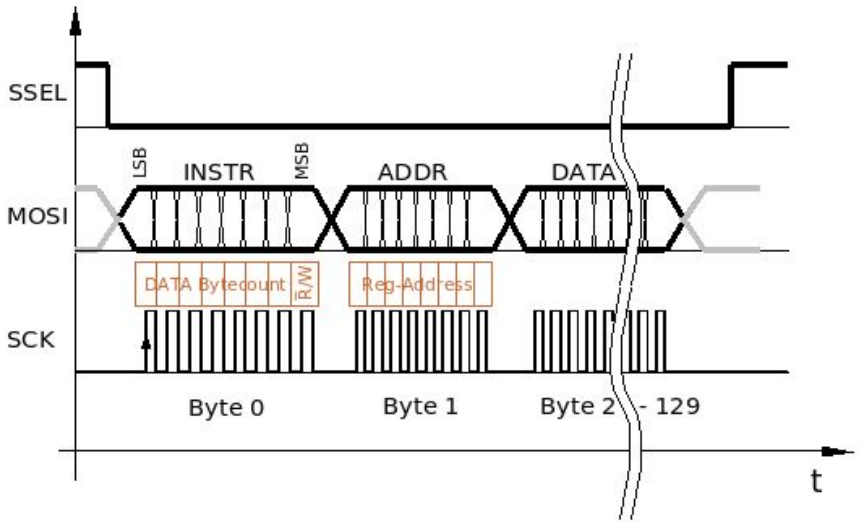

Abbildung 2.2: Paketstruktur des SPI-Protokolls

Bei einem Schreibzugriff werden nur Daten vom Mikrocontroller zum Funkmodul transferiert, die über MISO eingelesenen Bits haben keine Bedeutung und können verworfen werden. Wird ein Lesetransfer gestartet, sind die mit den ersten beiden Headerbytes ausgetauschten Daten aus dem Funkmodul bedeutungslos, ab dem dritten Byte legt dann das Funkmodul die vom zweiten Headerbyte adressierten Daten auf die MISO-Leitung. Der Mikrocontroller kann dabei gemäß Dokumentation einfach den Wert `0xFF` über MOSI an das Funkmodul senden und empfängt dafür die aus den Funkchip-Registern zu lesenden Bytes.

Jede Konfiguration, jedes Kommando und der Transfer der zu sendenden und empfangenen Daten wird so über Registerzugriffe durchgeführt (siehe dazu [NTRa]).

Bei der Treiber- und Applikationsimplementierung ist zu beachten, dass ein Schreib- oder Lesetransfer immer vollständig abgeschlossen werden muss und somit eine atomare Transaktion bildet. Wird ein Transfer beispielsweise durch einen Interrupt unterbrochen und in der ISR ein neuer Transfer gestartet, kommt es zu einer Protokollverletzung, die erst nach einer Inaktivität des SSEL-Signals mit dem nächsten Transfer wieder bereinigt ist. Dabei treten jedoch Datenkorruptionen auf und es können ganze Transfers verloren gehen.

Kapitel 3

Videoverarbeitung und Komprimierung

Digitale Videoanwendungen erfreuen sich zunehmender Beliebtheit. Beispiele dafür sind die sich schon seit längerem im Einsatz befindliche digitale Fernsehübertragung via Satellit DVB-S, oder die erst kürzlich vom Österreichischen Rundfunk gestartete digitale terrestrische Fernsehausstrahlung DVB-T oder aber auch das Fernsehen am Handy mittels IPTV über UMTS. Dazuzuzählen sind an dieser Stelle auch kabelgebundene Lösungen, die beispielsweise zentral gespeicherte Filme über Ethernet zum Fernseher ins Wohnzimmer transportieren.

Möglich ist dieser Fortschritt durch die stetige Entwicklung schnellerer Datenübertragungswege und leistungsfähigerer Prozessoren für die digitale Videoverarbeitung. Als wesentlichster Vorteil gegenüber analoger Videoübertragungen ist wohl der vollständige Erhalt der Videoqualität am Übertragungsweg zu nennen. Liegt das Material einmal in digitaler Form vor, gibt es bei normalen Betriebsbedingungen bis zur Rückwandlung in ein analoges Format keinen Qualitätsverlust mehr. Darüberhinaus kann das Video auch ohne zusätzliche Qualitätseinbußen gespeichert und vervielfältigt werden.
Natürlich gibt es auch Nachteile bei der digitalen Verarbeitung. Die Digitalisierung zu Beginn und die Analogwandlung am Ende sind selbstverständlich verlustbehaftet. Vor allem die Reduzierung der Datenmenge, um überhaupt eine zeitgerechte Übertragung zu ermöglichen, ist alles andere als verlustfrei bezüglich der Videogüte. Das Verhältnis von Videoqualität zu nötiger Übertragungsbandbreite ist bei analogen Übertragungsverfahren wesentlich größer. Dennoch scheinen die Vorteile digitaler Verarbeitung so schwer zu wiegen, dass man einiges an entstehender Anstrengung in Kauf nimmt.

Eine zentrale Komponente bei der digitalen Videoverarbeitung ist die Videokompression. Dabei werden nicht nur verlustfreie, sondern auch verlustbehaftete Verfahren angewendet. Ohne eine dadurch erreichte Datenreduktion könnte ein Video mit einer halbwegs ansprechenden Qualität und den heutzutage zur Verfügung stehenden Datenträgern und Übertragungsraten nicht gehandhabt werden.
Beim digitalen Videostandard ITU-R BT.601, der das europäische PAL-Signal digital beschreibt, ergibt sich eine Bitrate von 216MBit/s (früher bekannt als CCIR 601, siehe [ES98, Seite 2]). Ein solcher Datenfluss von 27MByte/s kann nur von wirklich schnellen Medien wie

zum Beispiel einer modernen Festplatte gelesen werden. Eine Funkübertragung ist bei diesen Datenmengen mit den derzeit üblichen Methoden praktisch unmöglich.

Um einen halbwegs handhabbaren Videodatenstrom zu erreichen, wird defacto in allen digitalen Videoapplikationen eine verlustbehaftete Kompressionstechnik eingesetzt (abgesehen von modernen, professionellen Videoproduktionen, wo kein Qualitätskompromiss zur Debatte steht). Je nach Anforderungen ist für jede Anwendung gesondert ein Kompromiss zwischen Videoqualität und Datenrate zu finden. Auch der eingesetzte Kompressionsalgorithmus spielt dabei eine große Rolle. Jede Lösung ist nur für einen bestimmten Anwendungsbereich optimal.
Bei Video-DVDs wird die Qualität beispielsweise nur soweit eingeschränkt, dass ein Kinofilm inklusive einigen Extras (mehrere Tonkanäle, Untertitel, ...) auf eine DVD passt. Die Datenrate darf aber 10,8MBit/s nicht übersteigen, um mit allen DVD-Playern abspielbar zu sein.
Mit wesentlich geringeren Datenraten ist bei langsameren Internetverbindungen auszukommen. Für ein Videostreaming muss die Qualität des Videos in diesem Fall dementsprechen eingeschränkt werden.

3.1 Videostandards

Die Entwicklung der Technik für den Fernseher hat verständlicherweise die analoge aber auch die digitale Videoverarbeitung sehr stark geprägt. Der vor allem in Europa eingesetzte PAL-Standard mit 625 Zeilen und einem Seitenverhältnis von 4:3 hat seinen Ursprung in der analogen Farbfernsehtechnik. Er ist eine Weiterentwicklung des in den USA heimischen NTSC-Standards. Auf Kosten der Farbauflösung werden bei PAL die durch Phasenverschiebungen auftretenden auffälligen Farbtonunterschiede durch zusätzlichen Schaltungsaufwand vermieden.

Als digitales Videoformat wird meist der ITU-R BT.601-Standard verwendet. Der Aufbau gleicht dem analogen Signal, wobei die Helligkeitskomponente Y mit 13,5Mhz und die beiden Farbkomponenten Cb und Cr wegen der geringeren Farbauflösung des menschlichen Auges mit 6,75Mhz zu jeweils 8Bit abgetastet werden. Daraus ergibt sich die bei ITU-R BT.601 eingesetzte, sogenannte 4:2:2-Abtastung. Die Zeilenzahl der ursprünglich analogen Quelle bleibt vorhanden, bei PAL ist dies 625. Durch die Abtastung ergibt sich eine horizontale Auflösung von 864 Pixeln. Darin sind aber auch die Bereiche der Austastlücken des analogen Signals enthalten, womit sich der eigentliche Bildinhalt auf 720x576 bei einer PAL-Quelle reduziert. Der ITU-R BT.601-Standard sieht zusätzlich vor, den Bereich der reinen Bildinformation mit den jeweils vier Byte langen Symbolen SAV und EAV zu markieren.

Der mit ITU-R BT.656 bezeichnete Standard definiert eine parallele und eine serielle Schnittstelle, die wahlweise für die Datenübertragung des ITU-R BT.601 Videoformats eingesetzt werden kann.

3.2 Digitalisierung

Um ein Videosignal digital verarbeiten zu können, muss es natürlich erst einmal digitalisiert werden. Spezielle Analog-Digital-Wandler für Videosignale werden Video Decoder genannt. Sie bieten zusätzlich zur eigentlichen Digitalisierung auch viele Zusatzfunktionen wie zum Beispiel automatische Videoformaterkennung, mehrere Videoeingänge oder Dekodierung von Signalen in der Austastlücke (VBI). Je nach geforderter Videoqualität muss wie bei jeder anderen Digitalwandlung ein entsprechend gut auflösender und genauer Konverter eingesetzt werden.

Die Digitalisierung ist naturgemäß immer mit einem gewissen Informationsverlust behaftet. Eine entsprechend gute Hardware kann aber die Qualitätsverluste auf ein für das menschliche Auge nicht mehr wahrnehmbares Maß reduzieren. Die Auflösung des Bildes ist großteils schon durch das analoge Videoformat festgelegt, bei PAL sind es 576 sichtbare Zeilen und eine durch die Abtastrate bestimmte horizontale Pixelanzahl des eigentlichen Bildsignals von 720.

3.3 MPEG-4

Die Moving Picture Experts Group ist ein Komitee von ISO/IEC, das sich mit der Standardisierung von Videokompressionsverfahren auseinandersetzt. Ihre Standards sind bereits weit verbreitet, darunter auch MPEG-2, das als Kompressionsformat für die Video-DVD oder bei DVB eingesetzt wird. Es hat sich mittlerweile die Abkürzung MPEG auch als Akronym für die Kompressionstechnik an sich eingebürgert.

MPEG-4 wurde ursprünglich für den Bereich geringerer Bitraten entworfen und enthält den auch für denselben Bereich entwickelten H.263-Standard der ITU. Dabei sollen nicht nur komprimierte analoge Videosignale unterstützt werden, sondern auch beliebige Objekte, aus denen gemeinsam bei der Dekodierung erst ein Video entsteht. Dieser Standard ist sehr umfangreich und enthält viele Methoden die Bitrate so weit wie möglich zu reduzieren. Er ist in viele Profile und Levels eingeteilt um nicht immer alle Funktionen unterstützen zu müssen. Rechenleistungsschwache Encoder und Decoder können somit beispielsweise auch nur die Basisprofile beherrschen.
Die erweiterten Profile sind vor allem in hardwarenäheren Bereichen wegen der begrenzten Rechenleistung so gut wie nie zu finden. Hier wird meistens nur das Simple Visual Profile und eventuell Teile vom Advanced Simple Profile (zum Beispiel ohne B-Frames) eingesetzt (eine Beschreibung der Profile ist in [PE02, ab Seite 597] zu finden).

Die nächsten drei Abschnitte behandeln nun die zugrundeliegende Technik der einfachen Videokompression, wie sie auch schon in früheren Versionen von MPEG und oftmals auch in anderen Verfahren eingesetzt wird. Für einen vollständigen Überblick sei auf eine entsprechende Fachliteratur verwiesen, das hier Festgehaltene soll lediglich eine theoretische Basis für die in dieser Arbeit auftretenden Begriffe bilden.

3.3.1 Intraframe Bilder

Ein Video besteht beim PAL-Standard aus 25 Bildern je Sekunde. Wie schon behandelt wäre das unkomprimierte Verarbeiten und Transportieren dieser Datenmengen mit den gängigen technischen Mitteln unmöglich. Die Bilder müssen also vor der Speicherung oder Versendung auf eine realistische Größe in Bytes reduziert werden.
Ein erster Schritt ist die Komprimierung eines Videobildes mittels verlustbehafteter JPEG-Technologie (siehe [Mil95, Seite 44]), wie dies bei einzelnen Bildern bereits weit verbreitet ist. Die so komprimierten Bilder werden I-Frames (intraframe-kodiert) genannt, weil sie unabhängig von anderen Bildern kodiert und dekodiert werden können.

3.3.2 Prädiktive Bilder

Dieser Bildtyp, bezeichnet als P-Frame, wird hingegen interframe-kodiert. Das bedeutet, dass dieses Bild nicht für sich alleine dekomprimiert werden kann, sondern dafür zumindest ein anderes, erfolgreich dekodiertes Bild zusätzlich benötigt wird. Durch die Verwendung eines vorherigen I- oder P-Frames als Referenz wird der Umstand ausgenutzt, dass aufeinanderfolgende Bilder von Videosequenzen sich oft sehr ähneln. Wenn also gewissermaßen nur die Änderungen kodiert werden, erreicht man in Summe eine wesentlich geringere Datenrate.

Beim Verfahren der sogenannten Bewegungskompensation werden Änderungen dabei nicht einfach durch Bildung von farblichen Bilddifferenzen ausgemacht, sondern durch genaue Untersuchung von kleinen Bildausschnitten, den sogenannten Makroblocks. Haben zwei Bilder fast denselben Makroblock an einer anderen Stelle, ist es wesentlich effizienter, nur den Bewegungsvektor zu übertragen. Weil dabei aber noch kleinere Bildfehler auftreten würden, wird zusätzlich noch der Prediction-Error kodiert, das ist der verbleibende Unterschied zum Originalbild.

3.3.3 Bidirektionale Bilder

Die sogenannten B-Frames benötigen zur Dekodierung einen zeitlich vorhergehenden und einen nachfolgenden I- oder P-Frame als Referenz. Durch Einsatz der interpolierten Bewegungskompensation für die Änderungen eines vorhergehenden Bildes auf ein nachfolgendes, kann meistens eine noch geringere Bitrate erreicht werden. Ein dadurch entstehender, verbleibender Bildfehler kann auch hier durch zusätzliche Kodierung der Restabweichung vermieden werden.
Dieser Bildtyp ist jener, der am meisten Rechenleistung, sowohl beim Encoder als auch beim Decoder erfordert. Aufgrund der Tatsache, dass auch zeitlich nachfolgende Bilder in die Kodierung mit eingerechnet werden und dadurch zusätzlicher Bildspeicher benötigt wird, ist auch die erreichbare minimale Verzögerung vom Eingangsvideo zum Ausgangsvideo auf jeden Fall größer als zum Beispiel bei reinem Vorwärtsbezug von P-Frames. Das sind vermutlich die Hauptgründe dafür, dass viele resourcenbeschränkte Encoder oftmals keine B-Frames unterstützen.

3.4 MP4 als MPEG-4-Container

In früheren MPEG-Versionen wurde der zugrundeliegende Datenstrom einfach ohne Zusatzinformationen in eine Datei geschrieben, alle für die Dekodierung nötigen Informationen sind darin schon enthalten. Für beliebige Audio- und Videodaten ihres MPEG-4 Standards hat die ISO/IEC ein eigenes MP4-Dateiformat definiert. Es basiert auf dem Apple QuickTime-Dateiformat und ist nun wesentlich flexibler als die Speicherung der nackten Videodaten. Dabei müssen beispielsweise die Daten nicht zwangsläufig in der Datei selbst enthalten sein, es können auch externe Dateien und anderen Quellen referenziert werden. Aber selbst innerhalb einer Datei sind die eigentlichen Videodaten von den Metadaten, die diese beschreiben, gewissermaßen getrennt gespeichert.

Die Grundlage für dieses Format sind ineinander geschachtelte Atome (englisch: Atoms), die eine hierarchische Struktur bilden (siehe [MPE, Seite 265]). Das Erzeugen einer MP4-Datei und damit einer solchen Struktur erfordert etwas Programmieraufwand, viele Atome müssen vorhanden sein und die Bitsequenz muss exakt stimmen. Hilfreich bei der Entwicklung einer MP4-Dateierzeugung ist das Sourceforge-Projekt `mpeg4ip` (http://mpeg4ip.sourceforge.net/). Das dort erhältliche Programm `mp4dump` analysiert eine MP4-Datei, gibt deren Inhalt textuell aus und zeigt auch enthaltene Fehler auf.
Die Atomstruktur enthält viele Informationen über die enthaltenen Videodaten, wie beispielsweise Bildformat, Bildwiederholungsrate, Länge, Anzahl und Position der I-Frames (für einen Überblick siehe [PE02, Abschnitt 7.3]). Dennoch dient die Datei nur dem Transport des MPEG-4-Datenstroms. Eine MPEG-4-Playersoftware extrahiert sich daraus den zugrundeliegenden Bitstrom und dessen MPEG-4-Decoder wertet diesen aus. Viele Informationen über das enthaltene Video sind auch hier noch enthalten, sodass der Decoder dadurch unabhängig von der MP4-Datei beispielsweise über das Bildformat Bescheid weiß und manche Player-Software das in der Atomstruktur des MP4-Files angegebene Bildformat einfach ignoriert.

Kapitel 4

Systemanforderungen und Architektur

Zunächst sind die Anforderungen der Anwendung zu analysieren und alle Lösungsmöglichkeiten genau abzuwägen. Dabei steht nicht so sehr die eigentliche Realisierung bezüglich Hard- oder Softwaregliederung im Vordergrund als viel mehr die eigentliche Funktion.
So ist es prinzipiell denkbar, die Videokompression sowohl in Hardware als auch in Software zu realisieren. Erst bei näherer Analyse muss hier die Softwarelösung ausgeschlossen werden, da die Rechenleistung der Mikrocontroller, die hier einsetzbar sind, dafür noch nicht ausreicht. An dieser Stelle anzumerken ist jedoch, dass auch eine Hardwarelösung die Kompression letztlich in Software durchführt. Allerdings in einer besonderen, dafür zugeschnittenen Umgebung in Form eines speziellen, hochintegrierten Schaltkreises. Die Softwarekomponente wird in diesem Fall als sogenannte Firmware bezeichnet und wird meist vom Hersteller des Chips entwickelt und zur Verfügung gestellt.

4.1 Systemverhalten

Bevor eine genauere Einteilung in Funktionskomponenten erfolgt, sollte zunächst das Gesamtsystemverhalten, welches ja die eigentliche Applikationsanforderung erfüllen soll, betrachtet werden.

Das auf der Serverseite von außen zur Verfügung gestellte analoge Video soll an jener Clientseite, die für das Video zuständig ist, wieder ausgegeben werden. Die Verzögerung des Videosignals darf dabei 500ms nicht überschreiten. Außerdem hat die Übertragung der Daten digital und im weltweit lizenzfreien ISM-Band bei 2,4GHz drahtlos zu erfolgen.

Zusätzlich sollen mindestens drei Übertragungskanäle mit einer Datenrate von jeweils etwa 12,8kBit/s zur Übermittlung von Steuerdaten zur Verfügung stehen. Steuerdaten können dabei zwischen allen beteiligten Knoten ausgetauscht werden. Dabei sollen einzelne Datenpakete eine Verzögerungszeit von 30ms für die vollständige und korrekte Übertragung nicht überschreiten. Steuerdaten, die länger für ihre Übermittlung benötigen, haben keine weitere

Relevanz mehr.

Systeme, deren korrekte Funktion nicht nur von der logischen, sondern auch von der zeitlichen Korrektheit abhängt, werden Echtzeitsysteme genannt. Durch die in dieser Applikation auftretenden zeitlichen Bedingungen handelt es sich hier um ein "festes Echtzeitsystem" (auf Englisch "Firm Real-Time System", siehe [WB05, Seite 321]). Auch die Videoübertragung stellt diese Anforderungen an das System, da zu spät eintreffende Videodaten auch nicht mehr verwendbar sind.

4.2 Schnittstellen

Die Schnittstellen nach außen sind Teil der Applikationsanforderungen. Betrachtet man das System als Black-Box, dann bilden sie die nötigen Ein- und Ausgänge, um alle Anforderungen erfüllen zu können.

Die Anwendung definiert zwei Schnittstellentypen. Der erste ermöglicht den Austausch eines analogen Videos mittels S-Video-Anschluß und der zweite behandelt den Informationsaustausch der Steuerdaten. Am Server ist die Video-Schnittstelle ein Eingang, am Videoclient ein Ausgang. Die Steuerdatenschnittstelle soll immer Ein- und Ausgang sein.

4.2.1 Videoschnittstelle

Das Video liegt auf der Serverseite im analogen Format vor, als Anschluß wird S-Video verwendet. Bei diesem Anschluß wird das Helligkeitssignal (Luminanz Y) und das Farbsignal (Chrominanz C) getrennt geführt, um eine bessere Videoqualität zu erreichen. Als Steckverbindung findet der allgemein übliche 4-polige Mini-DIN-Stecker, wie er in Abbildung 4.1 zu sehen ist, Verwendung.

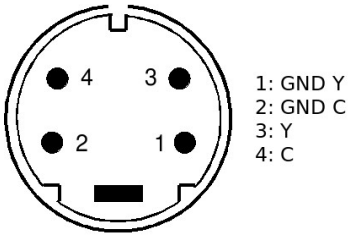

Abbildung 4.1: 4-poliger Mini-DIN-Stecker für S-Video

4.2.2 Steuerdatenschnittstelle

Die zeitkritischen Steuerdaten sollen über diese Schnittstelle mit dem System ausgetauscht werden. Steuerdaten von externen Geräten sind im System an die richtige Stelle zu transportieren, wo sie an dort angeschlossene externe Ziel-Geräte weitergeleitet werden. Für den dabei nötigen bidirektionalen Datenaustausch wird der Feldbus CAN (Control Area Network) eingesetzt.

4.3 Betriebsbedingungen

Die mit Funklösungen erreichte und mit Kabelverbindungen unvergleichbare Mobilität durch Verwendung des umgebenden Mediums bringt aber auch einen großen Unsicherheitsfaktor mit sich. Jede noch so ausgereifte Funktechnik kann mit einem entsprechenden Aufwand zumindest gestört, meistens sogar lahmgelegt werden. Auch wenn es nicht absichtlich herbeigeführt wird, können bei parallelem Betreiben zweier oder mehrerer Funktechnologien auf demselben Frequenzband empfindliche Störungen auftreten. Auch nicht für den Funk bestimmte Geräte können in den verwendeten Frequenzbereich hineinstören, ein Mikrowellenherd beeinträchtigt beispielsweise den Funkverkehr im ISM-Band (siehe dazu [Ste05, Abschnitt 6.2]). Der Datendurchsatz ist dabei je nach Funktechnik und Stärke der Störung eingeschränkt bis komplett unterbrochen.

Weitere wesentliche Faktoren bei Funklösungen sind die Dämpfung und Reflexionen der elektromagnetischen Wellen, die je nach Umgebung ganz unterschiedlich ausfallen. Neben der in jedem Medium (nicht im leeren Raum) mit steigender Entfernung zunehmenden Signalabschwächung werden die Wellen zum Beispiel innerhalb eines Gebäudes durch Wände, Decken und Böden noch stärker gedämpft und auch reflektiert. Im Allgemeinen hat jedes Objekt einen speziellen Einfluß auf die durchdringenden Wellen. Eine Reichweitenangabe ist somit nur für ganz bestimmte, nicht vollständig reproduzierbare Umgebungsbedingungen gültig.

Die Betriebsbedingungen können somit nur recht allgemein formuliert werden und halten die minimalen Anforderungen an die Funktechnologie fest.

- Indoor-Reichweite 10m ohne Zwischenwand
- Outdoor-Reichweite 30m mit Sichtverbindung
- zwischen oder im Umkreis von 10m um jeden Funkknoten keine Störfunker im gleichen Frequenzband
- Einhaltung der gesetzlichen Bestimmungen von Störfunkern und eigenen Funkknoten

4.4 Serversystem

Dieser Teil des Systems soll das analoge Video über die Video-Schnittstelle (siehe Abschnitt 4.2.1) entgegennehmen und an das Clientsystem per Funk digital aussenden. Weiters sollen bereitgestellte Steuerdaten an den betreffenden Client übermittelt werden, der diese gewissermaßen transparent an das Zielsystem weiterleitet. Auch ein beliebiger Client kann seinerseits Steuerdaten an den Server senden, der sie dann wieder in ein für das Zielsystem verständliches Format zurück übersetzt und an der für die Steuerdaten vorgesehenen Schnittstelle (siehe Abschnitt 4.2.2) bereitstellt.

4.5 Clientsystem

Wie bereits erwähnt gibt es zwei Typen von Clients. Der Video-Client hat die spezielle Aufgabe, die vom Server ausgesendeten Videodaten zu empfangen und analog an der Video-Schnittstelle auszugeben.
Die zweite Aufgabe müssen sowohl der Video-Client als auch der andere Client-Typ, der "normale" Client, erfüllen. Dabei handelt es sich um das Steuerdaten-Handling, das auch der Server in gleicher Art und Weise unterstützt. Die Steuerdaten sollen somit transparent über Funk empfangen und über die Steuerdatenschnittstelle weitergeleitet werden. Der Datentransfer findet dabei auch in umgekehrter Richtung statt.

4.6 Systemkomponenten

Die zu verwendenden Komponenten sind nun genauer zu bestimmen. Dies ist bei den meisten Anwendungen kein besonders einfacher Vorgang, da dabei sehr viele Faktoren eine Rolle spielen und meistens mehrere Lösungsvarianten in Frage kommen. Ein iteratives Verfahren zur Bestimmung der Komponenten ist hier zu empfehlen.
Zunächst sind solche Teile eines Systems festzulegen, die als kritisch bezüglich der Systemfunktion eingestuft werden können. Für die vorliegende Anwendung betrifft dies sicherlich die verwendete Funktechnologie, da die Anforderungen nahezu alle Techniken ausschließen. Gefordert sind hier nämlich neben einer garantierten, maximalen Latenzzeit von 30ms eine Übertragungsrate, die eine Übermittlung eines Videos plus Steuerdaten ermöglicht.

Was bedeutet nun die Forderung nach einer ausreichenden Datenrate? Wie schon im Abschnitt 1 erwähnt, ist derzeit keine Funktechnik verfügbar, die eine für die Übermittlung eines unkomprimierten Videos nötige Datenrate unterstützt. Für die Übertragung ist demnach eine Videokompression erforderlich.
Das geht natürlich auf Kosten der Qualität, da die Komprimierung eines Videos immer verlustbehaftet ist. Es bleibt jedoch nichts anderes übrig, als diesen Preis zu zahlen um die extremen Datenraten zu vermeiden. Kompressionsalgorithmen von aktuellem Stand der Technik leisten aber schon Erstaunliches, sodaß sich der Qualitätsverlust in Grenzen hält und dennoch eine vernünftige Datenrate erreicht wird.
Nach Analyse aktueller Kompressionsalgorithmen erscheint schließlich eine Datenrate von etwa 600kBit/s für eine annehmbare Qualität als ausreichend.
Die verwendete Funktechnologie sollte also unter Einberechnung der Video- und Steuerdatenraten und einer Reserve bei Übertragungsfehlern eine Datenübertragungsrate von mindestens 1MBit/s ermöglichen.

4.6.1 Funktechnologie

Die Funktechnologie ist zweifellos die Kernkomponente des Systems. Die Anwendungsanforderungen legen die Latte für eine einsetzbare Funktechnik sehr hoch. Es gibt die perfekte Technik nicht, die eine maximale Datenübertragungsrate, eine maximale Störsicherheit, eine garantierte minimale Latenzzeit und das bei minimaler Bandbreite aufweist. So bleibt nichts

anderes übrig, als für jede Anwendung den bestmöglichen Kompromiß zu finden.

Viele Entwicklungen im Bereich der Funktechnologien legen ihr primäres Augenmerk wegen des wachsenden Bedarfs vermehrt auf höhere Bitraten. Andere Parameter wie beispielsweise Störungssicherheit oder belegte Bandbreite werden dadurch natürlich negativ beeinflusst. Für die in dieser Arbeit behandelten Anwendung ist eine bestimmte Anforderung aber ganz wesentlich. Dabei handelt es sich um die garantierte maximale Latenzzeit eines Datenpaketes. Aktuelle Technologien wie WLAN oder Bluetooth genügen dieser Anforderung leider nicht, oder zumindest nicht immer (siehe [Wai02]). Es muss neben einer Videoübertragung eine Reaktionszeit für Steuerdaten von maximal 30ms erreichbar sein, dies aber natürlich nur bei definierten Betriebsbedingungen.
Durch die doch relativ hohe Datenübertragungsrate, die für das Video benötigt wird, muss der Entwurf des Übertragungsprotokolls gut durchdacht sein. Auch der Funktionsumfang des Funkmoduls spielt eine entscheidende Rolle, da in der Steuereinheit nicht übermäßig viel Rechenleistung für Fehlererkennungs-, Fehlervermeidungs- und Fehlerkorrekturmaßnahmen zur Verfügung steht.

Die sich für dieses Projekt am besten eignende Funktechnologie ist die von der deutschen Firma *Nanotron Technologies* entwickelte Technik. Die angebotenen Module der Produktreihe *nanoNET TRX RF Transceivers* belegen durch die eingesetzte Frequenzaufspreizung zwar das gesamte ISM-Band bei 2,4GHz (aktuelle Module unterstützen aber bereits drei nicht überlappende Kanäle, siehe [NAL]), sind jedoch ungleich störsicherer bezüglich leistungsbegrenzter, schmalbandiger Störer als digitale Modulationsverfahren, die keine Bandspreizung durchführen.
Die erzielbare maximale Datenübertragungsrate hält sich dabei in Grenzen. Es werden zwei Modi unterstützt, 1MBit/s und 2MBit/s, wobei die Symbolrate gleich bleibt, jedoch im schnelleren Modus vier statt zwei Symbole eingesetzt werden. Der schnellere Modus mit der doppelten Anzahl an Symbolen ist aber naturgemäß störungsempfindlicher als der mit zwei Symbolen. Die Funkmodule erlauben auch die Reduktion der Symbolrate auf die Hälfte, dabei erfüllt aber die damit erreichbare Datenübertragungsrate von 500kBit/s nicht die Anwendungsanforderungen.

Abbildung 4.2: Funk-Testmodul nanoPAN 5361

In Abbildung 4.2 ist das letztlich eingesetzte Testmodul zu sehen. Der dem Funkmodul *nanoPAN 5361* nachgeschaltene Sendeverstärker ermöglicht eine Sendeleistung von maximal 100mW (entspricht 20dBm).

Das gewählte Funkmodul unterstützt speziell für diese Anwendung wichtige Funktionen wie Virtual Carrier Sensing, Physical Carrier Sensing, Automatic Repeat Request, Forward Error Correction, Unicast-, Multicast- und Broadcast Transmission (siehe 2.6.1).

Im Folgenden wird nun auf die spezielle Bedeutung dieser Funktionen in Bezug auf die zu realisierende Anwendung eingegangen.

- **Virtual- und Physical Carrier Sensing**
  Diese Funktionen werden für die CSMA/CA-Fähigkeit des Funkmoduls benötigt. Dadurch kann die Belegung des Mediums erkannt und eine effiziente Nutzung desselben erreicht werden. Im Gegensatz dazu würde man mit einem TDMA-Verfahren, welches von diesem Funkmodul auch unterstützt wird, ein aufwendiges Management bei mehreren beteiligten Funkknoten benötigen. Wenn ein Knoten gerade keine Daten zum Senden hätte, wäre sein dedizierter Zeitschlitz unbenutzt. Für diese Applikation würde dies heißen, dass wichtige Datenübertragungsmöglichkeiten für das Video ausgelassen werden. Nimmt man an, dass ein gewisser Prozentsatz an Datenpaketen durch Übertragungsfehler verloren geht, womit immer zu rechnen ist, stünde nur eine geringe Datenübertragungsrate für das Video zur Verfügung und die Bildqualität würde darunter leiden.

- **Automatic Repeat Request**
  Durch das automatische erneute Senden eines verlorengegangenen oder gestörten Datenpaketes kann sichergestellt werden, dass die Daten beim Empfänger ankommen. Da schon das Funkmodul diese Funktion übernimmt und nicht erst die Steuereinheit, kann eine optimale Zeiteffizienz erreicht werden. Das ist speziell für die Übertragung der Steuerdatenpakete wichtig, weil sie eine feste Deadline einzuhalten haben.

- **Forward Error Correction**
  Die Übertragung eines Videos erzeugt ein erhebliches Datenaufkommen. Die Wahrscheinlichkeit für das Auftreten einer Störung von mindestens einem Datenpaket eines Video-Frames ist auch unter normalen Betriebsbedingungen nicht vernachlässigbar. Für ein unkomprimiertes Video sind einzelne Bitfehler nicht sehr störend, es treten nur Pixelfehler auf. Bei Fehlern im Datenstrom eines komprimiertes Videos sind allerdings gröbere Auswirkungen zu beobachten. Manche Decoder-Software, die das komprimierte Video wieder in unkomprimiertes Material zurückwandeln soll, bricht die Dekodierung sogar ab.
  Am Besten wäre es demnach, wenn eine unversehrte Übertragung des komprimierten Videos garantiert werden könnte. Dies würde aber bei vermehrt auftretenden Fehlern wieder das Übertragungsmedium zu sehr in Anspruch nehmen und die Übermittlung der bevorzugt zu behandelnden Steuerpakete wäre gefährdet.
  Abhilfe kann hier die Forward Error Correction ohne zusätzlichem Einsatz von Automatic Repeat Request schaffen, wenn gelegentliche Fehler im Datentrom noch akzeptabel sind und der Decoder dadurch nicht stoppt, sondern nur kleinere Bildfehler produziert. Der Nachteil dieser Methode bei der Datenübertragung ist die dafür nötige Redundanz und damit der erhöhte Übertragungsratenbedarf. Allerdings können damit am Empfänger innerhalb eines Bytes zwei Bitfehler korrigiert werden, auch wenn es sich um aufeinanderfolgende Bits handelt (siehe [NTRb, Abschnitt 5.4]). Sinnvoll ist diese Funktion, wenn wiederholtes Senden von Datenpaketen mehr Verkehr erzeugen würde, als die redundanten Daten.

- **Multicast Transmission**
  Dabei können zu sendende Daten gleich an mehrere Empfänger gleichzeitig adressiert werden. Pakete mit Videodaten als Inhalt sind so für mehrere Knoten gleichzeitig zu empfangen. Auf die Bestätigung des Empfangs muss hier jedoch klarerweise verzichtet werden.

4.6.2 Videokompression

Wie schon in vorhergehenden Abschnitten behandelt, kann ein Video in akzeptabler Qualität aus Mangel an Datenübertragungsrate nur in komprimierter Form drahtlos übermittelt werden. Eine verlustfreie Komprimierung, bei der zwar ein kodierungsbedingter Kompressionserfolg erzielbar ist, wäre für die gegebene Anwendung nicht ausreichend, da der zugrundeliegende Informationsgehalt des Videos einfach zu hoch ist. Bleibt somit nur mehr der Einsatz von verlustbehafteten Kompressionsverfahren übrig. Die in dieser Anwendung eingesetzte Technik soll bei der zur Verfügung stehenden Bitrate ein optimales Ergebnis bezüglich der Videoqualität erzielen.

Es gibt auch noch andere Anforderungen, die bei der Auswahl eines Kompressionsalgorithmus beachtet werden müssen. Die zu bedienende Anwendung soll ein Video-Live-Streaming durchführen. Das heißt, bestenfalls soll das am Server eingespeiste Videosignal ohne oder nur mit sehr geringer Verzögerung am Client gleich wieder ausgegeben werden. Es ist sicher einleuchtend, dass dies nicht realisierbar ist, da schon ohne die notwendige Videokompression die Daten eines einzelnen Bildes nur seriell übertragen werden können und das nicht beliebig schnell durchführbar ist. Bei Verwendung eines Encoder-Decoder-Paares, das eingangsseitig eine Videokompression und ausgangsseitig eine Videodekompression durchführt, sind noch zusätzliche Laufzeiten einzukalkulieren.

Die aller unterste Grenze der theoretisch erreichbaren Verzögerungszeit ist in Tabelle 4.1 zu sehen, wobei die durch die Steuereinheit auftretenden Verzögerungen vorerst vernachlässigt wurden.

Verarbeitungsschritt	**Verzögerung**
Digitalisierung	bis zu 40ms
Komprimierung	typ. 20-40ms
Funkübertragung	etwa 60ms
Dekomprimierung	typ. 10-30ms
Analogwandlung oder Bildaufbau	bis zu 40ms
in Summe ist zu rechnen mit	**> 150ms**

Tabelle 4.1: Unterste Grenzen der Video-Verzögerungszeiten

In dieser Tabelle fällt auf, dass bei der Funkübertragung eine Verzögerung der Dauer von zwei Einzelbildern bei einer Bildwiederholrate von 25 Bildern pro Sekunde zu erwarten ist. Auf den ersten Blick scheint es unmöglich zu sein, dass die Übertragung eines Bildes die Zeit für die Anzeige zweier Bilder benötigt. Man könnte so nicht die ganze Bildrate übertragen, da nur eine serielle Übertragung der Bilder möglich ist. Tatsächlich ist diese Verzögerungszeit für

ein Bild zwar korrekt, jedoch nur für einen I-Frame (siehe 3.3.1). Die im Schnitt wesentlich kleineren P-Frames (siehe 3.3.2) benötigen hingegen bei einer darauf abgestimmten Bitrate des Encoders weniger als 40ms Übertragungszeit. Wären diese Pakete auch größer, dann könnte man kein Live-Streaming mit 25 Bildern pro Sekunde durchführen, man müsste dann Bilder verwerfen.
Nun ist aber leicht nachzuvollziehen, dass die letztlich bleibende mittlere Videoverzögerung nicht geringer sein kann als die mittlere Übertragungsverzögerung aller I-Frames. Somit bestimmen die gegenüber den P-Frames größeren I-Frames das auftretende Video-Delay.

Das einzusetzende Videokompressionsformat muss neben möglichst guten Livestreamingfähigkeiten auch die Speicherung in eine Datei ermöglichen, sodass eine solche Datei zum Beispiel auf einem PC wiedergegeben werden kann.

Eine Lösung, die sich für die gegebenen Anforderungen besonders gut eignet, ist der Einsatz eines eigenen integrierten Schaltkreises der koreanischen Firma *INTIME*. Der MPEG-4-Chip des Typs *IME6500* benötigt zusätzlich nur noch SDRAMs mit einer der Anwendung entsprechenden Größe. Für ein Einkanalvideo in voller PAL-Auflösung von 720x576 benötigt man mindestens 64MByte SDRAM (siehe [IMEa, Seite 2]).
Der Chip hat viele nützliche Funktionen wie Bitratenkontrolle (Constant Bitrate, Variable Bitrate, Hybrid Bitrate), Group-of-Pictures Konfiguration (Anzahl P-Frames zwischen I-Frames), On-Screen-Display und einen dreistufigen Rauschunterdrückungsfilter.

4.6.3 Steuereinheit

Eine entscheidende Rolle spielt die Steuereinheit. Sie muss in der Lage sein auf jedes auftretende Ereignis blitzschnell zu reagieren und jede beteiligte Komponente gemäß Prioritätsliste zu bedienen. Für die Interaktion mit den Komponenten sind viele Schnittstellen nötig und eine angemessene Reserve soll für zukünftige Anforderungen zusätzlich eingerechnet werden. Weiters muss die Steuereinheit entweder genügend Rechenleistung für eine eventuelle softwareseitige MPEG-4-Dekodierung aufweisen oder relativ einfach gegen eine leistungsstärkere Einheit austauschbar sein.
Die Entwicklung soll so einfach und effizient wie möglich durchführbar sein und dementsprechende Werkzeuge wie eine übersichtliche Integrierte-Entwicklungsumgebung und ein Echtzeitdebugger sollen zum Einsatz kommen.

Ein Mikroprozessor der Firma *Analog Devices* vom Typ *Blackfin BF537* bringt alle nötigen Voraussetzungen mit. Er hat alle Schnittstellen onboard und kann mit einer Core-Frequenz von bis zu 600MHz arbeiten. Mehrere, unabhängige DMA-Controller binden einerseits periphere Schnittstellen wie SPI, PPI, UART oder Ethernet direkt an den Speicher an und ermöglichen andererseits ein sehr effizientes Kopieren von RAM-Bereichen im ein- oder zweidimensionalen Modus. Durch die Möglichkeit der direkten Einbindung des Adress-Daten-Busses in den Speicherbereich des Prozessors kann der Datentransfer zwischen RAM und MPEG-4-Chip auch prozessorschonend über Memory-DMA erfolgen.

Weiters spricht für die genannte Lösung das bereits am ICT erarbeitete Know-How bezüglich der Blackfin-Prozessorfamilie und die hier entwickelten Core-Module, die den Mikrocontroller

mit vielen wichtigen Komponenten erweitern. Das Core-Modul für den Blackfin BF537 ist in Abbildung 4.3 zu sehen. Dieses Modul verfügt zusätzlich über 4MByte Flash, 32MByte SDRAM und den für Ethernet nötigen Physical-Tranceiver-Chip.

Abbildung 4.3: Am ICT entwickeltes Core-Modul CM-BF537E

Mittels eigenem JTAG-Interface kann man das Core-Modul flashen, aber auch debuggen ohne den Flash-Speicher beschreiben zu müssen. Die Entwicklung ist damit sehr zeitsparend, da das sich in der Entwicklung befindliche und dabei relativ häufig geänderte Programm in einen viel schnelleren Speicher im JTAG-Interface geladen wird. Das Entwicklungskit der Firma Analog Devices enthält auch eine vergleichsweise mächtige integrierte Entwicklungsumgebung namens *VisualDSP++*.

4.6.4 Videodigitalisierung und Analogwandlung

Um das Video komprimieren zu können, muss es erst mittels Video Decoder digitalisiert werden. Der MPEG-4-Chip benötigt laut Herstellerangaben am Eingang ein digitales Video im Format ITU-R BT.601 oder ITU-R BT.656 (siehe [IMEb, Seite 1]). Dasselbe Format liefert der Chip gemäß Dokumentation am Ausgang, wenn er als MPEG-4-Dekoder verwendet wird. Dieses digitale Video muss für die analoge Schnittstelle wieder in ein analoges Video mittels Video Encoder zurückgewandelt werden.

Ein Video-Decoder-Chip der Firma *Analog Devices*, der mit *ADV7183B* bezeichnet wird, kann für die Digitalisierung eingesetzt werden. Er besitzt mehrere analoge Eingänge, die beliebig konfiguriert werden können, sodass Videosignale in Form von Composite-Video, S-Video oder YPrPb-Component in das gewünschte digitale Format ITU-R BT.656 gewandelt werden können (siehe [ADVb, Seite 1]). Dieser Chip lässt sich über die serielle Schnittstelle I^2C relativ einfach konfigurieren.

Der für die Analogwandlung eingesetzte Video Encoder *ADV7171* ist auch von der Firma *Analog Devices*. Dieser Chip führt das in digitaler Form im Format ITU-R BT.656 vorliegende Video in ein analoges Videosignal über (siehe [ADVa, Seite 1]). Die serielle Schnittstelle I^2C des Chips wird unter Anderem zur Auswahl der Ausgabeformate Composite-Video, S-Video, RGB, YUV und einige Kombinationen davon verwendet.

4.7 Protokollaufbau

Das Übertragungsprotokoll hat eine ganz zentrale Aufgabe, wobei es nicht einfach nur alle zu übertragenden Daten an ihr Ziel bringen soll.
Die Kernaufgaben des Protokolls sind eine garantierte Übertragung der Steuerdaten und ein möglichst vollständiger Videodatentransfer. Höchste Priorität haben die Steuerdaten, diese müssen unbedingt fehlerfrei und so schnell wie möglich übertragen werden. Hierfür bietet sich ein gewöhnlicher Unicast-Transfermodus mit positiver Empfangsbestätigung und automatisch wiederholtem Senden an.
Um eine möglichst hohe Übertragungsrate und damit möglichst gute Videoqualität zu erreichen, wird beim Senden der Videodaten auf die Bestätigung verzichtet und der Broadcastmodus eingesetzt. Gehen Videodatenpakete verloren oder sind sie fehlerhaft, kann der aktuelle Frame verworfen werden. Besser wäre, wenn der MPEG-4-Decoder auch die fehlerhaften Daten zum Dekodieren bekäme. Abhängig von der Schwere des Fehlers würden zwar Beeinträchtigungen des Ausgangsvideos auftreten, für den Betrachter wird dies aber meistens als besser empfunden, als wenn das Bild erst gar nicht angezeigt wird.
Der eingesetzte MPEG-4-Decoder hat aber verglichen mit anderen üblichen Softwaredecodern keine gute Fehlerkorrektur. Sobald ein Datenfehler in den Decoder des Chips geladen wird, hört er auf zu arbeiten. Um das Video nun nicht unvorhersehbar stillstehen zu lassen, müssen alle hineingeladenen Daten korrekt übertragen worden sein.

4.7.1 Steuerdaten

Ein Steuerdatenpaket bildet einen Container für Daten einer externen Anwendung. Ins System gelangen sie über die in Abschnitt 4.2.2 behandelte Schnittstelle.
Jedes Steuerdatenpaket umfasst eine Standardgröße von 32 Bytes, die aber parametrisierbar sein soll. Diese Pakete haben einen eindeutigen Empfänger und müssen garantiert ankommen. Dafür wird der normale Unicast-Modus mit der Funktion ARQ (Automatic Repeat Request, siehe Abschnitt 2.6.1) des Funkchips verwendet. Somit ist die Wahrscheinlichkeit des Paketverlusts unter normalen Betriebsbedingungen (siehe 4.3) verschwindend gering.

Die zweite, sehr wichtige Bedingung bezüglich Steuerdaten ist die möglichst geringe Übertragungsverzögerung. Dies wird durch den vom Funkchip unterstützten Medienzugriff über die Methodik von CSMA/CA (Carrier Sense Multiple Access, Collision Avoidance, siehe auch 2.6.1) gewährleistet. Steuerpakete vom Client können so unter Zuhilfenahme von Physical- und Virtual-Carrier-Sensing zwischen den Videopaketen des Servers das Medium belegen. Die maximale Verzögerung der Steuerdaten am Medium ist somit, abhängig von der Videopaketgröße, in diesem Fall bei 128Byte Nutzdaten lediglich 673μs (@2MBit/s brutto, siehe auch Abbildung 7.1). Der Videodatendurchsatz sinkt auch bei der vorgesehenen Maximalbelastung von Clientpaketen (5 Clients, jeweils 32 Byte alle 20ms → 64.000 Bit/s) dadurch theoretisch nur um 10%.

4.7.2 Videodaten

Durch die große Menge an Videodaten und der Annahme, dass einzelne Übertragungsfehler bei der Videoübertragung nur eine untergeordnete Rolle spielen, wird beim Senden der

Videodatenpakete der Broadcastmodus eingesetzt. Hierbei gibt es keine Bestätigung vom Empfänger über korrekte Ankunft der Videodaten. Es muss lediglich gewährleistet sein, dass der Videoempfänger auch mit teilweise korrupten Daten noch etwas anfangen kann, was bei den meisten MPEG-4-Decodern der Fall ist. Wenn nicht, muss der betreffende Frame bei der Dekodierung ausgelassen werden. Dies setzt aber voraus, dass für den Empfänger die falschen Daten auch erkennbar sind, was sich als nicht ganz trivial herausstellt. Einfach hingegen ist das Erkennen eines fehlenden Paketes.

Je nach Fehlertoleranz des MPEG-4-Decoders ist auch eine entsprechende Behandlung der Videodaten erforderlich, dabei können Fähigkeiten der eingesetzten Funklösung mit einbezogen werden (CRC, FEC).
Um alle Anforderungen abdecken zu können, werden zwei Pakettypen für die Videodatenübertragung definiert.

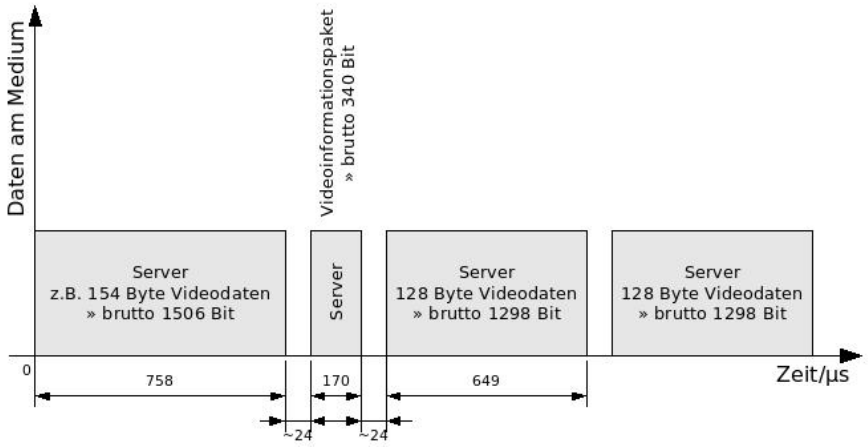

Abbildung 4.4: Videoinformations- und Videodatenpakete

Jeweils vor den Videodaten eines Frames wird ein Videoinformationspaket (siehe Abbildung 4.4) im Broadcastmodus ausgesendet. Der Videoclient erkennt dieses Paket anhand der Größe und der Headerinformationen. Dieser Header enthält die Anzahl der folgenden Videodatenbytes, woraus man nach Erhalt des nächsten Videoinformationspakets relativ einfach schließen kann, ob ein einzelnes Videodatenpaket verlorengegangen ist. Weiters ist es für den Videoclient ein Leichtes, sich auf das Videoinformationspaket aufzusynchronisieren. Das heisst, nach dem Einschalten des Clients muss dieser bei bereits aktivem Server nur auf ein Informationspaket eines I-Frames warten, um mit der Dekodierung beginnen zu können.

Der zweite Videopakettyp ist eine einfache Hülle für die Videodaten mit einer parametrisierbaren Standardgröße von 128 Bytes. Die Videodaten werden nach der Aussendung des zugehörigen Videoinformationspakets einfach in Videopakete verpackt und nacheinander auch im Broadcastmodus gesendet. Das letzte Videodatenpaket kann größer sein als die Standardgröße, wenn die über die Standardgröße hinausgehenden, verbleibenden Videodaten kleiner oder gleich der halben Standardgröße sind. Das bedeutet, wenn zum Schluss 128 bis 192 Bytes verbleiben, sind im letzten Videodatenpaket alle restlichen Videodaten zu senden.

Kapitel 5

Hardwareentwicklung

Am ICT wurde ein Testboard (Blackfin Video Board V1.0) entwickelt, das über alle nötigen Hardwarekomponten verfügt. Abbildung 5.1 zeigt das Schema der enthaltenen Hardware, wobei nicht alle Komponenten in jedem Funkknoten eingesetzt werden. Während der Server den Video Encoder ADV7171 nicht benötigt, findet am Video-Client der Video Decoder ADV7183B keine Verwendung und für den reinen Steuerclient sind Mikrocontroller und Funkmodul völlig ausreichend.

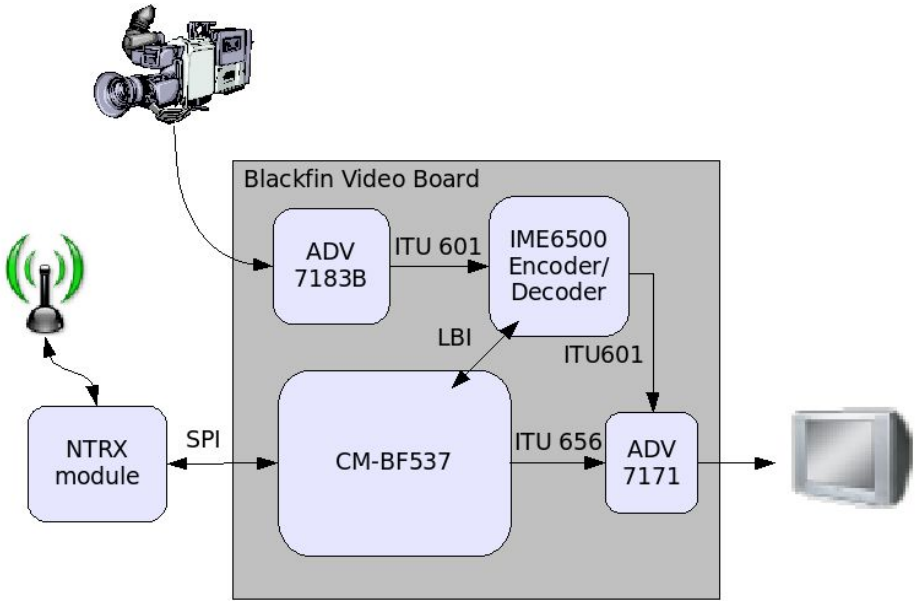

Abbildung 5.1: Schema der Hardwarekomponenten am Blackfin Video Board

Durch fehlende Erfahrungen mit dem MPEG-4-Chip musste das vorhandene Board erst getestet werden, was bereits Teil meines Aufgabenbereichs war. Nach längerer Testphase und Kontakt mit dem Support des MPEG-4-Chip Herstellers *INTIME* zeigte sich, dass der Chip mit den ursprünglich eingesetzten SDRAM-Bausteinen des Typs MT48LC4M16A2TG-75F der Firma *Micron* inkompatibel ist, obwohl im Referenzdesign des MPEG-4-Chips auch SDRAMs derselben Firma verwendet wurden. Als Ersatz wurden pinkompatible Bausteine der Firma *Samsung* des Typs K4S281632F-U75 herangezogen, die dann letztlich vom MPEG-4-Chip

auch erkannt wurden.

Nach einer genaueren Analyse aller für den MPEG-4-Chip zur Verfügung stehender Dokumentationen, Application-Notes und nach Feedback des Firmen-Supports konnten einige Fehler des Boards ausfindig gemacht werden. Einige Pins des Chips sollten anders verbunden werden, provisorische Änderungen am bestehenden Board wären nahezu undurchführbar gewesen, nachdem auf der Unterseite des MPEG-4-Chips 256 Pins auf 16x16mm für die Oberflächenmontage untergebracht sind. Ein neues Board musste somit am ICT entwickelt werden.

5.1 Korrekte Kontaktierung

Es zeigte sich, dass das ursprünglich entwickelte Board nicht für die MPEG-4-Kodierung und auch nicht für die Dekodierung eingesetzt werden konnte. Der Grund dafür sind spezielle Kontaktierungen, ohne die der Chip nicht, oder nur eingeschränkt arbeitet.
Die Tabelle 5.1 zeigt auch Pinbelegungen, die in dieser Form in keiner Dokumentation zu finden sind, jedoch für die korrekte Funktion in eingesetzter Konfiguration unbedingt richtig verbunden sein müssen.

IME6500	**MPEG-4-Chip-Pins**
von	**auf**
GPIO3	VSYNC
GPIO4	FIELD
GPIO5	GND
GPIO6	GND
BW	GND
HMODE	VDD
TMS	VDD

Tabelle 5.1: Nötige Kontaktierung bestimmter MPEG-4-Chip-Pins

Im Folgenden wird festgehalten, was bezüglich Eingangs- und Ausgangsseite des MPEG-4-Chips wesentlich ist.

5.1.1 Videoeingang

Ursprünglich war der ITU-R BT.656-Modus für den Transfer der digitalen Videodaten vom Video-Decoder zum MPEG-4-Chip vorgesehen. Allerdings konnte der MPEG-4-Chip auch nach intensiven Test- und Konfigurationsanstrengungen das Video mit ITU-R BT.656 als spezifizierten Videoeingangsmodus des MPEG-4-Chips nicht korrekt aufnehmen.
Der einzige Ausweg war, den ITU-R BT.601-Modus einzustellen, der die Kontaktierung von drei zusätzlichen Signalen zwischen Video Decoder und MPEG-4-Chip erforderte. Dabei handelt es sich um die Synchronisationssignale FIELD (Halbbildindikator), VS (Vertical Synchronization) und DV (Data Valid). Das Signal HS (Horizontal Synchronization) wäre an dieser

Stelle üblicher als DV, der MPEG-4-Chip benötigt aber letzteres für den zu verwendenden Modus. Der Video-Decoder muss jedoch so umkonfiguriert werden, dass dieser am Ausgang HS eigentlich ein DV ähnliches Signal ausgibt (siehe Abschnitt 6.3.2).

Die korrekte Kontaktierung des Video Decoders mit dem MPEG-4-Chip ist in Tabelle 5.2 ersichtlich.

ADV7183B Video-Decoder-Pin	**IME6500 MPEG-4-Chip-Pin**
P0-P7	CA_DIN0-CA_DIN7
LLC1	CA_PCKI
FIELD	CA_FIELD
VS	CA_VSYNC
HS	CA_DVALID

Tabelle 5.2: Nötige Kontaktierung der Schnittstelle zwischen Video-Decoder und MPEG-4-Chip

5.1.2 Videoausgang

Der Videoausgang des MPEG-4-Chips wird beim Dekodieren des MPEG-4-Videodatenstroms verwendet. Damit dieser überhaupt arbeiten kann, benötigt er die korrekte Belegung von zwei seiner GPIO-Pins, die nun nicht mehr für beliebige IO-Operationen zur Verfügung stehen (siehe Tabelle 5.1).
Das dekomprimierte Video soll über den Videoausgang im ITU-R BT.656-Format an den Video Encoder weitergeleitet werden, welcher aus diesen Daten wiederum ein analoges Video erzeugt. Auch hier scheint es so, als ob sich der MPEG-4-Chip unter diesem digitalen Videoformat etwas anderes vorstelle, da der Video Encoder daraus kein korrektes analoges Video erzeugen kann. Somit muss auch hier der alternative Modus ITU-R BT.601 mit zumindest einem zusätzlichen Synchronisationssignal (siehe Tabelle 5.3) eingesetzt werden.

IME6500 MPEG-4-Chip-Pin	**ADV7171 Video-Encoder-Pin**
D0-D7	P0-P7
PCLK0	CLOCK
HSYNC	HSYNC

Tabelle 5.3: Nötige Kontaktierung der Schnittstelle zwischen MPEG-4-Chip und Video Encoder

5.1.3 Local Bus Interface

Das LBI ist die Schnittstelle zwischen dem MPEG-4-Chip IME6500 und dem Mikrocontroller BF537.

Diese Schnittstelle wird eingesetzt für:

- Transfer der nötigen Firmware in den MPEG-4-Chip (siehe Abschnitt 6.2.2)
- Initialisierung und Konfiguration der MPEG-4-Encoder und Decoder-Funktionen (siehe Abschnitt 6.2.3)
- Transfer des MPEG-4-Datenstromes aus dem MPEG-4-Chip beim Einsatz als MPEG-4-Encoder (siehe Abschnitt 6.2.4)
- Transfer des MPEG-4-Datenstromes in den MPEG-4-Chip beim Einsatz als MPEG-4-Decoder (siehe Abschnitt 6.2.4)

Das LBI besteht aus einem gewöhnlichen Adress-Datenbus und einer Interruptsignalleitung. Der Zugriff soll für den Austausch großer Datenmengen so effizient wie möglich erfolgen. Aus diesem Grund wird der 16-Bit-Modus verwendet, die niederwertigste Adressleitung ist somit obsolet und der Pin HADDR0 des IME6500 bleibt unkontaktiert. Es werden somit immer ganze Wörter zu je 2 Bytes transferiert.
Durch die verwendete Interruptleitung, die auf ein GPIO-Pin des Mikrocontrollers gelegt wird, kann diese Schnittstelle gezielt eingesetzt werden und es treten keine zusätzlichen Verzögerungen und unnötige Busbelegungen beim Einsatz eines Polling-Verfahrens auf.

In Tabelle 5.4 ist die Kontaktierung des MPEG-4-Chips mit dem Mikrocontroller festgehalten.

IME6500 MPEG-4-Chip-Pin	**CMBF-537 Mikrocontroller**
HADDR1	ABE3
HADDR2-HADDR5	A2-A5
HDATA0-HDATA15	D0-D15
NCS	AMS2
NRD	ARE
NWD	AWE
HINTB	PG10

Tabelle 5.4: Nötige Kontaktierung der Schnittstelle zwischen MPEG-4-Chip und Mikrocontroller

Das ältere Entwicklungsboard (Blackfin Video Board V1.0) weist einen Fehler bei der Kontaktierung der Adressleitungen auf. Dabei ist die Leitung ABE3 des Mikrocontrollers auf den Pin HADDR0 und die Leitungen A2-A6 auf HADDR1-HADDR5 geführt. Da aber im 16-Bit-Modus der Pin HADDR0 vom MPEG-4-Chip ignoriert wird, weil dies die 8-Bit-Adresse spezifiziert, und die Leitung ABE3 des Mikrocontrollers aber die 16-Bit-Adresse angibt, muss eine Adressübersetzung durchgeführt werden. Dabei ist ein gewünschter Adressoffset um ein Bit Richtung MSB zu shiften, um die richtige Adressierung im MPEG-4-Chip zu erreichen.
Ohne Adressumsetzung ist eine Adresserhöhung um 16 Bit (Adressleitung ABE3), von einer 32 Bit ausgerichteten Adresse ausgehend, wirkungslos. Beim durchgehenden Auslesen eines betroffenen Adressbereichs sind also zwei aufeinanderfolgende 16 Bit Datenwörter immer ident, was sich auch im Blackfin-Memory Fenster der Entwicklungsumgebung VisualDSP++ dementsprechend widerspiegelt.
Dieser Kontaktierungsfehler kann von der Software aus detektiert und die Adressumsetzung

aktiviert werden, womit das Entwicklungsboard erster Version dann einsatzfähig ist.

5.2 Verwendetes Entwicklungsboard

In Abbildung 5.2 ist das bestückte, aktuelle Entwicklungsboard zu sehen.

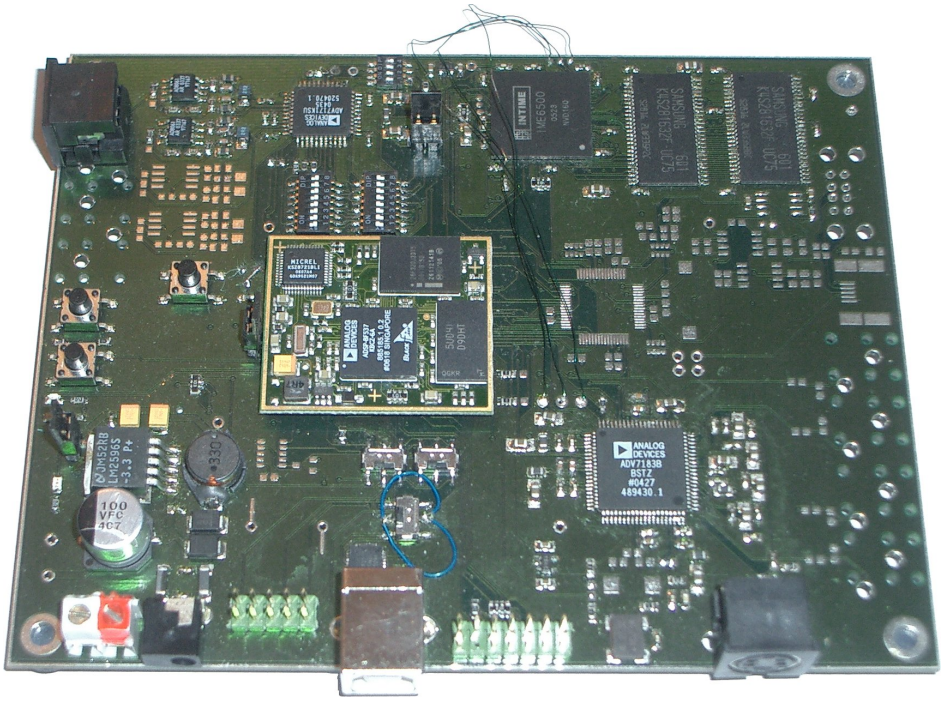

Abbildung 5.2: Am ICT entwickeltes und aktualisiertes Blackfin Video Board

Auf diesem neuen Entwicklungsboard findet man unter Vernachlässigung unwesentlicher Bauteile folgende Elemente:

- IME6500 MPEG-4-Chip
- 2x SAMSUNG K4S281632F-U75 SDRAMs
- ADV7183B Video Decoder
- ADV7171 Video Encoder
- Steckverbindungen für das Mikrocontrollermodul CM-BF537
- Kontaktstifte für das JTAG-Interface, das SPI und vier GPIO-Pins des BF537
- Mini-DIN-Kupplung jeweils für Analogvideoeingang und Analogvideoausgang (siehe 4.2.1)

5.3 Eval-Blackfin Board

Für den in Abschnitt 4.5 beschriebenen einfachen Client, der keine Videoverarbeitung durchzuführen hat und sich nur auf die Steuerdaten konzentriert, kann eine schon am ICT vorhandene Hardwarelösung verwendet werden. Die Basis besteht aus dem in Abbildung 5.3 gezeigten *EVAL-BF5XX Blackfin Evaluation Board*.

Abbildung 5.3: Am ICT entwickeltes EVAL-BF5XX Blackfin Evaluation Board

Um die Funktionen des Clients ohne Videoverarbeitung erfüllen zu können, ist noch eine Verbindung mit dem Funkmodul herzustellen. Die zu verbindenden Leitungen für die dabei eingesetzte SPI-Schnittstelle werden auf einem ebenfalls am ICT entwickelten, aufsteckbaren *EXT-BF5xx-Experimental Extension Board* einfach aufgelötet. Die Gesamtlösung für den einfachen Client ist in Abbildung 5.4 zu sehen.

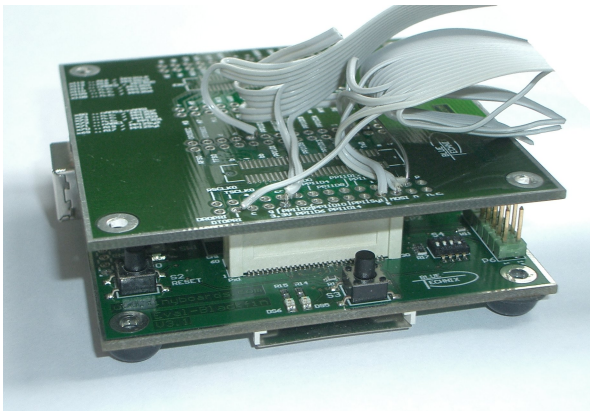

Abbildung 5.4: Hardwarelösung für den Client ohne Videoverarbeitung

5.4 Funkmodul

Das in Abbildung 4.2 gezeigte Funkmodul wird über SPI (Serial Peripheral Interface) mit dem Mikrocontroller verbunden. Zusätzlich zu den vier SPI-Leitungen gibt es aber auch noch zwei weitere Signale, die für den Betrieb des Chips erforderlich sind. Die Tabelle 5.5 gibt Aufschluß über die korrekte Kontaktierung mit dem Mikrocontroller.

CMBF-537 Mikrocontroller	**nanoPAN 5361 Funkmodul**
SCK	SPICLK
MISO	SPITXD
MOSI	SPIRXD
SSEL2	SPISSN
PF2	UCIRQ
PF3	PWRUPRESET

Tabelle 5.5: Nötige Kontaktierung der Schnittstelle zwischen Mikrocontroller und Funkmodul

Das mit dem Pin UCIRQ verbundene Signal ist der vom Funkchip ausgelöste Interrupt beim Auftreten eines vorkonfigurierten Ereignisses im Funkchip. Für die Applikation ist in diesem Zusammenhang wichtig, dass ein Interrupt beim Empfang eines Datenpaketes ausgelöst wird, um ein effizientes Handling durch den Mikrocontroller zu erreichen.
Das Signal PWRUPRESET (PowerUpReset) wird speziell für diese Hardwarekomponente benötigt, um einen definierten Anfangszustand zu erreichen. Dieses Signal muss vor der Initialisierung des Funkchips entsprechend getrieben werden (siehe Abschnitt 6.1.1).

Kapitel 6

Treiberentwicklung

Die eingesetzten Komponenten werden von der zu implementierenden Anwendungssoftware im Mikrocontroller initialisiert und der Funktion entsprechend angesprochen. Die dafür notwendigen Vorgänge sollen nicht direkt in die Anwendung mit eingebaut werden, sondern aus Gründen der Wiederverwendbarkeit und Wartbarkeit in eigene Software-Module ausgelagert und von der Anwendungssoftware lediglich verwendet werden. Diese Software-Module, die die Kommunikation mit der Hardware in gewisser Weise abstrahieren, nennt man Treiber. Die Anwendungssoftware benützt im besten Fall nur von den Treibern zur Verfügung gestellte Zugriffsmethoden.

Die Realisierung der Anwendung erfordert Treibersoftware für die verwendeten Hardwarekomponenten. Dieser Abschnitt beschäftigt sich nun mit der Implementierung dieser benötigten Treiber.

6.1 Funkchip nanoPAN5361

Der Funkchip ist mittels SPI mit dem Mikrocontroller verbunden. Auch die zwei zusätzlichen Signalleitungen müssen vom Treiber entsprechend behandelt werden.
Dieser Abschnitt zeigt nun, wie eine Kommunikation mit dem Funkchip über diese Schnittstellenleitungen möglich ist.

6.1.1 Initialisierung

Die Initialisierung des Funkchips ist ein relativ komplizierter Vorgang der aus vielen Einzelschritten besteht.
Eine grobe Auflistung der sequentiell durchzuführenden Maßnahmen:

- Hardware-Reset mittels korrektem PowerUpReset-Signal
- SPITXD-Leitungstreiber des Funkchips aktivieren
- Oszillator starten und Taktverteilung aktivieren

- Sender und Empfänger kalibrieren
- Schreiben der Symbolwerte in den Symbolgenerator
- Festlegen des Synchronisationswortes
- Festlegen der Funkknotenadresse
- Sender- und Empfängerkonfiguration

Um mit dem Funkchip überhaupt arbeiten zu können, muss zuerst der GPIO-Pin für das nullaktive Signal PowerUpReset entsprechend der Abbildung 6.1 getrieben werden. Diese Aufgabe ist aus Sicht der logischen Zuständigkeit eher dem Treiber zuzuordnen. Da jedoch durch die Verwendung eines GPIO-Pins des Mikrocontrollers die Portierbarkeit des Treibers auf andere Mikrocontroller nicht mehr gewährleistet wäre, wird dieser Vorgang nicht in den Treiber integriert. Er ist vor der ersten Verwendung des Treibers von der Applikation auszuführen (siehe dazu auch 7.2.1).

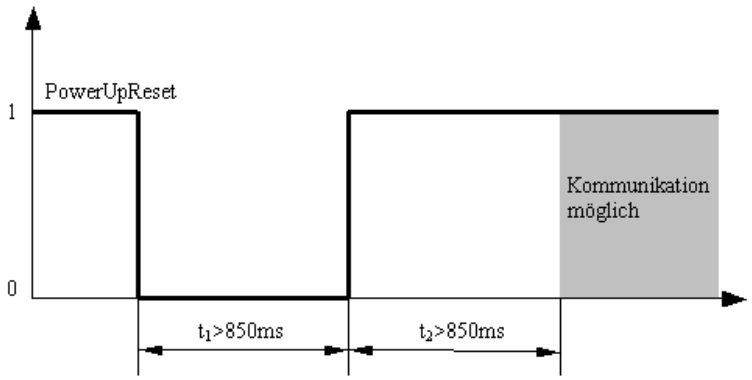

Abbildung 6.1: Timing des PowerUpReset-Signals für den Funkchip

Das Signal ist am Beginn, vor der Initialisierungsphase aktiv (auf 0), und für den restlichen Betrieb inaktiv (auf 1) zu setzen. Die aus der Abbildung ersichtlichen Timingparameter t_1 und t_2 wurden am Funkchip-Entwicklungsboard *PAN5460 - Panasonic Developmentboard V2.1* beobachtet, Tests zeigten jedoch, dass auch wesentlich geringere Zeitabstände von etwa 200ms ausreichen.

Nach diesem Hardware-Reset ist der Treiber am Werk. Er benötigt zur Initialisierung des Funkchips eine funktionierende SPI-Kommunikation. Der Zugriff auf diese Schnittstelle ist natürlich plattformabhängig. Um die Portierbarkeit des Treibers trotzdem so gut wie möglich zu gewährleisten, wird die eigentliche Kommunikation der Anwendung selbst überlassen. Dabei wird eine im Treiber als extern definierte, im Applikationshauptprogramm implementierte Methode `transSPI()` für alle Schreib- und Lesetransfers verwendet (siehe dazu auch 7.2.1).

Die erste Aktion der Treibersoftware ist die Aktivierung des Leitungstreibers für das Signal SPITXD des Funkchips. Nach dem Reset ist dieser deaktiviert, es können dadurch keine Daten vom Funkchip an den Mikrocontroller übermittelt werden. Die Aktivierung erfolgt durch das Setzen des Bits 7 der Registeradresse `0x0B` (`SciTxPushPull`) im Speicher des Funkchips.

Somit ist dies der erste Schreibzugriff auf den Chip, der gemäß SPI-Protokoll (siehe Abschnitt 2.6.3) durchzuführen ist.

Danach sind zahlreiche Initialisierungsmaßnahmen über unzählige Registertransfers durchzuführen, um den Funkchip in den betriebsbereiten Zustand zu versetzen. Die genaue Abfolge ist dem Quellcode im Anhang A zu entnehmen. Wichtig dabei ist jedoch die Initialisierung mit den korrekten Werten für den Symbolgenerator (ChirpBytes) und das Synchronisationswort (SyncWord),

6.1.2 Empfang eines Datenpaketes

Wurde der Funkchip korrekt initialisiert, der enthaltene Receiver gestartet und ein Datenpaket empfangen, wird über die Interruptleitung der Empfang vom Receiver angezeigt. Dieser Interrupt löst direkt die Bearbeitung einer ISR (Interrupt Service Routine) aus, innerhalb der das Datenpaket aus dem Funkchip in einen zusammenhängenden Speicherbereich des Mikrocontrollers gelesen und der Chip für den Empfang des nächsten Pakets wieder bereit gemacht wird.

Besonders zeitkritisch ist der Empfang von Datenpaketen, die größer als 128Byte sind. Im eingesetzten Auto-Modus des Funkchips stehen nämlich nur maximal 2x64Byte im Receive-Buffer zur Verfügung (siehe [NTRa, Seite 20]). Während ein Teil des Buffers schnell genug ausgelesen werden muss, speichert der Funkchip die gerade empfangenen Daten im anderen Bufferteil ab. Mithilfe zweier zusätzlicher Flags für jeden Bufferteil (`RxBufferCmd` - Bufferteil bereit zum Senden, `RxBufferRdy` - Bufferteil bereit zum Beschreiben) können durch das alternierende Auslesen theoretisch Datenpakete bis zur Maximalgröße von 8192Byte empfangen werden, obwohl kein Speicher dieser Größenordnung im Chip vorhanden ist.
Pakete dieser Größe machen für diese Anwendung nicht viel Sinn, da die Wahrscheinlichkeit für Störungen mit der Dauer der Belegung des Mediums zunimmt. Werden kleine Pakete gestört, können diese wieder schneller erneut gesendet werden.

6.1.3 Senden eines Datenpaketes

Der Treiber stellt eine Funktion zur Verfügung, die aus einem zusammenhängenden Speicherbereich des Mikrocontrollers eine angegebene Anzahl an Bytes an die spezifizierte Funkknotenadresse schickt. Zusätzlich kann auch der Typ des Pakets angegeben werden, entweder ist dies ein Unicast- oder eine Multicast- bzw. Broadcast-Message. Ähnlich wie beim Empfang eines Paketes der Größe von über 128Byte muss auch beim Senden wegen des gleich großen Sende-Buffers dieselbe Buffer-Zugriffstechnik eingesetzt werden. Während ein Bufferteil vom Funkchip ausgesendet wird, muss der andere Teil so schnell wie möglich durch SPI-Transfers vom Mikrocontroller beschrieben werden. Durch zwei Flags je Bufferteil wird die Synchronisation bei diesem Vorgang sichergestellt (`TxBufferCmd` - Bufferteil bereit zum Senden, `TxBufferRdy` - Bufferteil bereit zum Beschreiben).

6.2 MPEG-4-Chip IME6500

Diese Komponente wird zur Komprimierung und Dekomprimierung des zu übertragenden Videos eingesetzt. Es ist eine Hardwarekomponente, die ihre Aufgaben intern mittels eigener Software, in diesem Zusammenhang als Firmware bezeichnet, realisiert. Dieser Abschnitt behandelt die entwickelte Abstraktionsschicht zwischen dem direktem Buszugriff auf den MPEG-4-Chip und der Applikationssoftware. Die nötigen Initialisierungs- und Konfigurationsschritte werden durch diesen Treiber auf einfache Funktionsaufrufe reduziert.

6.2.1 Interaktion

Über das LBI (Local Bus Interface) wird ein Speicherbereich von 36 Bytes Größe in den adressierbaren Bereich des Mikrocontrollers eingeblendet. Die so zur Verfügung gestellten Register des MPEG-4-Chips sind in Tabelle 6.1 aufgelistet.

Adress-offset	**Register**	**Breite** /Bit	**Zweck**
0x00	CMD	32	Befehlskommunikation
0x04	DR0	32	Ein-, Ausgabeparameter
0x08	DR1	32	Ein-, Ausgabeparameter
0x0c	DR2	32	Ein-, Ausgabeparameter
0x10	STAT	32	Interruptstatus
0x14	FSR0	32	nicht benötigt
0x18	FSR1	32	nicht benötigt
0x1c	FCR	32	FIFO-Control
0x20	FDR	16	FIFO-Data
0x22	FAR	16	FIFO-Address

Tabelle 6.1: MPEG-4-Chip IME6500 Registeradressen

Hier werden nur Register und deren Bits beschrieben, die für die Erfüllung der erforderlichen Aufgaben relevant sind. Nicht Erwähntes kann in der offiziellen Dokumentation nachgeschlagen werden (siehe dazu [IMEb] und [IMEa]).

Das CMD-Register ermöglicht einen Software-Reset des Chips und steuert die Befehlsausführung. Vor der ersten Operation ist ein Software-Reset gemäß Tabelle 6.2 auszulösen.

Vorgang	**Beschreibung**
0x8000 → CMD	Bit CMD[15] setzen
delay 5ms	etwas warten
0x0000 → CMD	Bit CMD[15] rücksetzen
delay 5ms	etwas warten

Tabelle 6.2: MPEG-4-Chip Software-Reset

Ein Kommando an den MPEG-4-Chip wird, wie in Tabelle 6.3 gezeigt, schrittweise durchgeführt.

Schritt	Vorgang	Beschreibung
1	in1 → DR0 in2 → DR1 in3 → DR2	ersten Parameter festlegen zweiten Parameter festlegen dritten Parameter festlegen
2	task → CMD[31:16]	Task festlegen (nur bei type=0x20)
3	type → CMD[7:2] 1 → CMD[0]	Kommandotyp festlegen Befehl ausführen
4	while(CMD[0] == 1);	Ausführungsbeendigung abwarten
5	DR0 → out1 DR1 → out2 DR2 → out3	erster Rückgabewert zweiter Rückgabewert dritter Rückgabewert

Tabelle 6.3: MPEG-4-Chip Kommandoausführung

Alle nicht angeführten Bits des Registers CMD sind mit 0 zu beschreiben, besonders das Reset-Bit 15 und das Command-Mode Bit 1. Letzteres sollte es ermöglichen, den Chip einen Interrupt nach Ausführung des Befehls auslösen zu lassen, was jedoch mit den verwendeten MPEG-4-Chip-Samples nie funktioniert hat. Der Polling-Modus erscheint aber für geforderte Zwecke ausreichend zu sein.

An dieser Stelle sei noch angemerkt, dass durch den eingesetzten 16Bit Buszugriff natürlich nicht ein einzelnes Bit eines Registers alleine gesetzt werden kann, bei einem Schreibvorgang sind immer alle 16Bit beteiligt. Die Dokumentation weist speziell darauf hin, dass das Bit 0 des Registers CMD für die Ausführung eines Befehls als letztes zu beschreiben ist (siehe [IMEb, Abschnitt 11.2.1]). Das bedeutet konkret, dass alle Bits des Registers CMD korrekt gesetzt sein sollten, bevor CMD[0] von 0 auf 1 übergeht. Da das Register 32Bit breit ist und ein Registerzugriff nur 16 Bits auf einmal schreiben kann, sind die höherwertigen Bits des Registers (CMD[31:16]) auf Adressoffset 0x02 zuerst zu übertragen.

6.2.2 Download Firmware

Der eingesetzte Chip IME6500 besitzt keinen integrierten remanenten Speicher. Die für seine Funktion benötigte Firmware muss also extern vorhanden und vor der Inbetriebnahme in den Chip geladen werden. Sie wird von der Herstellerfirma als binäre Datei zur Verfügung gestellt und ist genau 480kByte groß.

Nach dem Software-Reset ist der Bootloader (die ersten 1024 Bytes der Firmware) über das Register DR1 und Kommandos im Register CMD, wie in [IMEb, Abschnitt 11.6.1] beschrieben, hineinzuladen. Der dort dokumentierte Pseudo-Code hat jedoch zwei gravierende Fehler. Bei der Berechnung der Check-Summe sollte statt `if(k & 4)` eigentlich `if(!(k & 4))` stehen und weiters soll
`if((read_reg(CMD) & 0x4000) == 1)` in Wirklichkeit

`if((read_reg(CMD) & 0x4000) == 0)` sein.
Letzterer ist definitiv als Fehler auszumachen, da die Operation niemals `true` liefern würde. Der andere Fehler konnte nur durch eine Analyse des von der Chiphersteller-Firma *INTIME* zur Verfügung gestellten Linux-Referenz-Sourcecodes gefunden werden.

Der übertragene Bootloader ermöglicht nun den Transfer der restlichen Firmware, sinngemäß steht es so in der Dokumentation. Tatsächlich sollte der nächste Schritt nun nicht nur den Rest der Firmware (480kByte Firmware - 1kByte Bootloader) in den Chip laden, sondern die gesamten 480kByte. Dieser aus der Dokumentation interpretierte Irrtum konnte wiederum durch genaue Analyse der Referenz-Sourcen ausgeräumt werden.

Für die Übertragung der 480kByte mittels funktionstüchtigem Bootloader stehen zwei Möglichkeiten zur Verfügung. Entweder über gewöhnliche Registertransfers, wie dies auch beim Laden des Bootloaders durchgeführt wurde, oder mittels FIFO Datentransfers. Die zweite Variante ist dabei wesentlich schneller, da hierbei sogenannte Burst-Transfers in Blöcken zu je 1024 Bytes durchgeführt werden. Das bedeutet, dass die Daten in einfachen, direkt aufeinanderfolgenden Schreibzyklen auf das Register FDR geschrieben werden. Die Daten landen in einem chipinternen 1024Byte großem FIFO-Buffer, von dem aus sie durch die Firmware in einen dafür vorgesehenen SDRAM-Speicherbereich weiter transferiert werden. Die Synchronisation dieses Vorgangs ist etwas aufwendiger als der Transfer über Registerkommandos, allerdings bedeutet dies einen geringeren Initialisationszeitbedarf. Der Chip ist durch die Übertragung mittels FIFO0 (erste FIFO des IME6500) somit schneller einsatzbereit.

Die erste Version der von der Firma *INTIME* zur Verfügung gestellten Firmware wies Fehler bei FIFO-Operationen auf. Dabei gingen sporadisch 1024Byte-Blöcke verloren. Dies konnte verifiziert werden, indem die Firmware zuerst nicht im FIFO-Verfahren in den Chip geladen wurde, sondern mit normalen Registertransfers, und die vom MPEG-4-Encoder gelieferten Videodatenpakete analysiert wurden. Der *INTIME*-Support stellte daraufhin eine adaptierte Version der Firmware zur Verfügung, die auch mit dem eingesetzen Local-Bus-Interface funktioniert.

6.2.3 Initialisierung und Konfiguration

Wenn die Firmware nun komplett hineingeladen wurde, sind Initialisierungen und Konfigurationen mittels Task-Operationen durchzuführen. Jede Aufgabe hat eine bestimmte Task-ID, Funktionsparameter und Rückgabewerte (siehe [IMEa, Seite 21ff]). Eine Aufgabe wird ausgeführt, indem gemäß Tabelle 6.3 ein spezielles Kommando an den Chip gesendet wird. Dabei ist für `task` die entsprechende Task-ID und für `type` der für Tasks zu verwendende fixe Wert `0x20` einzusetzen. Selbstverständlich sind die Übergabeparameter entsprechend der genannten Tabelle vor der Ausführung des Tasks zuerst in die entsprechenden Register zu übertragen, die nach der Ausführung die Rückgabewerte enthalten.

Zu Beginn der Initialisierung wird entsprechend dem Einsatz als Server- oder Video-Client-System ein MPEG-4-Encoder oder ein MPEG-4-Decoder ausgewählt. Zu diesem Zweck enthält die Firmware vier Module, zwei Video- und zwei Audiomodule, wobei jeweils eines davon der

Encoder und eines der Decoder ist (MPEG-4-Encoder, MPEG-4-Decoder, ADPCM-Encoder, ADPCM-Decoder). Es kann nur ein Video- und ein Audiomodul zur gleichen Zeit betrieben werden. Für Video und Audio kann aber auch auf ein Modul verzichtet werden, wovon in dieser Anwendung Gebrauch gemacht wird, da nur Videokodierung bzw. Videodekodierung ohne Ton erforderlich ist.

Nach der Auswahl eines Videomoduls wird der Videoeingang bzw. der Videoausgang des Chips konfiguriert und eine MPEG-4-Encoder- bzw. eine MPEG-4-Decoder-Instanz erzeugt. Der MPEG-4-Encoder verfügt über zahlreiche Konfigurationsmöglichkeiten, von denen viele für die bearbeitete Anwendung relevant sind.

Der genaue Initialisierungs- und Konfigurationsvorgang ist recht einfach aus dem Treiberquellcode im Anhang B zu entnehmen.

Der implementierte Treiber stellt einfache Funktionen zur Verfügung, um aus der Anwendung die Initialisierung, Konfiguration und den Betrieb des MPEG-4-Chips IME6500 durchzuführen.

6.2.4 Transfer MPEG-4-Datenstrom

Für den schnellen Datentransfer der MPEG-4-komprimierten Videodaten wird der erste FIFO-Buffer des Chips verwendet (FIFO0). Der vom MPEG-4-Encoder generierte MPEG-4-Datenstrom wird aus diesem Buffer blockweise ausgelesen und der für den MPEG-4-Decoder bestimmte MPEG-4-Datenstrom wird in diesen Buffer blockweise hineingeschrieben. Die Handhabung ist bei beiden Varianten etwas verschieden.

Der MPEG-4-Encoder liefert den MPEG-4-Datenstrom in Paketen, die jeweils ein Bild (Frame) enthalten. Mittels Interrupt wird angezeigt, dass Daten im FIFO-Buffer zum Auslesen bereit stehen. Die ersten acht Bytes der Daten bilden einen Paketheader (siehe [IMEa, Figure 7]). Neben Encoder-ID, Paketnummer und Frame-Typ (I oder P) ist darin auch die Paketgröße enthalten. Der FIFO-Buffer wird nun so oft in Blöcken der Buffergröße 1024Byte ausgelesen, bis die ganze Paketlänge übertragen wurde. Die nicht benötigten Daten im letzten Block, die laut Paketgröße nicht mehr dazugehören, können entweder verworfen oder auch gar nicht erst ausgelesen werden.

Dem MPEG-4-Decoder werden die reinen MPEG-4-Daten über den FIFO0-Buffer hineingeladen. Der Rest des letzten 1024Byte-Blocks des aktuell zu ladenden Frames muss mit Nullen aufgefüllt werden (gemäß Linux Referenz-Source-Code). Vor dem Laden muss der Decoder noch über den Frame-Typ (I oder P) und die Größe des zu ladenden, komprimierten Bildes informiert werden. Dies geschieht mittels Task STREAM_WRITE (Task-ID `0x2c00`) über eine Task-Operation. Der MPEG-4-Decoder speichert die Frames in einem Videoframe-Buffer und benachrichtigt die Anwendung mittels Interrupt über den aktuellen Bufferstand, wenn dieser einen konfigurierbaren Wert unterschreitet.

Der Transfer der Daten zu und vom FIFO0-Buffer wird im Treiber mittels Mikrocontroller-DMA durchgeführt. Dazu wird zuerst der Transfer vorbereitet und dann der DMA-Controller so konfiguriert, dass dieser 512 Schreib- oder Lesezugriffe auf der einen Speicheradresse des FDR mit 16Bit Breite durchführt (Burst Transfer). Damit wird der Transfer der betroffenen 1024 Byte zur Gänze vom DMA-Controller übernommen und der Mikroprozessor damit entlastet, um sich dann zwischenzeitlich anderen Aufgaben zuwenden zu können.

6.3 Video Decoder ADV7183B

Der Video Decoder ADV7183B der Firma *Analog Devices* wandelt viele analoge Videoformate in das digitale Format ITU-R BT.656 um. Er hat aber auch noch zusätzliche Ausgänge für die Synchronisationssignale HS, VS und FIELD (Horizontal-Synchronization, Vertical-Synchronization und Halbbildindikator). Diese Signale können in weitem Maße konfiguriert werden um dadurch auch das Videoformat ITU-R BT.601 mit zusätzlichen Synchronisationsleitungen für den MPEG-4-Chip verwendbar zu machen.

Die Konfiguration des Chips wird über die serielle Schnittstelle I^2C durchgeführt. Hierbei werden bestimmte Registeradressen im Video Decoder mit Bitmustern gemäß Dokumentation (siehe [ADVb]) vom Mikrocontroller aus über diese Schnittstelle beschrieben.

Der I^2C-Schnittstellentreiber ist am ICT schon als Komponente des *BLACKSheep*-Frameworks vorhanden und kann somit im Videogerätetreiber dieses Projektes ganz einfach eingebunden werden. Der entwickelte Gerätetreiber stellt eine einfache Funktion zur Verfügung, die alle nötigen Initialisierungen des Video Decoders und auch des Video Encoders durch simplen Aufruf durchführt.

6.3.1 Analogvideoeingang

Standardmäßig ist der ADV7183B auf ein Composite-Video-Eingagssignal eingestellt, kann aber über einen einfachen Registerzugriff auf S-Video oder YPbPr umkonfiguriert werden. Da das Eingangssignal im S-Video-Format vorliegt, muss der Video Decoder darauf eingestellt werden.

Durch Schreiben des Wertes `0x06` auf die Registeradresse `0x00` des Video Decoders (`0x00` ← `0x06`) wird der S-Video-Eingang am Entwicklungsboard korrekt konfiguriert.

6.3.2 Modus ITU-R BT.601

Schon sehr bald zeigten sich Inkompatibilitäten zwischen Video Decoder und MPEG-4-Chip. Der Videoeingang des IME6500 konnte das über die ITU-R BT.656 Schnittstelle gelieferte digitale Video des ADV7183A nicht korrekt aufnehmen. Abbildung 6.2 zeigt ein so erzeugtes, MPEG-4-komprimiertes blaues Standbild.

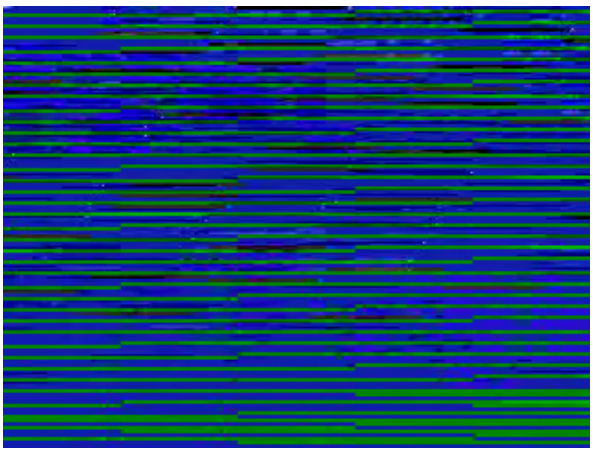

Abbildung 6.2: MPEG-4-komprimiertes blaues Standbild unter Verwendung der Schnittstelle ITU-R BT.656

Intensive Fehlersuche brachte leider keine Ergebnisse, der MPEG-4-Chip IME6500 scheint das vom Video Decoder generierte digitale Video über die Standard-Schnittstelle ITU-R BT.656 nicht korrekt zu interpretieren. Daraufhin musste auf zusätzliche Synchronisationssignale, und somit auf den Modus ITU-R BT.601 für die MPEG-4-Chip Konfiguration zurückgegriffen werden. Der MPEG-4-Chip unterstützt dafür die Signale VS (Vertical Synchronization), DV (Data Valid) und FIELD. Das DV-Signal kann aber vom Video Decoder nicht standardmäßig geliefert werden. Es ist jedoch möglich, das HS-Ausgangssignal des Video Decoder so umzukonfigurieren, dass es dem vom MPEG-4-Chip erwarteten DV-Signal recht nahe kommt.

Um den HS-Ausgang als DV-Signal für den MPEG-4-Chip verwendbar zu machen, muss das HS-Signal so umkonfiguriert werden, dass es genau während der Übermittlung der aktiven Videodaten (zwischen Start-of-Active-Video SAV und End-of-Active-Video EAV) auf logisch 0 liegt.
Dabei muss gemäß Dokumentation (siehe [ADVb, Seite 39]) der Parameter HSB mit dem Wert 284 und der Parameter HSE mit dem Wert 1724 initialisiert werden. Aus diesen Werten folgen die Registereinstellungen `0x34` ← `(0x1<<4)|(0x6)`; `0x35` ← `0x1c`; `0x36` ← `0xbc`.

Eine wichtige Einstellung ist auch das Timing des Synchronisationssignals VS. Mit den Standardeinstellungen wäre das Signal unsymmetrisch bezüglich der Halbbilder, was sich im MPEG-4-kodierten Video in einem um die halbe Bildbreite verschobenen Halbbild äußert.
Folgende Register sind für die symmetrische Einstellung der Vertikalsynchronisation zu setzen: `0x32` ← `0xc1`; `0x33` ← `0xc4`.

Weiters ist die Polarität des VS- und FIELD-Signals anzupassen, standardmäßig nimmt der MPEG-4-Chip sie als nullaktiv an. Die dafür nötige Registereinstellung ist `0x37` ← `0x29`.

Am Entwicklungsboard ist der MPEG-4-Chip und der Mikrocontroller BF537 mit der parallelen Digitalvideoschnittstelle des Video Decoders verbunden. Damit beide Komponenten mit dem digitalen Video versorgt werden können, muss das Taktsignal des Video Decoders stärker getrieben werden. Das geschieht durch die Registereinstellung `0xf4` ← `0x1d`.

6.4 Video Encoder ADV7171

Auch der Video Encoder lässt sich über die serielle I^2C-Schnittstelle konfigurieren. Er besitzt vier analoge Videoausgänge, standardmäßig werden zwei davon mit einem Composite-Videosignal belegt und zwei mit den beiden Komponenten Y und C des S-Video-Signals. Am Entwicklungsboard sind die beiden Signale für S-Video mit einer Mini-DIN-Kupplung (siehe 4.2.1) verbunden, somit ist für den korrekten Analogvideoausgang keine spezielle Einstellung nötig.

Konfigurieren muss man jedoch das Timing der Synchronisationssignale, die vom MPEG-4-Chip generiert werden. Wie auch beim Video Decoder nicht die ursprünglich angedachte ITU-R BT.656 Standardschnittstelle verwendet werden kann, muss der Video Encoder auch entsprechend konfiguriert werden. Der MPEG-4-Chip liefert am Ausgang kein vom Video Encoder korrekt verstandenes Videoformat, sofern man nicht die zusätzlichen Synchronisationssignale zuhilfe nimmt. Das BLANK-Eingangssignal wird dabei allerdings nicht benötigt, kann also deaktiviert werden. Gemäß Dokumentation des Video Encoders (siehe [ADVa, Seite 23]) ist für das ITU-R BT.601-Videoformat der Timing-Mode 2 zu verwenden. Dies geschieht durch die Registereinstellung `0x07` ← `0x0c`. Tests zeigten, dass auch das VS-Signal unverbunden bleiben kann, es reicht in diesem Fall das HS-Signal alleine offensichtlich aus.

Wie schon erwähnt, stellt der entwickelte Gerätetreiber eine Funktion zur Verfügung, die alle für dieses Entwicklungsboard nötigen Konfigurationen durchführt.

Kapitel 7

Softwareentwicklung

Dieser Abschnitt behandelt die Entwicklung der Anwendungssoftware, die auf einem Mikrocontroller der Firma *Analog Devices* des Typs Blackfin lauffähig ist. Sie steuert den logischen Ablauf aller Vorgänge, die zur Erfüllung aller Anforderungen nötig sind, unter Zuhilfenahme aller beteiligten Software- und Hardwarekomponenten. Dies ist somit die Kernkomponente der Anwendung, die viele Aufgaben zeitgerecht zu erfüllen hat. Der eingesetzte Mikrocontroller muss leistungsfähig sein, viele Schnittstellen besitzen und möglichst viele Vorgänge unabhängig voneinander gleichzeitig durchführen können. Realisiert werden kann dies nur mithilfe eines modernen Mikrocontrollers, der viele Funktionen unterstützt, wie zum Beispiel DMA-Transfers (Direct Memory Access).

Die Anwendung erfordert drei mehr oder weniger ähnliche Softwareteile für jeden Funktionstyp. Die Serversoftware muss etwas andere Vorgänge durchführen als der Video-Client oder der Steuer-Client. Es gibt gemeinsame Grundfunktionen, der Ablauf der Vorgänge ist jedoch unterschiedlich. Aus Gründen der Wartbarkeit werden aber alle Softwareteile in einem Softwareimage integriert. Es gibt nur ein Software-Projekt, unterschieden werden die zu aktivierenden Funktionen nur anhand der Hardware, auf der das Programm läuft. So können alle Mikrocontrollermodule (Core-Module, z.B. CM-BF537E) ohne diese neu zu flashen unter den verschiedenen Board-Typen einfach ausgetauscht werden.
Die Software muss somit zuerst die Hardware, bzw. das Board detektieren um die gewünschte Funktion aktivieren zu können. Dies kann etwa mittels einfachem Jumper auf zwei GPIO-Pins durchgeführt werden, oder zum Beispiel über unterschiedliche Verhaltensweisen der zwei Blackfin Video Boards. Beim älteren Board sind die Adressleitungen zum MPEG-4-Chip um ein Bit verschoben, was per Software detektiert werden kann (siehe dazu 5.1.3).

7.1 Blackfin Mikrocontroller

Einige Typen dieser Mikrocontrollerserie der Firma *Analog Devices* unterstützen vieles mehr als die für diese Anwendung benötigten Funktionen. Sie sind mit bis zu 600MHz Core-Clock sehr schnell, haben viele Schnittstellen mit eigenen DMA-Kanälen und ermöglichen eine effiziente Programmierung, rasches Debugging ohne den langsamen Flash-Speicher zu strapazieren. Als integrierte Entwicklungsumgebung wird VisualDSP++ eingesetzt. Diese IDE erleichtert die Entwicklungsarbeit mit vielen nützlichen Funktionen. Der eingebaute Image-Viewer

ermöglicht beispielsweise die Anzeige eines unkomprimierten Bildes aus dem Mikrocontroller-RAM. Eine wesentliche Zeitersparnis konnte auch durch die Möglichkeit des direkten Auslesens und Hineinschreibens von RAM-Bereichen erreicht werden. Dies ist sehr nützlich bei einer schrittweisen Inbetriebnahme der Komponenten. Im Mikrocontroller-RAM abgelegte MPEG-4-Videodaten können so direkt über VisualDSP++ in eine Datei am PC geschrieben und dort verifiziert werden (siehe dazu Abschnitt 7.5).
Umgekehrt ist es auch möglich, die Videodaten für den MPEG-4-Dekoder aus einer Datei am PC in das Mikrocontroller-RAM zu laden, um so speziell für die Fehlersuche immer dieselben Ausgangsbedingungen schaffen zu können. Effizientes Arbeiten wird somit durch diese jederzeit reproduzierbaren Verhältnisse ermöglicht.

7.2 Server

Die Applikationssoftware für den Server hat nun die ihr logisch untergeordnete Schicht der Treiber korrekt zu bedienen. Dabei wird die bereits im Abschnitt 6 beschriebene, selbst entwickelte Treibersoftware zur Ansteuerung der im Server zum Einsatz kommenden Komponenten wie Video Decoder, MPEG-4-Encoder und Funkmodul angesprochen. Die Serversoftware ist nun dafür zuständig, alle benötigten Einzelkomponenten zu einem Gesamtsystem zusammenzusetzen um dadurch die geforderten Anforderungen erfüllen zu können.

In diesem Abschnitt werden nun die Aufgaben der Serverapplikation behandelt und wie die hohen zeitlichen Anforderungen erfüllt werden können.

7.2.1 Initialisierung

Zu Beginn muss natürlich eine Initialisierung erfolgen, das heißt alle Treiber müssen ihre Komponenten initialisieren und in einen betriebsbereiten Zustand versetzen. Darüber hinaus sind alle für das Gesamtsystem benötigten Resourcen anzufordern.
Folgende Routinen sind durchzuführen:

- `initConverter()`: Initialisierung des Video Decoder-Treibers
- `resetIME()`: Software-Reset (MPEG-4-Chip Treiber)
- `loadIME6500FW ()`: hineinladen der Firmware (MPEG-4-Chip Treiber)
- `initReadDMA()`: konfiguriert den DMA-Channel zum Lesen vom MPEG-4-Chip FIFO0-Buffer
- `PwUpReset()`: treibt das Hardware-Reset-Signal für das Funkmodul
- `initSPI()`: konfiguriert die SPI-Schnittstelle zum Funkmodul
- `initSPIComm()`: liefert die Funkchipversion nach der SPI-Initialisierung (Funkchip-Treiber)
- `initNTRX()`: kalibriert und initialisiert Sender und Empfänger (Funkchip-Treiber)

- `initIRQ()`: konfiguriert alle Interruptquellen und registriert ISRs
- `startNTRX()`: konfiguriert IRQ-Event und startet den Empfänger (Funkchip-Treiber)

Die Methoden `initReadDMA()`, `PwUpReset()` und `initSPI()` greifen auf die Mikrocontroller-Hardware zu und sind somit plattformabhängig. Bei einem Wechsel auf einen anderen Mikrocontrollertyp müssen in diesen Methoden die Schnittstellenzugriffe angepasst werden. Leider kann keine strikte Trennung zwischen Applikationssoftware und Hardwarekomponenten gemacht werden, `PwUpReset()` gehört eigentlich logisch gesehen zum Funkchip-Treiber.

Bezüglich logischer Trennung wurde hier ein anderer Weg eingeschlagen. Man trennt die Treiber-Software von den mikrocontrollerspezifischen Hardwarezugriffen, um bei einer eventuell notwendigen Portierung auf einen anderen Mikrocontrollertyp den Anpassungsaufwand für die Treiber in Grenzen zu halten. So können meist annähernd die gleichen Treibersourcen für mehrere Plattformen verwendet werden, was auch den Wartungsaufwand sehr stark reduziert.
Demnach werden alle hardwareabhängigen Zugriffe im Applikationshauptprogramm zusammengefasst, um alle Änderungen im Zuge einer Portierung an einer Stelle durchführen zu können. Auch die Kommunikation des Funkchip-Treibers mit dem Funkchip über das SPI wird durch die Methode `transSPI()` im Hauptprogramm, die das Schreiben und Lesen auf die Schnittstelle durchführt, bewerkstelligt. Der Treiber führt nun einfach diese für ihn externe Funktion aus, um mit dem Funkchip kommunizieren zu können.

`initIRQ()` initialisiert den Interruptcontroller und registriert die aufzurufenden Interrupt-Service-Routinen (ISR). Der Interrupt des Funkmoduls ist hierbei gegenüber dem MPEG-4-Chip-Interrupt zu priorisieren, da sonst zu empfangende Datenpakete relativ leicht verloren gehen, wenn der Funkchip nicht rechtzeitig bedient wird. Beim MPEG-4-Chip ist eine Unterbrechung nicht so kritisch, es bleibt genügend Zeit übrig, um die MPEG-4-Daten rechtzeitig auszulesen.
Der MPEG-4-Encoder löst einen Interrupt aus, wenn der 1024 Byte große FIFO0-Buffer mit Daten voll ist. Unter der Annahme, dass die mittlere Bitrate etwa 700kBit/s beträgt, wäre das im Schnitt alle 11,4ms der Fall. Der Funkchip des Servers hingegen würde durchgehend Videodatenpakete etwa der Größe von 128 Bytes aussenden (die Paketgröße ist konfigurierbar). Nach jedem gesendeten Server-Paket könnte bei Bedarf ein Client seine Steuerdaten an den Server übermitteln. Somit könnte am Server im Extremfall etwa jede Millisekunde (@2MBit/s, 128 Byte Videodaten, 32 Byte Steuerdaten mit Acknowledge-Paket) ein Interrupt des Funkchips ausgelöst werden. Wird nun die Bearbeitung der Service-Routine für den Funkchip-Interrupt auch nicht mal eine Millisekunde lang hinausgezögert, kann dies Schwierigkeiten in der Datenübertragung auslösen. In Abbildung 7.1 ist die zeitliche Abfolge der Datenpakete am Medium ersichtlich.

7.2.2 MPEG-4 Videokompression

Der MPEG-4-Encoder wird über den Treiber aktiviert. Zuerst wird der MPEG-4-Chip und dessen Videoeingang über `initEncoder()` konfiguriert und eine Videoencoder-Softwareinstanz erzeugt. Danach ist diese Instanz nur mehr über `activateEncoder()` in Betrieb zu setzen. Daraufhin beginnt der MPEG-4-Encoder sofort mit der Verarbeitung der Videodaten, sobald

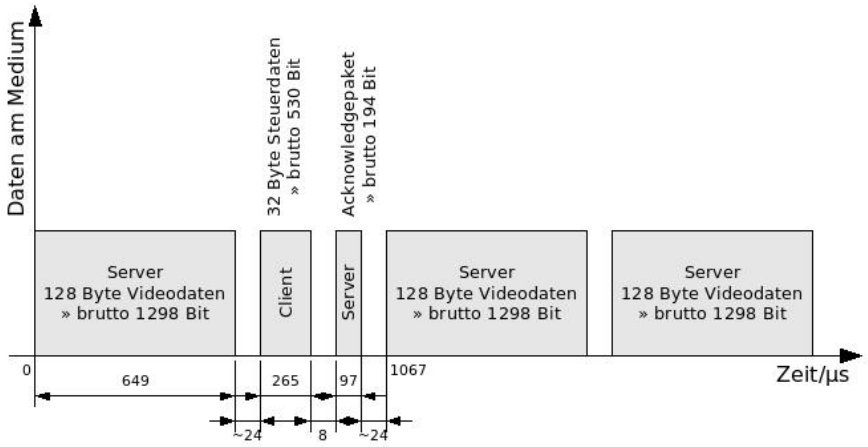

Abbildung 7.1: Beispiel für eine zeitliche Abfolge der Mediumsbelegung im 2MBit/s-Modus

er Daten zur Verfügung hat, schreibt er diese in den FIFO0-Buffer und löst bei korrekter Konfiguration einen Interrupt aus. Wurden vom Mikrocontroller diese ersten 1024 Bytes mittels DMA-Zugriff abgeholt, kann der FIFO0-Buffer und damit der Interrupt wieder aktiviert werden. Der MPEG-4-Encoder kopiert anschließend die nächsten 1024 Bytes in den Buffer und der Transfervorgang beginnt von vorne. Dies geschieht nun solange, bis der Encoder keine neuen Daten mehr zur Verfügung hat, das heißt, bis das aktuelle Paket und damit das aktuelle Bild komplett übertragen wurde. Die nächsten Daten kommen während oder nach der Komprimierung des darauf folgenden Frames.

Die vom MPEG-4-Encoder gelieferten Datenpakete enthalten jeweils ein Bild, wobei zwei Typen relevant sind. Der erste enthält einen I-Frame, der einen intra-coded Frame darstellt und der zweite einen P-Frame, der inter-coded ist (siehe 3.3). Die Datenpakete werden so wie sie sind, inklusive dem acht Byte Paketheader (siehe dazu [IMEa, Figure 7]) über einen MDMA-Zugriff (Memory-DMA) ins RAM des Mikrocontrollers geladen. Dies ist deshalb die Aufgabe der Applikationssoftware, da spezielle Werkzeuge des Blackfin-Mikrocontrollers (MDMA) verwendet werden und die Speicherverwaltung auch Anwendungssache ist.

Die gespeicherten Videodaten können nur begrenzt im RAM verbleiben, da die Live-Videodaten nicht komplett abgelegt werden können. Dies ist auch nicht nötig, da sie gleich per Funk weiter gesendet werden, um ein Video-Live-Streaming mit möglichst geringer Verzögerung zu erreichen. Der verwendete Speicherbereich hat aus diesem Grund auch nur eine relativ begrenzte Größe (siehe auch Abschnitt 7.2.3) und wird als Ringbuffer eingesetzt. Wenn man beim Schreiben also am Ende des Speichers angelangt ist, wird einfach wieder von vorne begonnen.

7.2.3 Senden der MPEG-4-Videodaten

Die im RAM zwischengespeicherten Videodaten sind nun so schnell wie möglich an den Video-Client zu senden. Dazu wird das in Abschnitt 4.7 behandelte Protokoll eingesetzt. Wie

schon in Abschnitt 4.6.2 behandelt, muss man für einen I-Frame in dieser Anwendung etwa mit 60ms Funkübertragungszeit rechnen. Bei einer P-Frame-Übertragungszeit von etwa 30ms würde man einen Zwischenspeicher benötigen, der zumindest einen I-Frame und zwei P-Frames vorübergehend aufnehmen könnte.
Da dies jedoch die absolute Untergrenze in diesem Fall wäre, ist ein mindestens doppelt so großer Speicher mit etwa 40kByte zu empfehlen. Ein noch größerer RAM-Bereich ist dann anzuraten, wenn die Bitrate des MPEG-4-Datenstromes dynamisch durch Rekonfiguration des MPEG-4-Chips verändert wird. Sinnvoll ist dies zum Beispiel dann, wenn die maximale Videoqualität erreicht werden soll, bei welcher die Videodaten gerade noch mit der zur Verfügung stehenden Funkübertragungsrate transportiert werden können. Der Indikator für die Nachjustierung der Bitrate könnte in diesem Fall die Byteanzahl der noch nicht gesendeten Videodaten sein. Ist diese Anzahl im Mittel sehr gering, kann eine höhere Bitrate und damit eine höhere Videoqualität eingestellt werden. Geht die Byteanzahl andererseits langfristig nicht mehr von selbst auf kleinere Werte zurück, muss die Bitrate verringert werden. Für diese dynamische Regelung benötigt man aber eine gewisse Speichergröße, um einen entsprechenden Ausregelungsspielraum zu haben.

7.2.4 Verarbeitung der Steuerdaten

Über die in Abschnitt 4.2.2 beschriebene Schnittstelle werden Steuerdaten empfangen und sind an die anderen Funkknoten weiterzuleiten. Dazu werden sie in 32Byte große Steuerdatenpakete aufgeteilt und als solche per Funkchip ausgesendet. Umgekehrt sind die über Funk empfangenen Steuerdatenpakete eines Clients über dieselbe Schnittstelle wieder auszusenden.

7.3 Video-Client

Der erste Client-Typus hat neben der Steuerdatenverarbeitung, die beim Server in Abschnitt 7.2.4 schon behandelt wurde, auch noch eine andere Aufgabe. Der vom Server per Funk ausgestrahlte Videodatenstrom (siehe 7.2.3) ist in ein Video zurückzuwandeln und über die vorgesehene Ausgangs-Schnittstelle (siehe 4.2.1) zur Verfügung zu stellen.
Die dafür entwickelte Applikation setzt auf die selbst entwickelten Treiberkomponenten (siehe Kapitel 6) für das Funkmodul, den MPEG-4-Decoder und den Video-Encoder auf.

7.3.1 Initialisierung

Die Komponenteninitialisierung gestaltet sich ganz analog der in Abschnitt 7.2.1 beschriebenen. Die Funktion `initConverter()` enthält dabei nicht nur Initialisierungskommandos für den Video Decoder, sondern auch entsprechende Befehle für den Video Encoder (siehe 6.4). Nachdem nur eine unidirektionale I^2C-Verbindung eingesetzt wird, haben die Initialisierungssequenzen bei Nichtvorhandensein der Video Encoder- oder Decoder-Komponenten keine negativen Auswirkungen.

7.3.2 Empfang der MPEG-4-Videodaten

Die vom Server gemäß Protokoll aus Abschnitt 4.7 gesendeten Videodaten werden paketweise vom Funkchip empfangen und der Applikation mittels Interrupt angezeigt. In der Interrupt-Service-Routine (ISR) werden die Pakete aus dem Funkchip gelesen und in einen RAM-Buffer geschrieben. Der Verlust eines oder mehrerer Pakete kann durch das eingesetzte Protokoll leicht detektiert werden. Fehlerhaft empfangene Pakete werden verworfen und gelten daher auch als verloren. Je nach eingesetztem MPEG-4-Decoder können unvollständige Videodaten zur Dekodierung bereitgestellt werden oder es ist der ganze Frame auszulassen.
Da der verwendete MPEG-4-Chip die Dekodierung teilweise falscher Videodaten in den meisten Fällen unangemeldet abbricht, muss bei Unvollständigkeit der Daten gleich der gesamte Frame verworfen werden. Ein einzelnes ausgelassenes Bild merkt der Betrachter des Videos kaum, bei mehreren fehlenden Frames wirkt das Video teilweise nicht mehr ganz flüssig.

7.3.3 Bedienung des MPEG-4-Decoders

Bevor eine Dekodierung erfolgen kann, muss der Video Ausgang und die Decoder-Instanz zuerst mittels `initDecoder()` initialisiert werden. Danach startet die Methode `activateDecoder()` den eigentlichen Decoding-Vorgang.

Die für die erfolgreiche MPEG-4-Video-Dekodierung nötige Kommunikation mit dem MPEG-4-Chip läuft nach folgendem Schema ab.
Der MPEG-4-Chip signalisiert der Applikationssoftware mittels Interrupt, dass sich in dessen internem Decoder-Framebuffer weniger Frames befinden, als das während der Initialisierung spezifizierte Limit. Aus einem Register kann dann der genaue Bufferstand eruiert werden.
Daraufhin wird das Laden des nächsten Frames vorbereitet. Dies geschieht durch Kommunikation über ein spezielles Task-Kommando, das den Typ des Frames (I- oder P-Frame) und dessen Größe in Bytes angibt. Die Übertragung der Videodaten geschieht dann mithilfe des FIFO0-Buffers und dessen ausgelösten Interrupts, welche die Bereitschaft zum Transfer des nächsten FIFO0-Blocks anzeigen. In der zugehörigen ISR wird ein DMA-Transfer vom Mikrocontroller-RAM auf die FIFO0-Datenregisteradresse gestartet.

7.4 Steuer-Client

Dabei handelt es sich um einen vergleichsweise einfachen Client-Typus. Er hat lediglich die Aufgabe, eine Steuerdatenverarbeitung, wie sie in Abschnitt 7.2.4 behandelt wurde, durchzuführen.
Dafür benötigt dieser Client nur die Initialisierung des Funkmoduls und der beteiligten Mikrocontrollerkomponenten, die mit den letzten sechs Einzelschritten aus Abschnitt 7.2.1 erledigt wird.

7.5 Verifikation der MPEG-4-Videodaten

Die vom MPEG-4-Encoder erzeugten MPEG-4-Daten mussten während der Entwicklungsphase laufend verifiziert werden. Besonders zu Beginn, bei der schrittweisen Inbetriebnahme

war eine Funktionsprüfung jeder Einzelkomponente unerlässlich. Aus diesem Grund wurde eine Möglichkeit geschaffen, die generierten Videodaten auch einfach überprüfen zu können.

7.5.1 Videodaten speichern

Dafür werden zuerst die Daten der einzelnen Frames aus dem Mikrocontroller-RAM über das JTAG-Interface und die Entwicklungsumgebung VisualDSP++ direkt in eine Datei auf dem PC gespeichert. Dies wird zum Beispiel ganz einfach über einen Klick mit der rechten Maustaste auf das erste Byte im Fenster "Blackfin Memory" und der Auswahl von "Dump..." durchgeführt. Im erscheinenden Fenster ist noch das Format, die Byteanzahl (Count) und die Zieldatei anzugeben. Rein binäres Speichern ist bis dato leider nicht unterstützt, als Format ist nun "Unsigned Integer" auszuwählen, um Fehlinterpretationen zu vermeiden. Außerdem ist die Option "Write format to file" hier zu deaktivieren und "Stride" sollte auf 1 belassen werden.

7.5.2 Umwandlung in eine MP4-Datei

Aus der im vorigen Abschnitt generierten Datei wird mithilfe einer selbst entwickelten Java-Applikation eine MP4-Datei erzeugt. Wie schon kurz in Abschnitt 3.4 erwähnt, besteht eine MP4-Datei aus einer hierarchischen Struktur von Atomen. Diese Struktur wurde mit der objektorientierten Programmiersprache Java durch Klassen und deren Vererbung nachgebildet. Es wurden alle hierfür benötigten Atomtypen implementiert. Durch die modulare Programmierung ist eine Erweiterung recht einfach möglich.

Die Umwandlung einer Datei namens `video_320x240.dat` würde man wie folgend starten:
`java at.kohlweiss.ime.Converter video_320x240.dat`

Dabei wird eine MP4-Datei mit dem Namen `video_320x240.dat.mp4` geschrieben, die mit allen gängigen Video-Playern abspielbar ist, sofern ein MPEG-4-Codec zur Verfügung steht. Zu empfehlen ist der kostenlose VLC media player (http://www.videolan.org), der alle nötigen Fähigkeiten schon standardmäßig mitbringt und sehr robust bezüglich Datenfehler ist.

Kapitel 8

Ergebnisse und Ausblick

Durch die Tatsache, dass die gesamte Anwendung ein recht komplexes System darstellt, in welchem alle Komponenten für die Erfüllung der geforderten Aufgaben gut zusammenarbeiten müssen, war die praktische Realisierung doch recht zeitraubend. Hinzu kam noch der Umstand, dass der eingesetzte MPEG-4-Chip noch brandneu und unerprobt war. Es gab auch wenige Informationen und falsche Dokumentationen seitens des Chipherstellers. Der rege Emailverkehr mit dem Firmensupport war unerlässlich und erlaubte nur ein schleppendes Vorankommen.

Waren diese zahlreichen Schwierigkeiten schließlich überwunden, lieferte der MPEG-4-Chip eine erstaunlich gute Bildqualität bei moderater Bitrate. Ein Beispiel zeigt Abbildung 8.1, hier ist ein Bild eines Videos mit der Auflösung 320x240 zu sehen. Mit 25 Bildern/s wurde eine Bitrate von 669kBit/s erreicht.

Abbildung 8.1: Schnappschuss eines MPEG-4-komprimierten Videos mit 669kBit/s

Die Funkübertragung der Videodaten und der zeitkritischen Steuerdaten, vor allem auch deren Priorisierung stellt eine Herausforderung für die eingesetzte Funktechnologie dar. Einerseits soll ein konstant großer Videodatenstrom übertragen und andererseits kleinere Steuerdatenpakete dazwischen gesendet werden. Während die Steuerdaten sicher beim Empfänger ankommen müssen, können bei Videodaten gewisse Verluste zugelassen werden. Allerdings

ist aber auch bei Videodaten Vorsicht geboten, da bei der Entwicklung der geforderten Anwendung der MPEG-4-Chip auch als MPEG-4-Decoder verwendet wurde, und dieser für eine durchgängige Dekodierung eine fehlerfreie Übertragung aller Videodaten benötigt. Sind hier nur kleine Fehler im Bitstrom, stoppt der Decoder ohne es vom Mikrocontroller aus unmittelbar zu merken. Bessere Implementierungen eines MPEG-4-Decoders, wie zum Beispiel in der PC-Software VLC media player zu finden, ignorieren die fehlerhaften Daten so gut es geht (eventuell mit Bildstörungen), setzten aber jedenfalls die Dekodierung fort.

8.1 Implementierung

Die Implementierung des Protokolls aus Abschnitt 4.7 stellte kein großes Problem dar, allerdings kann das Medium nicht ganz so effizient genutzt werden. Zusätzlich einzurechnende Zeiten für die Übertragung der zu sendenden Daten vom Mikrocontroller zum Funkchip verringern die maximal erreichbare Bitrate, wie in Abbildung 8.2 ersichtlich.

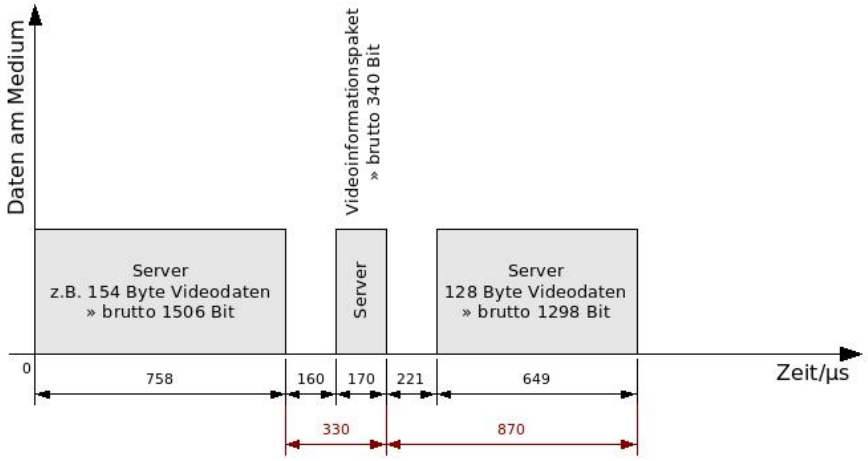

Abbildung 8.2: Gemessene Zeitabfolge am Medium

Der Server kann demnach nicht im schnellst möglichen Abstand von etwa 24μs erneut senden, wie die Abbildung 4.4 zeigt, sondern bei einem nachfolgenden 128Byte-Paket erst nach 221μs. Das ist die Zeit, die für den Transfer der Daten und der Sende-Konfiguration über SPI und für die ausführende Anwendungssoftware im Mikrocontroller benötigt wird. Statt einer Nettodatenrate von theoretisch 1,45MBit/s für die 128Byte-Pakete wird praktisch nur 1,12MBit/s erreicht (natürlich im 2MBit/s-Modus). Bei größeren Datenpaketen ist mit weniger Einbußen zu rechnen, da der Transfer der über 128Byte hinausgehenden Daten erst laufend während dem Senden stattfindet. Bei einer Paketgröße von 512Byte würde man statt theoretisch 1,788MBit/s praktisch 1,614MBit/s erreichen.

8.2 Server-Client Timing

Die Abfolge aller Ereignisse im Server und in den verschiedenen Clients ist sehr komplex. Die Extremwerte der Steuerdaten-Verzögerung zeigt Abbildung 8.3, die alle betreffenden Vorgänge im Server und in einem Client ohne Videodatenverarbeitung darstellt.

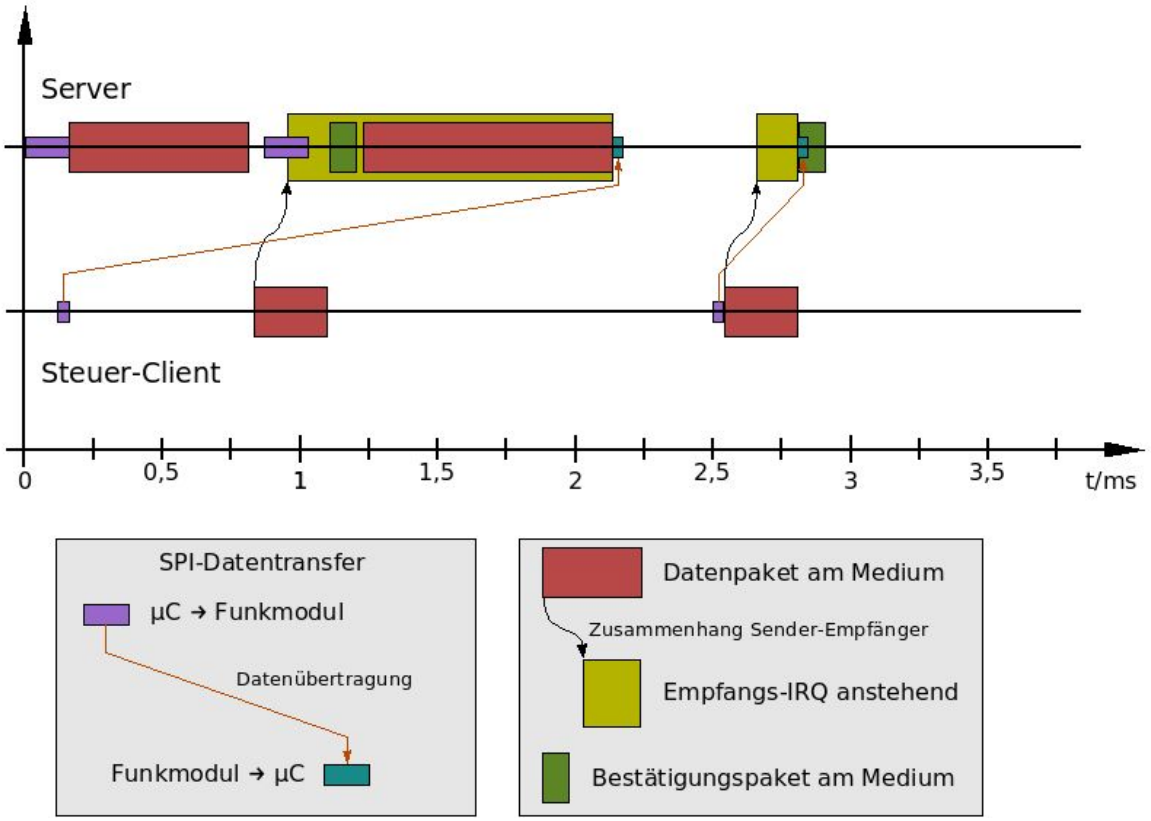

Abbildung 8.3: Server-Steuerclient Timing

Aufgrund der Intention des Servers, ein durchgehendes Videostreaming durchzuführen, erhöht sich die Gesamtdauer der Vorgänge für die Steuerdatenübermittlung. Die maximal auftretende End-to-End Verzögerung eines Steuerdatenpakets ist in Tabelle 8.1 ersichtlich.
Zu beachten ist hierbei, dass die maximale Videodatenpaketgröße 192 Bytes Nutzdaten ist (siehe Abschnitt 4.7.2). Damit ergibt sich die maximale Wartezeit auf das Freiwerden des Mediums zu 905μs. Nicht mit eingerechnet ist hier die Verarbeitungszeit der Steuerdaten im Mikrocontroller. Wenn dieser mit dem Senden oder Empfangen beschäftigt ist, kann die Bedienung der Steuerdatenschnittstelle auf Server- und Clientseite noch dementsprechend zusätzlich Zeit beanspruchen.

Die minimale Verzögerung tritt bei freiem Medium auf und wenn der Server gerade keine Videodaten zum Senden zur Verfügung hat (siehe Abbildung 8.3, rechter Bereich). In diesem Fall verzögern sich die Steuerdaten vom Sender zum Empfänger nur um etwa 345μs.

Ein grober Überblick über die noch komplexeren Vorgänge in einem Client mit zusätzlicher Videoverarbeitung ist in Abbildung 8.4 zu sehen.

Vorgang	**maximale Verzögerung/μs**
SPI-Transfer $\mu C \rightarrow$ Funkmodul	40
warten auf freies Medium	905
IFS (InterFrame Spacing)	24
Übertragung am Medium	265
warten auf Bearbeitung im Server	778
SPI-Transfer Funkmodul $\rightarrow \mu C$	40
Summe	**2.051**

Tabelle 8.1: Maximale End-to-End Verzögerung eines Steuerdatenpakets

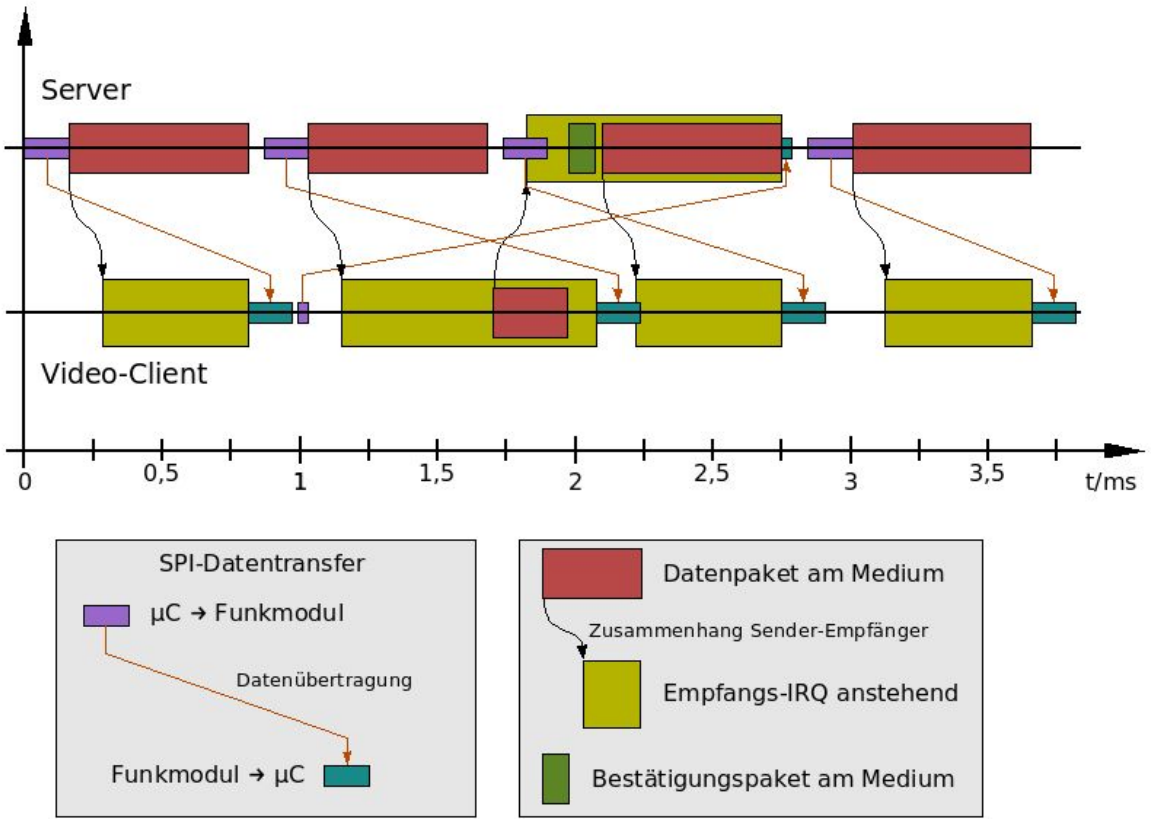

Abbildung 8.4: Server-Videoclient Timing

Wie aus dieser Abbildung und der Tabelle 8.1 zu schließen, ist der Sende- und Empfangsvorgang in der Applikation nicht ganz optimal implementiert. Der Grund dafür ist, dass der Datentransfer zwischen Mikrocontroller und Funkmodul beim Senden und Empfangen nicht gleichzeitig durchführbar ist. Für beide Vorgänge muss dieselbe Schnittstelle exklusiv verwendet werden.
Eine zusätzliche Schwierigkeit ist auch, dass ein einzelner Schreib- oder Lesetransfer über das SPI nicht durch einen anderen unterbrochen werden darf. Dies würde beispielsweise auftreten, wenn Videodaten in den Funkchip geschrieben werden und währenddessen am Server ein Steuerdatenpaket ankommt und den Empfangs-Interrupt auslöst. In der dadurch umgehend aktivierten ISR würden sofort Lesetransfers über das SPI gestartet werden.

Um die atomare Transaktion über das SPI zu schützen wird während dem Senden eines Datenpakets der Empfangs-Interrupt im Mikrocontroller deaktiviert und erst nach dem Sendevorgang wieder aktiviert. Ein Sendevorgang beginnt vor dem Transfer der Zieladresse und der zu sendenden Daten in das Funkmodul und endet nach der Meldung des Moduls über den erfolgten Abschluss (Flag `TxEnd` des *Transmitter Interrupt Status*-Registers, siehe [NTRa, Abschnitt 9.14]).
Es kann demnach kein Auslesen des Empfangs-Buffers erfolgen, wenn gerade ein Sendevorgang läuft und zuvor der Empfangs-Interrupt noch inaktiv war. Diese Verhaltensweise ist zwar etwas ineffizient, weil der Sender während dem Warten auf das Freiwerden des Mediums oder nach erfolgtem Transfer der zu sendenden Daten den Empfangs-Buffer bereits auslesen könnte. Ein flexiblerer Ablauf aller Vorgänge ist jedoch mit erheblich mehr Softwareaufwand verbunden.

8.3 Anforderungserfüllung

Die in Abschnitt 4 festgehaltenen Anforderungen an die zu entwickelnde Anwendung sind durchaus recht anspruchsvoll. Vor allem die Implementierung des Übertragungsprotokolls und die Sicherstellung der korrekten Steuerdatenübertragung während dem Video-Streaming sind nicht einfach zu realisieren und benötigen eine maximale Kontrolle der Sende- und Empfangsoperationen. Der eingesetze Funkchip ermöglicht all dies und bietet noch viele zusätzliche Funktionen an. Wie im letzten Abschnitt schon ersichtlich, konnte somit die Anforderung der maximalen Reaktionszeit für Steuerdaten bestens erfüllt werden.

Auch die maximale Verzögerung des Videos von 500ms konnte eingehalten werden, eine Verringerung ist durch Optimierung aller Vorgänge durchaus noch möglich. Die Tabelle 8.2 zeigt eine grobe Aufschlüsselung der auftretenden Verzögerungen in den einzelnen Verarbeitungsschritten, wobei es sich hierbei aber nicht um genaue Messwerte handelt. Mit den zur Verfügung stehenden Mitteln war keine genaue Messung möglich, lediglich die Gesamtverzögerung konnte mit ungefähr 0,5s oder sogar etwas darunter beobachtet werden.

Verarbeitungsschritt	**Verzögerung**
Digitalisierung	40ms
Komprimierung	100ms
Transfer vom Encoder zum Funkmodul	10ms
Funkübertragung	80ms
Transfer vom Funkmodul zum Decoder	10ms
Dekomprimierung	100ms
Analogwandlung	40ms
Summe	**380ms**

Tabelle 8.2: Erreichte Video-Verzögerungszeiten

8.4 Anwendungsgebiete

Das Anforderungsprofil dieser Anwendung ist auch in vielen anderen Applikationen wiederzufinden. Die entwickelte Technik kann überall dort eingesetzt werden, wo die Funkübertragung eines Videos und ein zeitgerechter Datentransfer über denselben digitalen Übertragungskanal benötigt wird. Hier seien nur zwei Beispiele angeführt, bei welchen der Einsatz eines solchen Systems absolut Sinn macht.

8.4.1 Kamerafernsteuerung

Die mit diesem System erreichte Mobilität macht sich überall dort bezahlt, wo laufend Ortswechsel durchgeführt werden und die Verkabelung viel Zeit beanspruchen würde.
Ein praxisnahes Beispiel ist der Einsatz einer Fernsteuerung für Kameras, die bei Filmdrehabeiten verwendet werden. Hier kann ein Drehortwechsel bei Verwendung einer Funkfernsteuerung viel schneller erfolgen. Systeme, die nur eine Steuerung der Kamera erlauben, sind heutzutage schon zahlreich im Einsatz. Ein System, das mit nur einer eingesetzten digitalen Funktechnologie aber auch das aufgenommene Video zur Kontrolle auf die Fernsteuerung überträgt und die Steuerung dadurch jedoch nicht negativ beeinflusst, ist derzeit am Markt noch nicht zu finden.

8.4.2 Prozessüberwachung und Steuerung

Die entwickelte Technik könnte auch in der Industrie eingesetzt werden, wo bestimmte Fertigungsabläufe videoüberwacht werden müssen und eine zeitgerechte Steuerung über denselben Funkkanal erforderlich ist.
Prinzipiell wird im Industriebereich gerne verkabelt, weil sich der Initialaufwand schon bald rechnet. Wenn aber Mobilität für einen fahrenden Roboter unbedingt gefordert ist, werden derzeit beispielsweise Schienenkontaktlösungen eingesetzt, die jedoch wieder viele andere Nachteile haben. Durch die robuste Auslegung von nanoNET wäre diese Funklösung aber auch für rauhe Industrieumgebungen durchaus geeignet.

8.5 Verbesserungsmöglichkeiten

Das System lässt sich natürlich noch optimieren, um die Anforderungen noch besser erfüllen zu können. Einige Angriffspunkte seien hier noch festgehalten.

8.5.1 Videokompression

Um einen optimalen Kompromiss zwischen verfügbarer Datenübertragungsrate und der Videoqualität zu erreichen, könnte man die Videobitrate dynamisch nachstellen. Ist noch genügend Reserve auf der Funkschnittstelle vorhanden, würde man durch eine einfache Rekonfiguration des MPEG-4-Encoders und einer damit erhöhten Videobitrate auch die Videoqualität verbessern. Wenn umgekehrt die vom Encoder gelieferten Daten aus Mangel an Übertragungsrate nicht schnell genug gesendet werden können, ist auch eine Verringerung der Videobitrate

gleichermaßen möglich.
Dieser Regelungsalgorithmus ist jedoch vor allem unter der Nebenbedingung, dass eine minimale Videoverzögerung erreicht werden soll, nicht ganz einfach zu implementieren.

8.5.2 Videodekompression

Wie in dieser Arbeit schon erwähnt, ist der eingesetzte MPEG-4-Chip als MPEG-4-Decoder durch seine geringe Fehlertoleranz nicht die optimale Wahl. Die Implementierung eines MPEG-4-Decoders in Software und die direkte Integration in die Anwendungssoftware wäre zu empfehlen. Der verwendete Mikrocontroller BF537 könnte mit seiner verfügbaren Rechenleistung die MPEG-4-Dekodierung mit einem gut optimierten Algorithmus durchaus erledigen. Der wesentliche Vorteil einer solchen Lösung wäre, dass der Videoclient weniger Hardware benötigen würde (kein MPEG-4-Chip und keine zusätzlichen SDRAM-Bausteine).

8.5.3 Funkübertragung

Die Funkübertragung an sich arbeitet generell schon recht zufriedenstellend. Es ist jedoch bei Einsatz eines besseren MPEG-4-Decoders noch eine Feinabstimmung möglich. Je nach Fehlertoleranz des Decoders können die Videodaten im Broadcast-Modus ohne oder mit Forward-Error-Correction übertragen werden.
Die Verwendung von FEC bedeutet allerdings einiges an Einbußen bezüglich der verfügbaren Übertragungsrate. So ist bei 128Byte großen Videodatenpaketen ohne FEC im 2MBit/s-Modus eine Bitrate von 1,12MBit/s erreichbar, mit FEC jedoch nur mehr 746kBit/s. Dadurch, dass der hier als Decoder eingesetzte MPEG-4-Chip keine Datenfehler erlaubt, konnte für ungehindertes Videostreaming leider nicht der Broadcast-Modus eingesetzt werden. Es wurde die Funktion Automatic-Repeat-Request aktiviert, um jedes Videodatenpaket garantiert fehlerfrei zu übertragen (der Empfänger wurde so konfiguriert, dass er den Empfang nur bestätigt, wenn beide CRCs erfolgreich waren). Dabei ist aber nur eine maximale Bitrate von 1MBit/s zu erreichen.

Eine kleine Optimierung ist auch beim Transfer der Daten zwischen Mikrocontroller und Funkchip möglich. Weil es nicht unbedingt notwendig war, wurde beim SPI-Transfer auf den DMA-Controller verzichtet. Durch dessen Einsatz würde sich auch die maximal erreichbare Datenübertragungsrate auf der Funkschnittstelle erhöhen, da eine effizientere Nutzung des Mediums durch verkürzte Vorbereitungszeit vor dem Senden eines Videodatenpakets erreicht wird (siehe Abbildung 8.2).
Statt einer Periodendauer von 870μs beim Senden von 128Byte Videodaten könnte man durch Einsatz des DMA-Controllers für den SPI-Transfer 790μs erreichen, was einer maximalen Nettobitrate von 1,236MBit/s entspricht. Dies ist deshalb möglich, weil ein SPI-Transfer innerhalb einer gewöhnlichen Schleife im Anwendungsprogramm zwischen den einzelnen übertragenen Bytes eine Verzögerung von etwa 500ns verursacht. Beim Transfer durch den DMA-Controller tritt diese Verzögerung nicht auf, wodurch eine Zeitersparnis von circa 80μs bei der Übertragung von 128Byte möglich ist.

Bezüglich SPI gibt es ein weiteres zeitliches Einsparungspotential. Bei einer besseren Koordination von Sende- und Empfangsvorgängen wäre noch etwas Zeit gut zu machen. Wie schon

in Abschnitt 8.2 gezeigt, wird nicht jede Wartezeit der beteiligten Mikrocontroller so effizient wie möglich genutzt. Da die Anforderungen jedoch trotzdem erfüllt werden, im speziellen die Steuerdatenverzögerung, wurde in diesem Fall eine dadurch einfachere Applikationssoftware bevorzugt.
Um die hier liegen gebliebene Zeit auch noch auszunützen müsste eine atomare Transaktion über jeden einzelnen SPI-Transfer gespannt werden. Beim Senden würde man auf die prinzipielle Deaktivierung des Empfangs-Interrupts verzichten und könnte somit zwischendurch auch den Empfänger bedienen.
Die aktuelle Anwendung wartet nach dem Senden auch auf die Meldung des Funkchips, dass alle Daten abgeschickt wurden (`TxEnd`-Flag). Bei Verwendung des ARQ-Modus für die Steuerdaten beinhaltet dies auch den Bestätigungsvorgang. Das bedeutet, dass der Steuer-Client beim Senden der Steuerdaten auch auf das Bestätigungspaket des Empfängers wartet ehe er mit der Applikationssoftware fortfährt. Wenn das Datenpaket nicht beim ersten Mal ankommt, ist die Wartezeit dementsprechend länger. Antwortet der Empfänger garnicht, bricht der Steuer-Client nach der eingestellten Maximalanzahl an Sendeversuchen ab. Eine effizientere Nutzung der verfügbaren Mikrocontrollerkapazität wäre nun dadurch möglich, dass nicht bis zur Beendigung des Sendens gewartet wird, sondern die Synchronisation der Sendevorgänge über einen Interrupt erfolgt, der durch das vom Funkchip gemeldete `TxEnd`-Ereignis ausgelöst wird. Der Funkchip ist dann dafür entsprechend zu initialisieren.

Derzeit werden die vom MPEG-4-Encoder gelieferten Datenpakete (siehe [IMEa, Figure 7]) ohne Weiterverarbeitung direkt über die Funkschnittstelle gesendet. Da derselbe MPEG-4-Chip auch für die Dekodierung eingesetzt wird, können die Daten so gleich wieder verwendet werden. Bei Einsatz eines anderen Decoders, eventuell einem Software-Decoder im Mikrocontroller, wäre es aber besser, einen standardisierten Datenstrom zur Verfügung zu stellen. Der MPEG-2 Transport Stream (MPEG-2 TS) wurde genau für diesen Zweck entwickelt und wird beispielsweise auch beim digitalen Satellitenfernsehen oder bei der digitalen terrestrischen Fernsehtechnik eingesetzt. Eine Umsetzung der vom MPEG-4-Encoder gelieferten Daten in einen MPEG-2 TS wäre hierbei zu empfehlen, da auch Verbesserungen bei fehlerhaft übertragenen MPEG-4-Videodaten zu erwarten sind.

8.6 Ausblick

Die vorliegende Diplomarbeit und vor allem deren praktischer Teil analysierte den Einsatz einer eigenen Hardware für die Echtzeit-Videokompression. Es wurde dabei nicht mit übermäßigem Aufwand für die Inbetriebnahme gerechnet, da diese Hardwarelösung bereits einen Kompressionsalgorithmus betriebsbereit zur Verfügung stellen sollte.
Viele Schwierigkeiten und Unstimmigkeiten schaltungs- und softwaretechnischer Natur mussten jedoch überwunden werden, die sehr viel Entwicklungszeit kosteten.
In Anbetracht des Umstands, dass die Hardwarelösung auch zusätzliche Bauteile und damit zusätzliche Kosten, sowie Platz- und Stromverbrauch verursacht, wäre eine reine Softwarelösung für den Kompressionsalgorithmus, optimiert auf die Zielplattform vermutlich in vielen Belangen effizienter gewesen.

Mit einem schnelleren und leistungsfähigeren Mikrocontroller, wie beispielsweise mit dem Dual-Core BF561 von *Analog Devices* könnte unter Einsatz eines geeigneten und optimierten Kompressionsalgorithmus eine Videokompression und Dekompression in Software durchgeführt werden.

Ein wesentlicher Fortschritt für die in dieser Arbeit behandelte Anwendung könnte auch durch die neue Funktechnologie UWB erreicht werden. Dieser relativ neue Standard erreicht durch eine extreme Bandbreitennutzung, jedoch mit geringer Sendeleistung, sehr hohe Datenraten und eine sehr gute Störsicherheit.
Durch diese hohen Datenraten (bis zu 1.320MBit/s) würde er auch eine unkomprimierte Videoübertragung ermöglichen, womit diese Technologie für die bearbeitete Anwendung vermutlich besser geeignet wäre. Die Auswirkungen auf die Videoverzögerung wären durch die fehlende Kompression und Dekompression überaus vorteilhaft. Ein entsprechend schneller Datentransfer zwischen Video Decoder und Funkmodul muss dabei vorausgesetzt werden.
In den USA ist UWB gemäß FCC zumindest unter gewissen Randbedingungen schon seit Februar 2002 nutzbar. Da alle anderen Regulierungsbehörden jedoch durch den radikalen Ansatz von UWB für das Zulassungsprozedere viel Zeit benötigten, ist erst jetzt ab Ende 2006 die lizenzfreie Nutzung in Europa zu erwarten.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ADVa] *Digital PAL/NTSC Video Encoder with 10-Bit SSAF^{TM} and Advanced Power Management ADV7170/ADV7171* 26, 45

[ADVb] *Multiformat SDTV Video Decoder ADV7183B* 26, 43, 44

[ES98] Effelsberg, Wolfgang ; Steinmetz, Ralf: *Video Compression Techniques.* dpunkt - Verl. für digitale Technologie, 1998 13

[IMEa] *IME65XX Firmware Manual, Version 1.00* 25, 39, 41, 42, 49, 60

[IMEb] *Real-time MPEG4 Multimedia Codec IME65XX* 26, 39, 40

[Mil95] Milde, Torsten: *Videokompressionsverfahren im Vergleich.* dpunkt - Verl. für digitale Technologie, 1995 16

[MPE] *ISO/IEC 14496-1: Information technology - Coding of audio-visual objects - Part 1: Systems* 17

[NAL] *nanoLOC TRX Transceiver (NA5TR1) Datasheet Version 1.00* 7, 8, 22

[NAS] *nanoNET Serial Peripheral Interface Specification for NA2TR1 Version 1.04* 11

[NTRa] *nanoNET TRX (NA2TR1) Register Description Version 1.04* 12, 38, 57

[NTRb] *nanoNET TRX PHY and MAC System Specifications* 11, 23

[PE02] Pereira, Fernando ; Ebrahimi, Touradj: *The MPEG-4 Book.* Prentice Hall PTR, 2002 15, 17

[Ste05] Stepping, Christoph: *Drahtlose Netze.* J. Schlembach Fachverlag, 2005 7, 20

[Wai02] Waitz, Martin K.: *Funkmodule im Echtzeitbetrieb*, Institute of Computer Technology, Vienna University of Technology, Diplomarbeit, 2002 2, 6, 7, 22

[WB05] Wörn, Heinz ; Brinkschulte, Uwe: *Echtzeitsysteme.* Springer, 2005 6, 19

Anhang A

Funkchiptreiber Quellcode

```
/**
 * @file ntrxInit.c
 *
 * contains methods for initialization of the TRX-chip
 *
 * @author Georg Kohlweiss, 9926105, E754, Vienna University of Technology
 * @date 2005-2006
 */

#include "ntrxInit.h"
#include "ntrxComm.h"

/**
 * TRX-chip has to be initialized with this values to be able to transmit
 * (values out of nanotron source code).
 * The values are chip-version dependent, ChirpBytes[0] for chip 0x6301 (board1),
 * ChirpBytes[1] for chip 0x6101 (board2)
 */
byte ChirpBytes[2][3][CHIRPDATACOUNT] = {
                                        {
                                            {0x20,  // CONFIG_SAW_E_SERIES
                                             0xDF,
                                             ...
                                             ...
                                             ...
                                             0xDF,
                                             0xDB}
                                        }
                                    };

/**
 * mapping of desired output power to the allowed steps (out of nanotron source code)
 */
const byte allowedPower[64] =
    {
        0, 0, 0, 0, 0, 0,
        1, 1, 1, 1, 1, 1,
        2, 2, 2, 2, 2, 2,
        3, 3, 3, 3, 3, 3,
        4, 4, 4, 4, 4, 4,
        5, 5, 5, 5, 5, 5,
        21,21,21,21,21,21,
        22,22,22,22,22,
        23,23,23,
        39,39,39,
        40,40,
        41,41,
        57,
        58,
        59,
        60,
        61,
        62,
```

```
        63
    };


/**
 * writes the used synchronization sequence to TRX-chip registers (out of nanotron source code)
 */
void setSyncWord ()
{
    byte syncWord[] = {0x96,0x69,0x56,0xDB,0x54,0x96,0x24,0xAB};

    trans (INSTR_WR |8, Silbadero_TxRxSyncWord_O, syncWord);
}

/**
 * writing desired output power for transmission to the TRX-chip
 *
 * @param value output power level between 0x00 and 0x3F, values>0x3F were changed to this maximum value
 */
void setOutputPower(byte value)
{
    if (value > 0x3F)
        value = 0x3F;

    writePart (Silbadero_RfTxOutputPower0_O,
            Silbadero_RfTxOutputPower0_MSB, Silbadero_RfTxOutputPower0_LSB,
            allowedPower[value]);

    writePart (Silbadero_RfTxOutputPower1_O,
            Silbadero_RfTxOutputPower1_MSB, Silbadero_RfTxOutputPower1_LSB,
            allowedPower[value]);
}

/**
 * writing chirp-bytes used by the transmitter to the TRX-chip
 *
 * @param chirpBytes pointer to a 2D-array of chirp-bytes to be written to the TRX-chip
 */
void setRfChirpGenData(byte chirpBytes[3][CHIRPDATACOUNT])
{
    byte i, n;
    byte data[2];

    // enable local osc
    writeBit(Silbadero_RfLoEn_O, Silbadero_RfLoEn_B, TRUE);

    // enable frequency divider (needed for chirp generator)
    writeBit(Silbadero_RfLoDiv10En_O, Silbadero_RfLoDiv10En_B, TRUE);

    // init the chirp-generator
    writeBit(Silbadero_RfLoChirpGenClkEn_O, Silbadero_RfLoChirpGenClkEn_B, TRUE);
    writeBit(Silbadero_RfChirpReset_O, Silbadero_RfChirpReset_B, TRUE);
```

```
    writeBit(Silbadero_RfChirpReset_O, Silbadero_RfChirpReset_B, FALSE);

    for (i = 0; i < 3; i++)
    {
        data[1] = 0x2 << i;

        for (n = 0; n < CHIRPDATACOUNT; n++)
        {
            data[0] = chirpBytes[i][n];
            trans(INSTR_WR |2, Silbadero_RfChirpGenData_O, data);
        }
        writeBit(Silbadero_RfChirpGenAddrInit_O, Silbadero_RfChirpGenAddrInit_B, TRUE);
    }

    writeBit(Silbadero_RfLoChirpGenClkEn_O, Silbadero_RfLoChirpGenClkEn_B, FALSE);
    writeBit(Silbadero_RfLoDiv10En_O, Silbadero_RfLoDiv10En_B, FALSE);
    writeBit(Silbadero_RfLoEn_O, Silbadero_RfLoEn_B, FALSE);
}

/**
 * adjusting base frequency of the receiver,
 * calibrates the values of the 12 switched capacitors according to
 * the required RX target frequency of the local oscilator
 *
 * @param frq should be a fixed frequency value of 0x0F48 for legal use,
 * decoded frequency value only known to the producer (nanotron),
 * some tests with a spectrum analyzer showed a frequency dependency
 * of about 10Mhz/0x04
 *
 * @return the determined values of the capacitors
 */
word rx_lo_adj (word frq)
{
    word rx_lo_caps;

    writeBit(Silbadero_RfLoEn_O, Silbadero_RfLoEn_B, TRUE);
    writeBit(Silbadero_RfLoDiv10En_O, Silbadero_RfLoDiv10En_B, TRUE);
    writeBit(Silbadero_RfLoAdjClkEn_O, Silbadero_RfLoAdjClkEn_B, TRUE);
    writeBit(Silbadero_RfLoRxMode_O, Silbadero_RfLoRxMode_B, TRUE);

    delay_ms(10);

    writeWord(Silbadero_RfLoAdjIncValue_O, frq);

    delay_ms(10);  // wait at least 771 us

    rx_lo_caps = readWord(Silbadero_RfLoRxCaps_O);

    writeBit(Silbadero_RfLoRxMode_O, Silbadero_RfLoRxMode_B, FALSE);
    writeBit(Silbadero_RfLoAdjClkEn_O, Silbadero_RfLoAdjClkEn_B, FALSE);
    writeBit(Silbadero_RfLoDiv10En_O, Silbadero_RfLoDiv10En_B, FALSE);
    writeBit(Silbadero_RfLoEn_O, Silbadero_RfLoEn_B, FALSE);
```

```
    return rx_lo_caps;
}

/**
 * adjusting base frequency of the transmitter,
 * calibrates the values of the 12 switched capacitors according to
 * the required TX target frequency of the local oscilator
 *
 * @param frq should be a fixed frequency value of 0x090B for legal use,
 * decoded frequency value only known to the producer (nanotron),
 * some tests with a spectrum analyzer showed a frequency dependency
 * of about 10Mhz/0x04
 *
 * @return the determined values of the capacitors
 */
word tx_lo_adj (word frq)
{
    word tx_lo_caps;

    writeBit(Silbadero_RfLoEn_O, Silbadero_RfLoEn_B, TRUE);
    writeBit(Silbadero_RfLoDiv10En_O, Silbadero_RfLoDiv10En_B, TRUE);
    writeBit(Silbadero_RfLoAdjClkEn_O, Silbadero_RfLoAdjClkEn_B, TRUE);
    writeBit(Silbadero_RfLoRxMode_O, Silbadero_RfLoRxMode_B, FALSE);

    delay_ms(10);

    writeWord(Silbadero_RfLoAdjIncValue_O, frq);

    delay_ms(10);    // wait at least 771 us

    tx_lo_caps = readWord(Silbadero_RfLoTxCaps_O);

    writeBit(Silbadero_RfLoAdjClkEn_O, Silbadero_RfLoAdjClkEn_B, FALSE);
    writeBit(Silbadero_RfLoDiv10En_O, Silbadero_RfLoDiv10En_B, FALSE);
    writeBit(Silbadero_RfLoEn_O, Silbadero_RfLoEn_B, FALSE);

    return tx_lo_caps;
}


/**
 * finds out and sets the value of the chirp DAC filter
 *
 * @param chirp_caps_in pointer to LO's capacitors store
 *
 * @return TRUE if the real filter value was found,
 *         FALSE if adjustment aglorithm failed
 */
byte rcosc_adj (byte* chirp_caps_in)
{
    byte i,j;
    word caps_sum;
    byte rc = FALSE;
```

```
    byte chirp_caps = *chirp_caps_in;

    writeBit(Silbadero_RfTxEn_O, Silbadero_RfTxEn_B, TRUE);
    writeBit(Silbadero_RfTxFctEn_O, Silbadero_RfTxFctEn_B, TRUE);

    for (i=0; i<15; i++)
    {
        writePart(Silbadero_RfTxChirpFilterCaps_O,
                Silbadero_RfTxChirpFilterCaps_MSB, Silbadero_RfTxChirpFilterCaps_LSB,
                chirp_caps);

        caps_sum = 0;
        for (j=0; j<4; j++)
        {
            writeBit(Silbadero_RfTxFctMeasCmd_O, Silbadero_RfTxFctMeasCmd_B, TRUE);

            delay_ms(1);   // wait at least 2.8 us

        caps_sum += readPart(Silbadero_RfTxFctCnt_O, Silbadero_RfTxFctCnt_MSB, Silbadero_RfTxFctCnt_LSB);
        }
        if (caps_sum > RCOSC_UPPERLIMIT)
        {
            chirp_caps--;
        }
        else if (caps_sum < RCOSC_LOWERLIMIT)
        {
            chirp_caps++;
        }
        else
        {
            rc = TRUE;
            (*chirp_caps_in) = chirp_caps;
            break;
        }
    }
    writeBit(Silbadero_RfTxFctEn_O, Silbadero_RfTxFctEn_B, FALSE);
    writeBit(Silbadero_RfTxEn_O, Silbadero_RfTxEn_B, FALSE);

    writePart(Silbadero_RfTxChirpFilterCaps_O,
            Silbadero_RfTxChirpFilterCaps_MSB, Silbadero_RfTxChirpFilterCaps_LSB,
            *chirp_caps_in);

    return rc;
}

/**
 * this method should be implemented externally
 * to drive the required initializing signal PowerUpReset.
 * this signal has to be active (low) a certain time (e.g. 200ms)
 * and additionally inactive (high) again for a certain time (e.g. 200ms)
 *
 * has to be called before the first communication
 * with the nanotron chip over SPI occurs
```

```
 */
extern void PwUpReset(void);


/**
 * manages calibration of receiver and transmitter,
 * calls specific methods, calculates and writes RfTxPaBiasAdj-value
 */
void calibrateNTRX()
{
    byte caps;
    byte bias;
    word rx_lo_caps, tx_lo_caps;

    rx_lo_caps = rx_lo_adj (0x0F48); // out of mydata.c: rx_inc_val[] = {0x48,0x0F};
    tx_lo_caps = tx_lo_adj (0x090B); // out of mydata.c: tx_inc_val[] = {0x0B,0x09};

     caps = RfTxChirpFilterCaps_DEFAULT; // out of trx_default_settings.h: #define RfTxChirpFilter-
Caps_DEFAULT 0x6 for 1099
    (void)rcosc_adj (&caps);     /* if rc=FALSE exception ? */

    bias = ((byte)(((64755-2317*caps)*(8371+tx_lo_caps)
                - 368550000)/23034375));
  writePart (Silbadero_RfTxPaBiasAdj_O, Silbadero_RfTxPaBiasAdj_MSB, Silbadero_RfTxPaBiasAdj_LSB,
             bias);

  writePart (Silbadero_RfTxPaFreqAdj_O, Silbadero_RfTxPaFreqAdj_MSB, Silbadero_RfTxPaFreqAdj_LSB,
             RfTxPaFreqAdj_DEFAULT);
}

/**
 * sets chirp symbols and calls the chirp-bytes writing method
 *
 * @param ver chip version read previously out of chip for selecting the right chirp-bytes array
 */
void setNTRXChirp(word ver)
{
    /* init station values */
  writePart(Silbadero_TxRxChirpMatrix0_O, Silbadero_TxRxChirpMatrix0_MSB, Silbadero_TxRxChirpMatrix0_LSB,
            Silbadero_TxRxChirpDown_VC);
  writePart(Silbadero_TxRxChirpMatrix1_O, Silbadero_TxRxChirpMatrix1_MSB, Silbadero_TxRxChirpMatrix1_LSB,
            Silbadero_TxRxChirpUp_VC);
  writePart(Silbadero_TxRxChirpMatrix2_O, Silbadero_TxRxChirpMatrix2_MSB, Silbadero_TxRxChirpMatrix2_LSB,
            Silbadero_TxRxChirpFoldMinus_VC);
  writePart(Silbadero_TxRxChirpMatrix3_O, Silbadero_TxRxChirpMatrix3_MSB, Silbadero_TxRxChirpMatrix3_LSB,
            Silbadero_TxRxChirpOff_VC);

  writePart(Silbadero_TxPreTrailMatrix0_O, Silbadero_TxPreTrailMatrix0_MSB, Silbadero_TxPreTrailMatrix0_LSB,
            0);
  writePart(Silbadero_TxPreTrailMatrix1_O, Silbadero_TxPreTrailMatrix1_MSB, Silbadero_TxPreTrailMatrix1_LSB,
            1);

    if (ver < 0x6301)
```

```
        setRfChirpGenData(ChirpBytes[1]);
    else
        setRfChirpGenData(ChirpBytes[0]);
}

/**
 * writes 6 bytes from specified buffer to both station address registers
 *
 * @param sta pointer to 6 byte-buffer containing the station address to be set for this node
 */
void setStaAddress(byte *sta)
{
    trans (INSTR_WR |6, Silbadero_RamStaAddr0_O, sta);
    trans (INSTR_WR |6, Silbadero_RamStaAddr1_O, sta);
}


/**
 * initializes the transmitter parameters of the TRX-chip
 *
 * @param ver specifies the chip version which identifies the two different boards used
 */
void initTx(word ver)
{
    /*
     * set the RF transmit power for data frames
     * value range is from 0.63 here, with 63 representing the maximum power value
     * the 64 input values are mapped to the alowed 19 values
      */
    if (ver < 0x6301)
        setOutputPower(0x3F);   // max value without external amplifier
    else
         setOutputPower(0x31);   // chip with external amplifier should not be overdriven (distortion),
value leads to 0x17 as amp value

    //    writeBit(Silbadero_TxUnderrunIgnore_O, Silbadero_TxUnderrunIgnore_B, TRUE);


    /*
     * select which of the two local station addresses
     * will be used as source address for any packets transmitted
     */
    writeBit(Silbadero_TxAddrSlct_O, Silbadero_TxAddrSlct_B,
            0);

    // enable the external amplifier on board1
    if (ver == 0x6301)
        writeBit(Silbadero_RfTxExtPampEnOutEn_O, Silbadero_RfTxExtPampEnOutEn_B, TRUE);


    // set the maximum number of retransmissions, 14=0xE is maximum value
    writePart(Silbadero_TxArqMax_O, Silbadero_TxArqMax_MSB, Silbadero_TxArqMax_LSB,
             MAX_ARQ);
```

```
    // we don't need scrambling here, turn it off by using value zero
    writePart (Silbadero_TxScrambInit_O, Silbadero_TxScrambInit_MSB, Silbadero_TxScrambInit_LSB, 0);

    // set respect ACK-packet in virtual carrier sensinging mode
   writeBit (Silbadero_TxVCarrSensAck_O, Silbadero_TxVCarrSensAck_B, Silbadero_TxVCarrSensAckOn_BC);
}

/**
 * initializes the receiver parameters of the TRX-chip
 *
 * @param ver specifies the chip version which identifies the two different boards used
 */
void initRx(word ver)
{
    if (ver < 0x6301)
        // Silbadero_RfRxDownCmpThres_O@0x63 (board1) is address 0x1F
        // @0x61(board2)=Silbadero_RfRxUpDownCmpThres_O
        // use variable of new chip version to avoid extra header include of old version
        writePart(Silbadero_RfRxDownCmpThres_O,
                Silbadero_RfRxDownCmpThres_MSB, Silbadero_RfRxDownCmpThres_LSB,
                RfRxUpDownCmpThres_DEFAULT);
    else
    {
        writePart(Silbadero_RfRxUpCmpThres_O,
                Silbadero_RfRxUpCmpThres_MSB, Silbadero_RfRxUpCmpThres_LSB,
                RfRxUpCmpThres_DEFAULT);
        writePart(Silbadero_RfRxDownCmpThres_O,
                Silbadero_RfRxDownCmpThres_MSB, Silbadero_RfRxDownCmpThres_LSB,
                RfRxDownCmpThres_DEFAULT);
    }

    writePart(Silbadero_RfRxLnaFreqAdj_O,
            Silbadero_RfRxLnaFreqAdj_MSB, Silbadero_RfRxLnaFreqAdj_LSB,
            RfRxLnaFreqAdj_DEFAULT);
    writePart(Silbadero_RfRxUpSymAdjust_O,
            Silbadero_RfRxUpSymAdjust_MSB, Silbadero_RfRxUpSymAdjust_LSB,
            RfRxUpSymAdjust_DEFAULT);
    writePart(Silbadero_RfRxDownSymAdjust_O,
            Silbadero_RfRxDownSymAdjust_MSB, Silbadero_RfRxDownSymAdjust_LSB,
            RfRxDownSymAdjust_DEFAULT);

    writePart(Silbadero_RfRxUpDownCmpAdj_O,
            Silbadero_RfRxUpDownCmpAdj_MSB, Silbadero_RfRxUpDownCmpAdj_LSB,
            RfRxUpDownCmpAdj_DEFAULT);


    // set up how many bit errors in the syncword of a received packet will be tolerated
    writePart(Silbadero_RxCorrErrThres_O,
            Silbadero_RxCorrErrThres_MSB, Silbadero_RxCorrErrThres_LSB,
            RxCorrErrThres_DEFAULT);

    // set up the bit detector
```

```
    writePart(Silbadero_RxDetBitSyncPulsesThres_O,
            Silbadero_RxDetBitSyncPulsesThres_MSB, Silbadero_RxDetBitSyncPulsesThres_LSB,
            RxDetBitSyncPulsesThres_DEFAULT);
    writePart(Silbadero_RxDetBitSyncLostThres_O,
            Silbadero_RxDetBitSyncLostThres_MSB, Silbadero_RxDetBitSyncLostThres_LSB,
            RxDetBitSyncLostThres_DEFAULT);
    writePart(Silbadero_RxDetGateAdjThres_O,
            Silbadero_RxDetGateAdjThres_MSB, Silbadero_RxDetGateAdjThres_LSB,
            RxDetGateAdjThres_DEFAULT);

    writePart(Silbadero_RxDetUnSyncGate_O,
            Silbadero_RxDetUnSyncGate_MSB, Silbadero_RxDetUnSyncGate_LSB,
            RxDetUnSyncGate_DEFAULT);
    writePart(Silbadero_RxDetBitSyncGate_O,
            Silbadero_RxDetBitSyncGate_MSB, Silbadero_RxDetBitSyncGate_LSB,
            RxDetBitSyncGate_DEFAULT);
    writePart(Silbadero_RxDetFrameSyncGate_O,
            Silbadero_RxDetFrameSyncGate_MSB, Silbadero_RxDetFrameSyncGate_LSB,
            RxDetFrameSyncGate_DEFAULT);

    writeBit(Silbadero_RxDetBitSyncAdj_O, Silbadero_RxDetBitSyncAdj_B, RxDetBitSyncAdj_DEFAULT);


    /*
     * set up the receiver to check whether the destination address
     * of a received packet matches one of the local station addresses
     */
    writeBit(Silbadero_RxAddrMode_O, Silbadero_RxAddrMode_B, TRUE);

    /*
     * If possible configure the receiver to stop only on correct crc2 packets received.
     * This feature is not available in firmware versions up to 1.65
     */
    writeBit(Silbadero_RxCrc2Mode_O, Silbadero_RxCrc2Mode_B, TRUE);


    // set the TRX chip to either never transmitt an ACK when a packet with is received,
    // or only after correct CRC1 or CRC2
    writePart(Silbadero_RxArqMode_O, Silbadero_RxArqMode_MSB, Silbadero_RxArqMode_LSB,
            //      Silbadero_RxArqModeNone_VC
            //      Silbadero_RxArqModeCrc1_VC
            Silbadero_RxArqModeCrc2_VC
           );

}

/**
 * initializes the TRX-chip according to the chip-version, calibrates and initializes receiver and transmitter,
 * setting chirp bytes, sync word and station address, everything to startup the TRX-chip
 *
 * @param nodeAddress pointer to 6 byte-buffer containing the station address to be set for this node
 * @param modSystem specifies either the 1Mbps mode (Silbadero_TxRxModSystem2ary_BC) or the
```

```
 * 2Mbps mode (Silbadero_TxRxModSystem4ary_BC)
 */
void initNTRX (byte *nodeAddress, byte modSystem)
{
    word ver;

    ver = readWord(Silbadero_Ver_O); // reading chip-version and chip-revision (2 bytes)

    // IRQ initialization
    writeBit (Silbadero_IrqPol_O,      Silbadero_IrqPol_B,      FALSE);   // set IRQ to low-active mode
    writeBit (Silbadero_IrqPushPull_O, Silbadero_IrqPushPull_B, TRUE);    // push-pull driver for IRQ-
output



    //-----------------------------------------------------------------
    // out of AppNote08_TRX_Clk_Supply_Config_V100_Prelim_2005_08_17.pdf
    //
    // start the crystal oscillator
    writeBit(Silbadero_PwrOnOffCrystal16MHz_O, Silbadero_PwrOnOffCrystal16MHz_B, TRUE);

    //allow 5 ms for the start up of the oscillator
    delay_ms(5);

    //disable the clock switcher's reset
    writeByte(Silbadero_RstCrystal16MHz_O, 0x00);

    //enable the clock distribution
    writeBit(Silbadero_PwrOnOffClock16MHz_O, Silbadero_PwrOnOffClock16MHz_B, TRUE);
    //-----------------------------------------------------------------

    calibrateNTRX();

    writePart(Silbadero_RfAgcGainMax_O, Silbadero_RfAgcGainMax_MSB, Silbadero_RfAgcGainMax_LSB,
            RfAgcGainMax_DEFAULT);

    writePart(Silbadero_RfAgcAmp_O, Silbadero_RfAgcAmp_MSB, Silbadero_RfAgcAmp_LSB,
            RfAgcAmp_DEFAULT);
    writePart(Silbadero_RfAgcIntCapsUnSync_O,
            Silbadero_RfAgcIntCapsUnSync_MSB, Silbadero_RfAgcIntCapsUnSync_LSB,
            RfAgcIntCapsUnSync_DEFAULT);
    writePart(Silbadero_RfAgcIntCapsSync_O,
            Silbadero_RfAgcIntCapsSync_MSB, Silbadero_RfAgcIntCapsSync_LSB,
            RfAgcIntCapsSync_DEFAULT);
    writePart(Silbadero_RfAfcThres_O, Silbadero_RfAfcThres_MSB, Silbadero_RfAfcThres_LSB,
            RfAfcThres_DEFAULT);
    writeBit (Silbadero_RfAgcHoldFrameSyncCtrl_O, Silbadero_RfAgcHoldFrameSyncCtrl_B,
            RfAgcHoldFrameSyncCtrl_DEFAULT);
    writePart(Silbadero_RfAgcHoldBitSyncCtrl_O,
            Silbadero_RfAgcHoldBitSyncCtrl_MSB, Silbadero_RfAgcHoldBitSyncCtrl_LSB,
            RfAgcHoldBitSyncCtrl_DEFAULT);
    writePart(Silbadero_RfAgcHoldBitSyncTimeOutCtrl_O,
           Silbadero_RfAgcHoldBitSyncTimeOutCtrl_MSB, Silbadero_RfAgcHoldBitSyncTimeOutCtrl_LSB,
            RfAgcHoldBitSyncTimeOutCtrl_DEFAULT);
```

```
    writePart(Silbadero_RfAgcPeakDetResetAdj_O,
            Silbadero_RfAgcPeakDetResetAdj_MSB, Silbadero_RfAgcPeakDetResetAdj_LSB,
            RfAgcPeakDetResetAdj_DEFAULT);


    if (ver >= 0x6301)
    {
        writeBit(Silbadero_RfAgcUpDis_O, Silbadero_RfAgcUpDis_B,
                RfAgcUpDis_DEFAULT);
        writeBit(Silbadero_RfAgcDownDis_O, Silbadero_RfAgcDownDis_B,
                RfAgcDownDis_DEFAULT);
    }

    setNTRXChirp(ver);

    writeBit(Silbadero_TxRxModSystem_O, Silbadero_TxRxModSystem_B,
            modSystem);
    //        Silbadero_TxRxModSystem2ary_BC);
    //        Silbadero_TxRxModSystem4ary_BC);
    writeBit(Silbadero_TxRxDataRate_O, Silbadero_TxRxDataRate_B,
            Silbadero_TxRxDataRate1MSymbols_BC);
    writePart(Silbadero_TxRxCrcType_O, Silbadero_TxRxCrcType_MSB, Silbadero_TxRxCrcType_LSB,
            //        Silbadero_TxRxCrcType1_VC);
            Silbadero_TxRxCrcType3_VC);

    setSyncWord();

    // set the two local station addresses
    setStaAddress(nodeAddress);

    // configure forward error correction
    writeBit(Silbadero_TxRxFwdEc_O, Silbadero_TxRxFwdEc_B,
            Silbadero_TxRxFwdEcOff_BC);
    //        Silbadero_TxRxFwdEcOn_BC);

    initTx(ver);

    initRx(ver);
}

/**
 * clears any pending interrupts of the receiver and transmitter, configures the
 * receiving of different packet-types, specifies which events causes an interrupt (external interrupt of
µC)
 * and finally let the receiver listen
 */
void startNTRX()
{
    byte rxIRQ, txIRQ;

    // clear any pending interrupts in receiver of the TRX chip
    rxIRQ = readByte(Silbadero_RxIntsRawStat_O);
    writeTransientByte(Silbadero_RxIntsReset_O, rxIRQ);
```

```
    // clear any pending interrupts in transmitter of the TRX chip
    txIRQ = readByte(Silbadero_TxIntsRawStat_O);
    writeTransientByte(Silbadero_TxIntsReset_O, txIRQ);

    //  enable reception of special packets
    writeBit(Silbadero_RxDataEn_O, Silbadero_RxDataEn_B, TRUE);
    writeBit(Silbadero_RxBrdcastEn_O, Silbadero_RxBrdcastEn_B, TRUE);
    //    writeBit(Silbadero_RxTimeBEn_O, Silbadero_RxTimeBEn_B, TRUE);


    // configure IRQ event
    writeBit (Silbadero_IrqEnableRxInt_O, Silbadero_IrqEnableRxInt_B, TRUE);    // Rx drives interrupt
line
    writeBit (Silbadero_RxIntsEn_O,       Silbadero_RxHeaderEnd_B,    TRUE);    // interrupt at header
end
   writeBit (Silbadero_RxIntsEn_O,       Silbadero_RxOverflow_B,     TRUE);   // interrupt at rx overflow


    // start the receiver of the TRX chip
    writeTransientBit(Silbadero_RxCmdStart_O, Silbadero_RxCmdStart_B, TRUE);
}
```

(* LaTeX generated by highlight 2.4.5, http://www.andre-simon.de/ *)

Anhang B

MPEG-4-Chip-Treiber Quellcode

```
 #include "ime6500.h"
#include <cdefbf537.h>

BYTE int_wait;
signed short oldBoard = -1;

/** waits for the resetting of a flag, done by the
  * IME6500 ISR imeISR() after the occurance of an interrupt
  */
void waitIRQ()
{
    while(int_wait)
        ;
}

/**
 * Due to address mismatch on Blackfin Video Board Revision V1.0
 * offset addresses have to be shifted.
 */
int getIMEAddr(int offset)
{
    if (oldBoard)
        return ADDR_IME65xx + (offset << 1);
    else
        return ADDR_IME65xx + offset;
}

/** writes a 16 bit value to the specified register
  *
  * @param offset 16 bit register to be written to, specified as register offset, e.g. 0x20 for FDR-register
  * @param data value to write
  */
void writew(int offset, WORD data)
{
    BYTE i;
    *((volatile unsigned short *)getIMEAddr(offset)) = data;
    for (i=0; i<10; i++)
        asm( "NOP;");
}

/** reads a 16 bit value from the specified register
  *
  * @param offset 16 bit register to be read from, specified as register offset, e.g. 0x20 for FDR-register
  * @param data value to write
  */
WORD readw(int offset)
{
    return *((volatile unsigned short *)getIMEAddr(offset));
}

/** reads a 32 bit value from the specified register
  *
  * @param offset 32 bit register to be read from, specified as register offset, e.g. 0x00 for CMD-register
```

```
  * @return value read
  */
DWORD reg_read(int offset)
{
    DWORD val = readw(offset);
    return val |(readw(offset+2) << 16);
}

/** writes a 32 bit value to the specified register
  *
  * @param offset 32 bit register to be written to, specified as register offset, e.g. 0x00 for CMD-register
  * @param data value to write
  */
void reg_write(int offset, DWORD data)
{
    writew(offset, data);
    writew(offset+2, data >> 16);
}

/** writes the complete command register
  * @param cmd complete command value to be written to the CMD register
  */
void cmd_write(DWORD cmd)
{
    writew(CMD+2, cmd >> 16);
    writew(CMD, cmd);
}

/** @return 1 if the old Blackfin Video Board v1.0 is used, 0 when using the new one
  */
BYTE isOldBoard ()
{
    if (oldBoard < 0)
        resetIME();

    return oldBoard;
}

/** gets the direkt mapped structure of the lower part address of the CMD register
  *
  * @return direkt changeable structure of the lower part of the CMD register
  */
CMD_T* get_cmd_reg()
{
    return ((CMD_T *)(getIMEAddr(CMD)));
}

/** gets the direkt mapped structure of the task address part of the CMD register
  *
  * @return direkt changeable structure of the upper part of the CMD register
  */
CMD_TASK_T* get_cmd_task_reg()
{
```

```
    return ((CMD_TASK_T *)(getIMEAddr(CMD+2)));
}

/** gets the direkt mapped structure of the FAR register address
  *
  * @return direkt changeable FAR register structure
  */
FAR_T* get_far_reg()
{
    return ((FAR_T *)(getIMEAddr(FAR)));
}

/** gets the direkt mapped structure of the FCR register address
  *
  * @return direkt changeable FCR register structure
  */
FCR_T* get_fcr_reg()
{
    return ((FCR_T *)(getIMEAddr(FCR)));
}


/** gets the direkt mapped structure of the STAT register address
  *
  * @return direkt changeable STAT register structure
  */
STAT_T* get_stat_reg()
{
    return ((STAT_T *)(getIMEAddr(STAT)));
}


/** reads the fifo control structure from specified channel register
  *
  * @param ch fcr channel number
  * @return fcr structure read from register
  */
FCRCH_T read_fcr(BYTE ch)
{
    BYTE d;

    d = (BYTE)(reg_read(FCR) >> (ch * 8));

    return *((FCRCH_T *)(&d));
}

/** writes the fifo control structure to specified channel register
  *
  * @param ch fcr channel number
  * @param fcr fcr structure to write to register
  */
void write_fcr(BYTE ch, FCRCH_T fcr)
{
```

```
    BYTE d = *((BYTE *)(&fcr));

    reg_write(FCR, d << (ch * 8));
}

/** executes a software reset for IME6500 and detects the devboard version
  */
void resetIME()
{
    cmd_write(0x8000);
    delay_ms(5);
    cmd_write(0x0000);
    delay_ms(5);

    oldBoard = (*((WORD*)(ADDR_IME65xx+2)) == 0x2000);

    // reset FCR
    reg_write(FCR, reg_read(FCR));

    STAT_T* stat = get_stat_reg();

    if (stat->pend)
        stat->pend = 0;

    int_wait = FALSE;
}

/** start a command transaction in polling mode
  * and waits until it finishes
  *
  * @param command specifies the command type
  * @param task specifies the task ID
  * @return 0 if successful, 1 otherwise
  */
BYTE start_command_poll (BYTE command, WORD task)
{
    CMD_T *cmd = get_cmd_reg();
    CMD_TASK_T *cmdT = get_cmd_task_reg();

    cmdT->task = task;
    cmd->type = command;
    /// this executes the command, must be last
    cmd->fwint = 1;

    WORD timeout=16;
    WORD delay=10;
    WORD i=0;

    while(--timeout)
    {
        if (!(reg_read(CMD) & 1))
            return 0;
```

```
        /// delays program execution for the specified amount of us
        delayus(delay);
        delay <<= 1;
    }

    return 1;
}

/**
 * executes a task by IME6500 with specified input parameters
 *
 * @param out array address of output parameters
 * @param task task ID of task to execute (see firmware manual)
 * @param in array address of input parameters
 * @return 0 if successful, 1 if execution failed
 */
BYTE run_task (DWORD out[3], WORD task, DWORD in[3])
{
    reg_write(DR0, in[0]);
    reg_write(DR1, in[1]);
    reg_write(DR2, in[2]);

    if (start_command_poll (0x20, task))
        return 1;

    out[0] = reg_read(DR0);
    out[1] = reg_read(DR1);
    out[2] = reg_read(DR2);

    return 0;
}

/** read N BYTEs from FIFO starting from
  * the current FIFO address (FAR)
  *
  * @param buf address of the buffer where to be written to
  * @param N byte amount to read
  */
void fifo_read_current(BYTE *buf, WORD N)
{
    WORD k;
    WORD d;

    for(k=0; k<N; k+=4)
    {
        d = readw(FDR); // must read the lower 16 bits first
        buf[k]   = d;
        buf[k+1] = d >> 8;

        d = readw(FDR); // and read the upper 16 bits next
        buf[k+2] = d;
        buf[k+3] = d >> 8;
    }
```

```
}

/** write N BYTEs to FIFO starting from
  * the current FIFO address (FAR)
  *
  * @param buf address of the buffer to be read from
  * @param N byte amount to write
  */
void fifo_write_current (BYTE *buf, WORD N)
{
    WORD k;

    for(k=0; k<N; k+=4)
    {
        writew(FDR, (buf[k+1] << 8) |buf[k]); // must write the lower 16 bits first
        writew(FDR, (buf[k+3] << 8) |buf[k+2]); // and write the upper 16 bits next
    }
}

/** enables FIFO0 and IME6500 internal transfer
  */
void fifo_enable()
{
    BYTE val = 0;
    FCRCH_T *wr = (FCRCH_T *)&val;
    FCRCH_T rd = read_fcr(0);
    wr->fint_enable = ~(rd.fint_enable);
    wr->reserved = ~(rd.fint_enable);
    write_fcr(0, *wr);
}

/** receive N BYTEs from FIFO0, with burst size of BLKSIZ BYTEs
  *
  * @param buf address of the buffer where to be written to
  * @param BLKSIZ transfer is made in blocks of this specified size
  * @param N byte amount to read
  */
void fifo_receive(BYTE *buf, WORD BLKSIZ, WORD N)
{
    DWORD fcr = reg_read(FCR);
    reg_write(FCR, fcr ^0x8);

    // set FAR
    writew(FAR, 0);

    int_wait = TRUE;

    fifo_enable();

    WORD k = 0;

    while(k < N)
    {
```

```
        waitIRQ();  // this call returns after a FIFO interrupt

        fifo_read_current(buf, BLKSIZ); // wait_fifo_intr() also disables the FIFO

        buf = buf + BLKSIZ;
        k = k + BLKSIZ;

        int_wait = TRUE;

        fifo_enable(); // so the FIFO must be enabled again
    }
}

/** send N BYTEs to FIFO ch, with burst size of BLKSIZ BYTEs
  *
  * @param buf address of the buffer where to be read from
  * @param BLKSIZ transfer is made in blocks of this specified size
  * @param N byte amount to write
  */
void fifo_send(BYTE *buf, WORD BLKSIZ, DWORD N)
{
    DWORD fcr = reg_read(FCR);
    reg_write(FCR, fcr ^0x8);

    // set FAR
    writew(FAR, 0);

    fifo_write_current(buf, BLKSIZ); // data must be ready before the FIFO is enabled

    buf = buf + BLKSIZ;
    DWORD k = BLKSIZ;

    int_wait = TRUE;

    fifo_enable();

    while(k < N)
    {
        waitIRQ();  // this call returns after a FIFO interrupt

        fifo_write_current (buf, BLKSIZ); // wait_fifo_intr() also disables the FIFO

        buf = buf + BLKSIZ;
        k = k + BLKSIZ;

        int_wait = TRUE;

        fifo_enable(); // so the FIFO must be enabled again
    }
}

/** activates video firmware module to use
  *
```

```
  * @return 0 if successful, 1 otherwise
  */
BYTE setupCoderFW()
{
    DWORD param[] =
        {
            0x00010000, // selecting video and no audio encoder
            0,
            0
        };

    return run_task (param, 0, param);
}

/** setup of the video input of IME6500
  *
  * @return 0 if successful, 1 otherwise
  */
BYTE setupRecord()
{
    DWORD param[] =
        {
            0,
            0,
            0
        };

    // SETUP_RECORD: Set up Video Recording Path from Video Input Port A
    // Video Input Format Control
    param[0] = (0<<31)   // CA_FIELD_POL
             |(0<<30)  // CA_CHID_EN
             |(0<<29)   // CA_FIELD_IGNORE (1=ignore)
             |(0<<7)   // CA_MODE (1=656, 0=601)
             |(1<<6)   // CA_ENABLE
             |(0x1<<4)  // CA_BYTESEQ 0x1=Cb, Y, Cr, Y; 0x0=Y, Cb, Y, Cr
             |(0<<3)   // CA_PCKI_P (0 in 656-mode)
             |(0<<2)   // CA_DVALID_P (0=active high)
             |(0<<1);  // CA_VSYNC_POL (0=active low)
    param[1] = 0;
    param[2] = 0;
    if (run_task (param, 0x1800, param))
        return 1;

    // Specify Input Picture Size in Pixels
    param[0] = ((FIELD_SIZE_X-1) << 16) |(FIELD_SIZE_Y-1); // CA_NPIXELS, CA_NLINES
    param[1] = 0;
    param[2] = 0;
    if (run_task (param, 0x1804, param))
        return 1;

    // Specify Active Picture Region Horizontally
    param[0] = (FIELD_SIZE_X-1) << 16; // CA_X_LAST, CA_X_FIRST
    param[1] = 0;
```

```
    param[2] = 0;
    if (run_task (param, 0x1808, param))
        return 1;

    // Specify Active Picture Region Vertically
    param[0] = (FIELD_SIZE_Y-1) << 16; // CA_Y_LAST, CA_Y_FIRST
    param[1] = 0;
    param[2] = 0;
    if (run_task (param, 0x180C, param))
        return 1;

    /// ---------------------------------------------------------------------------
    /// following was suggested by INTIME-support, but seems to have no positive effect
    /*
     // Specify Delay from VSYNC to the beginning of internal field data processing
    #if (PICTURE_SIZE_X == 720)
     param[0] = 0x000073F1; // CA_FDP_DLY
    #else
     param[0] = 0x00008DE9; // CA_FDP_DLY
    #endif
     param[1] = 0;
     param[2] = 0;
     if (run_task (param, 0x1820, param))
      return 1;

     // Specify position of Channel ID during VBI for TOP field
    #if (PICTURE_SIZE_X == 720)
     param[0] = 0x00006D29; // CA_CHID_POS_TOP
    #else
     param[0] = 0x0000872D; // CA_CHID_POS_TOP
    #endif
     param[1] = 0;
     param[2] = 0;
     if (run_task (param, 0x1830, param))
      return 1;

     // Specify position of Channel ID during VBI for BOTTOM field
    #if (PICTURE_SIZE_X == 720)
     param[0] = 0x000073E9; // CA_CHID_POS_BOT
    #else
     param[0] = 0x00008DE1; // CA_CHID_POS_BOT
    #endif
     param[1] = 0;
     param[2] = 0;
     if (run_task (param, 0x1834, param))
      return 1;
    */
    return 0;
}

/** setup of an IME6500 internal video input channel
  *
  * @return 0 if successful, 1 otherwise
```

```
 */
BYTE setupRecordChannel()
{
   // SETUP_REC_CH: Set up Respective Recording Input Channels
   // Specify Horizontal/Vertical Scaling of Port A Input Channels
   DWORD param[] =
      {
         1,
         (PICTURE_SIZE_X << 16) |FIELD_SIZE_X, // REC_OUTSIZE_X, REC_INSIZE_X
#if (PICTURE_SIZE_X == 720)
         ((PICTURE_SIZE_Y/2) << 16) |FIELD_SIZE_Y // REC_OUTSIZE_Y, REC_INSIZE_Y
#else
         (PICTURE_SIZE_Y << 16) |FIELD_SIZE_Y // REC_OUTSIZE_Y, REC_INSIZE_Y
#endif
      };

   if (run_task (param, 0x2000, param))
      return 1;

   // Enable/Disable Port A Input Channels
   param[0] = 1;
   param[1] = 0;
   param[2] = 0;

   return run_task (param, 0x2004, param);
}

/** creates an video encoder instance
 *
 * @return 0 if successful, 1 otherwise
 */
BYTE createVideoEncoder()
{
   // CREATE_VEC: Create a Video Encoder
   DWORD param[] =
      {
         (1<<15), // according email from david kim 2006-08-11
#if (PICTURE_SIZE_X == 720)
         (PICTURE_SIZE_X << 16) |(PICTURE_SIZE_Y) |(1<<15),
#else
         (PICTURE_SIZE_X << 16) |(PICTURE_SIZE_Y),// no field mode according email from david
kim 2006-08-11,
#endif
         0
      };
   if (run_task (param, 0x2400, param))
      return 1;

   return 0;
}

/** setup of the video encoder instance previously created
 *
```

```
  * @return 0 if successful, 1 otherwise
  */
BYTE setupVideoEncoder()
{
    // SETUP_VEC: Set up Modes of Video Encoder Instances.
    // These tasks must be called before the encoders are activated.
    DWORD param[3];

    param[0] = 1;   // CH 0
    param[1] = (2 << 29) |(0 << 8) |5; // 1=Hybrid, 2=CBR, 4=VBR
    param[2] = 0;
    // Select Bit Rate Control Mode
    if (run_task (param, 0x2814, param))
        return 1;

    param[0] = 1;
    param[1] = 28000; // for constant bitrate (CBR) bitrate (bits/frame)
    // param[1] = (40<<16) |24; //for hybrid bitrate (HBR): max, min bitrate (kbits/frame)
    param[2] = 0;
    // Select Bit Rate Control Mode
    if (run_task (param, 0x2818, param))
        return 1;

    /// additional filters may be activated here
    // 0x04 Select Noise Filtering Modes
    // 0x08 Set up L1 Filter Level. A higher level results in less noise at the output.
    // 0x0C Set up L2 Filter Level. A higher level results in less noise at the output.
    // 0x10 Set up L3 Filter Level. A higher level results in less noise at the output.
    // 0x00 Set up Motion Detection(MD)

    param[0] = 1;
    param[1] = VEC_PP_COUNT << 16; // VEC_PP_COUNT: Number of P(not B) Pictures between
I pictures, 0 means no P-frames
    param[2] = 0;
    // Set up GOP(Group Of Picture) Mode
    if (run_task (param, 0x281C, param))
        return 1;

    // VEC_FR_INT=262144 (65536*4) would decrease the output video
    // frame rate to 1/4th of the input video frame rate
    param[0] = 1;
    param[1] = VEC_FR_INT;
    param[2] = 0;
    // Specify Output Frame Interval
    if (run_task (param, 0x2820, param))
        return 1;

    return 0;
}

/** activates previously created video encoder instance
  *
  * @return 0 if successful, 1 otherwise
```

```
  */
BYTE activateVideoEncoder()
{
    // RUN_VEC: Activate/deactivate Video Encoders.
    // An encoder sends out compressed video streams to FIFO 0 only when it is activated.
    DWORD param[] =
        {
            1,
            0,
            0
        };
    return run_task (param, 0x3400, param);
}

/** setup of the video output of IME6500
  *
  * @param source specifies the source to be put out,
  *             0 specifies video input port A, 1 specifies video decoder output
  * @return 0 if successful, 1 otherwise
  */
BYTE setupVideoOutput(BYTE source)
{
    DWORD param[3];

   param[0] = (source<<31) |(0<<2) |(1<<1) |1; // VOP_INPATH, VOP_SIGNAL (0=656, 1=601),
PAL, enable port
    param[1] = (100<<16) |(100<<8) |100; // background color cr, cb, y
    param[2] = 0;

    if (run_task (param, 0x0C00, param))
        return 1;

    param[0] = (0<<16) |0; // VOP_INOFFSET_Y, VOP_INOFFSET_X
    param[1] = ((PICTURE_SIZE_Y)<<15) |PICTURE_SIZE_X; // VOP_INSIZE_Y, VOP_INSIZE_X
  param[2] = ((1024*(PICTURE_SIZE_Y)/288)<<15) |(1024*(PICTURE_SIZE_X)/720); // VOP_STEP_Y,
VOP_STEP_X

    if (run_task (param, 0x0C04, param))
        return 1;

    return 0;
}

/** initializes video encoder by executing all necessary setup methods
  *
  * @return 0 if successful, 1 otherwise
  */
BYTE initEncoder()
{
    if (setupCoderFW())
        return 1;

    if (setupRecord())
```

```
        return 1;

    if (setupRecordChannel())
        return 1;

    if (createVideoEncoder())
        return 1;

    if (setupVideoEncoder())
        return 1;

    return 0;
}

/** initializes video decoder by selecting decoder firmware module
  *
  * @return 0 if successful, 1 otherwise
  */
BYTE initDecoder()
{
    DWORD param[] =
        {
            0x00020000, // selecting video decoder and no audio module
            0,
            0
        };

    if (run_task (param, 0, param))
        return 1;

    return 0;
}

/** activates video decoder after executing of needed setup methods
  *
  * @return 0 if successful, 1 otherwise
  */
BYTE activateDecoder ()
{
    DWORD param[3];

    if (setupVideoOutput(1))
        return 1;

    // CREATE_VDC: Create a Video Decoder.
    param[0] = 1<<15; // destroy all other decoders
    param[1] = (PICTURE_SIZE_X<<16) |PICTURE_SIZE_Y; // VDC_SIZE_X, VDC_SIZE_Y
    param[2] = 0; // VDC_OFFSET_X, VDC_OFFSET_Y
    if (run_task (param, 0x2400, param))
        return 1;

    // SETUP_VDC: Set up Modes of Video Decoders Instances.
    param[0] = 1; // VDC_ID
```

```
    param[1] = (65536); // VDC_FR_INT. Frame interval to be used. Must be equal to or greater than
65536. Output Frame Interval = (VDC_FR_INT/65536) * Standard output frame Interval (25 or 30)
    param[2] = (1<<31); // VDC_FR_USE. Use this frame interval value (Must be 1)
    if (run_task (param, 0x2800, param))
        return 1;

    //  Specify minimum data block count
    param[0] = 1; // VDC_ID
    param[1] = (1<<31) |(16<<8) |4; // VDC_VBUF, VDC_ABUF(no audio needed)
    param[2] = 0;
    if (run_task (param, 0x2804, param))
        return 1;

    // set FAR
    writew(FAR, 0);

    // resetting FCR
    writew(FCR, readw(FCR) ^0xE);

    // RUN_VDC Activate/Deactivate Video Decoders.
    param[0] = 1; // VDC_ID
    param[1] = 0;
    param[2] = 0;
    if (run_task (param, 0x3400, param))
        return 1;

    return 0;
}

/** supply video decoder with infos of next frame
  *
  * @param frameType specifies type of frame, 0 means I-frame, 1 means P-frame
  * @return 0 if decoder now expects the new frame, 1 if decoder is not ready to accept a new frame
  */
BYTE prepareDecoderPacket(BYTE frameType, DWORD frameSize)
{
    DWORD param[3];

    // STREAM_WRITE.
    param[0] = (0<<16) |(frameType&0xf); // DEC_ID. Decoder channel ID
    param[1] = frameSize;
    param[2] = 0;
    if (run_task (param, 0x2C00, param))
        return 1;

    return !param[0]; // DR0=0: The decoder is NOT ready to accept a new frame
}

/** activates video encoder instance
  *
  * @return 0 if successful, 1 otherwise
  */
BYTE activateEncoder()
```

```
{
    // set FAR
    writew(FAR, 0);

    // reset FCR
    writew(FCR, readw(FCR) ^0xE);

    if (activateVideoEncoder())
        return 1;

    return 0;
}

/** deactivates video encoder or video decoder instance
  *
  * @return 0 if successful, 1 otherwise
  */
BYTE deactivateCoder()
{
    DWORD param[3];

    // deactivating encoder/decoder
    param[0] = 1; // encoder/decoder ID
    param[1] = 0;
    param[2] = 0;
    if (run_task (param, 0x3404, param))
        return 1;

    return 0;
}

/**
 * writes the boot loader part of the firmware (first 1024 BYTEs) to the IME6500
 *
 * @param FW address of the first byte of the bootloader
 * @param size size in bytes of the bootloader
 */
void writeBL(BYTE *FW, DWORD size)
{
    WORD bcnt, i;
    DWORD eflag0, eflag1, d, k;

    // download the first 1024 BYTEs of the firmware
    bcnt = 0;
    while(bcnt < size)
    {
        //wait until IME65XX is ready to accept the next 512 BYTEs
        while((reg_read(CMD) & 0x2000) == 0)
            ; // wait for CMD[13]=1

        eflag0 = eflag1 = 0;

        for(k=0; k<512; k=k+4)
```

```
        {
            i = bcnt + k;
            d = (FW[i]<<24) |(FW[i+1]<<16) |(FW[i+2]<<8) |FW[i+3];
            reg_write(DR1, d);
            start_command_poll(0, 0);

            eflag0 = eflag0 ^d;

            if(!(k & 4))
                eflag1 = eflag1 ^d;
        }

        // send checksum data to IME65XX
        reg_write(DR1, eflag0);
        start_command_poll(0, 0);

        reg_write(DR1, eflag1);
        start_command_poll(0, 0);

        // checksum result:
        // if checksum is !!NOT!! OK, CMD[14] will be 1, otherwise continue with the next 512B block
        // else (bit=1), resend the current block
        if((reg_read(CMD) & 0x4000) == 0)
            bcnt = bcnt + 512;
    }

    // wait for download check
    while((reg_read(CMD) & 0x1000) == 0)
        ; // wait for CMD[12]=1

    // read firmware info
    /* DWORD uExtraInfo = reg_read(CMD);
     DWORD uLinkDate = reg_read(DR0);
     DWORD uLinkTime = reg_read(DR1);
     DWORD uDramType = reg_read(DR2);

     printf("Bootloader installed successfully: DR0=0x%08x DR1=0x%08x DR2=0x%08x\n",
       uLinkDate, uLinkTime, uDramType);
     printf("DramType : 0x%08X (%d banks/%dMbits)\n", uDramType, (uDramType>>26)+1,
       ((1+(uDramType>>26))*(uDramType&0x3fffff)*8)>>20);*/
}

/**
 * writes the complete firmware to the IME6500 (inclusive bootloader!)
 * using FIFO0
 *
 * @param FW address of the first byte of the firmware
 * @param rSize size in bytes of the firmware
 */
BYTE writeFW(BYTE *FW, DWORD rSize)
{
    reg_write (DR0, rSize);
```

```
    BYTE mB = 0;
    FCRCH_T *mode = (FCRCH_T *)&mB;
    mode->hint_enable = 1;

    start_command_poll(1, 0);

    fifo_send (*mode, FW, 1024, rSize);

    return 0;
}

/** loads firmware into IME6500
  *
  * @param addr_fw address of the first byte of the firmware
  * @param bl_size size in bytes of the bootloader part of the firmware
  * @param fw_size size in bytes of the complete firmware
  */
void loadIME6500FW (BYTE *addr_fw, DWORD bl_size, DWORD fw_size)
{
    // writing boot loader to chip (first 1024 BYTEs of firmware)
    writeBL (addr_fw, bl_size);

    // writing complete firmware to chip
    writeFW (addr_fw, fw_size);
}

/** ISR to be executed for IME6500 interrupt during firmware download
  */
void imeISR()
{
    STAT_T* stat = get_stat_reg();
    BYTE fcrchB = 0;

    if (stat->pend)
        stat->pend = 0;

    fcrchB = readw (FCR);

    if (fcrchB & 0x40)
        writew (FCR, fcrchB ^0xA);

    int_wait = FALSE;
}
```

(* LaTeX generated by highlight 2.4.5, http://www.andre-simon.de/ *)

Index