DIPLOMARBEIT

Aufbereitung, Übertragung und Darstellung von Videobildern im Rahmen des SENSE-Projektes

Eingereicht an der
Fakultät für Elektrotechnik und Informationstechnik,
Technischen Universität Wien.
Ausgeführt zur Erlangung des akademischen Grades eines
Diplom-Ingenieurs

unter der Leitung von

O. Univ. Prof. Dipl.-Ing. Dr. techn. Dietmar Dietrich
Institutsnummer: 384
Institut für Computertechnik

und

Univ. Ass. Dipl.-Ing. Dr. techn. Dietmar Bruckner
Institutsnummer: 384
Institut für Computertechnik

von

Franz Winter
0425806
Eiselsbergstraße 36
4641 Steinhaus bei Wels

24. März 2012 _____

Kurzfassung

Das SENSE-Projekt beschäftigt sich mit der Entwicklung von Methoden für vernetzte, intelligente Sensoren. Das besondere an diesem Projekt ist, dass sich das Sensornetzwerk selbstständig auf eine unbekannte Umgebung einstellt. Um die entwickelten Methoden zu testen wurde eine Testplattform entwickelt, welches als Sicherheitsüberwachungssystem konzipiert wurde. Um das Überwachungssystem in einer realistischen Umgebung zu testen, wurde es dazu in einem Flughafen installiert.

Ziel dieser Arbeit ist, weitere Funktionen in die Testplattform zu integrieren. So soll es dem Benutzer des Überwachungssystems ermöglicht werden, Videobilder von der Videokamera im Sensorknoten über eine Benutzeroberfläche am PC zu betrachten. Dabei soll es einerseits möglich sein, Live-Videobilder von einem Knoten zu betrachten, sowie On-Demand-Videobilder aus der im knoteninternen durchgeführten Videoaufnahme.

Bei der Entwicklung der Software musste beachtet werden, dass das in den Knoten eingebettete Computersystem nur eine begrenzte Größe an Speicher und Rechenleistung besitzt. Ein weitere Einschränkung stellte die limitierte Datenrate der mit Funk angebunden Sensorknoten dar.

Für die Umsetzung der Anforderungen müssen die Videodaten komprimiert werden, da die Kommunikationsinfrastruktur nicht ausreichend Bandbreite besitzt. Dazu bietet sich der MPEG-4/H.263-Encoders des i.MX31 an. Die Übertragung der komprimierten Videodaten erfolgt mit dem Streaming-Protokoll RTP. Für die Wiedergabe des Video-Streams wird der VLC Media Player verwendet, welcher in das Visualisierungsprogramm am PC eingebettet wurde. Die positive Begleiterscheinung der Videokompression ist, dass für die Speicherung der Videodaten ebenfalls weniger Speicherplatz benötigt wird. Die Speicherung der Videodaten kann auf der RAM/Flash-Disk beziehungsweise einen Massenspeicher welcher über USB angebunden ist, durchgeführt werden.

Durch die Implementierung eines RTP Streaming-Services besteht die Möglichkeit auch andere Endgeräte in das System zu integrieren. So wäre es möglich, Videos auf ein Handy oder Tablet zu streamen.

Danksagung

An erster Stelle möchte ich mich bei meinen Eltern bedanken, die es mir ermöglicht haben, an der TU Wien zu studieren.
Weiteres möchte ich mich bei meinem Betreuer, Univ. Ass. Dipl.-Ing. Dr. techn. Dietmar Bruckner für die geduldige Zusammenarbeit bedanken, bei Herrn Linder Thomas von der Firma Bluetechnix für die sehr hilfreichen Tipps rund um den i.MX31 sowie Dipl.-Ing. Josef Mitterbauer für das eine oder andere Problem mit Linux.

Inhaltsverzeichnis

Abkürzungen

AC	Alternating Current
API	Application Programming Interface
ARM	Advanced RISC Machine
ASCII	American Standard Code for Information Interchange
ATA	Advanced Technology Attachment with Packet Interface
ATM	Asynchronous Transfer Mode
B-Frame	Bidirectionally predictive-coded-Frame
BSP	Board Support Package
BV	Bewegungsvektor
CIF	Common Intermediate Format
CCD	Charge-coupled Device
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DC	Direct Current
DKT	Diskrete Kosinus-Transformation
DMA	Direct Memory Access
DLL	Dynamic Linked Library
DSP	Digitaler Signal-Prozessor
EAS	Embedded Audio Subsystem
EVS	Embedded Video Subsystem
FAT	File Allocation Table
FIFO	First in - First out
FPGA	Field Programmable Gate Array
FPS	Frames per second
GPL	General Public License
GUI	Graphical User Interface
HD	High Definition
HDD	Hard Disk Drive
ID	Identification
IEC	International Electrotechnical Commission
IO	Input/Output
IP	Internet Protocol
IPU	Image Processing Unit
IPv6	Internet Protocol Version 6
ISDN	Integrated Services Digital Network
ISO	International Organization for Standardization
ITU	International Telecommunication Union
IETF	Internet Engineering Task Force
JPEG	Joint Photographic Experts Group
LCD	Liquid crystal display
LGPL	Lesser General Public License
LTIB	Linux Target Image Builder
MPA	MPEG Audio
MPV	MPEG Video
MP2T	MPEG 2 Transport
MPEG	Moving Picture Experts Group
MTU	Maximum Transmission Unit
MMC	Multimedia Card
MMU	Memory Management Unit
OpenGL ES	Open Graphic Library for Embedded Systems

OSI	Open Systems Interconnection
P-Frame	Predictive-coded-Frame
PC	Personal Computer
PCMU	Pulse Code Modulation (μ-Law)
Protobuf	Protocol Buffers
QCIF	Quarter Common Intermediate Format
QVGA	Quarter Video Graphics Array
QQVGA	Quarter-QVGA
RAM	Random Access Memory
RDU	Reasoning and Decision making Unit
RFC	Requests for Comments
RISC	Reduced Instruction Set Computer
RTMP	Real-Time Messaging Protocol
RTP	Real-Time Transport Protocol
RTCP	Real-Time Control Protocol
RTSP	Real-Time Streaming Protocol
RVLC	Reversible Variable Length Coding
SD-Karte	Secure Digital-Karte
SDP	Session Description Protocol
SENSE	Smart Embedded Network of Sensing Entities
SIP	Session Initiation Protocol
STL	Standard Template Library
SVH	Short Video Header
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
USB	Universal Serial Bus
USB-OTG	Universal Serial Bus-On-the-go
VGA	Video Graphics Array
VCSU	Video Compression and Streaming Unit
VSUV	Video Streaming Unit Visualization
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

1. Einführung SENSE-Projekt

Die Überwachung von Gebäuden wird in der Regel durch Sicherheitspersonal gewährleistet. Dieses ist durch ihre direkte Anwesenheit und ihre sensorischen Fähigkeiten in der Lage, sicherheitskritische beziehungsweise potentiell sicherheitskritische Situationen zu erkennen und auf diese angemessen zu reagieren. Diese sensorischen Fähigkeiten sind auch der Grund, weshalb es zwingend notwendig ist, dass Sicherheitspersonal überall dort anwesend sein muss, wo keine technischer Hilfsmittel für die Überwachung bereitgestellt werden. Eine einzelne Person des Sicherheitsdienstes ist dabei nur in der Lage, ein bestimmtes Areal zu überwachen. Als logische Schlussfolgerung gilt, dass man mit steigender Größe des zu überwachenden Areals mehr Sicherheitspersonal benötigt. Dies bedingt auch, dass die Personalkosten für die Überwachung steigen und bei sehr großen Arealen wie zum Beispiel einem Kaufhaus untragbar werden können.

Um konkurrenzfähig zu bleiben, müssen die Kosten für die Gebäudeüberwachung gesenkt werden, ohne dabei den Sicherheitsstandard zu senken. Dies kann durch elektronische Hilfsmittel wie zum Beispiel in Form von Videokameras erreicht werden. Dazu werden Kameras überall dort montiert, wo früher Sicherheitspersonal für die direkte Überwachung eingestellt werden musste. Die Videobilder laufen in einer Sicherheitszentrale zusammen, wo sie auf Bildschirmen aufbereitet werden. Das Sicherheitspersonal ist dadurch nicht mehr gezwungen, im zu überwachenden Areal direkt anwesend zu sein. So kann in solchen Systemen durch einen Blickwechsel auf einen anderen Monitor in ein anderes Überwachungsareal eingeblickt werden. Damit ist eine einzige Person in der Lage, verschiedenste Bereiche zu überwachen, was in Folge die Personalkosten verringert. Jedoch sind weiterhin die sensorischen Fähigkeiten des Menschen gefragt, um Gefahrensituationen zu erkennen. Doch wie sieht es mit Gebäuden aus - wie zum Beispiel einem Flughafen - welche durch ihre enorme Ausdehnung und ihren hohen Sicherheitsanforderungen ebenfalls kostengünstig überwacht werden sollen?

Die Sicherheitsbestimmungen an Flughäfen wurden mit dem Anschlag auf das World Trade Center am 11. September 2001 schlagartig erhöht. So wurde das Sicherheitspersonal enorm aufgestockt und eine Reihe von technischen Hilfsmitteln eingeführt, um ähnliche Geschehnisse in Zukunft zu vereiteln. Doch sind die dadurch entstehenden Kosten langfristig nicht zu tragen. Man ist somit auf der Suche nach technischen Hilfsmitteln, um die Überwachung noch effizienter durchführen zu können. Mit dem im nächsten Kapitel vorgestellten Projekt wird ein erster Schritt in die Richtung eines intelligenten Überwachungssystems gemacht.

1.1 Motivation hinter dem SENSE-Projekt

Wie bereits in der Einführung erwähnt, können Personalkosten gespart werden, wenn das Sicherheitspersonal mit einfachen technischen Hilfsmitteln, wie zum Beispiel einem Videoüberwachungssystem, ausgerüstet wird. Die Architektur eines solchen Videoüberwachungssystems ist in Abbildung 1 dargestellt. Dabei werden die durch Videokameras gelieferten Videobilder in einem zentralen Überwachungsraum zusammengeführt und auf Bildschirmen dargestellt. Durch den Multiplexer wird die Darstellung von mehreren Überwachungsvideos ermöglicht, da meist eine größere Anzahl von Kameras als Überwachungsbildschirme zur Verfügung steht. Das Personal in der Überwachungszentrale spielt in solchen Systemen eine wichtige Rolle, da dieses für die Erkennung von Gefahren zuständig ist. Stellt das Personal in der Überwachungszentrale eine Gefahr fest, so gibt es diese Information dem Sicherheitspersonal weiter, welche für die Beseitigung der Gefahr zuständig ist.
Ein solches System hat mehrere Nachteile, die durch die Einbindung von Menschen ins System hervorgerufen werden. Ein Nachteil dieses Systems liegt in der monotonen Arbeit, welche das Sicherheitspersonal ermüden lässt und zum Schwinden der Aufmerksamkeit führt. Durch das Ermüden des Sicherheitspersonals ist es notwendig, dieses nach einer bestimmen Dauer durch aufmerksameres Sicherheitspersonal auszutauschen, was wiederum zu kritischen Momenten bei der Überwachung führt. Ein weiterer Nachteil kommt zu tragen, wenn eine enorme Anzahl an Videokameras erneut einen erheblichen personellen Aufwand erfordert, um alle Bildschirme im Auge zu behalten.

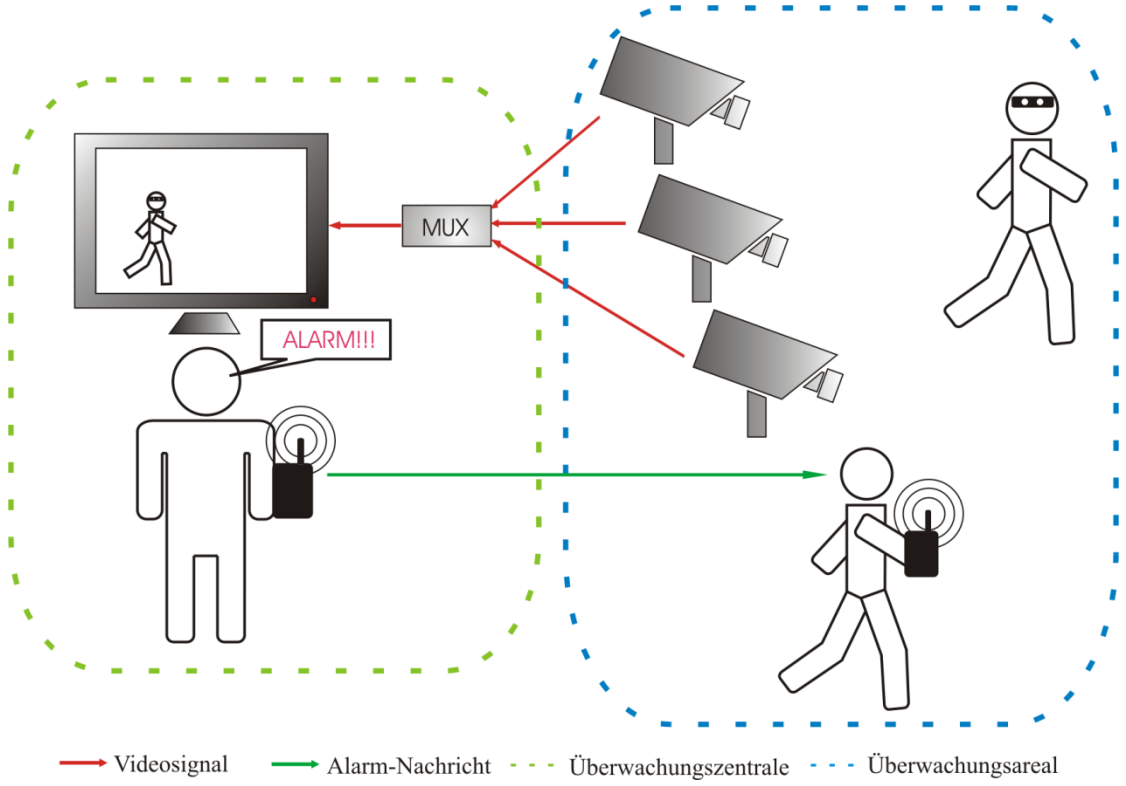


Abbildung 1: Klassisches Videoüberwachungssystem

Diese beiden Nachteile, welche klassische Überwachungssysteme haben, sollen bei SENSE (Smart Embedded Network of Sensing Entities) ausgeschlossen werden. Die Idee, welche hinter dem SENSE-Projekt steckt, ist, dem Menschen die monotone Überwachungstätigkeit abzunehmen. Anstelle des Menschen kommt ein intelligentes Überwachungssystem für die Erkennung von Gefahren zum Einsatz. Idealerweise wird in so einem System nur noch Personal für die Gefahrenbeseitigung benötigt. Abbildung 2 zeigt einen Überwachungsknoten, welcher durch einen eingebetteten Computer die Sensordaten analysiert und bei Gefahr einen Alarm auslöst. Mit dem Einsatz eines kabellosen Kommunikationssystems wäre eine einfache nachträgliche Installation des Überwachungssystems möglich sowie die Einbindung des Sicherheitspersonals durch mobile Terminals. Durch den Einsatz von mobilen Terminals würde damit auch keine Sicherheitszentrale mehr notwendig sein.

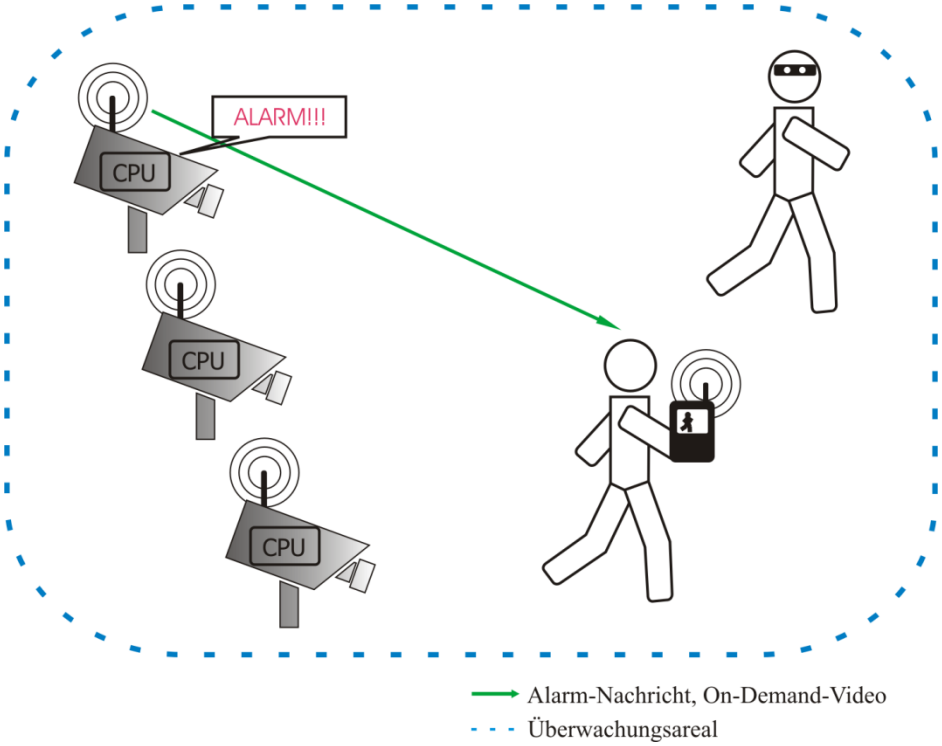


Abbildung 2: Intelligentes Videoüberwachungssystem

Die Vorteile des Überwachungssystems in Abbildung 2 sind klar. Es wird ermöglicht, die Anzahl der Videokameras zu erhöhen und damit die Sicherheit von großen Arealen zu gewährleisten und zugleich die Personalkosten gegenüber einem klassischen System niedrig zu halten. Ein weiterer entscheidender Vorteil ist, dass ein solches Überwachungssystem nicht müde werden kann und somit immer aufmerksam ist. Jedoch ist es aus technischer Sicht äußerst anspruchsvoll, eine Maschine mit den notwendigen Algorithmen auszustatten, um mit dem fortgeschrittenen Auffassungsvermögen eines Menschen mithalten zu können. Doch ist dies genau die Herausforderung, welche versucht wird, im SENSE-Projekt zu realisieren.

Das SENSE-Projekt ist ein von der Europäischen Union unterstütztes Projekt, an dem mehrere Universitäten und Forschungseinrichtungen beteiligt gewesen sind. Das Projekt startete mit dem 01.09.2006, hatte eine Laufzeit von 45 Monaten und endete offiziell mit dem 01.06.2010. Die betei-

ligten Universitäten sowie Forschungseinrichtungen kamen dabei aus Spanien, Polen, Griechenland und Rumänien. Darunter auch Österreich mit der Austria Institute of Technology GmbH. und dem Institut für Computertechnik der Technischen Universität Wien [Bur09, S. 3].
Das Ziel des SENSE-Projektes bestand darin, Werkzeuge für die Entwicklung, Implementierung und den Betrieb eines intelligenten, adaptiven Funknetzwerkes zu entwickeln. Die entwickelten Methoden umfassten dabei Algorithmen, welche das Hinzufügen eines Sensorknoten detektieren und die Nachbarschaftsbeziehung zu den Knoten aktualisiert, bis hin zu Algorithmen, welche verschiedenste Gefahrensituationen erkennen können [BH10, S. 3, 35 - 36].
Um die Entwicklungen zu testen, wurde zusätzlich eine Testplattform entwickelt, welche als Sicherheitsüberwachungssystem konzipiert wurde. Da sich das SENSE-Projekt unter anderem mit vernetzten Sensoren beschäftigt, bestand die Architektur der Testplattform aus einem verteilten, vernetzten System aus Überwachungsknoten. Damit die Überwachungsknoten untereinander beziehungsweise dem Zentralrechner kommunizieren können, steht einerseits ein kabelgebundenes Kommunikationssystem sowie ein Funkkommunikationssystem zur Verfügung. Ein für stationäre Installationen ausgelegter Überwachungsknoten verfügt dabei über eine Kamera sowie einer Reihe von Mikrofonen. Mithilfe dieser Sensoren und einem in jeden Knoten eingebetteten Computersystem ist jeder Knoten in der Lage, seine Umgebung zu überwachen. Durch die Vernetzung der Sensorknoten ist es möglich, knotenübergreifende Aufgaben zu lösen, wie zum Beispiel das Verfolgen einer verdächtigen Person über verschiedene Überwachungsbereiche hinweg. Dazu ist es erforderlich, die lokalen Sichten eines jeden Sensorknoten in eine kohärente globale Sicht zusammenzuführen [Bur10, S. 3].
In dieser ersten Ausbaustufe stellt das System die erste Instanz bei der Detektion von Gefahren dar. Der Benutzer des Systems wird jedoch auf die Alarme aufmerksam gemacht und entscheidet als zweite Instanz, ob tatsächlich eine Gefahr vorliegt. Zum Projektende hin wurde mit dem entwickelten Überwachungssystem ein Feldversuch durchgeführt. Dazu wurden zwölf Überwachungsknoten am polnischen Flughafen Kraków installiert. Die Tests zeigten, dass nicht alle geplanten Funktionalitäten erreicht werden konnten. Doch war das System in der Lage, zum Beispiel herrenlose Gepäcksstücke sowie laufende beziehungsweise schreiende Personen automatisch zu erkennen [Bur10, S. 34].

1.2 Funktionsanforderungen

Es ist natürlich klar, dass nur ein Teil der Gesamtfunktionalität des SENSE-Projektes in dieser Arbeit behandelt werden kann. Die Anforderungen dieser Arbeit beschränken sich hauptsächlich auf die Videobildaufbereitung.

Die funktionalen Anforderungen sind Folgende: Der Benutzer des Überwachungssystems soll die Möglichkeit haben, über eine GUI am PC Videos betrachten zu können. Dabei soll es ihm ermöglicht werden, entweder Live-Videobilder oder aufgenommene Videobilder On-Demand betrachten zu können. Durch die Aufnahme wird die Voraussetzung geschaffen, Geschehnisse, welche vor mindestens 25 Minuten stattgefunden haben, nachträglich On-Demand abzurufen.

Die Anforderungen haben folgende Randbedingungen. Die Videobilder werden im SENSE-Knoten mit einer Kamera erfasst und mit einem im Knoten eingebetteten Videobildanalysesystem analysiert.

Um die knoteninterne Kommunikationsbandbreite nicht zu überlasten, werden die Videobilder komprimiert. Für die Kompression der Videobilder, welche eine Auflösung von 640 x 480 Pixel besitzen, wird auf den JPEG-Standard zurückgegriffen. Das Videobildanalysesystem stellt die JPEG-Bilder zusätzlich über TCP/IP zur Verfügung.

Die Speicherung der Videobilder soll lokal in jedem SENSE-Knoten durchgeführt werden. Dafür ist ein geeignetes Speichermedium zu wählen und bei Bedarf in den Knoten zu integrieren.

Das zu implementierende Programm im SENSE-Knoten soll als eigenständige Anwendung realisiert werden. Dadurch, dass weitere Anwendungen parallel ausgeführt werden, ist darauf zu achten, dass die Anwendung eine möglichst geringe CPU Auslastung aufweist.

Damit der Benutzer des Systems in der Lage ist, Videosequenzen eines jeden SENSE-Knoten betrachten zu können, ist eine Übertragung der Videobilddaten notwendig. Für die Übertragung soll die vorhandene Kommunikationsinfrastruktur verwendet werden. Dabei ist darauf Rücksicht zu nehmen, dass die bereits geringe Datenrate der verwendeten Funkmodule so gering wie möglich beansprucht wird.

Als Übertragungsprotokoll der Videobilder an sich soll ein standardisiertes Protokoll verwendet werden, welches so gewählt werden sollte, dass eine Wiedergabe mit einem gängigen Software Multimedia-Player möglich ist.

Zusammenfassung der Anforderungen:

- Einlesen von JPEG-Bildern
- Zeitlich beschränkte Speicherung der Videobilddaten
- Videobildübertragung mit einem standardisiertes Protokoll
- Anzeige der Videodaten am PC

1.3 SENSE-Systemkomponenten

Dieses Kapitel soll eine Übersicht über den Aufbau der im SENSE-Projekt zusammenspielenden Komponenten geben. Ziel dieses Kapitels ist es, ein Verständnis für den bereits bestehenden Aufbau von Software und Hardware zu bekommen und um die Aktivitäten, welche in dieser Arbeit gemacht wurden, besser abzugrenzen. Da bereits zu Beginn dieser Arbeit die Hardware zur Verfügung stand, liegt hier kein typischer Systementwurf wie im Lehrbuch vor. Dieser sieht vor, dass erst, nachdem die Anforderungen spezifiziert worden sind, die zu entwickelnde Software designt wird und daraus die notwendige Hardware festlegt wird [TH07, S. 17-18]. Da dies in diesem Fall nicht gegeben ist, wird im Einführungsteil die vorhandene Hardware beschrieben.

Die Grundarchitektur des SENSE-Projektes basiert auf einem verteilten System, welches aus Videoüberwachungsknoten, den sogenannten SENSE-Knoten, besteht. Ein Kommunikationssystem stellt sicher, dass die SENSE-Knoten sowohl untereinander kommunizieren können als auch mit einem zentralen PC. Eine einfache Systemkonfiguration ist zum Beispiel in Abbildung 3 dargestellt. Im Folgenden werden die drei wichtigsten Systemkomponenten beschrieben.

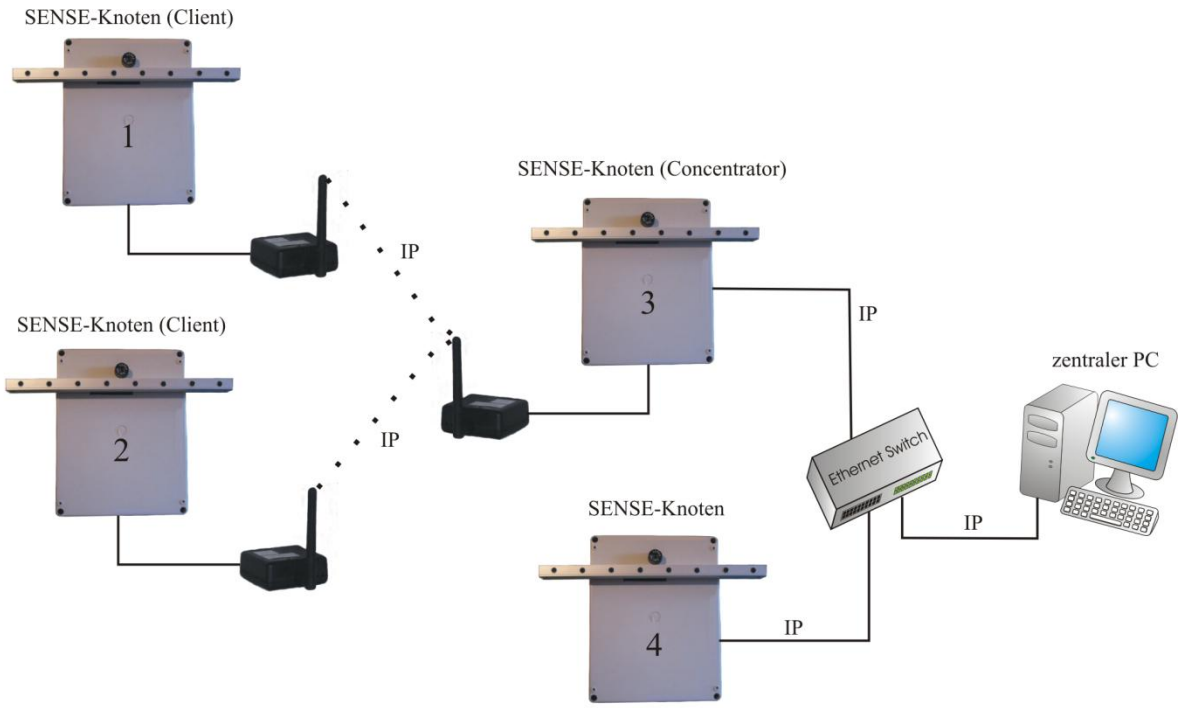


Abbildung 3: Einfache Systemkonfiguration

1.3.1 SENSE-Knoten

Im Wesentlichen besteht ein SENSE-Knoten aus einer Kamera, mehreren Mikrofonen, sowie einem eingebetteten Computersystem. Solch ein Knoten kann überall dort montiert werden, wo eine Überwachung stattfinden soll. Durch die im eingebetteten Computersystem ausgeführte Software ist ein Knoten in der Lage, einen Bereich zu überwachen und bei einer Sicherheitsverletzung Alarm zu geben.

Jeder Knoten ist in der Lage, verschiedenste Gefahren zu erkennen. Durch die Vernetzung der Knoten ist es auch möglich, knotenübergreifende Aufgaben zu lösen. Wie zum Beispiel das Verfolgen einer Person hinweg von verschiedenen Überwachungsbereichen oder der akustischen Ortung von schreienden Menschen.
Dadurch, dass jeder Knoten die Analyse der Sensordaten knotenintern durchführt, werden im Normalfall keine unverarbeiteten Sensordaten direkt über die Kommunikationsinfrastruktur übertragen. Eine Ausnahme stellt das Streamen von Videos dar. Das Kommunikationsaufkommen beschränkt sich auf Nachrichten, welche den Status der Knoten beziehungsweise Event-Nachrichten beschränkt. Dadurch, dass die Datenmengen der Nachrichten zwischen den Knoten, verglichen mit den Sensordaten, äußerst gering sind, ist es möglich mit geringer Bandbreite bereits größere Überwachungssysteme zu realisieren.
Wird von einem Überwachungsknoten eine Gefahr erkannt, ist in dieser Ausbaustufe angedacht, die Alarmnachrichten an einen Zentralcomputer zu schicken. Eine Visualisierungssoftware am Zentralcomputer ist dafür zuständig die verschiedensten Informationen dem Benutzer anzuzeigen.

1.3.2 Visualisierungssoftware

Über die Visualisierungssoftware soll es dem Benutzer ermöglicht werden, die SENSE-Knoten zu verwalten sowie auf verschiedenste Informationen zuzugreifen. Der PC, welcher diese Software ausführt, ist über Ethernet mit den SENSE-Knoten verbunden. Die SENSE-Knoten besitzen in diesem System die Rolle eines Servers, durch die verschiedenste Informationen über Dienste angeboten werden. Die Visualisierungssoftware greift auf diese Dienste zu, um so an die notwendigen Informationen zu gelangen.

Die Visualisierungssoftware stellt einen schematischen Gebäudeplan dar, in den jeder SENSE-Knoten eingezeichnet ist. Sendet ein SENSE-Knoten einem Alarm an die Visualisierungssoftware, so wird das Sicherheitspersonal darauf aufmerksam gemacht. Mit einem Alarm bekommt das Sicherheitspersonal eine Reihe von Information zur Verfügung gestellt. Darunter die Position des Knoten, der die Alarmnachricht gesendet hat sowie weitere Detailinformationen.

Die gestellten Funktionsanforderungen in dieser Arbeit erweitern dabei die Informationsdetails zu einem Alarm. So wird es dem Sicherheitspersonal ermöglicht, vergangene Geschehnisse von einem beliebigen SENSE-Knoten abrufen zu können. Das Betrachten von Aufnahmen der Videokamera dient dazu, um das entscheidende Bildmaterial wiedergeben zu können, welche Grundlage für die Alarmgenerierung war.

1.3.3 Kommunikationsinfrastruktur

Im SENSE-Projekt stehen zwei verschiedene Kommunikationstechnologien für den Datenaustausch zur Verfügung. Einerseits ein kabelgebundenes Kommunikationsmedium sowie ein speziell für das SENSE-Projekt entwickeltes Funknetzwerk auf Basis von NanoNet. SENSE-Knoten können dabei entweder ausschließlich über ein Kabel oder eine Funkverbindung verfügen sowie eine Kombination dieser beiden [BZP+09, S.11]. Abbildung 3 zeigt die verschiedenen Kombinationsmöglichkeiten.

Besitzt ein SENSE-Knoten sowohl eine Ethernet-Verbindung als auch ein Funkmodul, so handelt es sich um einen Concentrator. Der Concentrator zusammen mit bis zu vier Clients bildet eine Gruppe. Innerhalb dieser Gruppe ist keine Client zu Client Kommunikation möglich. Sämtliche Kommunikationsabläufe erfolgen über den Concentrator. Die Funkmodule erreichen dabei eine Datenrate von bis zu 1 MBit/s [BZP+09, S. 11, 52].

Als kabelgebundenes Kommunikationsmittel wird Ethernet verwendet. Die Datenrate des verwendeten Ethernets ist 100 MBit/s. Das Ethernet-Netzwerk stellt somit den Backbone dar, welcher die einzelnen Concentrator und SENSE-Knoten untereinander verbindet sowie die einfache Einbindung eines PCs ermöglicht [BZP+09, S. 11].

1.4 Aufbau der SENSE-Knoten

Um die Umgebung überwachen zu können, ist jeder SENSE-Knoten mit zwei verschiedenen Sensorsystemen ausgestattet. Für die optische Erfassung kommt eine Kamera zum Einsatz, welche eine Auflösung von 640 x 480 Pixel (VGA) besitzt. Ergänzend zu dem optischen Sensor steht jedem

Überwachungsknoten eine Reihe von Mikrofonen zur Verfügung, die zur akustischen Überwachung eingesetzt werden. Abbildung 4 zeigt einen SENSE-Knoten (graue Box), dessen Innenleben äußerst umfangreich ist. Um die für das SENSE-Projekt gewünschten Funktionalitäten implementieren zu können, wurden mehrere Prozessoren als erforderlich erachtet. Diese sind auf drei Boards verteilt, welche über ein Verbindungs-Board verbunden sind. Jede dieser Platinen ist für einen bestimmten Bereich der Funktionalitätsanforderungen des SENSE-Projektes zuständig. Die folgende Beschreibung gibt eine Übersicht über die verschiedenen Boards und den jeweiligen Funktionen, die sie im Knoten erfüllen.

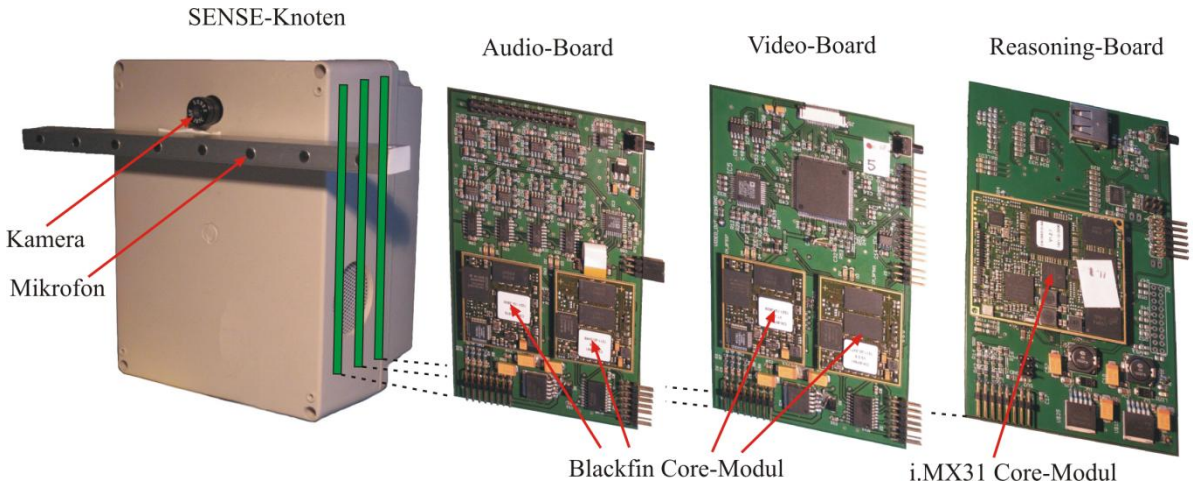


Abbildung 4: Aufbau eines SENSE-Knoten

1.4.1 Embedded Audio Subsystem

Das Embedded Audio Subsystem (EAS) besteht aus dem Audio-Board, mehreren Mikrofonen sowie der Software, welche die Analyse der Sensordaten durchführt. Auf dem Audio-Board wird sowohl die analoge Aufbereitung der Mikrofon-Signale durchgeführt als auch die digitale Verarbeitung mit Hilfe zweier Core-Module der Firma Bluetechnix [5]. Diese Core-Module bauen wiederum jeweils auf einen Digitalen Signal-Prozessor (DSP) der Firma Analog Devices [16] auf. Die gewonnen Daten des EAS werden über einen Dienst der Reasoning and Decision making Unit (RDU) zugänglich gemacht. Durch das EAS lassen sich Gefahren, wie zum Beispiel schreiende Personen, wahrnehmen [BH10, S. 8, 10, 14].

1.4.2 Embedded Video Subsystem

Das Embedded Video Subsystem (EVS) besteht aus dem Video-Board, einer Kamera sowie der Software, welche die Analyse der Videobilder durchführt. Die Analysesoftware wird auch hier wieder auf zwei Core-Module ausgeführt, die jeweils eine DSP enthalten. Für die Aufteilung des Videobilddatenstromes auf die beiden DSPs kommt ein FPGA (Spartan III) zum Einsatz.
Die Aufgabe des EVS besteht darin, Objekte aus einem Videobild zu extrahieren. Um auf die Ergebnisse des EVS zugreifen zu können, stellt dieses verschiedene Services über verschiedene Ports zur Verfügung, welche mit der folgenden Übersicht kurz beschrieben werden [BH10, S. 8, 10].

Video Streaming (JPEG)

Über dieses Service stellt das EVS das in JPEG komprimierte Bildmaterial zur Verfügung. Die Videobilder besitzen eine Auflösung von 640 x 480 Pixel (VGA) und eine Videobildrate von 3 - 7 FPS. Das EVS stellt über Port 30003 JPEG-Bilder zur Verfügung.

Video Streaming (YUV)

Als Alternative zum Video Streaming (JPEG) Service stellt das EVS auch Kamerabilder im YUV (4:2:0) planar Format bereit. Dieses Service stand nicht von Anfang an zur Verfügung und wurde aus Umständen der Machbarkeit hinzugefügt. Da die knoteninterne Kommunikation teilweise nur 10 MBit/s beherrscht, musste die Auflösung des Services verringert werden. Dadurch stellt das Video Streaming (YUV) Service die Videobilder nur mit einer Auflösung von 320 x 240 Pixel (QVGA) bereit. Es wird an dieser Stelle hingewiesen, dass eine gleichzeitige Nutzung der beiden Video Streaming Services nicht empfohlen wird und zu Stabilitäts- beziehungsweise Performance-Problemen der EVS führen kann. Dieses Service kann standardmäßig über Port 30005 abgerufen werden.

Low Level Symbols Streaming

Die gewonnen Daten durch das EVS werden über ein eigenes Service zugänglich gemacht. Für den Austausch der Daten zwischen dem EVS und der RDU wird XML verwendet. Da das Video Streaming (JPEG) unabhängig vom Low Level Symbols Streaming Service ist, wird mittels einer Sequenznummer sicher gestellt, dass die XML-Daten mit den JPEG-Bildern synchronisiert werden kann. Die XML-Daten sind über den Port 30002 erreichbar.

Commands

Die ersten drei Services bieten ausschließlich Daten für die Weiterverarbeitung an. Mit dem Konfiguration-Service ist man in der Lage, verschiedenste Parameter der Videobildverarbeitungssoftware zu ändern. So können zum Beispiel die Belichtungszeit, Framerate der Kamera sowie weiter Parameter konfiguriert werden. Dieses Service ist unter dem Port 30001 erreichbar.

1.4.3 Reasoning and Decision making Unit

Die Reasoning and Decision making Unit (RDU) wurde für zwei Aufgaben konzipiert, die auf zwei getrennten Programmen ausgeführt werden, nämlich die gleichnamige Software namens Reasoning and Decision making Unit sowie der Video Compression and Streaming Unit (VCSU). Diese beiden Programme werden unter Linux ausgeführt. Auf den verwendeten Prozessor sowie Betriebssystem wird im nächsten Kapitel kurz eingegangen. Zuvor jedoch eine kurze Beschreibung der beiden Programme.

Reasoning and Decision making Unit

Die RDU Software ist für die Erkennung von Gefahren zuständig. Dabei baut die Software auf den gewonnen Daten des EVS und EAS auf. Durch die Daten wird es möglich, die Handlungen in einer Umgebung erkennen zu können. Will man zum Beispiel in der Lage sein, herrenlose Koffer in einem

Flughafen erkennen zu können, ist es notwendig, einerseits Personen als auch Koffer detektieren zu können. Diese Aufgabe wird dabei durch die EVS erfüllt. Der zeitliche Zusammenhang zwischen der Position des Koffers und der Person wird durch die RDU hergestellt, um in Folge einen Alarm auszulösen. Die RDU schickt diese Alarme schlussendlich an den zentralen Rechner, wo der Benutzer darauf aufmerksam gemacht wird.

Video Compression and Streaming Unit

Die VCSU ist einer von zwei Programmen, welche in dieser Arbeit erstellt wurden. Die VCSU ist für die Aufbereitung der Videobilder zuständig. Dazu greift das Programm auf das Video Streaming Services des EVS zu. Die VCSU ist einerseits für die Speicherung als auch für die Übertragung der Videobilder zuständig. Dem Benutzer der Visulisierungssoftware am PC wird bei einem Alarm dadurch die Möglichkeit geboten, die gespeicherten Videobilder entweder live oder zu einem späteren Zeitpunkt zu betrachten.

1.5 Hardware und Software der Reasoning and Decision making Unit

Das Herzstück des Reasoning-Boards ist ein Core-Modul (CM-i.MX31C) der Firma Bluetechnix. Das Core-Modul basiert auf einem Prozessor der i.MX Familie aus dem Hause Freescale. Das Core-Modul besitzt eine Größe von 55 x 45 mm und beherbergt neben dem i.MX31 128 MB NAND Flash sowie 32 MB RAM [5].

Das folgende Unterkapitel beschreibt einige technische Merkmale des verwendeten Prozessors. Für die effektive Nutzung des Prozessors werden im letzten Unterkapitel einige wesentliche Eigenschaften des verwendeten Betriebssystems erklärt.

1.5.1 Freescale i.MX Prozessorfamilie

Die i.MX Serie wurde für mobile Geräte entwickelt, welche eine hohe Rechenleistung benötigen. Der i.MX31 im Speziellen basiert auf einem ARM1136JF-S CPU, welcher mit 532 MHz betrieben wird. Die CPU verfügt über eine acht-stufige Pipeline und eine Vector Floating Point Unit. Nebenbei beschleunigt ein unified L2 Cache mit 128 kB die Ausführung von Programmen. Dem von ARM lizenzierten CPU-Kern stellt Freescale [9] eine breite Palette von Peripherie beiseite. Neben gängigen Schnittstellen wie USB-OTG und ATA (HDD) besitzt der i.MX31 eine Image Processing Unit (IPU) sowie ein Multimedia and Human Interface [Fre07, S. 1].

Mit der IPU ist der i.MX in der Lage, eine Kamera über das dafür vorgesehene CMOS/CCD-Kamera Interface, anzusteuern. Dabei stehen Funktionen zur Farbraum-Konvertierung, Videobild-Invertierung, -Rotation und sowie -Skalierung zur Verfügung. Als Gegenstück zum Kamerainterface besitzt der IPU auch ein Interface um Displays für Ausgaben anzusteuern. All diese Funktionen erledigt die IPU bei minimaler beziehungsweise geringfügiger Beteiligung des ARM-Prozessors [Fre07, S. 1-2].

Die erwähnenswerten Features des Multimedia and Human Interface sind der Graphics Accelerator und der MPEG-4/H.263-Encoder. Mit dem Graphics Accelerator erfahren anspruchsvollere 3D-

Anwendungen, die auf OpenGL ES oder Java Mobile 3D basieren, eine Ausführungsbeschleunigung. Wie auch die Funktionen der IPU benötigen auch die Funktionen des Graphics Accelerator und MPEG-4/H.263-Encoder nur eine minimale Rechenzeit der ARM-CPU [Fre07, S. 1-2].

1.5.2 Betriebssystem – Linux

Durch die Verwendung eines leistungsfähigen Prozessors und umfangreicher Ressourcen ist die Verwendung eines Betriebssystems sinnvoll. Freescale bietet für den i.MX, Windows CE als auch Linux, in Form von zwei verschiedenen Board Support Paketes (BSP) an.

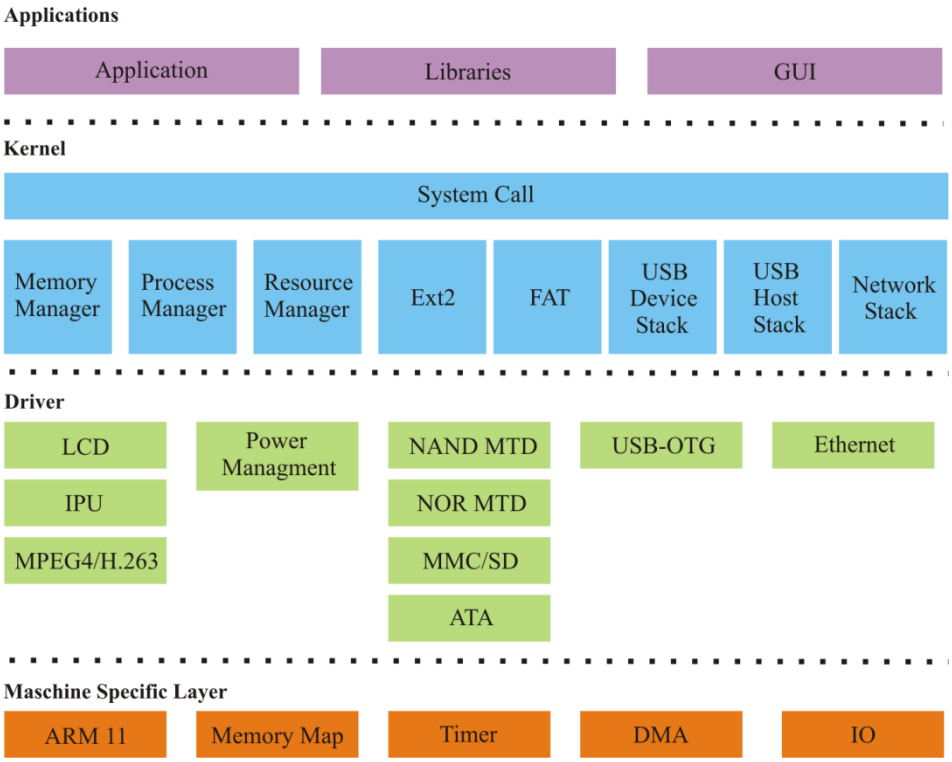


Abbildung 5: Schichten des BSPs (Quelle: [Fre09, S. 3-1)

Mit den bereitgestellten BSPs steht somit eine umfangreiche Softwarebasis zur Verfügung, ohne den Linux Kernel neu auf den i.MX zu portieren beziehungsweise Treiber entwickeln zu müssen. Somit wird der Einstieg erleichtert und man kann schnell eigene Anwendungen implementieren. Die Wahl des Betriebssystems ist im SENSE-Projekt auf Linux gefallen. Die Gründe dafür sind der kostenlose und der Quellcode offene Zugang zum Linux Kernel. Die i.MX Linux Portierung basiert auf einem Standard Linux Kernel. Die verwendete Kernel-Version ist 2.6.22 und bietet alle Features, welche man von einem modernen Betriebssystem erwartet. Diese sind Funktionen wie:

- Prozess- und Thread-Management
- Memory Management (Speichereinblendung, Belegung und Freigabe von Speicher, MMU und L1/L2 Cache-Steuerung)
- Resource Management (Interrupts)

- File System (ext2, NFS, FAT)
- Driver model
- Standardisierte APIs
- Verschiedenste Kommunikations-Stacks

Abbildung 5 zeigt die Architektur des Linux-BSP für die i.MX Prozessorfamilie. Sie besteht aus Anwender Programmen (user-space executables), Standard Kernel Komponenten, welche von der Linux Community kommen sowie Hardware spezifische Treiber und Funktionen, die von Freescale bereitgestellt werden [Fre09, S. 3-1,3-2].

1.6 Anforderungsanalyse

Dieses Kapitel beschäftigt sich mit den grundsätzlichen Überlegungen, um die geforderten Funktionalitäten zu erfüllen. Um den Zusammenhang der Teilsysteme besser zu erkennen, gibt Abbildung 6 einen Überblick und zeigt die wesentlichen Domänen des Systems. Der SENSE-Konten erfasst mit Hilfe der Sensoren die Umgebung (linker Bildteil) und interagiert auf der rechten Seite über die Visualisierungssoftware am PC mit dem Benutzer.

Die Anforderungen lassen sich in die drei wichtigsten funktionalen Teile aufteilen. Diese sind die Speicherung, die Videoübertragung und die Anzeige der Videobilder. Der Zusammenhang zwischen den Anforderungen und den sich daraus ergebenen Teilanforderungen wird im Folgenden näher dargestellt.
Die erste Anforderung, betrifft die Übertragung der Videodaten. Wie bereits in Kapitel 1.3 angeführt, kommen grundsätzlich zwei verschiedene Arten von Kommunikationstechnologien zum Einsatz. Als Verbindung zwischen PC und den SENSE-Knoten steht eine drahtgebundene 100 MBit/s Ethernet-Verbindung zur Verfügung. Daneben besitzen manche Knoten einen drahtlosen Zugang zum Netzwerk. Die Datenrate der drahtlosen Kommunikation beträgt dabei 1 MBit/s. Somit ist klar, dass die Datenrate des drahtlosen Kommunikationssystems für die weiteren Überlegungen herangezogen wird, da diese maßgeblich die effektive Bandbreite im Netzwerk bestimmt. Zusätzlich muss neben der Videoübertragung auch noch Bandbreite für den Austausch der Nachrichten zwischen den SENSE-Knoten berücksichtigt werden. Erste Tests haben gezeigt, dass die gelieferten JPEG-Bilder des EVS mit einer Auflösung von 640 x 480 Pixel eine Größe von 20 bis 200 kByte besitzen. Daraus ergibt sich eine maximal benötigte $\text{Datenrate} = 200\ \text{kByte} \cdot 7\ \text{FPS} = 1400\ \text{kByte/s}$.
Diese erste Abschätzung der benötigten Datenrate macht klar, dass die Videodaten einer besseren Komprimierung unterzogen werden müssen. Als entscheidendes Auswahlkriterium für das Komprimierungsverfahren muss die Auslastung der CPU beachtet werden. Eine grundsätzliche Anforderung ist, dass die Videoverarbeitung eine CPU Auslastung von unter 10 % erzeugen soll. Die Wahl des Komprimierungsverfahrens wird in Kapitel 4 näher betrachtet.
Ein weiterer wichtiger Bestandteil der zu entwickelnden Software ist die Videoübertragung, welche auf einem standardisierten Protokoll aufbauen soll. Dieses Protokoll soll so gewählt werden, dass eine Wiedergabe mit einem Multimedia-Player möglich ist. Diese Teilaufgabe wurde dabei als kritisch eingestuft, da es nicht klar war, ob es bereits verfügbare Lösungen für dieses Problem gibt und

diese den Anforderungen gerecht werden. Deshalb wird auch dieser Punkt in Kapitel 4 näher untersucht.

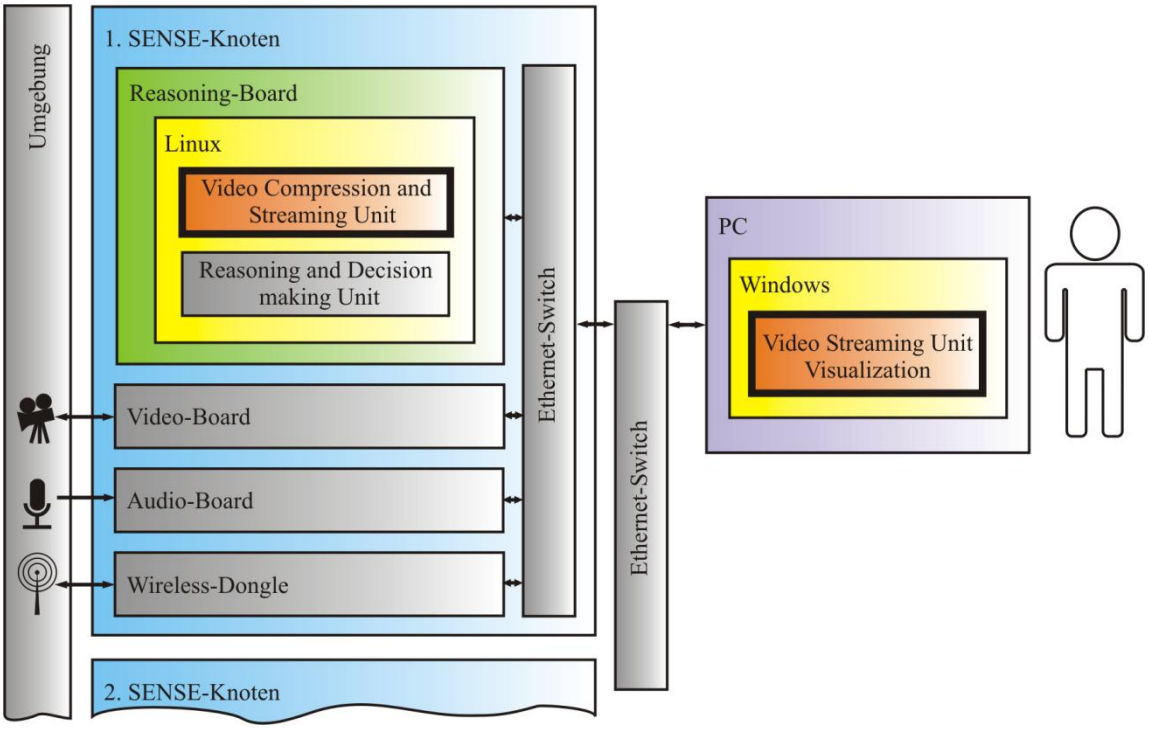


Abbildung 6: Systemübersicht

Die zweite Anforderung betrifft die temporäre Speicherung der Videobilder. Die positive Begleiterscheinung der notwendigen Videokomprimierung ist, dass auch der Bedarf an benötigtem Speicherplatz sinkt. Da jeder SENSE-Knoten Videobilder mit einer Dauer von mindestens 25 Minuten Video speichern können soll, würde man bei einer Speicherung von JPEG-Bildern, mit dem oben angenommen Randbedingungen, einen Speicher mit einer $\text{Größe} = 200\,\text{kByte} \cdot 7\,\text{FPS} \cdot 1500\text{s} \approx 2\,\text{GByte}$ benötigen. Der schlussendliche Bedarf an Speicher kann somit erst ermittelt werden, wenn das Video Komprimierungsverfahren festgelegt wurde. Für die Realisierung der temporären Speicherung scheint ein Ringpuffer die beste Wahl zu sein.

Die letzte Anforderung betrifft die Anzeige der Videodaten. Diese Anforderung wird durch die geforderte Verwendung eines Multimedia-Players vereinfacht. Das einzige Teilproblem, welches sich aus der Visualisierung ergibt, ist, dass ein eigenes Kommunikationsprotokoll festgelegt werden muss, welche den Datenaustausch zwischen SENSE-Knoten und Visualisierungssoftware ermöglicht.

Die zu implementierenden Funktionen sind auf zwei Computersysteme aufgeteilt. Die Anwendung am SENSE-Knoten ist für die Videokomprimierung, -Speicherung sowie Streamen zuständig. Das Programm am PC ist für den Empfang, Decodierung und Anzeige des Video-Streams zuständig. In Abbildung 6 werden die zu entwickelnden Programme durch die dick umrandeten Kästchen hervor gehoben.

2. Videokompression

Um digital aufgenommen Videobildern so natürlich wie möglich wiederzugeben, wird versucht, die Auflösung der digitalen Bilder soweit wie möglich zu erhöhen, sodass das menschliche Auge die körnige Struktur der Pixel nicht mehr wahrnehmen kann. Das Verlangen nach hochauflösenden Bildern und Videos hat dazu geführt, dass heute in vielen Wohnzimmern hochauflösende Fernsehgeräte zu finden sind. Würde man dabei ein nicht komprimiertes, hochauflösendes Video mit einer Bildrate von 25 FPS abspielen wollen, so würde dies einen Datenstrom mit einer Datenrate von über 100 MByte/s hervorrufen. Denkt man daran, ein Video mit dieser Datenrate zu speichern, würde man einerseits eine Festplatte benötigen, die auch mindestens diese Datenrate aufweist und zusätzlich auch eine sehr große Speichergröße besitzt. Diese Datenraten würden in heutiger Zeit zwar keine Probleme darstellen, jedoch stehen nicht immer diese Bandbreiten zur Verfügung. Man denke dabei nur an eine Videoübertragung per Funk. Um trotzdem Videos mit hoher Datenrate über Kommunikationskanäle mit geringer Datenrate übertragen zu können, ist es ganz klar, dass die Videobilddaten komprimiert werden müssen. Da von Anbeginn der ersten Kommunikationssysteme man diesem Problem gegenüberstand, wurden bis heute unzählige Verfahren zur Videokomprimierung entwickelt.

Im Folgenden werden einige grundlegende Kompressionstechniken vorgestellt, um Daten im Allgemeinen sowie Videodaten im Speziellen zu komprimieren. Anschließend werden einige derzeit gängige Videokomprimierungsstandards kurz vorgestellt.

2.1 Grundlegende Verfahren für die Videokompression

Videobildkompressionsverfahren können grundsätzlich in drei Kategorien unterteilt werden. Nämlich der Entropie-, Quellen- und Hybrid-Codierung. Quellen-Codierungsverfahren setzen sich zum Ziel, die Ausgangsdaten so aufzubereiten, dass die Trennung von relevanten und irrelevanten Daten möglich wird. Die Entfernung der irrelevanten Daten führt zwar dazu, dass die Datenmenge reduziert wird. Jedoch passiert dies mit einem bestimmten Grad an Informationsverlusten. Solche Verfahren berücksichtigen die Eigenschaften des menschlichen Auges und sind somit nicht allgemein auf Daten anwendbar.
Verfahren, die der Entropie-Codierung angehören, zeichnen sich dadurch aus, dass es bei der Codierung/Decodierung zu keinen Informationsverlusten kommt. Diese Verfahren sind dabei unabhängig vom Typ der Daten. Hybrid-Codierungen bauen auf den Techniken der Entropie- und Quellen-

Codierung auf. Tabelle 1 zeigt einen Überblick über verschiedene Verfahren für die Videokompression, gegliedert in die drei verschiedenen Codierungskategorien [ES98, S. 5-6], [Mil95, S. 5-6].

Tabelle 1: Kategorisierung von Codiertechniken (Quelle: [ES98, S. 8])

Kategorie	Beispiele
Quellen-Codierung	Farbunterabtastung Transformations-Codierung
Entropie-Codierung	Lauflängen-Codierung Codierung mit variable Länge
Hybrid-Codierung	H.261, H.263, H.264 MPEG-1, MPEG-2, MPEG-4

2.1.1 Farbunterabtastung

Die einfachste Kompressionsmethode berücksichtigt die physischen Eigenschaften des menschlichen Auges. So ist das menschliche Auge in der Lage, Helligkeitsunterschiede besser zu unterscheiden als Farbunterschiede.

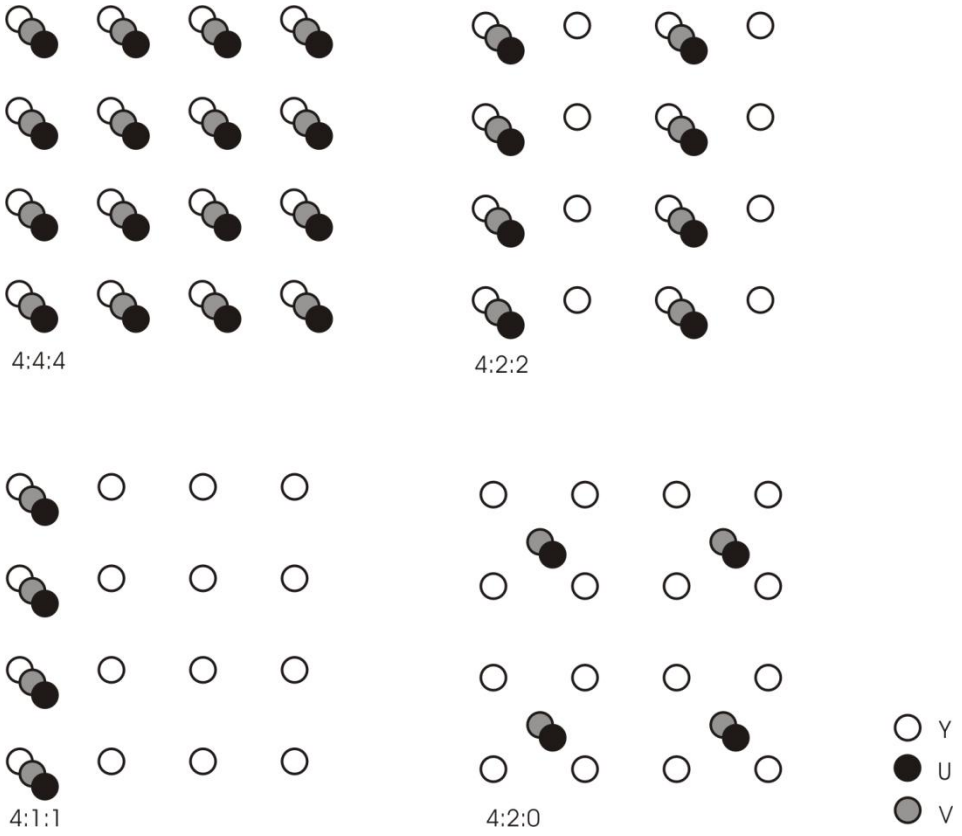


Abbildung 7: Unterabtastformate (Quelle: [Haf99, S. 20])

Um diese Eigenheit für die Videodatenkompression zu nützen, muss das Bild in einem entsprechenden Farbraum dargestellt werden. Dies kann zum Beispiel der YUV-Farbraum sein. In diesem Farbraum setzt sich ein Bildpunkt (Pixel) aus einer Helligkeitskomponente (Y) und zwei Farbdifferenzkomponenten (U und V) zusammen. Im YUV-Farbraum ist es nun einfach möglich, die Farbdifferenzkomponenten mit einer geringeren Rate abzutasten als die Helligkeitskomponenten. Abbildung 7 zeigt dazu gängige Unterabtastformate. Angenommen, man verwendet eine Komponentenauflösung von acht Bit, so werden für die Speicherung von YUV (4:4:4) 4 x 4 Y-Werte + 4 x 4 U-Werte + 4 x 4 V-Werte = 48 Byte benötigt. Bei YUV (4:2:2) hingegen werden die Farbdifferenzkomponenten nur eines jeden zweiten horizontalen Bildpunktes für die Weiterverarbeitung heran gezogen. Bei YUV (4:2:0) wird durch die Platzierung inmitten vierer Bildpunkte angedeutet, dass die verwendeten Farbdifferenzkomponenten sich aus den jeweils angrenzenden Farbdifferenzkomponenten bestimmt. YUV (4:2:0), YUV(4:1:1) sowie YUV (4:2:0) benötigen im Gegensatz zu YUV (4:4:4) 4 x 4 Y-Werte + 2 x 2 U-Werte + 2 x 2 V-Werte = 32 Byte. Trotz der geringeren Anzahl an Bytes ist in einem Bild kein sichtbarer Unterschied zwischen dem Bild, welche sämtliche Farbkomponenten verwendet und dem Bild mit unterabgetasteten Farbkomponenten zu erkennen [ES98, S. 9, 40], [Haf99, S. 19-20].

2.1.2 Transformations-Codierung

Die meisten modernen Kompressionsstandards nutzen Transformationen, um die im Zeitbereich vorliegenden Daten in eine andere mathematische Darstellung umzuwandeln, welche für die Kompression geeigneter ist. Durch die Transformation an sich wird zwar keine Komprimierung erreicht, sondern erst nachdem die Daten der Transformation einer Quantisierung unterzogen wurden, können die Bilddaten effektiv mit einer Entropie-Codierung komprimiert werden [ES98, S. 17].

Diskrete Kosinus-Transformation

Für die Transformation mit der diskreten Kosinus-Transformation (DKT) muss ein Bild in Blöcke geteilt werden. Angenommen $F(x,y)$ ist so ein Block mit der Größe 8 x 8 Pixel, wobei x die Zeile und y die Spalte des Blockes angibt. Die zweidimensionale DKT ist für diesen Block durch

$$f(k,n) = \frac{C(k)}{2}\frac{C(n)}{2}\sum_{x=0}^{7}\sum_{y=0}^{7} F(x,y)\cos\left(\frac{(2x+1)\pi k}{16}\right)\cos\left(\frac{(2y+1)\pi n}{16}\right) \quad (1)$$

definiert. Die inverse DKT hingegen ist mit

$$F(x,y) = \sum_{k=0}^{7}\sum_{n=0}^{7} \frac{C(k)}{2}\frac{C(n)}{2} f(k,n)\cos\left(\frac{(2x+1)\pi k}{16}\right)\cos\left(\frac{(2y+1)\pi n}{16}\right) \quad (2)$$

definiert, wobei

$$C(z) := \begin{cases} \frac{1}{\sqrt{2}} & f\ddot{u}r\ z = 0 \\ 1 & f\ddot{u}r\ z > 0 \end{cases} \quad (3)$$

gilt. Der Block $F(x,y)$ kann dabei als Matrix mit der kanonischen Standardbasis aufgefasst werden. Durch die Transformation wird nun $F(x,y)$ durch eine andere Basis dargestellt. Die 64 Elemente der neuen Basis sind die sogenannten Basisbilder. Um das Bild $F(x,y)$ aus den Basisfunktionen zu kon-

struieren, wird die Koeffizientenmatrix $f(k,n)$ benötigt. Der erste Koeffizient dieser Matrix $f(k,n)$ heißt DC-Koeffizient und gibt die Grundwerte von $F(x,y)$ an. Die restlichen Koeffizienten sind AC-Koeffizienten und geben die steigende vertikale und horizontale Frequenz in $F(x,y)$ an.
Durch die Verwendung der Kosinus-Funktion sowie der Division durch zwei ergeben sich nicht ganzzahlige Koeffizienten aus den zu transformierenden ganzzahligen Werten. Für die Speicherung dieser Koeffizienten werden somit mehr Bits benötigt als für die Darstellung der Bildpunkte an sich. Erst nachdem die Koeffizienten einer Quantisierung und Rundung unterzogen wurden, kann damit eine Kompression durchgeführt werden. Die Quantisierung entfernt in Folge hochfrequente irrelevante Bildinformationen, was dazu führt, dass ein Großteil der Koeffizienten in der $f(k,n)$ Matrix Null wird.
Die DKT ist theoretisch verlustfrei invertierbar, was jedoch praktisch durch die begrenzte Rechengenauigkeit nicht allgemein erfüllt ist. Deswegen wird die Berechnung der DKT häufig ganzzahlig durchgeführt, womit anstelle von 64 Multiplikationen und 63 Additionen, durchschnittlich eine Multiplikation und neun Additionen für die Berechnung eines DKT-Koeffizienten benötigt werden [Mil95, S. 18-22].

Integer-Transformation

Eine Weiterentwicklung der DKT ist die Integer-Transformation. Die Integer-Transformation stellt dabei eine Approximation der DKT dar. Die Herleitung der Integer-Transformation ist in [HCS07, S. 416- 417] aufgeführt. Durch die Verwendung einer Festkomma-Arithmetik ist die Definition der inversen Transformation exakt definiert und es wird die Entstehung von Rundungsfehler durch verschiedene Implementierungen vermieden. Die Transformation wird wie folgt definiert:

$$Y = C_f X {C_f}^T \tag{4}$$

wobei C_f wie folgt definiert ist:

$$C_f = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \tag{5}$$

Durch die Transformation werden die Werte in der Matrix X durch eine neue Basis in Y dargestellt. Infolge der Skalierung der Werte ergeben sich ausschließlich ganzzahlige Werte. Dadurch, dass die einzelnen Zeilen eine unterschiedliche Norm besitzen, muss dies in der darauffolgenden Quantisierung ausgeglichen werden.
Die Inverse Transformation ist hingegen wie folgt definiert:

$$X = {C_i}^T Y C_i \tag{6}$$

und C_i

$$C_i = \begin{bmatrix} 1 & 1 & 1 & 0{,}5 \\ 1 & 0{,}5 & -1 & -1 \\ 1 & -0{,}5 & -1 & 1 \\ 1 & -1 & 1 & -0{,}5 \end{bmatrix} \tag{7}$$

Durch die Integer-Transformation wird zwar die Leistung der Kompression, verglichen zur DKT, nicht verbessert, jedoch liegt der Vorteil in der starken Reduzierung des Rechenaufwandes. So benötigt die Transformation ausschließlich Additionen und Bit-Schiebe Operationen für die Multiplikation mit Zwei und Division durch Zwei [Str09, S. 303-305], [HCS07, S. 415-416], [TK96, S. 40].

2.1.3 Lauflängen-Codierung

In digitalisierten Bildern kommen des Öfteren Folgen von identischen Zeichen vor. Durch das Ersetzen solcher Zeichen durch Codes können Daten erheblich reduziert werden. Das Beispiel in Abbildung 8 illustriert dies zum besseren Verständnis.
Die Abarbeitung erfolgt dabei zeichenweise von links nach rechts. Wenn beim schrittweisen Durchgehen der Originaldaten mindestens vier gleiche Zeichen auftreten, so werden diese durch eine spezielle Zeichenfolge ersetzt. Diese Zeichenfolge beginnt in diesem Beispiel mit einem einzigartigen Steuerzeichen „/“, gefolgt von der Anzahl der zu ersetzenden Zeichen „4“ und dem Zeichen, welches in den Originaldaten ersetzt wird „C“. Das Ergebnis der Lauflängen-Codierung ist im unteren Teil der Abbildung zu sehen. Beim Decodieren der komprimierten Daten erkennt der Decoder anhand des Steuerzeichens den Beginn einer Sequenz und wandelt diese wieder in die ursprünglichen Originaldaten um.

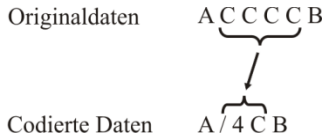


Abbildung 8: Beispiel Lauflängen-Codierung (Quelle: [ES98, S. 10])

Die Effizienz der Lauflängen-Codierung ist dabei sehr stark abhängig von der Folge von gleichen Zeichen im Datenstrom. Deswegen wird dieses Verfahren nur ergänzend zu anderen Entropie-Codierungen eingesetzt [ES98, S. 9-10], [Tor95, S. 11].

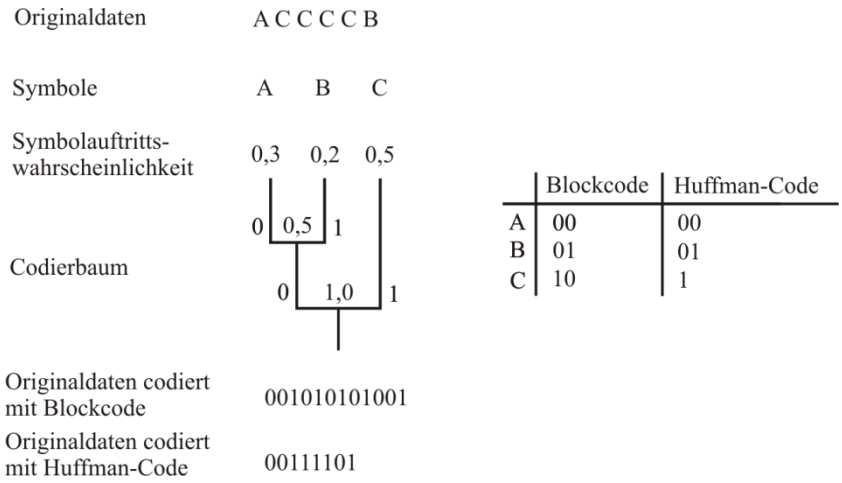
2.1.4 Codierung mit variabler Länge

Zeichen in Form von Datenbytes müssen grundsätzlich nicht mit einer fixen Anzahl von Bits codiert werden. Grundlage für Kompressionsverfahren, die eine variable Codewortlänge einsetzen, ist die unterschiedliche Auftrittswahrscheinlichkeit von Zeichen in einer Zeichenfolge. So wird bei der Codierung dem Zeichen mit der höchsten Auftrittswahrscheinlichkeit ein kürzeres Codewort zugeteilt als einem Zeichen mit niedriger Auftrittswahrscheinlichkeit. Zwei Vertreter für diese Klasse der Codierung sind die Huffman-Codierung sowie die Arithmetische Codierung, die im Folgenden näher erklärt werden [ES98, S. 9].

Huffman-Codierung

Die Huffman-Codierung setzt voraus, dass für die zu codierenden Zeichen die jeweiligen Auftrittswahrscheinlichkeiten vorliegen. Abbildung 9 zeigt ein einfaches Beispiel, welches die Vorgehensweise der Huffman-Code Generierung zeigt.
Für die Codierung geht man folgendermaßen vor. Man ersetzt in jedem Durchgang sukzessive zwei

Symbole mit der niedrigsten Auftrittswahrscheinlichkeit durch ein neues Symbol. Dabei wird auch die Auftrittswahrscheinlichkeit addiert, wobei für den nächsten Durchgang jeweils die Auftrittswahrscheinlichkeit dieser zusammengefassten Symbole heranzuziehen ist. Dieses Ersetzen wird solange gemacht, bis kein Ersetzen mehr möglich ist. Dadurch entsteht ein Graph mit einer Wurzel und mehreren Ästen. Werden die Äste systematisch mit „0“ und „1“ beschriftet, so ergibt sich beim Durchschreiten von der Wurzel bist zum Astende ein Codewort, welches dem Symbol am jeweiligen Astende zugeordnet wird. Daraus bildet sich für die Codierung und Decodierung notwendige Huffman-Code-Tabelle. Diese Tabelle muss dem Decoder bekannt sein und gegebenenfalls im Datenstrom enthalten sein [Mil95, S.11-15].



	Blockcode	Huffman-Code
A	00	00
B	01	01
C	10	1

Abbildung 9: Beispiel Huffman-Codierung (Quelle: [Mil95, S. 14])

Arithmetische Codierung

Die Huffman-Codierung besitzt den Nachteil, dass bei einem binären Alphabet die Entropie wegen der ganzzahligen Codewortlängen nicht erreicht wird. Dieser Nachteil tritt bei der arithmetischen Codierung nicht auf.

Die Vorgehensweise bei der Arithmetischen-Codierung ist, dass die Folge der Eingangsdaten auf eine Gleitkommazahlbereich von (0,1] abgebildet werden (siehe Abbildung 10). In jedem Schritt wird das Intervall in Teile aufgeteilt, welcher der Auftrittswahrscheinlichkeit der Symbole entspricht. Dabei legt das in jedem Schritt zu codierende Symbol fest, welches Intervall ein weiteres Mal mit dem Teilungsverhältnis der Auftrittswahrscheinlichkeit unterteilt wird. Diese sukzessive Unterteilung wird solange gemacht, bis alle Eingangsdaten codiert wurden. Nachdem alle Symbole codiert wurden, wird den Eingangsdaten ein Wert aus dem schlussendlich erhaltenen Werteintervall zugeordnet. Dazu wird ein Wert gewählt, der sich aus einer möglichst geringen Anzahl an Bits darstellen lässt. Für den Decodierungsvorgang müssen dem Decoder die Eingabezeichen mit den dazugehörigen Auftrittswahrscheinlichkeiten bekannt sein [Mil95, S.15-17].

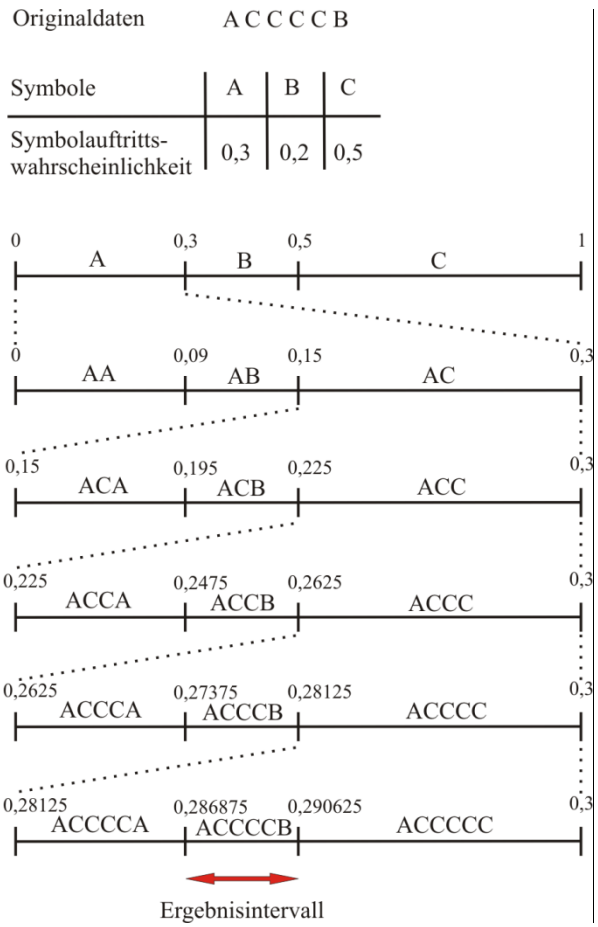


Abbildung 10: Beispiel Arithmetische Codierung (Quelle: [Mil95, S. 16])

2.2 Videokomprimierungsstandards

Die im vorigen Kapitel grundlegenden Codierungsverfahren werden zum Teil in verschiedenen Komprimierungsstandards verwendet. Als Stellvertreter für die Vielzahl an Videokomprimierungsstandards werden in Folge die Standards H.261, H.263, H.264 und MEPG-4 näher behandelt. Aus diesen ausgewählten Standards sind MPEG-4 als auch H.264 zwei sich am aktuellen Stand der Technik befindliche Videokompressionsstandards. Dabei sollte der MPEG-4-Standard nicht als reiner Videokomprimierungsstandard angesehen werden, da dieser wesentlich mehr beinhaltet. So beinhaltet MPEG-4 unter anderem mit Einschränkungen die Standards H.263 und H.264, weswegen H.263 und H.264 auch hier behandelt werden. Die Beschreibung des bereits älteren Standards H.261 ist dahin begründet, da dieser die wesentliche Grundstruktur der H.26x-Standards beschreibt, auf die die später entwickelten H.263 und H.264 Standards aufbauen. Dabei widmen sich die H.26x-Standards verglichen zu MPEG-4-Standards ausschließlich der Videokompression [Ric03, S. 92-93].

2.2.1 H.261

H.261 ist ein Videocodec, welcher von der International Telecommunication Union (ITU) für die Übertragung von digitalen Farbvideos entwickelt wurde. Der Codec wurde speziell für Datenraten von 64 kBit/s ausgelegt, welche bei Integrated Services Digital Network (ISDN) vorzufinden sind. H.261 unterstützt ausschließlich zwei Videobildauflösungen, nämlich 352 x 288 Pixel (CIF) und 176 x 144 Pixel (QCIF).
Das Blockdiagramm in Abbildung 11 zeigt die einzelnen Blöcke des H.261 Videocodierers. Dieser sind im Wesentlichen der Quellcodierer, der Video Multiplex Codierer und die Codierungssteuerung. Die einzelnen Blöcke werden in Folge noch näher beschrieben [HCS07, S. 239-240].

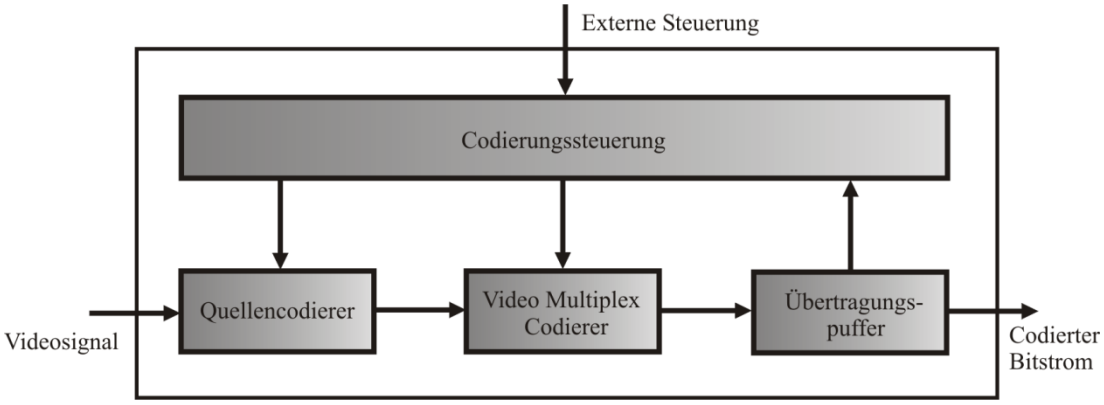


Abbildung 11: Blockdiagramm H.261 Videocodierer (Quelle: [HCS07, S. 240])

Quellencodierer

Der Quellcodierer besteht grundsätzlich aus drei Elementen. Diese sind die Bewegungsvorhersage, die Transformation und die Quantisierung. Abbildung 12 zeigt den Zusammenhang zwischen den Blöcken innerhalb des Quellcodierers.
Der erste Schritt der Quellcodierung besteht in der Videobildvorverarbeitung. Dazu wird das Video-

bild im Y, C_B, C_R Format codiert, um im Anschluss eine Farbunterabtastung durchführen zu können. Dazu wird das Abtastformat (4:2:0) verwendet, welches in Abbildung 7 gezeigt wird. Für die anschließende Weiterverarbeitung werden die einzelnen Komponenten aufgeteilt. Für die Transformation werden 8 x 8 Werte einer jeweiligen Komponente zu einem Block zusammengefasst und transformiert. Nebenbei gibt es noch sogenannte Makroblöcke, welche aus vier Helligkeitswertblöcken (Y) sowie jeweils einen Farbdifferenzblock (C_B, C_R) bestehen. Makroblöcke hingegen finden bei der Bewegungsvorhersage Verwendung.
Die einzelnen Blöcke können in zwei verschiedenen Modi codiert werden. Im Intraframe-Modus werden alle Blöcke des aktuellen zu codierenden Videobildes codiert, während im Interframe-Modus ausschließlich Videobilddifferenzen codiert werden. Eine einfache Implementierung des Interframe-Modus wäre das vorangehende Videobild als Vorhersage für das aktuelle Videobild zu benutzen. Durch die Subtraktion dieser beiden Videobilder erhält man die Änderungen zwischen den Videobildern. Infolgedessen wird nur die Änderung zwischen den beiden Videobildern codiert, was zu einer Reduktion des Informationsgehaltes führt.
Der Interframe-Modus kann verbessert werden, indem eine Bewegungsschätzung eingesetzt wird. Die Bewegungsschätzung vergleicht makroblockweise das letze im Videobildspeicher gespeicherte Videobild mit dem aktuellen Videobild und liefert daraus eine Menge von Bewegungsvektoren. Solch ein Bewegungsvektor wird dabei, wenn möglich, für jeden Makroblock berechnet und gibt dessen Translation in zwei Dimensionen an. Die x- und y-Koordinate der Translation werden durch einen ganzzahligen Wert im Bereich von ± 15 Pixeln angegeben und sind auf den Bildbereich beschränkt. Die Bewegungskompensation versucht, mit Hilfe der Bewegungsvektoren aus dem vorangegangen Videobild ein Prädiktionsbild zu erzeugen, welches dem derzeit zu codierenden Videobild möglichst ähnlich sehen soll. Die Differenz zwischen dem aktuellen Videobild und dem vorausgesagtem Videobild ergibt den Prädiktionsfehler.

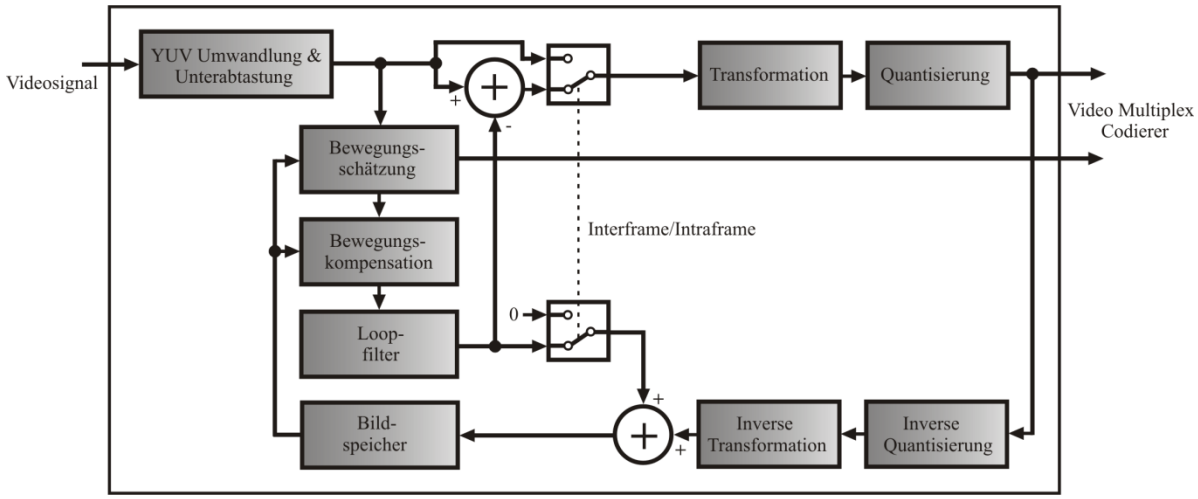


Abbildung 12: H.261 Quellencodierer (Quelle: [HCS07, S. 241])

Findet bei der Bewegungskompensation in einem Makroblock keine Übereinstimmung statt, so wird der Makroblock im Intraframe-Modus codiert. Ansonsten wird die Differenz zwischen dem geschätzten und dem aktuellen Videobild codiert.
Für die Codierung des ursprünglichen Videobildes beziehungsweise Differenzbildes wird die DKT

verwendet, wie sie in Kapitel 2.1 vorgestellt wurde. Die durch die Transformation erhaltenen Koeffizienten werden in Folge quantisiert. Dazu kommt ein Quantisierungskoeffizient zum Einsatz, welcher vom Codiermodus beziehungsweise von den einzelnen Koeffizienten abhängig ist. Dabei reduziert sich die Bitrate sowie die Videobildqualität, je höher der Quantisierungswert ist. Die quantisierten Koeffizienten werden einerseits an den Video Multiplex Codierer weitergegeben sowie auch der Inversen Quantisierung/Transformation zugeführt. Durch die Rücktransformation der Koeffizienten befindet sich im Videobildspeicher das gleiche Videobild, welches auch im Decoder ermittelt wird. Das Bild im Videobildspeicher wird in Folge für die nächste Bewegungsschätzung verwendet [HCS07, S. 240, 242], [Mil95, S. 32-34], [ES98, S. 59-62], [Str00, S. 262], [ITU93, S. 3-6].

Video Multiplex Codierer

Die Aufgabe des Video Multiplex Codierer, ist die Daten des Quellcodierers in Gruppen zusammenzufassen und dabei noch vorhandene Redundanzen zu entfernen. Die in den einzelnen Blöcken quantisierten DKT-Koeffizienten weisen durch eine Umordnung nach einem Zickzack-Muster eine große Anzahl von aufeinanderfolgenden Nullen auf. Durch eine Variante der Lauflängen-Codierung wird diese Folge von Nullen durch eine neue Symbolfolge dargestellt. Im Anschluss darauf werden diese Symbole durch eine Variante der Huffman-Codierung codiert. Die Codierung erfolgt anhand von fünf standardisierten Huffman-Tabellen, welche für die verschiedenen Daten wie Bewegungsvektoren, DKT-Koeffizienten usw. gegeben sind.
Schlussendlich werden die Daten in einer hierarchischen Struktur organisiert. H.261 verwendet vier Hierarchieebenen, welche in Abbildung 13 gezeigt werden. Der Bit-Stream besteht dabei aus Codewörtern variabler Länge. Diese sind jedoch anfällig gegenüber Übertragungsfehler, weswegen eine Übertragungssyntax gewählt wurde, welche eindeutige Codewörter in den verschiedenen Hierarchieebenen verwendet. Dadurch wird die Synchronisation bei Übertragungsfehlern wesentlich erleichtet [HCS07, S. 243-244], [Mil95, S. 35].

Codierungssteuerung

Die Codierungssteuerung verändert die Eigenschaften der einzelnen Funktionsblöcke des Codierers dahingehend, dass die gewünschte Videobildqualität beziehungsweise Bitrate eingehalten wird. Dazu wird der Inhalt des Übertragungspuffer (siehe Abbildung 11) überwacht, um daraus notwendige Eingriffe in die einzelnen Codierungsschritten durchzuführen. So kann die Codierungssteuerung zum Beispiel eine Vorverarbeitung der Videobilder dazu schalten, die Quantisierung verändern oder eine temporäre zeitliche Unterabtastung vornehmen [HCS07, S. 242-243].

2.2.2 H.263

H.263 ist eine Weiterentwicklung von H.261, welcher ebenfalls von der ITU standardisiert wurde. Dabei wurde die Grundstruktur von H.261 übernommen und durch eine Reihe von Verfahren erweitert beziehungsweise verbessert. Dadurch weist H.263, verglichen zu H.261, eine deutlich höhere Codierungseffizienz auf. Ebenfalls wurde die Unterstützung einer größeren Anzahl von Videobildauflösungen hinzugefügt, mit der die Codierung von hoch auflösenden Videobildern wie 16 CIF (1408 x 1152 Pixel) möglich wird. Dadurch, dass H.263 eine Weiterentwicklung von H.261 ist,

werden in Folge nur die wesentlichen Änderungen überblicksweise beschrieben. Dabei ist festzuhalten, dass H.261 nicht kompatibel zu H.263 und umgekehrt ist [HCS07, S. 295-297], [ES98, S. 63].

Quellencodierer

Ein wesentlicher Anteil der gesteigerten Codierungseffizienz wird durch die Quellencodierung erreicht. Dabei wurde die Grundstruktur der Quellcodierung von H.261 übernommen (siehe Abbildung 12) und durch einige Verfahren erweitert beziehungsweise vorhandene verbessert.
So ist die Bewegungskompensation bei H.263 als auch bei H.261 weiterhin optional. Die Genauigkeit der Bewegungskompensation an sich wurde jedoch gegenüber H.261 verdoppelt und besitzt eine Genauigkeit eines halben Pixels. Optional ist auch die Angabe von bis zu vier Bewegungsvektoren pro Makroblock, beziehungsweise die Angabe von Bewegungsvektoren ohne Einschränkung auf den Bildbereich, möglich. Dies führt dazu, dass der Vorhersagerestfehler reduziert werden kann, was wiederum zu einer Reduzierung der Datenrate führt.
Neben dem Interframe- und Intraframe-Modus ist es bei H.263 optional möglich, den P-B-Frame-Modus zu verwenden. Mit dem P-B-Frame-Modus ist es möglich, bidirektionale Vorhersagen zu machen. Ein P-B-Frame besteht aus zwei Videobildern, die als eine Einheit codiert werden. Das P-Frame ist dabei eine Vorhersage des zuletzt decodierten P-Frame. Das B-Frame hingegen ist eine Vorhersage aus dem zuletzt decodierten P-Frame und dem momentan decodierten P-Frame. Der Standard beschränkt die Anzahl der B-Frames auf ein Frame zwischen zwei P-Frames ein, um damit die Decodier-Verzögerung zu reduzieren. Der P-B-Frame-Modus kann verwendet werden, um bei geringfügiger Erhöhung der Bitrate die Videobildrate zu verdoppeln. Diese Anwendung führt zu einer signifikanten Erhöhung der subjektiven Videoqualität. Die zweite Anwendungsmöglichkeit besteht darin, die Videobildrate nicht zu verdoppeln, sondern die durch die B-Frames entstehende bessere Kompression die Datenrate zu reduzieren [HCS07, S. 297-298, 315-318], [ES98, S. 63-64], [ITU05, S. 11-12].

Video Multiplex Codierer

Neben der Möglichkeit der Codierung mittels Huffman-Codierung gibt es bei H.263 die optionale Möglichkeit, die Codierung mittels syntaxbasierender arithmetischer Codierung durchzuführen. Diese basiert grundsätzlich auf den in Kapitel 2.1 vorgestellter Methode der arithmetischen Codierung. Der Vorteil von der syntaxbasierenden arithmetischen Codierung liegt in der besseren Komprimierung der Daten als bei der Huffman-Codierung. Ein Nachteil ist bei der anspruchsvollen Implementierung zu finden.
Die Video Multiplexing Hierarchie ist hingegen zu H.261 identisch geblieben (siehe Abbildung 13). Die Änderungen gegenüber H.261 sind in den einzelnen Hierarchiestufen zu finden. So ist es bei H.263 zwingend Start-Codewörter byteweise auszurichten. Durch diese byteweise Ausrichtung vereinfacht sich das Finden des Start-Codewortes nach einem Übertragungsfehler und reduziert somit Hardwarekosten.
Die restlichen Veränderungen gegenüber H.261 betreffen die Optimierung und Ergänzung der hinzugekommenen optionalen Codierverfahren wie dem P-B-Frame-Modus. Dadurch, dass Teile der hierarchischen Struktur bei H.263 optional sind, kann der Codec so konfiguriert werden, dass der

Videodatenstrom entweder eine möglichst niedrige Datenrate oder eine bessere Fehlertoleranz aufweist [HCS07, S. 295-296, 298-306].

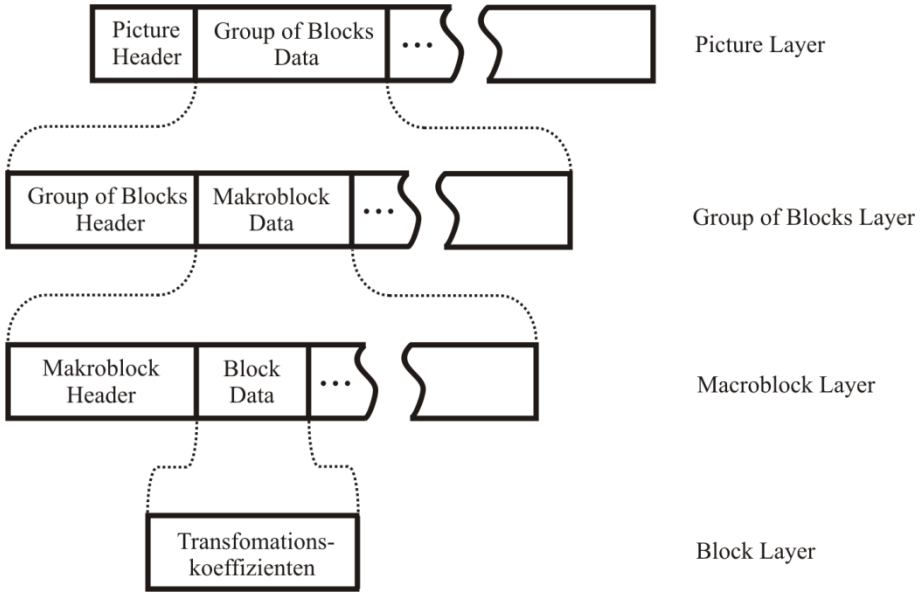


Abbildung 13: Hierarchische Struktur eines H.261 und H.263 Frames (Quelle: [HCS07, S. 244])

2.2.3 H.264

Das Ziel der ITU bei der Entwicklung des aktuellsten Videokompressions-Standard H.264 war, die Codierungseffizienz gegenüber H.263 nochmals zu steigern und dabei aber eine ähnliche Videobildqualität wie bei H.263 zu erzielen.
Ein fundamentaler Schritt war es das Design in zwei eigene Layer zu trennen, nämlich den Netzwerkadaptierungs-Layer sowie den Videocodierungs-Layer. Der Videocodierungs-Layer ist für die effektive Darstellung der Videobilder zuständig, während der Netzwerkadaptierungs-Layer für die Kapselung der Videodaten in Datenpakete zuständig ist [HCS07, S. 407-408].

Quellencodierer

Bei H.261 und H.263 wird im Intraframe-Modus die DKT auf die einzelnen Makroblöcke angewendet, um in Folge irrelevante Informationen aus den Daten entfernen zu können. Jedoch weisen Videobilder, welche im Intraframe-Modus codiert sind, verglichen zu denen im Interframe-Modus codierten Videobilder trotzdem eine große Anzahl von Bits auf. Bei H.264 wird nun eine Eigenschaft von Videobildern ausgenutzt, die darauf basiert, dass benachbarte Blöcke meist sehr ähnliche Helligkeitswerte- sowie Farbdifferenzwerte aufweisen. Aufbauend auf dieser Eigenschaft können örtliche Vorhersagen von benachbarten, bereits verarbeiteten, Blöcken gemacht werden, was zu einer Reduzierung der Daten im Interframe-Modus führt. Die Vorhersage wird entweder in 4 x 4 Pixel große Unterblöcke oder 16 x 16 Pixel große Makroblöcke durchgeführt. Für die Vorhersage sind für die Blöcke mit der Größe von 4 x 4 neun Modi und für die Blöcke mit 16 x 16 Pixel vier Modi definiert. Die Wahl des Modus ist dabei auf die Texturrichtung abzustimmen. Ein Beispiel für einen Modus mit einer Blockgröße von 4 x 4 Pixel wird in Abbildung 14 gezeigt. Im Modus 2 ergibt

sich zum Beispiel der Bildpunkt a durch $a = (A + B + C + D + E + F + G + H + 4)/8$. Die weiteren Modi können in [ITU10, S. 30-38] nachgelesen werden.

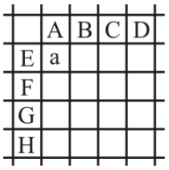


Abbildung 14: Örtliche Vorhersage (Quelle: [Str09, S. 297])

Bei der Codierung im Interframe-Modus wird bei H.264 weiterhin mit Datenreduktionsverfahren gearbeitet, welche auf Bewegungsschätzung und -kompensation basieren. H.264 unterstützt die meisten Methoden der Vorgängerstandards, sie werden jedoch durch eine Reihe von neuen Verfahren verbessert. Diese neuen Verfahren betreffen dabei ausschließlich die Bewegungsschätzung.
Eine Erneuerung bei der Bewegungsschätzung betrifft die Größe der Blöcke für die Bewegungsschätzung. So ist es bei H.264 möglich, jeden Makroblock wahlweise in eine unterschiedliche Anzahl von Blöcken, die unterschiedliche Blockgrößen und -formen besitzen, aufzuteilen. Die Anzahl der Blockgrößen und –formen ist dabei durch den Standard vorgegeben. So wird im Modus 1 ein Makroblock nicht unterteilt und nur ein Bewegungsvektor angegeben, während im Modus 7 der Makroblock in 16 gleich große Unterblöcke (4 x 4 Pixel) aufgeteilt werden, wo für jeden Unterblock ein Bewegungsvektor angegeben werden kann. Der Vorteil durch die Auswahl von verschiedenen Blockgroßen ist, dass die Vorhersage genauer gemacht werden kann. Dies führt dazu, dass die Datenrate reduziert werden kann und das bei feineren Bilddetails die subjektive Videobildqualität steigern lässt.
Eine weitere Verbesserung betrifft die Genauigkeit der Bewegungsvektoren. Diese wurde bei H.264 gegenüber H.263 ein weiteres Mal verdoppelt und besitzt somit bei H.264 eine Genauigkeit von einem Viertel Pixel. Dies erhöht die Codierungseffizienz bei Videos mit hoher Auflösung und Bitrate.
Des Weiteren wird bei der Bewegungsschätzung die Einschränkung bezüglich der Verwendung des ausschließlich zuletzt gespeicherten Videobildes aufgehoben. So ist es bei der Bewegungsschätzung in H.264 möglich, sich auf mehrere Referenzvideobilder bei der Codierung im Interframes-Modus zu beziehen. Dabei können bis zu vier unterschiedliche Referenzvideobilder ausgewählt werden, was ebenfalls zu einer Steigerung der Videobildqualität sowie der Codiereffizienz führt.
Ein schlussendlich letzter Punkt bei der Quellencodierung betrifft die Transformation. Bei H.264 wird nicht mehr die DKT für Transformation der 8 x 8 Pixel Blöcke verwendet, sondern diese durch die Integer-Transformation ersetzt. Die Funktionsweise wird in Kapitel 2.1 näher beschrieben. Die Transformation wird dabei auf 4 x 4 Pixel Große Blöcke angewendet, was die Güte der Voraussage im Intraframe- beziehungsweise Interframe-Modus steigert. Des Weiteren lassen sich Bilddetails besser lokalisieren und Blockartefakte reduzieren [HCS07, S. 410-414], [Str09, S. 296-298, 304-305].

Video Multiplex Codierer

Bei H.264 werden zwei neue Verfahren für die Entropiecodierung eingeführt. Eine dieser Methoden ist die kontextabhängige Lauflängencodierung. Diese verwendet im Gegensatz zur normalen Lauf-

längencodierung eine universal verwendbare Codiertabelle, die auf dem exponentiellen Golomb-Code basiert. Abhängig von den bereits verarbeiteten Symbolen wird zwischen zwei Codiertabellen gewechselt.
Die Effizienz der Entropiecodierung lässt sich weiter steigern, wenn die kontextabhängige binäre arithmetische Codierung verwendet wird. Diese Codierungsmethode ist eine Abwandlung der im Kapitel 2.1 vorgestellten arithmetischen Codierung, wobei die Codiertabellen dem vorliegenden Kontext angepasst werden können. Verglichen mit der kontextabhängigen Lauflängencodierung wird die Bitrate bei kontextabhängiger binärer arithmetischer Codierung um bis zu 15 % verringert [Str09, S. 306-308]. Details zu diesen Verfahren sind in [ITU10, ab S. 209] zu finden.

2.2.4 MPEG-4

Die Moving Picture Experts Group (MPEG) setzte sich neben der ITU ebenfalls zum Ziel, einen Standard für die Übertragung von digitalen Video- und Audio-Daten zu spezifizieren, der sowohl effizient, flexibel, aber auch für zukünftige Erweiterungen offen ist. MPEG-4 stellt den aktuellsten Standard der ISO/IEC dar, welcher MPEG-1 und MPEG-2 ablösen sollte. Während die Vorgängerstandards ausschließlich statische Daten erzeugten, ist bei MPEG-4 die dynamische Gestaltung der Informationen möglich. Dabei stellt die Videokompression lediglich einen Teilaspekt der unterschiedlichen Möglichkeiten dar [Str09, S.289], [ES98, S.49].
Ein wichtiges Konzept von MPEG-4 ist der objektorientierte Ansatz. Dieser Ansatz ergibt sich aus der Tatsache, dass es kein ideales und universales Komprimierungsverfahren für alle Typen von Daten gibt. So ist es zum Beispiel möglich, einen Untertitel eines Films auf zwei verschiedene Arten zu codieren.
Die herkömmliche Methode für die Codierung ist es, den Untertitel direkt im Videobild darzustellen und gemeinsam zu codieren. Durch die Verwendung einer blockweisen Transformation kann es aber dazu kommen, dass die Kanten des Untertitels nicht ideal codiert werden können und bei der Decodierung von diesen Kanten Artefakte entstehen. Durch den objektorientierten Ansatz kann zum Beispiel eine Videoszene aus verschiedenen Objekten bestehen. Wie zum Beispiel einem Video- und Untertitelobjekt. Durch die Trennung in zwei verschiedene Typen von Objekten kann auf jedes dieser Objekte die optimalste Codierungsmethode angewendet werden. So kann zum Beispiel der Text in ASCII codiert werden und das Video mit H.264 komprimiert werden. Dadurch kann in einem ersten Schritt eine deutlich bessere Kompression der Daten ermöglicht werden. Ein weiterer Vorteil dieses Ansatzes ist, dass die Darstellung des Untertitels deutlich besser möglich ist.
Abbildung 15 zeigt das Grundkonzept des objektorientierten Ansatzes. Auf der linken Seite werden in einem ersten optionalen Schritt Objekte erzeugt, indem aus dem Videoobjekt weitere Objekte extrahiert werden. Die einzelnen Objekte werden in Anschluss durch jeweils eigene Codierer komprimiert und mittels Multiplexer zusammengeführt. Auf der anderen Seite werden die Daten durch einen Demultiplexer auf die einzelnen jeweiligen Decodierer aufgeteilt. Die decodierten einzelnen Objekte werden in Anschluss zusammengeführt und angezeigt. Durch diese Vorgehensweise kann zwar eine optimale Codierung erreicht werden, jedoch werden eine Vielzahl von Encoder- und Decoder-Algorithmen benötigt. Das wiederrum macht die Realisierung eines MPEG-4-Decoders gegenüber eines MPEG-2 Decoders äußerst aufwendig [Sym04, S. 196-199].

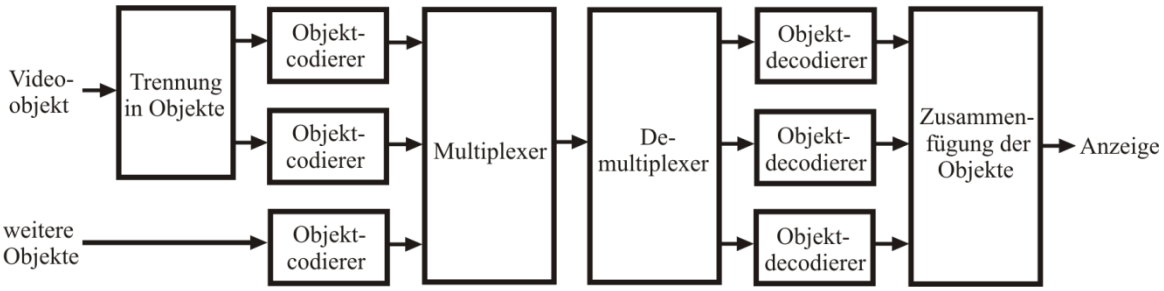


Abbildung 15: Objektcodierung in MPEG-4 (Quelle: [Sym01, S. 198])

Der MPEG-4-Standard an sich besteht derzeit aus 27 Teilen und wird kontinuierlich durch weitere Teile erweitert. Einige dieser Teile bauen wiederum auf anderen Standards auf. So baut zum Beispiel der Teil 2 (Visual) auf dem Konzept von H.263 auf, beziehungsweise Teil 10 (Advanced Video Coding) auf H.264 auf [Ric03, S. 92-93], [ISO04, S. 1], [ISO09, S. 1].

3. Streaming-Technologien

Durch den Siegeszug des Internets und der paketbasierenden Kommunikation gibt es verschiedenste Anforderungen für die Übertragung von Daten. Eine wichtige Anforderung ist der verlässliche Transport der Daten. Dies wird erreicht, indem eine eigene Kontrollverbindung zwischen Client und Server aufgebaut wird. Damit kann der Client das empfangene Paket bestätigen und bei Bedarf eine erneute Übermittlung der Daten erzwingen. Diese Vorgehensweise der Übermittlung ist für zeitdiskrete Daten, wie statische Inhalte einer Webseite optimal, jedoch nicht für Multimediadaten wie Audio und Video [Kün01, S. 11-12].

3.1 Streamen von Multimediadaten

Multimediadaten haben durch ihren zeitbasierten Inhalt andere Anforderungen als statische Inhalte. Dadurch, dass Multimediadaten neben den zeitbasierten Eigenschaften in der Regel auch eine umfangreiche Datenmenge aufweisen, ist es eine äußerst ineffiziente Vorgehensweise, die Daten vollständig zu übertragen und erst dann mit der Wiedergabe zu beginnen. Um die Wartezeit für die komplette Übertragung von Multimediadaten zu ersparen, wäre es möglich, dass der Empfänger bereits während der Übertragung der Daten mit der Wiedergabe beginnt. Und genau diese Strategie wird beim Streamen von Multimediadaten auch gemacht. Ein Server sendet einen kontinuierlichen Datenstrom an den Client. Sobald die Daten den Client erreichen, kann dieser sie decodieren und wiedergeben. Ein typisches Problem, welches sich beim Streamen von Daten ergibt, betrifft die unterschiedlichen Empfangszeitpunkte der einzelnen Pakete am Client. Diese kommen zustande, wenn der Server einerseits die Daten nicht rechtzeitig verschickt, beziehungsweise wenn die Pakete unterschiedliche Wege zum Empfänger nehmen. Deswegen ist es erforderlich, eine bestimmte Menge von Daten zu empfangen und diese zwischen zu puffern, bevor sie weiterverarbeitet werden. Durch die Pufferung am Empfänger können gewisse Bandbreitenschwankungen des Übertragungskanals toleriert werden, jedoch entsteht durch die Pufferung eine Wiedergabeverzögerung [Kün01, S.12-13].

3.1.1 Live- und On-Demand-Inhalte

Medien welche über einen Streaming-Dienst angeboten werden, sind in der Bereitstellung durch Live- und On-Demand-Inhalte zu unterscheiden. On-Demand-Inhalte liegen bereits vor dem Zeitpunkt des Streamen vollständig in einem bestimmten Format vor. Stehen diese Inhalte auf einem Server, so kann der Benutzer diese zu einem beliebigen Zeitpunkt abrufen. Dazu sendet der Nutzer eine Anforderung an diesen Server. Ist diese Anforderung korrekt, so beginnt der Server die gefor-

derte Datei an den Client zu übertragen. Damit sind vorproduzierte Inhalte zu jedem Zeitpunkt für den Benutzer verfügbar.
Bei Live-Inhalten liegen vor dem Zeitpunkt des Streamen noch keine Daten zur Verfügung. Dabei wird erst zum Startpunkt des Streamen ein Encoder gestartet. Dieser Encoder liefert in Echtzeit Daten an den Server. Der Server nimmt diese Daten entgegen und leitet diese an den Client weiter [Kün01, S. 14].

3.1.2 Unicast- und Multicast-Übertragung

Bei einer Unicast-Übertragung wird für jeden Teilnehmer eine eigene Verbindung geöffnet. Über diese Verbindung werden dann sowohl die Nutzdaten als auch die Steuerdaten übertragen. Bei On-Demand-Inhalten ist diese Vorgehensweise zwingend notwendig, da es dem Benutzer nur so ermöglicht werden kann, zu jeden beliebigen Zeitpunkten auf die Multimediadaten zuzugreifen. Nebenbei ist diese vollständige Kontrolle notwendig, damit der Benutzer individuell auf sein Verlangen die Übertragung steuern kann, wie zum Beispiel das Pausieren der Wiedergabe oder Änderung der Wiedergabeposition.
Multicast-Übertragungen hingegen zeichnen sich dadurch aus, dass alle Teilnehmer gleichzeitig mit den identischen Daten beliefert werden. Die Verbreitung der Information ähnelt dabei einer Rundfunksendung. Deswegen werden Live-Übertragungen mit Multicasting an die Teilnehmer versendet. Ein Vorteil von Multicasting ist, dass der Server nicht überlastet wird, da dieser nicht zu jedem Teilnehmer exklusiv eine Verbindung aufbauen muss. Der Hauptvorteil liegt jedoch darin, dass deutlich weniger Netzwerkressourcen benötigt werden [Kün01, S.27].

3.2 Streaming-Protokolle

Für die Übertragung von statischen Inhalten werden heut zu Tage üblicherweise TCP/IP basierte Protokolle eingesetzt. Diese besitzen Mechanismen, welche die zu verlässliche Übermittlung der Daten garantieren. Jedoch besitzen Multimediadaten andere Anforderungen. So macht bei der Übertragung von Multimediadaten das fehlende Echtzeitverhalten des Internets Probleme. Deswegen wurden spezielle Streaming-Protokolle für diese Art von Daten entwickelt, um den Kommunikationsprozess zwischen Server und Client zu optimieren.
Streaming Protokolle setzen meist auf der Transportschicht des ISO/OSI-Models auf. Verwendete Protokolle aus der Transportschicht sind UDP und TCP, wobei ersteres bevorzugt wird. Die Gründe dafür liegen darin, dass UDP im Gegensatz zu TCP keine Mechanismen besitzt, die garantieren, dass Datenpakete beim Empfänger ankommen. Da diese Mechanismen Bandbreite benötigen, wird UDP gegenüber TCP bevorzugt. Jedoch müssen bei der Verwendung von UDP am Empfänger spezielle Vorkehrungen getroffen werden, um die fehlerhaften oder fehlenden Datenpakete zu tolerieren [Kün01, S. 22-23].
Im Folgenden werden zwei gängige Streaming Protokoll-Familien näher beschrieben. Diese sind das Real-Time Transport/Control/Streaming-Protocol und das Real-Time Messaging Protocol.

3.2.1 Real-Time Transport Protocol

Das Real-Time Transport Protocol (RTP), Real-Time Control Protocol (RTCP) sowie das Real-Time Streaming Protocol (RTSP) wurde von der IETF als Audio und Videoübertragungsstand für Multicastkonferenzen entwickelt. Dabei beschreibt RFC 3550 RTP und RTCP sowie RFC 2326 RTSP. Diese Protokolle wurden so designt, dass diese unabhängig von den darunterliegenden Schichten des ISO/OSI-Models arbeiten können (siehe Abbildung 16). Da sich RTP bewährt hat, wird dieser Standard heute bei Videokonferenzübertragungen, Webcasts und Fernsehübertragungen eingesetzt. Die Gründe für die breite Verwendung von RTP liegen darin, dass RTP sehr gut skalierbar ist [Per08, S. 9-10].

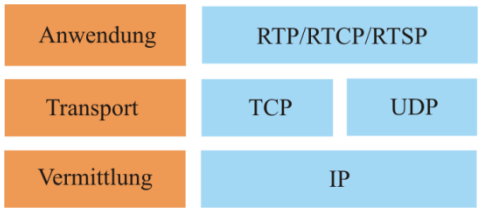


Abbildung 16: RTP/RTCP/RTSP im ISO/OSI-Protokollstapel

Real-Time Transport Protocol

Die Übertragung der Multimediadaten erfolgt bei RTP in Sitzungen. Für die Übertragung der einzelnen Medienarten wird jeweils eine eigene Sitzung initiiert. Die Verwaltung der Sitzungen erfolgt hingegen mit RTCP. Durch eine RTP-Sitzung ist es sowohl möglich, eine Unicast-Verbindung zwischen zwei Benutzern aufzubauen als auch Multicast-Übertragungen zu einer Gruppe von Teilnehmern. Einer Sitzung gehört im Allgemeinen eine Gruppe von Teilnehmern an, welche über RTP miteinander kommunizieren. Dabei kann ein Teilnehmer in mehreren Sitzungen aktiv sein. Die Identifizierung des Teilnehmers erfolgt durch ein Netzwerkadresse-Portnummer-Paar. Abbildung 17 zeigt dazu ein Beispiel einer Übertragung von Audio und Videodaten. In diesem Beispiel sendet der Teilnehmer 1 sowohl Audio- als auch Videodaten, Teilnehmer 2 und 3 Teilnehmer sind hingegen die Empfänger. Durch die Trennung der Echtzeitdaten in jeweils eigene Sitzungen ist es möglich, dass ein Teilnehmer nicht zwingenderweise an jeder Sitzung teilnehmen muss. So verzichtet zum Beispiel Teilnehmer 3 an der Teilnahme der Videositzung, da die Kommunikationsverbindung zu wenig Bandbreite besitzt [Per08, S. 67-68].
Dadurch, dass Netzwerke im Allgemeinen heterogen sind, wurde im Standard RFC 3550 das Konzept des Übersetzers und Mixers eingeführt. Durch einen Mixer besteht die Möglichkeit, einen Teilnehmer trotz seiner geringen Bandbreite mit anderen Teilnehmern, welcher eine größere Bandbreite zur Verfügung steht, zu kommunizieren. Der Mixer führt dabei eine Transcodierung der Multimediadaten durch, um sie für den Kommunikationskanal mit der geringeren Bandbreite anzupassen.
Mit einem Übersetzer ist es möglich, eine Übertragung zwischen verschiedenen Übertragungstechniken, wie zum Beispiel IPv6 und ATM, aber auch den Kommunikationsverkehr durch eine Firewall zu ermöglichen [SCF+03, S. 6-7].

Aufbau eines Real-Time Transport Protocol Paketes

Ein RTP-Paket besteht im Wesentlichen aus vier Teilen. Der erste Teil besteht aus einem zwingend erforderlichen RTP-Header, gefolgt von einer optionalen Header-Erweiterung, einem optionalen Nutzdaten-Header sowie der Nutzdaten an sich. Der genaue Aufbau eines RTP-Pakets wird in Abbildung 18 gezeigt. Im Folgenden werden einige wichtige Felder des RTP-Paketes näher beschrieben.

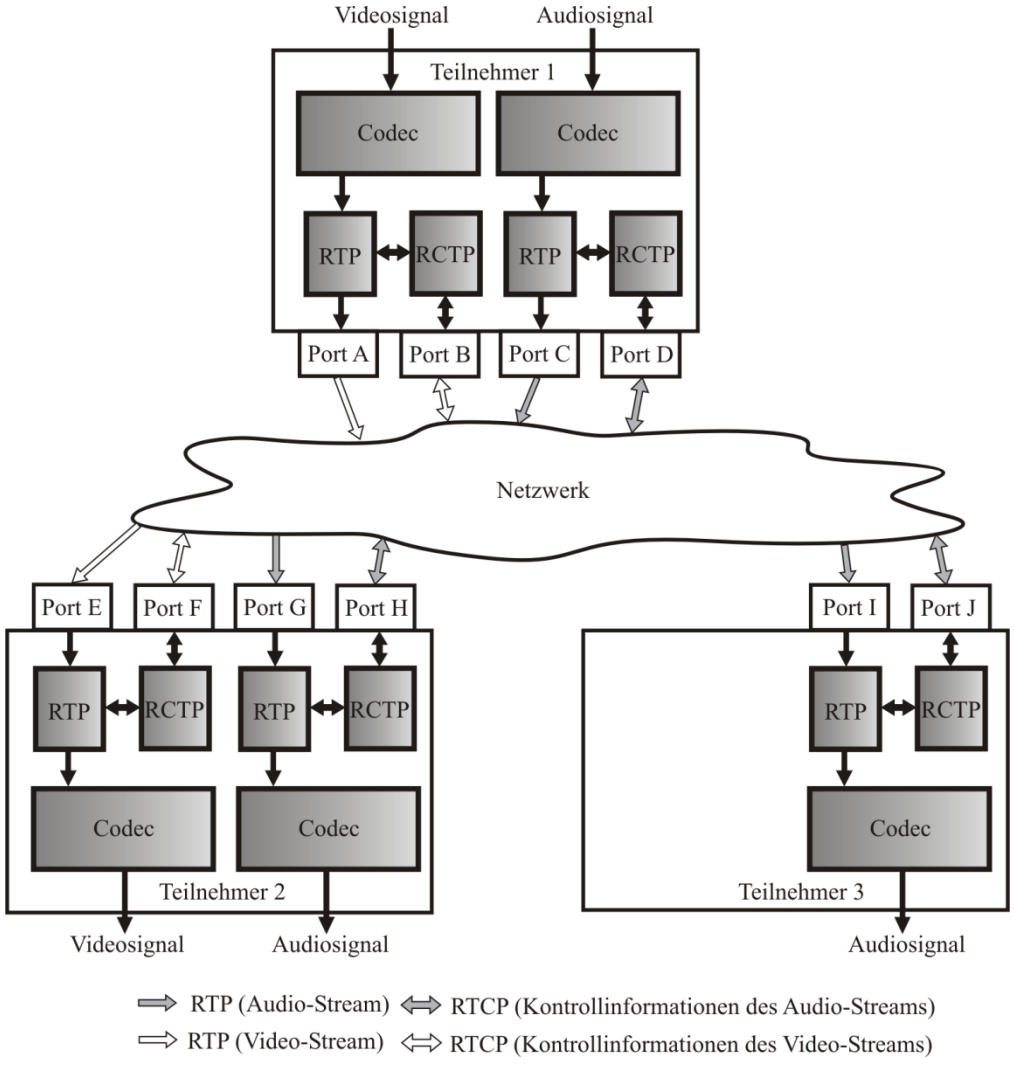


Abbildung 17: Beispiel einer Übertragung mit RTP/RTCP

Ein wichtiges Feld im RTP-Paket stellt das Nutzdatentypfeld dar. Durch dieses Feld wird es ermöglicht, den Typ der Nutzdaten, welcher durch das RTP-Paket transportiert wird, zu identifizieren. Damit kann eine Anwendung, die auf RTP aufbaut, die Daten an die für diesen Datentyp passende Verarbeitung zuordnen. Die Zuordnung des Nutzdatentyps erfolgt durch Profile, welche den Zusammenhang zwischen der Nutzdatentypnummer und des Multimediaformats regelt. Die Zuordnung der Formate wird im Standard RFC 3551 beschrieben. Tabelle 2 zeigt einen Ausschnitt dieser Zuordnungen. Der Wertebereich der Nutzdatentypen gliedert sich dabei in einen Bereich, wo eine stati-

sche Zuordnung festgelegt wurde und in einen Bereich für die dynamische Zuordnung der Datenformate. Für die Zuweisung der Formate im dynamischen Bereich werden oft wieder weitere Protokolle benötigt, wie zum Beispiel SDP [Per08, S. 70], [SC03, S. 31-32], [SCF+03, S.13].
Die Sequenznummer dient dazu, um einzelne RTP-Pakete zu identifizieren. Der Wertebereich der Sequenznummer umfasst einen 16-Bit-Integer ohne Vorzeichen. Der Initialwert der Sequenznummer ist aus verschlüsselungstechnischen Gründen zufällig. Die Sequenznummer wird bei jedem gesendeten Paket um eins erhöht und beginnt bei einem Überlauf des Wertebereichen wieder bei null.

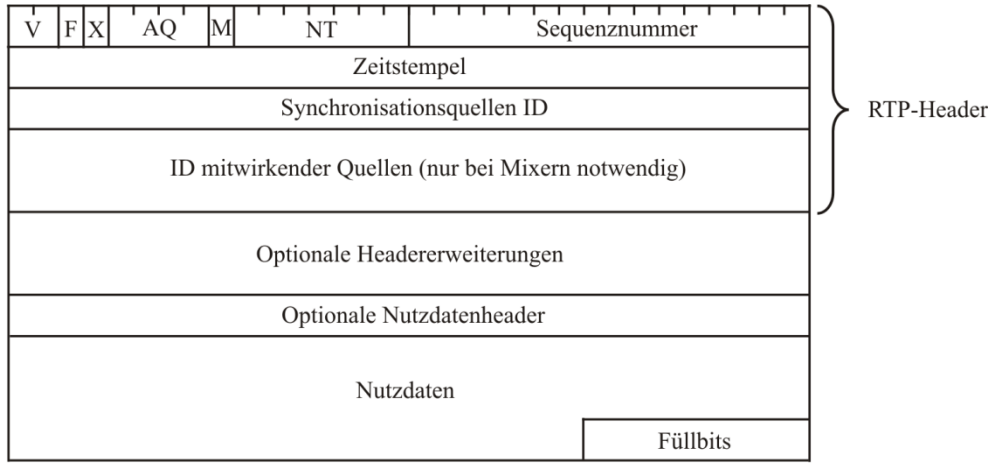


Abbildung 18: Aufbau eines RTP Paketes (Quelle: [Per08, S. 70])

Die hauptsächliche Funktion der Sequenznummer besteht darin, festzustellen, ob es Lücken in der Sequenznummernreihenfolge gibt. Stellt der Empfänger eine Lücke fest, so muss dieser durch eine spezielle Vorgehensweise reagieren, um die fehlenden Daten wiederherzustellen. Diese Vorgehensweise kann bei Audiodaten so aussehen, dass verlorengegangene Pakete durch zum Beispiel Hintergrundrauschen ersetzt werden.
Die zweite Funktion der Sequenznummer besteht darin, es dem Empfänger zu ermöglichen, die Sendereihenfolge der RTP-Pakete wiederherzustellen. Dabei ist hervorzuheben, dass die Sequenznummer nicht für die Wiedergabereihenfolge der in den RTP-Paketen beinhalten Nutzdaten zuständig ist. Dafür ist der Zeitstempel zuständig [Per08, S. 71-75, 229-232].

Der Zeitstempel ist ein 32-Bit-Integer, welcher mit einer medienabhängigen Rate erhöht wird. Der Initialisierungswert beziehungsweise Werteüberlaufverhalten ist dabei gleich wie bei der Sequenznummer. Hingegen gibt der Wert des Zeitstempelfeldes den Abtastzeitpunkt des ersten Bytes der Mediadaten im Nutzdatenfeld an. Der Zeitstempel an sich wird von einer Uhr mit einer bestimmten Genauigkeit abgeleitet, welcher monoton und linear fortlaufend ist. Durch den Zeitstempel wird der Wiedergabezeitpunkt am Empfänger festgelegt. Desweiteren wird der Zeitstempel für die synchronisierte Wiedergabe benötigt als auch für die Bestimmung des Jitters [Per08, S. 78-79], [SCF+03, S. 13-14].

Tabelle 2: Ausschnitt einiger Nutzdatentypnummern (Quelle: [SC03, S. 32-33])

Nutzdaten-typnummer	Nutzdatenformat	Beschreibung
0	PCMU (Audio)	Puls Code Modulation (µ-law)
14	MPA (Audio)	MPEG Audio (z.B.: Mp3)
31	H.261 (Video)	
32	MPV (Video)	MPEG I/II Video
33	MP2T (Audio/Video)	MPEG II Transport Stream
96-127	Dynamisch	

Der Umfang der Nutzdaten eines RTP-Paketes ist grundsätzlich nicht beschränkt. Die Beschränkung der Nutzdaten wird durch die MTU des Netzwerkes festgelegt. Dabei ist es möglich, dass mehrere Mediendaten-Frames in einem RTP-Packet zusammengefasst werden können. Jedoch ist im RTP-Header kein Feld vorgesehen um auf die Größe der Nutzdaten zu schließen. Dies erfordert somit einen zusätzlichen Rechenaufwand um die Nutzdatengröße zu Bestimmung, vor allem dann, wenn Füllbits am Ende des RTP-Paketes angehängt werden. Dem Empfänger stehen jedoch zwei Wege für die Bestimmung der Nutzdatengröße zur Verfügung.
Bei der ersten Methode wird eine fixe Größe der Mediendaten-Frames verwendet. Die Anzahl der Frames leitet sich aus den festgelegten Framegrößen und der RTP-Paketgröße ab. Die zweite Methode beruht darauf, dass manche Nutzdatenformate bereits Kennungen beinhalten, welche in den einzelnen Frames inkludiert sind und die Größe der Frames angibt. Der Empfänger der Nutzdaten muss die Nutzdaten parsen und erhält so den Startpunkt sowie die Anzahl der Frames. Dieses Vorgehen findet dann Verwendung, wenn die Frames eine variable Länge besitzen [Per08, S. 88-89].

Real-Time Control Protocol

Neben RTP kommt noch das Real-time Control Protocol zum Einsatz. RTCP ist für eine Reihe von Aufgaben zuständig. Diese Aufgaben umfassen die Teilnehmeridentifikation, Benachrichtigungen über Änderungen der Sitzungsteilnehmer, eine periodische Berichtserstattung über die Empfangsqualität und Informationen welche für die Synchronisation von verschiedenen Streams erforderlich sind. Der Datenaustausch erfolgt bei RTCP über einen von RTP getrennten Port. Jede RTP-Sitzung verwendet eine eigene RTCP-Verbindung, wobei RTP immer einen Port mit einer geraden Nummer und RTCP um eine eins größere Portnummer verwendet. Für den Datenaustausch mittels RTCP stehen fünf verschiedene Datenpakete zur Verfügung. Diese sind das Senderbericht-, Empfängerbericht-, Quellenangabe-, Teilnehmermanegment- und das Applikationsspezifische-Datenpaket [SCF+03, S. 19-21], [Per08, S. 97-98].

Real-Time Streaming Protocol

Das Real-Time Streaming Protocol ist ein flexibles Framework, um eine On-Demand-Übertragung von Multimediadaten zu ermöglichen. So kann RTSP als offene Schnittstelle zwischen Server und Client verschiedener Hersteller gesehen werden. RTSP ist einerseits für den Aufbau und andererseits für die Verwaltung von On-Demand-Streams zuständig [SRL98, S. 1, 5].

3.2.2 Real-Time Messaging Protocol

Das Real-Time Messaging Protocol (RTMP) wurde von Adobe [12] entwickelt, um die Übertragung von Audio- und Videodaten zwischen Adobe Flash Plattformen [13] wie Adobe Flash Media Server [14] und Adobe Flash Player [15] zu ermöglichen. Das ursprünglich proprietäre Protokoll wurde 2009 in einer ersten Spezifikation öffentlich zugänglich gemacht. RTMP baut typischerweise auf dem Real-Time Messaging Chunk Stream Protocol (RTMP Chunk Stream) auf (siehe Abbildung 19). Das Einsatzspektrum von RTMP und RTMP Chunk Stream reicht dabei von Unicast- und Multicast-Übertragungen bis hin zur Übertragung von On-Demand- und Live-Multimediadaten [Ado09b, S. 4].

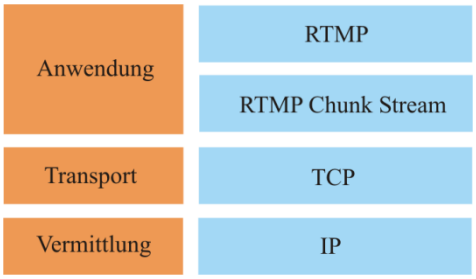


Abbildung 19: RTMP im TCP/IP-Protokollstapel

Real-Time Messaging Protocol

Das Real-Time Messaging Protocol ist für das Multiplexen und Paketieren der Multimedia-Streams zuständig. Dazu werden die Multimediadaten in Nachrichten gepackt und an ein Transportprotokoll weitergegeben. Für den Transport zwischen den Teilnehmern kann zum Beispiel RTMP Chunk Stream verwendet werden, jedoch ist auch der Einsatz eines anderen Protokolls möglich. Durch die Trennung in Nachrichten-Streams gibt es eine Möglichkeit, verschiedene Medienarten getrennt voneinander zu übertragen, wobei jeder Stream durch einen Identifier unterscheidbar gemacht wird. Dabei wird zwischen Video-, Audio-, Kommando-, Daten- und benutzerspezifischen Kommandonachrichten unterschieden.
Abbildung 20 zeigt ein Beispiel für Übertragung eines Audio- und Video-Streams mittels RTMP und RTMP Chunk Stream. In dem Beispiel sendet der Teilnehmer 1 die Daten zu den Teilnehmern 2 und 3. Wie auch bei RTP ist es bei RTMP möglich, dass ein Teilnehmer, welcher zum Beispiel eine zugeringe Bandbreite besitzt, auf bestimmte Medienarten verzichten kann (siehe Teilnehmer 3 in Abbildung 20). Verglichen mit RTP wird nur die halbe Anzahl an Ports benötigt, da bei RTMP das Multiplexen der Medien-Streams in der Anwendungsschicht durchgeführt wird [Ado09a, S. 2, 3], [Ado09c, S. 5].

Der Aufbau einer RTMP-Nachricht wird in Abbildung 21 gezeigt. Die Nachricht besteht dabei aus einem Headerteil und einem Nutzdatenteil. Die Anzahl der Felder einer RTMP-Nachricht sind verglichen mit einer RTP-Nachricht deutlich weniger und reduzieren sich auf die für das Streamen von Multimediadaten notwendigen Felder, wie Nachrichtentyp, Sequenznummer und Zeitstempel.

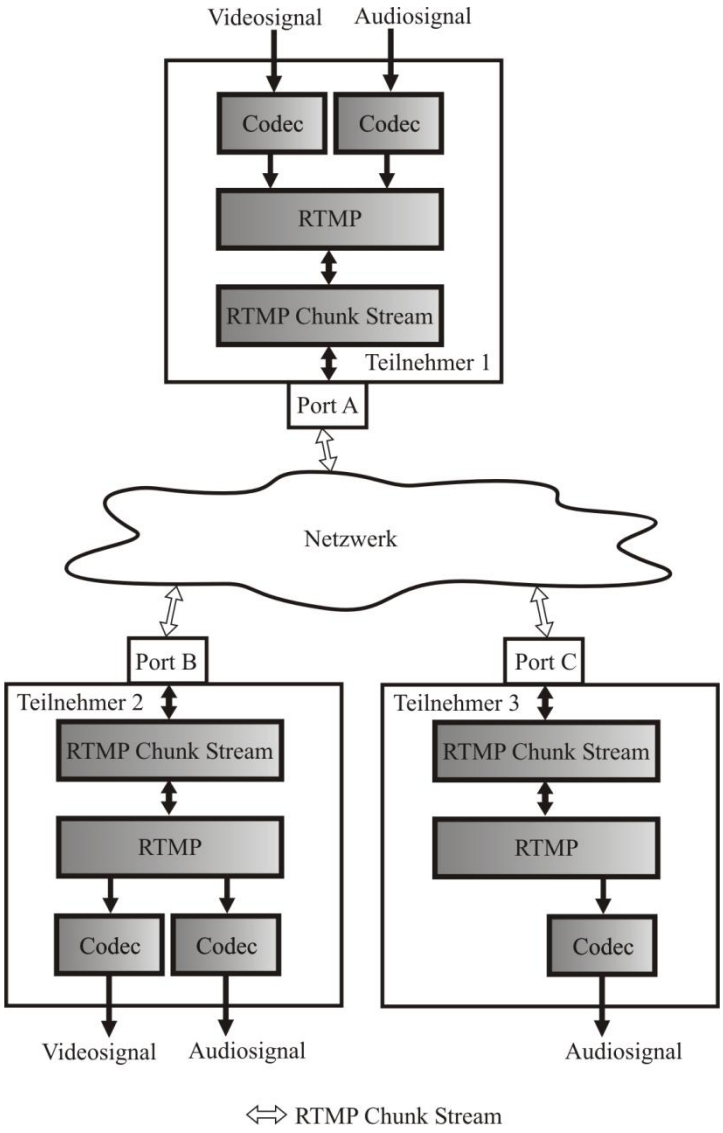


Abbildung 20: Beispiel einer Übertragung mit RTMP

Das erste Feld in einem Nachrichten-Paket gibt den Typ der Nachricht an. Dabei sind die IDs 1–7 für Steuerungsnachrichten reserviert. Das Nutzdatenlängenfeld mit einer Größe von drei Bytes repräsentiert die Anzahl der Nutzdatenbytes, welche nach dem Header folgen. Ähnlich wie bei RTP besitzt auch jede RTMP-Nachricht einen Zeitstempel mit einer Größe von 4 Bytes. Der Zeitstempel gibt bei RTMP einen Wert in Millisekunden an, der relativ zu einem nicht spezifizierten Zeitintervall steht. Typischerweise fängt der Zeitstempel mit dem Wert null an, was aber nicht zwingend erforderlich ist und steigt in Folge linear monoton an. Das Überlaufverhalten des Zeitstempels verhält sich gleich wie bei RTP. Durch den Zeitstempel wird es ermöglicht, die Nachrichten-Streams zu synchronisieren, Bandbreitenmessungen durchzuführen, Übertragungs-Jitter zu detektieren und Datenflusssteuerung durchzuführen. Die Unterscheidung der einzelnen Nachrichten-Streams wird durch die Nachrichten-Stream-ID ermöglicht, welche sich aus drei Bytes zusammen setzt. Gefolgt wird dieser Header schlussendlich von den Nutzdaten [Ado09a, S. 4-5].

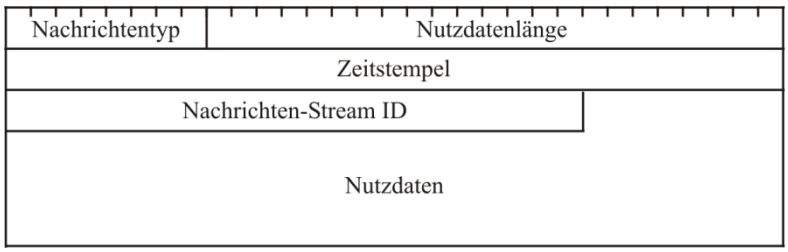


Abbildung 21: Aufbau eines RTMP-Nachrichten Paketes (Quelle: [Ado09a, S. 5])

Real-Time Messaging Protocol Chunk Stream

RTMP Chunk Stream stellt die Basis für Streaming-Anwendungen dar. Dabei wurde RTMP Chunk Stream für die Zusammenarbeit mit RTMP konzipiert, wobei aber auch andere Protokolle auf RTMP Chunk Stream aufsetzten können. Die Aufgabe des Protokolls besteht darin, Nachrichten für die darunter liegende Transportschicht zu multiplexen und paketieren.
Um Nachrichten über RTMP Chunk Stream übertragen zu können, ist eine spezielle Handshake-Prozedur notwendig. Die Identifizierung der Teilnehmer geschieht wie bei RTP über ein Netzwerkadressen-Portnummern-Paar. Nachdem eine Verbindung aufgebaut wurde, können eine Reihe von Nachrichten-Streams übertragen werden. Die Nachrichten-Streams werden dazu in einen Chunk-Stream umgewandelt und so die Nachrichten Stück für Stück übertragen. Damit der Empfänger die empfangen Chunks wieder zu Nachrichten zusammensetzen kann, werden die Chunks mit einer ID versehen.
Das Zerstückeln von großen Nachrichten in kleine Chunks ist notwendig, um große Nachrichten übertragen zu können. So besitzen große Nachrichten mit niedriger Priorität den Nachteil, dass sie den Kommunikationskanal für Nachrichten mit hoher Priorität zeitweise blockieren. Um dem entgegen zu wirken, kann die Chunk-Größe benutzerspezifisch für jede Verbindungrichtung individuell konfiguriert werden, wobei Größen von 128 bis 65536 Bytes möglich sind [Ado09b, S. 4, 13-14].

4. Machbarkeitsanalyse

Durch die in Kapitel 1.6 durchgeführte Aufteilung in Teilprobleme kristallisierten sich die Videokompression und die Videoübertragung als kritisch heraus. Deswegen ist das Ziel dieses Kapitels, diese beiden Teilprobleme genauer zu analysieren.

Da die beiden Teilprobleme im Zusammenhang stehen, bedarf es für ein weiteres Design der Software, diese auf einander abzustimmen. Deswegen wurde ein erster grober Entwurf der Software gemacht. Dieser wird im Blockdiagramm in Abbildung 22 gezeigt. Der linke Teil der Abbildung stellt die VCSU dar, welche am i.MX31 des SENSE-Knoten ausgeführt wird. Die rechte Seite stellt hingegen die VSUV am PC dar. Die Hauptaufgabe der VCSU ist es, Videobilder einzulesen und diese mit einem noch nicht näher bekannten Videokompressionsverfahren zu komprimieren. In Folge werden die komprimieren Videodaten in einem Ringpuffer gespeichert. Für die Speicherung der Daten steht an dieser Stelle bereits fest, dass ein Ringpuffer am besten für diese Aufgabe geeignet ist. Die Videoübertragung greift dabei ebenfalls auf den Ringpuffer zu, um an die für die Übertragung notwendigen Videodaten zu gelangen. Am PC wird der Video-Stream vom Multimedia-Player empfangen, decodiert und angezeigt. Die Bedienung der Funktionen erfolgt durch Steuerelemente in der VSUV, wobei der Datenaustausch zwischen der VCSU und VSUV über ein zu spezifizierendes Kommunikationsprotokoll erfolgt.
Im Folgenden wird auf die Videokompression und Videoübertragung näher eingegangen. Die Vorgehensweise war, sich einen Überblick über bestehende und brauchbare Lösungen zu machen und funktionsfähige Prototypen zu entwickeln. Als Kriterium für bestehende Lösungen wurde darauf geachtet, dass diese folgende Eigenschaften besitzen:

- kostenlos verfügbar
- quelloffener Programmcode
- passende Lizenzbestimmungen
- Performance
- gute Dokumentation

4.1 Videokompression

Im SENSE-Projekt ist es vorgesehen, dass bestimmte Videoknoten nur per Funk erreichbar sind. Diese Funkverbindung bietet eine Datenrate von ungefähr 1 MBit/s. Um bei dieser Datenrate eine Videoübertragung überhaupt zu ermöglichen beziehungsweise so gering wie möglich in Anspruch

zu nehmen, ist es notwendig, die Videodaten so gut wie möglich zu komprimieren.
Dabei stellt sich als erstes die Frage, welches Videokomprimierungsverfahren verwendet werden soll. Die Kriterien dafür sind äußerst konträr. So soll das Bildmaterial einerseits bestmöglich komprimiert werden, um so wenig Datenrate wie möglich zu beanspruchen, auf der anderen Seite soll die Bildqualität so gut sein, um Geschehnisse erkennen zu können. Dazu kommt, dass der Prozessor nur in geringster Weise in Anspruch genommen werden soll. Doch bietet sich für diese Aufgabe der integrierte MPEG-4/H.263-Encoder des i.MX31 gerade zu an.

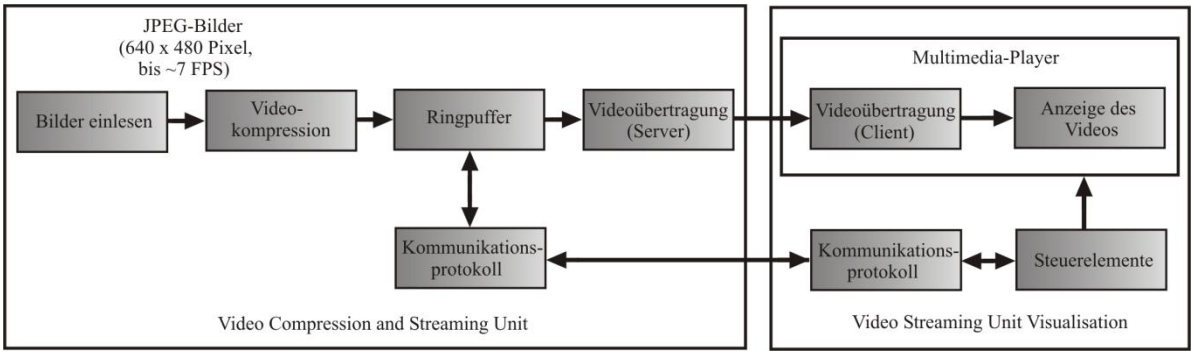


Abbildung 22: Abstraktes Blockdiagramm

4.1.1 Hantro MPEG-4/H.263-Encoder

Der Hantro MPEG-4/H.263-Encoder zeichnet sich vor allem durch seine in Hardware realisierte Videokomprimierung aus. Dadurch, dass die Komprimierung unabhängig von der CPU durchgeführt werden kann, belastet dieser die CPU kaum. Das Benutzerhandbuch des Encoders gibt Auskunft über die relevanten technischen Eigenschaften:

- MPEG-4 und H.263 kompatibel
- maximal zulässige Auflösung: 640 x 480 Pixel (VGA)
- maximal mögliche Framerate: ~30 FPS
- unterstützte Stream-Bitraten: 64, 128, 384, 768, … kBit/s

Die API des Encoders lässt auch benutzerdefinierte Auflösungen und Frameraten zu und deckt somit die Anforderungen bestmöglich ab. Jedoch benötigt der Encoder als Eingangsvideobilddaten eine YUV (4:2:0) planar Darstellung, was jedoch nicht gegen eine Verwendung des Encoders spricht [Han04, S. 8].

Durch die Festlegung für die Nutzung des Encoders ergibt sich ein neues Blockdiagramm der VCSU. So besteht die Möglichkeit, die Videokomprimierung wie in Abbildung 23 dargestellt, durchzuführen. Dazu werden die JPEG-Bilder eingelesenen und in Folge von JPEG nach YUV (4:2:0) transcodiert. Die YUV-Bilder können im Anschluss dem Encoder zugeführt werden.
Für die Aufgabe des Transcodierens musste jedoch eine softwaremäßige Lösung gefunden werden. Da das Erstellen eines JPEG-Decodier-Algorithmus den zeitlichen Rahmen dieser Arbeit sprengen würde, wurde versucht, auf bestehende Lösungen zurückzugreifen. Dazu wird in Folge FFmpeg sowie der Tiny JPEG Decoder näher betrachtet.

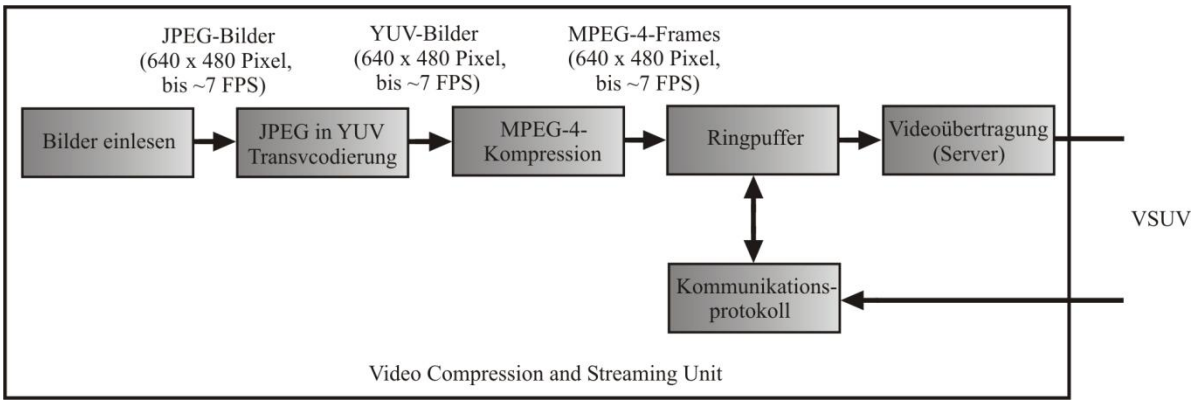


Abbildung 23: Abstraktes Blockdiagramm der VCSU mit MPEG-4-Encoder und JPEG-Bilder

4.1.2 FFmpeg

FFmpeg ist ein Open-Source-Projekt, welches sich durch seine Fülle von Audio- und Video Codecs einen Namen gemacht hat. Jedoch stellt FFmpeg nicht nur eine äußerst breite Palette von Codecs über den Transcoder (FFmpeg) zur Verfügung, sondern auch ein Anzeigeprogramm (FFplay) und Funktionalität für das Streamen von Multimedia (FFserver) [Lon11, S. 119]. Deshalb findet es in vielen Open-Source-Projekten wie zum Beispiel dem VLC Media Player Einsatz. Durch die GPL Lizenzbestimmungen gäbe es auch keine Probleme, diese zu nutzen [1].

Um zu evaluieren, ob FFmpeg für das Decodieren der JPEG-Bilder geeignet ist, wurde ein Testprogramm entwickelt. Dazu musste der Programmcode von FFmpeg mit dem Compiler des Linux-BSPs kompiliert werden, um FFmpeg unter Linux verwenden zu können. Die Kompilierung wurde vereinfacht, da FFmpeg vom BSP bereits unterstützt wird und somit nur eine Option im Linux Target Image Builder (LTIB) ausgewählt werden musste.
Das Testprogramm hatte die Aufgabe ein JPEG-Bild mit VGA-Auflösung zu laden und in ein YUV-Bild umzuwandeln. In einem ersten Test transcodierte das Programm einzelne Videobilder einwandfrei. Ein weiter Test welcher mehrmals das gleiche Videobild konvertierte, zeigte jedoch die Probleme deutlich auf. Dabei wurden folgende Messergebnisse gemacht:

- Transcodierrate (JPEG nach YUV (4:2:0) planar): ~ 5 FPS
- CPU Auslastung: > 95%
- Programmgröße: > 16 MByte (statische Einbindung der Bibliotheken)

Aus den Ergebnissen ist ersichtlich, dass das Transcodieren von JPEG-Bildern mit FFmpeg eine äußerst rechenintensive Aufgabe für den i.MX31 darstellt. Mit einer Transcodierrate von nur 5 FPS ist dies eines der beiden Kriterien, die gegen die Verwendung von FFmpeg spricht. Des Weiteren fiel zur Kompilierzeit auf, dass FFmpeg sehr umfangreich ist. Dies wirkte sich auf die Programmgröße aus. So besaß das Testprogramm bereits eine Größe von über 16 MByte und das obwohl beim Erstellen des Testprogrammes die Kompiliereinstellung für das statische Einbinden von Bibliotheken gemacht wurde. Ohne diese Einstellung lag die Programmgröße bei ungefähr 40 MByte. Dies ist ebenfalls ein Kriterium, das gegen die Verwendung von FFmpeg spricht.

4.1.3 Tiny JPEG Decoder

Wie auch FFmepg steht der Tiny JPEG Decoder unter GPL. Die Implementierung wurde dabei speziell für die Decodierung von JPEG-Bildern ausgerichtet. Bei der Entwicklung von Tiny JPEG-Decoder wurde darauf geachtet, kompakte Funktionen zur Verfügung zu stellen, die eine einfache Portierung auf verschiedenste Zielsysteme ermöglichen soll [2].

Um Tiny JPEG Decoder unter Linux nutzen zu können, musste der Programmcode mit dem ARM-spezifischen Compiler kompiliert werden. Die Kompilierung konnte dabei zügig und ohne Probleme durchgeführt werden. Um die Funktionalität zu testen, stellt der Entwickler bereits ein Demoprogramm zur Verfügung. Mit diesem ist es möglich, einzelne JPEG-Bilder in verschiedene Formate umzuwandeln, darunter auch YUV (4:2:0) planar. Dieser erste Test bestätigte die Funktion des Transcodieren von JPEG nach YUV (4:2:0). Des Weiteren stellte das Demoprogramm auch eine Benchmark-Funktion zur Verfügung. Diese wandelt immer das gleiche JPEG-Bild in das gewünschte Ausgangsformat um. Dazu wird dem Programm am Beginn des Benchmark-Durchlaufes die Anzahl der Umwandlungen angegeben. Daraufhin führt es die Umwandlungen durch und gibt die benötige Rechenzeit aus. Das Demoprogramm lieferte dabei folgende Ergebnisse:

- Transcodierrate (JPEG nach YUV (4:2:0) planar): > 100 FPS
- CPU Auslastung: > 95%
- Programmgröße: < 100 kByte

Zwar ist die CPU bei diesem Test praktisch auch voll ausgelastet, jedoch nur deswegen, weil sie ständig eine Umwandlung vornimmt. Da das EVS die Videobilder jedoch nur mit einer Framerate von bis zu 7 FPS liefert, würde die CPU-Belastung vermutlich immerhin noch bei ungefähr 14% liegen.

4.2 Videoübertragung

Neben der Videokomprimierung erfordert auch die Übertragung der Videodaten eine besondere Beachtung. So ist eine der Randbedingungen, dass die Videoübertragung über ein standardisiertes Verfahren erfolgen soll. Das gewählte Übertragungsprotokoll soll wiederum vom Multimedia-Player unterstützt werden, um den Video-Stream zu empfangen, zu decodieren und anzuzeigen zu können.
Als Übertragungsprotokoll wurde RTP gewählt. Die Gründe dafür sind, dass RTP einem offenen Standard unterliegt, weit verbreitet ist und von vielen Open-Source Projekten wiederzufinden ist. Mit der Festlegung des Streaming-Protokolls musste auch der Multimedia-Player festgelegt werden. Die Entscheidung fiel dabei auf den VLC Media Player. Dieser zeichnet sich dadurch aus, dass er einerseits RTP/RTCP/RTSP unterstützt und andererseits eine Vielzahl von Audio- und Video-Codecs mitbringt, um die Videodaten zu decodieren. Ein wichtiges Kriterium ist auch, dass der Player kostenlos erhältlich ist und die Möglichkeit bietet ihn in verschieden Anwendungen einzubetten. Da die Implementierung von RTP ein Thema für eine eigene Arbeit wäre, wurde auch hier versucht auf eine bestehende Lösung aufzubauen. Recherchen im Internet lieferten einige Ergebnisse. Unter anderem wurden folgende Open-Source-Projekte genauer betrachtet.

4.2.1 FFmpeg

Wie bereits im Kapitel 4.1.2 erwähnt, bietet FFmpeg ebenfalls Funktionen, Videos mittels RTP zu streamen. Jedoch konnte kein funktionsfähiges Demoprogramm in Betrieb genommen werden, sodass die Verwendung von FFmpeg verworfen wurde.

4.2.2 GStreamer

GStreamer bietet ähnliche Funktionalitäten wie FFmepg. Jedoch unterscheiden sich diese in der Architektur. Die GStreamer-Bibliothek wurde als Framework konzipiert, während FFmpeg über Bibliotheken Funktionen bereit stellt. Der Vorteil des Frameworks ist hingegen, dass schnell eine Videoverarbeitung realisiert werden kann, ohne Programmierkenntnisse zu besitzen. Die Beschreibung der Videoverarbeitungsschritte kann dabei sowohl über Konsolenbefehle als auch mit Hilfe von graphischen Editoren durchgeführt werden. Das Open-Source-Projekt welches unter LGPL steht, kann dabei durch die Plug-in-Architektur einfach erweitert werden [Kof08, S. 518]. GStreamer ist unter [3] beziehbar.

Eine erste Evaluierung von GStreamer zeigte, dass es möglich ist, Videos mittels RTP zu übertragen und diese mit dem VLC Media Player zu betrachten. Dazu wurde auf Vorteile, welche das Framework bietet, zurückgegriffen, sodass die beschriebene Funktionalität ohne schreiben einer einzigen Zeile Code auskam [LLS+08 S. 75]. Als Testdaten wurde eine Videodatei übertragen, welche eine Auflösung von 352 x 288 Pixel sowie eine Videobildrate von 25 FPS besitzt. Das Streamen mit GStreamer brachte dabei folgende Ergebnisse:

- CPU Auslastung: < 2%
- Programmgröße: konnte nicht bestimmt werden

Die Programmgröße konnte hingegen nicht festgestellt werden, da die Funktionen über verschiedene Dateien hinweg verteilt sind. Eine genauere Analyse wurde jedoch nicht durchgeführt.

4.2.3 Live 555 Streaming Media

Live 555 Streaming Media ist ein Open-Source-Projekt, welches ein Framework für das Streamen von Audio- und Videodaten bietet. Jedoch liegt hierbei nicht der Fokus auf der Entwicklung von Videoverarbeitungspipelines wie bei GStreamer, sondern dabei, ein Framework für die Integration in Programmen zur Verfügung zu stellen. Live 555 Media Streaming unterstützt eine Reihe von standardisierten Protokollen, wie zum Beispiel RTP/RTCP, RTSP, SIP usw., welche beim Streamen von Multimediadaten Verwendung finden [AAC+10, S. 1]. Der Programmcode des Live 555 Server ist unter [4] beziehbar und wurde unter LGPL veröffentlicht.

Für die Evaluierung der Streaming-Funktionen stehen einige Demoprojekte zur Verfügung. Diese wurden in einem ersten Schritt für die ARM-Plattform kompiliert. Nach der zeitintensiven Modifizierung der „make“ Dateien von Live 555 Media Streaming konnte ein Video-Stream erfolgreich vom i.MX31 an den PC übertragen werden. Als Testdaten wurde eine Videodatei übertragen, welche eine Auflösung von 640 x 480 Pixel besitzt und eine Videobildrate von 9 FPS aufweist. Die Wieder-

gabe des RTP-Streams wurde dabei mit dem VLC Media Player durchgeführt und lieferte folgende Ergebnisse:

- CPU Auslastung: < 1%
- Programmgröße: < 660 kByte

4.3 Ergebnisse der Machbarkeitsanalyse

Durch die gewonnen Erkenntnisse der Machbarkeitsanalyse konnten die wichtigsten Eigenschaften der verschiedenen Lösungsmöglichkeiten aufgezeigt werden. Ein Fazit über die Videokomprimierung, Videoübertragung und dem schlussendlichen Blockdiagramm schließen dieses Kapitel ab.

4.3.1 Videokomprimierung

Auf dem ersten Blick erscheint, dass der JPEG Tiny Decoder die Aufgabe des Transcodierens der JPEG- in YUV-Bilder am besten erfüllen zu könnten, da dieser eine gerade noch akzeptable Prozessorauslastung hervorruft. Doch stellte sich bei den späteren Tests heraus, dass die vom Entwickler bereitgestellte Benchmark sich fehlerhaft aufweist. Die Korrektur dessen und Wiederholung des Performancetests brachte zum Vorschein, dass die Transcodierrate unter dem Niveau von FFmepg liegt. Und zwar liegt diese nur bei ~3 FPS.
Durch die Tests von FFmpeg und dem Tiny JPEG Decoder ist durchaus eine Tendenz für den Rechenaufwand für das Transcodieren der Videobilder zu erkennen. Doch kann nicht ausgeschlossen werden, dass dies mit den verwendeten Bibliotheken besser möglich wäre. An dieser Stelle wird darauf hingewiesen, dass die verwendete Bibliothek vermutlich nicht optimal auf den i.MX31 portiert wurde. FFmpeg und der Tiny JPEG Decoder sind zwar für das Portieren auf andere Zielsysteme ausgelegt, jedoch wird nicht im Speziellen die i.MX-Prozessorfamilie von Freescale angeführt. Eine Eigenentwicklung beziehungsweise Optimierung dieser Bibliotheken auf die i.MX-Architektur wurde jedoch als zu zeitintensiv eingestuft und verworfen.

Tabelle 3: Bewertung von FFmpeg und dem Tiny JPEG Decoder

	FFmpeg	Tiny JPEG Decoder
Prozessorauslastung	−	−
Testprogrammgröße	−	+
Integration in eigene Anwendungen	+	+

Die relevanten Eigenschaften für das Transcodieren der JPEG- in YUV-Bilder werden in Tabelle 3 zusammengefassten und bewertet. Die Tabelle zeigt, dass beide Bibliotheken nicht für das Transcodieren geeignet sind, da beide den ARM-Prozessor des i.MX31 völlig auslasten würden.

Ursprünglich verfügte das EVS ausschließlich über ein Service, welches JPEG-Bilder über einen TCP/IP-Socket zur Verfügung stellt. Das Service wurde so ausgelegt, dass Videobilder mit VGA Auflösung und einer Videobildrate von mindestens einen doppelt so großen Wert als 7 FPS betrieben wird. Jedoch ist EVS kontenintern nur über 10 MBit an den Ethernet-Switch angebunden. Damit Videobilder trotzdem mit VGA-Auflösung und einer Videobildrate von mindestens 15 FPS übertragen werden können, wurde von den Entwicklern des EVS eine JPEG-Komprimierung implementiert.
Um trotzdem den MPEG-4-/H.263-Encoder verwenden zu können, ohne dazu eine Transkodierung der JPEG-Bilder vorzunehmen, wurde vom Entwickler des EVS ein weiteres Service hinzugefügt. Dieses Service ist in der Lage, YUV-Bilder für den verwendeten MPEG-4-/H.263-Encoder zu liefern. Dadurch, dass YUV-Bilder mit VGA-Auflösung und 7 FPS die knoteninterne Datenkommunikation überlastet würde, wurde die Videobildauflösung auf 320 x 240 Pixel (QVGA) reduziert. Die Reduktion der Auflösung wäre sowieso erforderlich gewesen, da die Übertragung eines Video-Streams mit VGA-Auflösung die Funkverbindung völlig Auslasten würde.

Das ursprüngliche Blockdiagramm aus Abbildung 22 Abbildung 24wurde somit ein weiteres Mal abgeändert und durch das Blockdiagramm in Abbildung 24 ersetzt. Durch die gewählte Vorgehensweise bei der Videoverarbeitung ergibt sich der Vorteil, dass keine Rechenzeit für das Transcodieren der JPEG- in YUV-Bilder benötigt wird. Durch die Verwendung des MPEG-4-/H.263-Encoders im i.MX31 kann auf ein äußerst fortgeschrittenes Komprimierungsverfahren zugegriffen werden. Jedoch hat diese vorgehensweises den Nachteil, dass die Videobildauflösung auf ein Viertel reduziert werden musste.

4.3.2 Videoübertragung

Für das Streamen mit RTP stehen drei verschiedene Open-Source-Projekte zur Verfügung. Die Zusammengefassten und bewerten Eigenschaften sind in Tabelle 4 tabellarisch aufbereitet. Dadurch, dass mit FFmpeg kein Demoprogramm erstellt werden konnte, werden für die Spalte keine Angaben gemacht, jedoch vollständigkeitshalber aufgelistet. Da in kurzer Zeit kein Demoprogramm erstellt werden konnte, kam eine Verwendung nicht weiter in Frage.

Tabelle 4: Bewertung von FFmpeg GStreamer und Live 555 Media Streaming

	FFmpeg	GStreamer	Live 555 Media Streaming
Prozessorauslastung	k.A.	+	+
Testprogrammgröße	k.A.	+	+
Integration in eigene Anwendungen	k.A.	−	+

Mit GStreamer gelang es hingegen durchaus, Videos zu übertragen. Jedoch ist eine geschlossene Softwareanwendung für die Funktionalitäten dieser Arbeit gewünscht, was die Integration in eigene Anwendungen erfordert. Dabei wurde beim Studieren des Codes erkannt, dass dafür es vermutlich ein zu großer Zeitaufwand notwendig gewesen wäre, um das Framework zu verstehen und die ge-

wünschte Funktionalität in die eigene Software zu integrieren. Deshalb wurde die Verwendung von GStreamer schlussendlich auch verworfen.
Die Wahl für das Streamen der Videodaten fiel auf Live 555 Media Streaming. Das Framework zeichnet sich gegenüber GStreamer vor allem dadurch aus, dass eine einfache Integration in eigene Programme möglich ist. Zwar bietet das Projekt, verglichen mit den anderen beiden vorgestellten Projekten, eine schlechtere Dokumentation und kleinere Internet-Community, jedoch eine umfangreiche Demoprojektesammlung, welche sich als hilfreicher herausstellte.

Nach der Festlegung des Kompressions- und Videoübertragungsverfahrens wurde ein weiteres Testprogramm verfasst. Dies umfasste bereits das Komprimieren der YUV-Bilder sowie das Übertragen und Darstellen durch den VLC Media Player. Dieser Test brachte zum Vorschein, dass die nicht konstante Videobildrate des EVS ein Problem darstellt. So bricht die Videoübertragung ab, sobald die Videobildrate des EVS über eine bestimmte Zeit unter der Streaming-Rate liegt. Damit dieses Problem in der schlussendlichen Software nicht auftritt, beinhaltet das Blockdiagramm in Abbildung 24 einen Videobildraten-Stabilisierung-Block, welcher Videobildraten-Schwankungen ausgleichen soll.

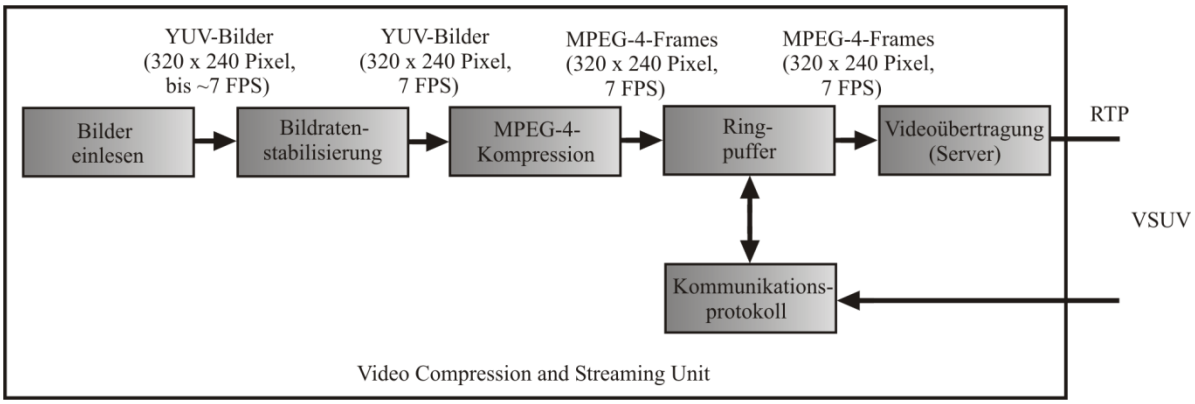


Abbildung 24: Abstraktes Blockdiagramm der VCSU mit MPEG-4-Encoder und YUV-Bilder

5. Software Design

Das Design der Software teilt sich in zwei Teile auf. So sind die VCSU im SENSE-Knoten und die VSUV am PC zu entwerfen. Als Basis für die Entwicklung werden die Ergebnisse aus der Machbarkeitsanalyse im Kapitel 4 herangezogen (Abbildung 22 und Abbildung 24).

5.1 Architektur der Video Streaming Unit Visualization

Der Benutzer des Systems interagiert in erster Instanz mit der VSUV. Die VSUV dient zur Anzeige des Video-Streams und Visualisierung einiger Betriebsparameter der VCSU. Durch die VSUV wird es dem Benutzer ermöglicht, On-Demand- beziehungsweise Live-Streams von der VCSU anzufordern.

Als Zielbetriebssystem der Anwendung wurde ein PC mit Windows festgelegt. Basierend auf dieser Festlegung und der Anforderungen fiel die Entscheidung auf die Programmiersprache Visual C#. Die Gründe für die Wahl dieser Programmiersprache sind, dass C# auf dem .NET-Framework aufbaut und so die unter .NET erstellten Programme unter den Windows-Varianten plattformunabhängig sind [Sch05, S. 3]. Desweitern unterstützt C# ActiveX, was die Einbindung von ActiveX-Steuerkomponenten, wie den VLC Media Player, in die Anwendung möglich macht. Nebenbei kann eine ActiveX-Komponente auch in verschiedenste andere Container eingebettet werden, wie zum Beispiel dem Internet Explorer, Word, Visual Basic usw. [LX08, S. 542]. Die Entwicklung von Programmen mit C# wird durch die kostenlose Entwicklungsumgebung Microsoft Visual Studio 2008 Express Edition vereinfacht. Die Entwicklungsumgebung ist dabei unter [11] beziehbar.

Die Anforderung, dass die Wiedergabe der Video-Streams mit dem VLC Media Player durchgeführt werden soll, vereinfachte die Entwicklung der VSUV deutlich. Dadurch fiel der Umfang der VSUV verglichen mit der VCSU wesentlich geringer aus. Im Folgenden wird das entwickelte Klassendiagramm beschrieben sowie der Zusammenhang zwischen den festgelegten Klassen.

5.1.1 Klassendiagramm

Abbildung 25 zeigt das entwickelte Klassendiagramm für die VSUV. Der Einstiegspunkt der Anwendung erfolgt durch die statische Klasse `Programm`. Die zentrale Klasse der VSUV stellt jedoch `MainForm` dar. Sie wird zu Beginn des Programmes instanziiert und stellt das Grundgerüst für die Visualisierungskomponenten dar. Sämtliche Steuerungskomponenten wie Schaltflächen, Menüs und so weiter werden in der Klasse `MainForm` instanziiert. Mit den Steuerkomponenten ist es dem Be-

nutzer möglich, mit dem System zu interagieren. Die Komponenten werden im Klassendiagramm durch die Klasse Steuerkomponenten beziehungsweise Anzeigekomponenten auf eine abstrakte Weise zusammengefasst.
Die Einbettung des VLC Media Player erfolgt mit der Klasse `AxAXVLC`. Diese Klasse stellt über die ActiveX-Schnittstelle des VLC Media Players Zugriff auf sämtliche Funktionen bereit. Die Klasse präsentiert sich innerhalb der Anwendung als Videoanzeigefenster, jedoch ohne den gewohnten Bedienelemente der VLC Media Player Anwendung. Die Steuerung des Players erfolgt dabei über Member-Funktionen der Klasse `AxAXVLC`.
Das Erstellen von Kommunikations-Nachrichten erfolgt mit den Klassen von Protocol Buffers. Mit den Klassen lässt sich eine Nachrichten erstellen und über die TCP/IP-Verbindung durch die Klasse `TcpClient` versenden.

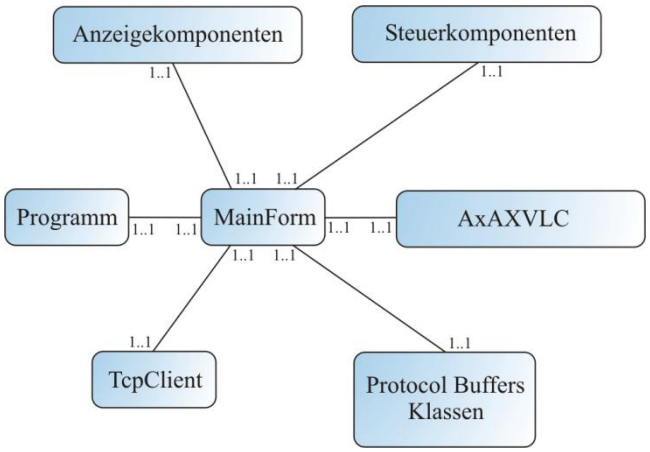


Abbildung 25: Klassendiagramm der VSUV

5.1.2 Kommunikationsablauf

Die vorhandene Kommunikationsinfrastruktur ermöglicht bereits die Kommunikation zwischen den SENSE-Knoten und dem PC. Somit ist nur noch ein Kommunikationsprotokoll zu entwickeln. Um dies zu realisieren, wurde Protocol Buffers (abgekürzt Protobuf) verwendet. Diese Festlegung kam unter anderem daher, dass Protobuf bereits zur Kommunikation zwischen den RDUs verwendet wird.

Protobuf wurde von Google Inc. als Open-Source-Projekt entwickelt, um den Datenaustausch zu vereinfachen [6]. Protobuf serialisiert Daten, welche in einer strukturierten Form vorliegen. Dabei ist Protobuf grundsätzlich mit XML vergleichbar. Jedoch ist Protobuf im Gegensatz zu XML nach Ansicht von Google flexibler, effizienter und besitzt Mechanismen, welche die Serialisierung automatisieren [KS09, S. 192]. Protobuf wird von Google für C++, Java und Phyton unterstützt. Dadurch, dass Google den Quellcode frei zugänglich macht, gibt es eine Reihe von Portierungen auf andere Programmiersprachen, wie zum Protobuf-Net für C# [7].

Das Hauptaugenmerk des Nachrichtenaustausches betrifft das Anfordern von Video-Streams durch die VSUV. Den zeitlichen Ablauf des Nachrichtenaustausches zwischen der VSUV und der VCSU zeigt das Transaktionsdiagramm in Abbildung 26. Dieser Ablauf unterteilt sich in drei Abschnitte

und setzt eine bereits aufgebaute TCP/IP-Verbindung voraus.
Im ersten Schritt schickt die VSUV eine Video-Stream-Anforderungsnachricht an die VCSU. Diese Nachricht enthält neben einem eindeutigen Kommando einen weiteren Parameter, welcher eine Zeit in Sekunden angibt. Dieser Zeitwert wird von der VCSU für die Bestimmung der Leseposition im Ringpuffer verwendet. Die angeforderte Leseposition ergibt sich durch das aktuellste, im Ringpuffer gespeicherte Videobild, abzüglich des Zeitwerts aus der Anforderungs-Nachricht, multipliziert mit der Streaming-Videobildrate. Live- beziehungsweise On-Demand-Video-Streams unterscheiden sich in diesem Fall nur von der Wiedergabeposition der im SENSE-Knoten gespeicherten Videos.
Um sicher zu stellen, dass die gewünschte Anforderung durch die VCSU umsetzbar ist, sendet die VCSU eine Bestätigung an die VSUV zurück. Durch eine positive Bestätigung kann die Video-Stream-Übertragung mit RTP gestartet werden. Dies erfolgt im zweiten Abschnitt. Die RTP-Sitzung dauert dabei solange, bis die VCSU eine Kommandonachricht erhält, welche die Beendigung der Übertragung signalisiert.

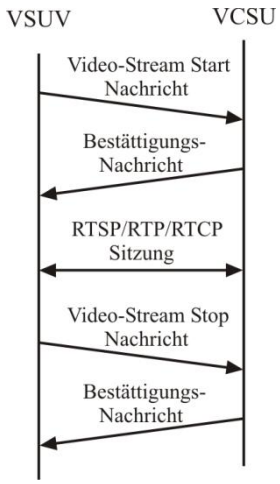


Abbildung 26: Nachrichten-Transaktionsdiagramm für das Anfordern eines Video-Streams

5.1.3 Aktionsabläufe

Die wesentlichen Abläufe der VSUV umfassen die Anforderung von Video-Streams und deren Anzeige, beziehungsweise die Visualisierung von Betriebszuständen der VCSU. Die Kommunikation erfolgt über eine TCP/IP-Verbindung. Dabei bauen die VSUV und VCSU auf betriebssystemspezifische Socket-Funktionen auf. Für die Serialisierung und Deserialisierung von Protobuf-Nachrichten greift die VSUV auf die Protobuf Portierungen Protobuf-Net zurück.

Anforderung von Video-Streams

Innerhalb der VSUV baut die Klasse `TcpClient` eine Verbindung mit dem am SENSE-Knoten ausgeführten VCSU auf. Die VSUV initiiert dazu eine Verbindung als Client, wobei die VCSU als Server eine Verbindung entgegennimmt. Aufbauend auf der Instanz der Klasse `TcpClient` greifen die Protobuf-Net Funktionen auf den Stream der Klasse `TcpClient` zu. Dadurch ist es über die Klasse `MainForm` möglich, Nachrichten an die VCSU zu schicken beziehungsweise zu empfangen. Das Sequenzdiagramm in Abbildung 27 zeigt die Interaktion zwischen dem Benutzer und der

VSUV. Dem Benutzer wird über die einzelnen Steuerkomponenten der Benutzeroberfläche unter anderem die Möglichkeit gegeben, Live-Videobilder und On-Demand-Videos anzufordern.

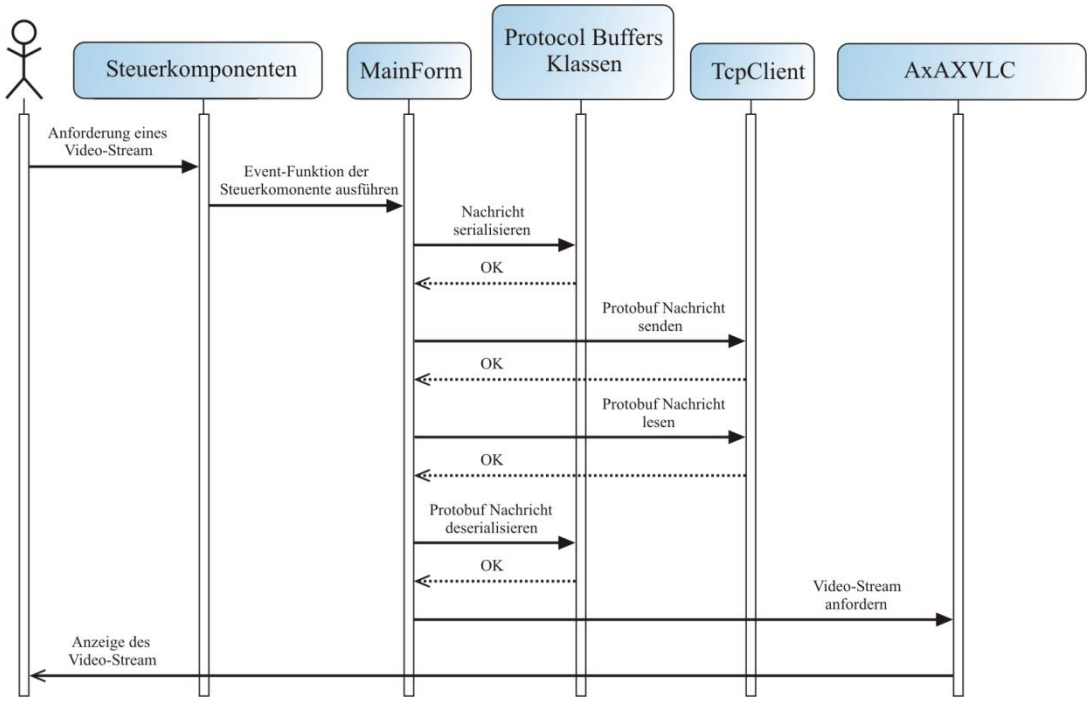


Abbildung 27: Sequenzdiagramm - Video-Stream anfordern

Bei der Betätigung der Schaltfläche, die für das Anfordern des Video-Streams zuständig ist, wird die zur Schaltfläche zugeordnete Event-Funktion in der Klasse `MainForm` aufgerufen. Diese Funktion erstellt in einem ersten Schritt eine Nachricht über die Protobuf-Net Klassen.

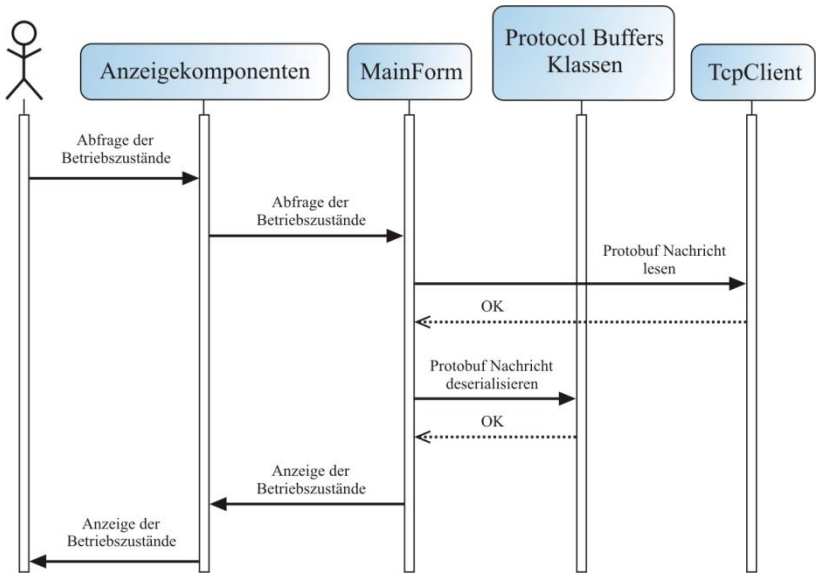


Abbildung 28: Sequenzdiagramm - Visualisierung der Betriebszustände

In Folge wird die Nachricht, welche die Abspielposition des gewünschten Video-Streams enthält, über den Socket an die VCSU geschickt. Die VCSU bestätigt diese Nachricht, wenn die gewünschte Anforderung für die VCSU ausführbar ist. Der Klasse `AxAXVLC` wird in Folge mitgeteilt, dass ein Video-Stream von der VCSU angefordert werden kann. Der in die VSUV eingebunden VLC Media Player beginnt mit der Anforderung des RTP-Streams und dessen Decodierung beziehungsweise Anzeige.

Visualisierung von Betriebszuständen der VCSU

Die zweite Aufgabe der VSUV besteht darin, über den Socket empfange Nachrichten zu verarbeiten. Das Sequenzdiagramm in Abbildung 28 zeigt die Interaktion zwischen den Instanzen der Klassen. Dazu überprüft die Klasse `MainForm` periodisch, ob neue Daten über den Socket empfangen wurden. Stehen dabei Daten zur Verfügung, so werden diese durch eine Funktion der Protobuf-Klassen deserialisiert. Konnte eine Nachricht richtig deserialisiert werden, so werden die gewonnen Daten dem Benutzer über die entsprechenden Anzeigekomponenten angezeigt.

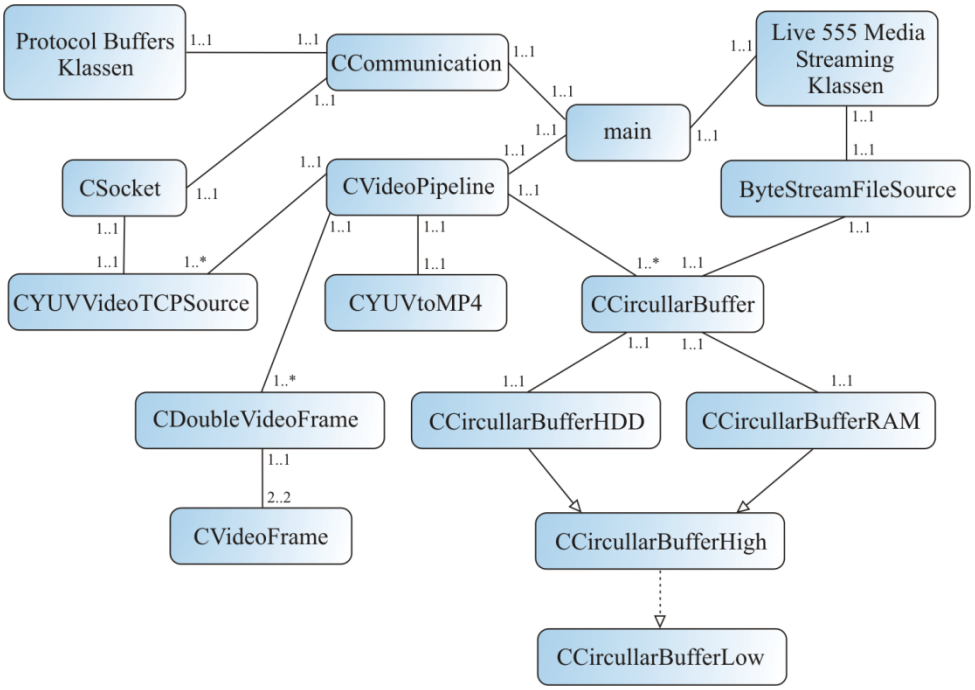


Abbildung 29: Klassendiagramm der VCSU

5.2 Architektur der Video Compression and Streaming Unit

Dadurch, dass Live 555 Streaming Media und Protobuf die Programmiersprache C++ verwenden, wird für die Implementierung der VCSU ebenfalls C++ verwendet. Durch die Verwendung von C++ stehen somit objektorientierte Konzepte zur Verfügung. Dadurch ist man im Stande, zusammengehörigen Funktionalitäten in Klassen zu kapseln und in einem Klassendiagramm darzustellen.

5.2.1 Klassendiagramm

Die Entwicklung des Klassendiagrammes (Abbildung 29) der VCSU basiert dabei auf dem Blockdiagramm aus Abbildung 24. An dieser Stelle wird auf die Notation der Klassennamen hingewiesen. So stellen alle Klassennamen mit dem Anfangsbuchstaben ‚C' (wie class) eine Klasse im objektorientierten Sinne dar.
Die Klasse `main` als Einstiegspunkt der VCSU-Anwendung besitzt unter anderem nicht diesen Buchstaben, da es sich um keine Klasse im klassischen Sinne handelt. Die restlichen Kästchen, welche kein „C" besitzen, sind Klassen von Live 555 Media Streaming und Protobuf, welche diese Notation nicht verwenden.

5.2.2 Grundsätzlicher Verarbeitungsablauf

Durch die Verwendung eines Betriebssystems können Funktionen in Prozessen und Threads grundsätzlich unabhängig voneinander abgearbeitet werden. Diese Gegebenheit erleichtert die Einbindung von Live 555 Media Streaming, erfordert jedoch wiederum Entwicklungsaufwand bei der Prozesssynchronisation. Die hauptsächlich abzuarbeitenden Tasks der Video Streaming Unit werden in Abbildung 30 dargestellt.

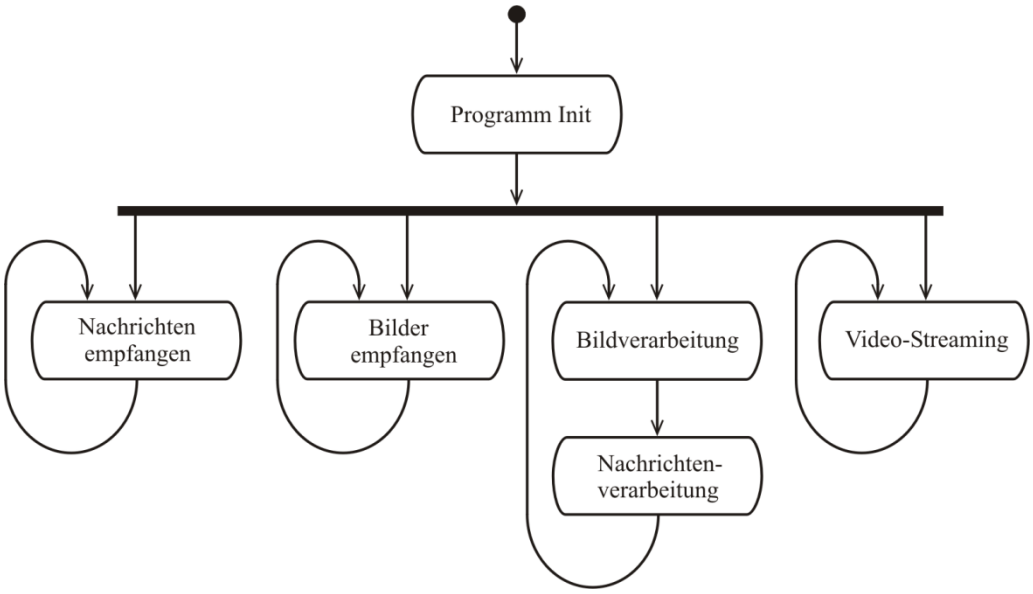


Abbildung 30: Grundsätzlicher Verarbeitungsablauf der VCSU

Nach dem Start der VCSU werden sämtlichen Initialisierungen durchgeführt. Danach teilt sich der Abarbeitungsablauf in mehrere parallel arbeitende Tasks auf. Dabei ist vorgesehen, dass ein Task für das Empfangen der Videobilddaten zuständig ist. Dieser Task liest ständig Daten vom EVS ein und puffert sie in einem Videobildpuffer zwischen. Ein weiterer Task ist für den Empfang von Nachrichten rund um die Kommunikation zwischen VCSU und VSUV zuständig. Der Task liest die Daten für die Nachrichten ein, decodiert sie und schreibt sie in einen Nachrichtenpuffer.
Für die Verarbeitung der Daten, welche durch die beiden ersten Tasks erzeugt werden, ist ein weite-

rer Task zuständig. Dieser ist für die Videobildverarbeitung zuständig, welche Videobilder vom ersten Task übernimmt, diese komprimiert und im Ringpuffer speichert. Gefolgt wird die Videodatenverarbeitung von der Nachrichtenverarbeitung, welche die empfangenen Nachrichten vom zweiten Task verarbeitet und wenn vorgesehen Nachrichten verschickt.
Das Streamen der Videobilder erfolgt über einen weiteren eigenständigen Task. Der Task greift dabei auf den Ringpuffer zu, in dem der dritte Task kontinuierlich schreibt und überträgt die gelesenen Daten bei Bedarf an die VSUV.

5.2.3 Leser-Schreiber-Problem

Die Aufteilung der Abläufe auf verschiedene Tasks erfordert zwischen dem Video-Streaming-Task und dem Videopipeline-Task besondere Aufmerksamkeit. Abbildung 31 zeigt die drei Funktionsblöcke und die jeweilige Zuordnung in die beiden verschiedenen Tasks.

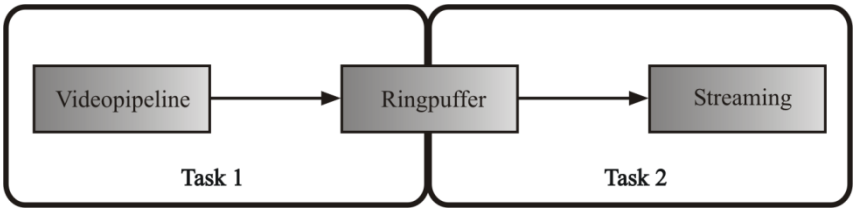


Abbildung 31: Leser-Schreiber-Problem

Mit dieser Struktur liegt ein allgemeines Leser-Schreiber-Problem vor. Bei einem solchen Problem gibt es einen Schreiber, der sowohl Daten in ein Datenfeld schreiben als auch Daten davon lesen darf. Daneben gibt es mehrere Leser, welche aus dem Datenfeld sowohl lesen als auch Daten schreiben dürfen. Das Datenfeld kann eine Datei, ein Speicherblock oder ähnliches sein [Sta04, S. 245]. Die Rolle des Schreibers nimmt in diesem konkreten Fall die Videopipeline in Task 1 ein. Task 2 auf der anderen Seite liest die Daten. Dabei teilen sich beide Tasks den Ringpuffer als gemeinsames Datenfeld. Für den Fall, dass beide Tasks auf den Ringpuffer zugreifen wollen, ist es notwendig diese zu synchronisieren.
Es gibt mehrere Methoden, dieses Problem zu lösen. So gibt es Ansätze, welche den Leser oder den Schreiber bevorzugen. Diese Methoden regeln den Zugriff mit Hilfe von Semaphoren. Jener Task, welcher auf den Ringpuffer zugreifen will, setzt ein Semaphore und schließt damit aus, dass ein anderer Task auf den Ringpuffer zugreifen kann. Sobald dieser den kritischen Programmteil verlassen hat, setzt dieser die Semaphore wieder zurück, was es einem anderen Task erlaubt, den kritischen Programmabschnitt auszuführen. Mit dieser Vorgehensweise kann sichergestellt werden, dass die Daten konsistent bleiben [Sta04, S. 246-249].
Jedoch kann diese Art von Vorgehensweise in dem vorliegenden Fall nicht direkt angewandt werden. Die Gründe dafür liegen bei der Funktionsweise von Live 555 Streaming Media. Um die Prozesssynchronisation für diese vorliegenden Anforderungen zu entwickeln, ist es notwendig, zuerst den Ablauf des Streaming-Tasks zu verdeutlichen.

Leser-Task

Durch die Verwendung von Live 555 Media Streaming wird dessen Arbeitsweise als gegeben angenommen. Eine Abänderung kam durch die Komplexität nicht in Frage. Die Funktionsweise des Streaming-Task wird in Abbildung 32 anhand der Zustandsmaschine gezeigt.
Der Streaming-Server befindet nach dem Start im Ruhezustand, wo dieser auf eingehende RTP-Verbindungen wartet. Kommt es zu einem Verbindungsaufbau, so versucht der Streaming-Server, seinen internen Datenpuffer zu füllen. Dieser Datenpuffer besitzt eine Größe von 100000 Byte. Um diesen Datenpuffer zu füllen, greift dieser wiederrum auf den Ringpuffer zu. Dabei versucht das Puffer-Management des Streaming-Servers Schwankungen auszugleichen.
Das Puffer-Management liest die Daten blockweise über die Klasse `ByteStreamFileSource` ein, wobei die Blockgröße variieren kann. Ist es dem Streaming-Server nicht möglich, weitere Daten in den internen Puffer zu laden und leert sich dieser, hat das zur Folge, dass die RTP-Sitzung abgebaut wird. Dieses Verhalten stellt ein Problem dar, wenn Live-Videobilder übertragen werden sollen. Durch die nicht konstante Videobildrate des EVS kann es dazu kommen, dass dem internen Datenpuffer von Live 555 Streaming Media zu wenig Videodaten zur Verfügung gestellt werden. Um dies zu verhindern, muss dafür gesorgt werden, dass die Schreibrate ähnlich beziehungsweise gleich der Leserrate ist.

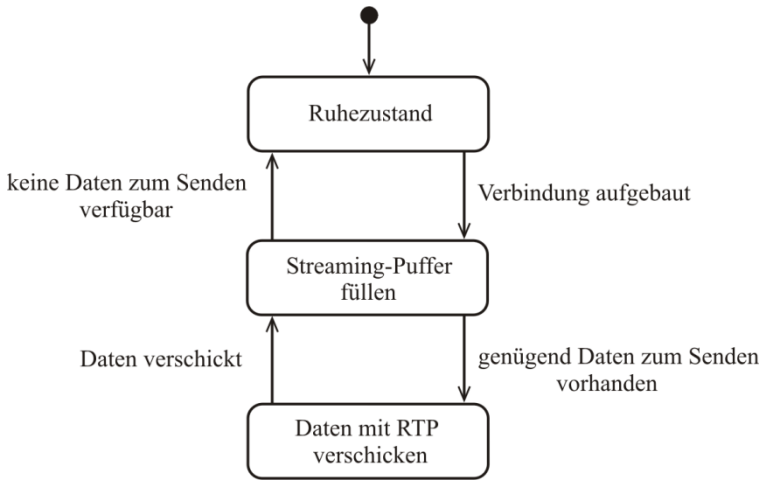


Abbildung 32: Abstrakte Zustandsmaschine des Streaming-Servers

Schreiber-Task

Der Schreiber-Task ist einerseits für die Abarbeitung der Videopipeline sowie für die Nachrichtenverarbeitung zuständig. Des Weiteren muss der Schreiber-Task dafür sorgen, dass die Videopipeline mit der Videobildrate arbeitet, um das im vorherigen Abschnitt erwähnte Problem bei der Live-Videobild Übertragung zu umgehen. Die Vorgehensweise für eine mögliche Realisierung für eine konstante Abarbeitungsrate des Schreiber-Tasks wird im darauf folgenden Unterkapitel Videobildratenstabilisierung näher betrachtet.

Prozess-Synchronisation

Damit der Ringpuffer richtig funktioniert und die Integrität der gespeicherten Daten gewährleistet werden kann, ist der Zugriff auf diesen zu regeln. Wie bereits im letzten Unterkapitel erwähnt, liest der Leser-Task Daten blockweise aus dem Ringpuffer. Somit kann es beim Lesevorgang dazu kommen, dass der Zugriff auf den Ringpuffer für längere Zeit nicht möglich ist. Zusätzlich muss jedoch die Möglichkeit gegeben sein, dass der Schreiber-Task periodisch Daten im Ringpuffer ablegen kann. Dies ist zwingend erforderlich, damit die Videopipeline ihre Videobildrate aufrechterhalten kann. Durch die bereits geringe Videobildrate durch das EVS ist ein Verwerfen von Einzelbildern, bei einem nicht erlaubten Zugriff auf den Ringpuffer nicht sinnvoll, beziehungsweise führt auf Dauer zu den erwähnten Problemen bei der Live-Videobild-Übertragung. Die Anforderung durch den Schreiber- und Leser-Task erfordern es, einen speziellen Ringpuffer zu entwickeln sowie die Prozesssynchronisation auf diese Anforderungen abzustimmen.
Um diese Anforderungen zu erfüllen, setzt sich die Klasse `CCircularBuffer` intern aus zwei Ringpuffern zusammen. Dabei nimmt ein Puffer die Rolle des eigentlichen Ringpuffers ein, der für die Speicherung von mindestens 25 Minuten Video zuständig ist und der andere die Funktion eines Zwischenpuffers. Um den Ablauf besser zu verstehen, werden in Abbildung 33 die wesentlichen Situationen gezeigt, welche beim Zugriff auf den Ringpuffer entstehen.
Die Abbildung stellt den Zugriff auf die einzelnen Instanzen innerhalb der Klasse `CCircular-Buffer` dar, nämlich den Zwischenpuffer als Instanz der Klasse `CCircularBufferRAM` sowie den Ringpuffer als Instanz der Klasse `CCircularBufferHDD`. Die farblich unterschiedlich dargestellten Flächen sollen die unterschiedlichen Tasks hervorheben. Die Implementierung soll folgende Funktionsweise an den Tag legen.

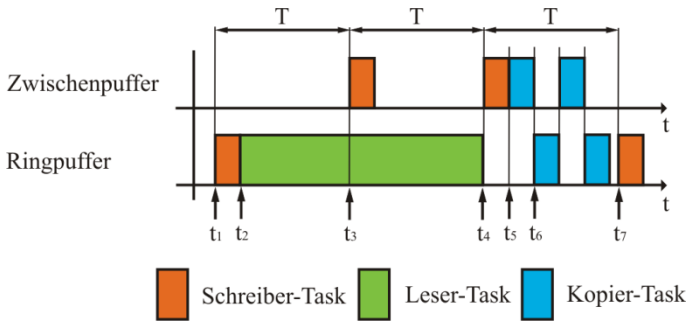


Abbildung 33: Prozesssynchronisation

Dem Schreiber-Task muss es ermöglicht werden, Videobilder periodisch mit dem Intervall T zu speichern. Angenommen, zum Zeitpunkt t_1 will der Schreiber-Task ein Videobild speichern, so wird es diesem auch gewährt, da zu diesem Zeitpunkt kein Task auf den Ringpuffer zugreift. Durch den Schreibzugriff auf den Ringpuffer muss jedoch ausgeschlossen werden, dass ein anderer Tasks auf den Ringpuffer zugreift, um die Konsistenz der Daten im Ringpuffer zu gewährleistet. Dadurch, dass die Videopipeline immer nur einzelne Videobilder in den Puffer schreibt, blockiert der Schreiber-Task den Ringpuffer nur für eine kurze Zeit. Die Zeit, welche für den Zugriff auf den Ringpuffer benötigt wird, ist von der Videobildgröße beziehungsweise der Schreib- und Lese-Rate auf das Speichermedium abhängig. Will nun der Leser-Task zu einem wenig späteren Zeitpunkt t_{1+} ebenfalls auf

den Ringpuffer zugreifen, so wird es diesem verwehrt. Dies ist nicht weiter schlimm, da der Schreibzugriff durch den Schreiber-Task nur kurze Zeit dauert und der Streaming-Server dies mit der internen Pufferung ausgleicht.
Sobald der Schreibzugriff durch den Schreiber-Task abgeschlossen ist, kann der Leser-Task Daten aus dem Ringpuffer lesen. Dies geschieht zum Zeitpunkt t_2. Durch die blockweise Arbeitsweise von Live 555 Streaming Media kann ein Lesezugriff durchaus mehrere Schreibversuche des Schreiber-Tasks blockieren, was von der Videobildauflösung beziehungsweise der Videobildwiedergaberate abhängt. In diesem Beispiel dauert der Lesezugriff von t_2 bis t_4.
Zum Zeitpunkt t_3 liegt nun der Fall vor, bei dem der Leser-Task den Ringpuffer blockiert. Jedoch muss der Schreiber-Task das Videobild speichern, um für die Verarbeitung des nächsten Videobildes bereit zu sein. Dadurch, dass nicht bestimmbar ist, wie lange der Leser-Task noch für das Lesen der Daten braucht, müssen die Daten vom Schreiber-Task im Zwischenpuffer zwischengespeichert werden. Während des Zeitintervalls, in dem ein Lesezugriffes stattfindet, müssen alle Videobilder im Zwischenpuffer abgelegt werden. Damit kann über ein bestimmtes Zeitintervall sichergestellt werden, dass die Videopipeline den Videobilderfluss aufrechterhalten kann. Das Zeitintervall, welcher vom Zwischenpuffer überbrückt werden kann, ist dabei von der Speichergröße des Zwischenpuffers, der Videobildrate und der Videobildgröße abhängig.
Der Zeitpunkt t_4 markiert die Situation, zu der der Lesezugriff durch den Leser-Task abgeschlossen ist. Zum gleichen Zeitpunkt will jedoch auch der Schreiber-Task ein Videobild im Ringpuffer speichern. Prinzipiell wäre der Zugriff auf den Ringpuffer möglich, jedoch muss an dieser Stelle auf die Speicherreihenfolge geachtet werden. Da sich ein Videobild im Zwischenpuffer befindet und das Videobild des Schreiber-Task so schnell wie möglich gespeichert werden muss, wird das Videobild vom Schreiber-Task wieder im Zwischenspeicher abgelegt.
Zum Zeitpunkt t_5 hat der Schreiber-Task den Schreibzugriff abgeschlossen und ein weitere Task kommt ins Spiel. Dieser sogenannte Kopier-Task ist dafür zuständig, die Daten vom Zwischenpuffer in den eigentlichen Ringpuffer zu übertragen. Dazu liest dieser die Videobilder einzeln aus dem Zwischenpuffer aus (t_5) und schreibt sie in Folge einzeln in den Ringpuffer (t_6). Dies passiert solange, bis alle Videobilder aus dem Zwischenpuffer in den Ringpuffer übertragen wurden.
Der Zeitpunkt t_7 markiert im Beispiel den Zustand, wo der Kopier-Task sämtliche Videobilder vom Zwischenpuffer in den Ringpuffer kopiert hat und der Schreiber-Task wieder direkt auf den Ringpuffer zugreifen kann.

5.2.4 Videopipeline

Die Videopipeline kapselt sämtliche Verarbeitungsschritte für die Videobildverarbeitung. Ein besonderes Augenmerkt wird auf die Einfachheit, Übersichtlichkeit, Flexibilität, Wiederverwendbarkeit sowie Performance gelegt. Der Aufbau der Videopipeline wurde so designt, dass bestimmte Verarbeitungsschritte zu einer Klasse zusammengefasst werden.
Die durch die Videopipeline benützten Klassen können in drei Gruppen eingeteilt werden. Diese sind die Gruppe der Videobildquellen, Videobildsenken sowie videobildverarbeitenden Klassen. Zu der Gruppe der Videobildquellen zählen die Klassen `CYUVVideoTCPSource` sowie `CJPEGVideoTCPSource`. Diese Klassen kapseln im speziellen die Funktionen ein, um auf das YUV- be-

ziehungsweise JPEG-Video-Service des EVS zuzugreifen. Zwar findet man die Klasse `CJPEGVideoTCPSource` im schlussendlichen Klassendiagramm nicht, wurde jedoch im Rahmen der Machbarkeitsanalyse implementiert und hier vollständigkeitshalber erwähnt. Als Gegenstück zu diesen Videobildquellen stehen Videobildsenken zur Verfügung. Die Klasse `CCircularBuffer` ist der einzige Vertreter dieser Gruppe. Grundsätzlich würde auch das Streaming-Service zu dieser Gruppe gehören, jedoch kann das Streaming-Service nur zusammen mit dem Ringpuffer eingebunden werden. Zur dritten Gruppe gehören die Klasse `CYUVtoMP4` und `CJPEGtoYUV`, wobei letztere im schlussendlichen Klassendiagramm ebenfalls keine Verwendung mehr fand.
Durch die Kapselung dieser Funktionen steht ein flexibles Gerüst für die Videobildverarbeitung zur Verfügung. So kann durch die Integration von weiterer Klassen die Funktion der Videopipeline einfach erweitern werden.

Datenaustausch

Der Datenaustausch zwischen den Videobildquellen, -senken und videobildverarbeiteten Klassen erfolgt über einen speziellen Videobildpuffer. Im Klassendiagramm wird dieser Puffer in den Klassen `CVideoBuffer` und `CDoubleVideoBuffer` realisiert. Die Klasse `CDoubleVideoBuffer` setzt sich aus zwei Instanzen der Klasse `CVideoBuffer` zusammen. Dabei agiert eine dieser Instanzen als Eingabe-Puffer und die zweite als Ausgabe-Puffer. Das Beispiel in Abbildung 31 zeigt den Datenaustausch zwischen der Klasse `CYUVVideoTCPSource` und `CYUVtoMP4`. Dafür sind folgende Schritte notwendig (siehe Abbildung 34).

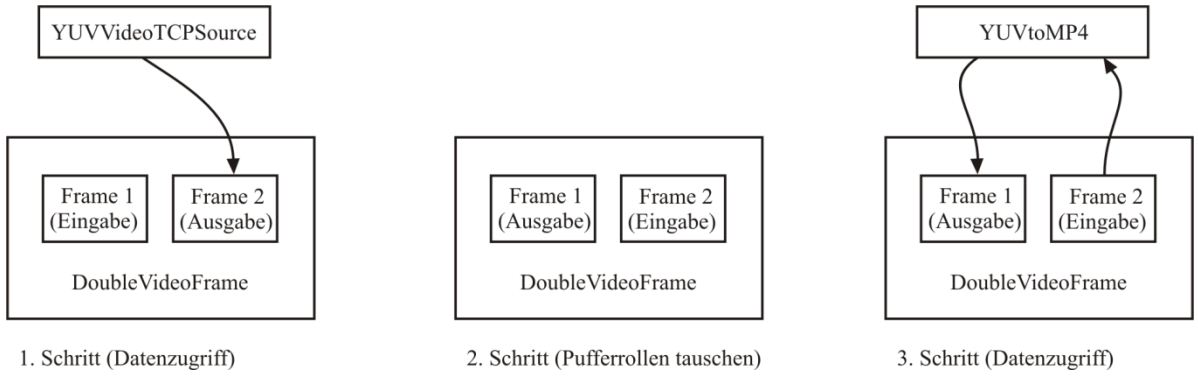


Abbildung 34: Ablauf Datenaustausch

Im ersten Schritt schreibt die Klasse `CYUVVideoTCPSource` ein YUV-Bild in den Puffer. Dazu wird die Regel festgelegt, dass jede Klasse, welche einer anderen Klasse Daten übergeben will, diese in den Puffer schreiben muss, dem zu diesem Zeitpunkt die Rolle des Ausgabe-Puffers zukommt.
Nachdem der Schreibvorgang abgeschlossen ist, tauschen die Puffer im zweiten Schritt ihre Rollen. Der Eingangs-Puffer nimmt dazu die Rolle des Ausgangs-Puffers ein. Gleiches gilt im umgekehrten Sinne auch für den Ausgangs-Puffer. Mit diesem Tausch besitzen die Puffer ihre logisch richtige Rolle für die darauf folgende Weiterverarbeitung. So stehen die im anfänglich im ersten Schritt geschrieben Daten im dritten Schritt im Eingangs-Puffer. Durch den Aufbau der Klasse `CDoubleVideoFrame` kann die Klasse `CYUVtoMP4` durch den Eingangs-Puffer sowohl auf das YUV-Bild zugreifen und besitzt mit dem Ausgangs-Puffer zugleich einen Puffer, in der das Ergebnis der Ope-

ration gespeichert werden kann. Nachdem Schritt 3 abgeschlossen ist, kann der DoubleVideoBuffer die Daten an eine weitere Klasse übergeben.

Videobildratenstabilisierung

Auf das Verhalten der EVS, welche eine nicht konstante Videobildrate aufweist, wurde bereits aufmerksam gemacht. Damit es bei Live-Video Übertragungen zu keinen Abbrüchen kommt, muss die Videopipeline mit der gleichen Videobildrate Daten in den Ringpuffer schreiben, mit der auch der Streaming-Task Daten aus dem Ringpuffer liest.
Die nicht konstante Videobildrate hat dabei einen weiteren Nebeneffekt, der sich bei dem Video-Wiedergabeverhalten zeigen kann. Angenommen, eine Person geht durch den Bildbereich der Kamera. Durch dieses Ereignis kann es dazu kommen, dass die Videobildanalyse im EVS mehr Rechenzeit in Anspruch nimmt und somit die Videobildrate abnimmt. Angenommen, die Videobilder werden mit einer Rate von 10 FPS mit RTP übertragen und das EVS bricht auf eine Videobildrate von 1 FPS ein, hat die verringerte Videobildrate des EVS jedoch in Folge keine Auswirkung auf die Videobildübertragungsrate des Streaming-Servers. So macht es dadurch den Anschein, dass die Abläufe bei der Wiedergabe in der VSUV schneller ablaufen als diese in der Realität passieren. In dieser Annahme um den Faktor 10. Dem Betrachter des Videos würde vorkommen, als würde diese Person, die eigentlich durch den Bildbereich geht, diesen durchlaufen.
Damit stellt sich nur noch die Frage, wie der Videopipeline Videobilder zur Verfügung gestellt werden soll, wenn das EVS keine liefert? Für dieses Problem gibt es zwei Lösungswege. Einerseits wäre es möglich, die niedrigste Videobildrate des EVS als Wiedergaberate der VSUV nehmen. Dabei würden jedoch sämtliche Videobilder, die über dieser Videobildrate liegen, verworfen. Da jedoch die Videobildrate bereits einen sehr niedrigen Wert aufweist, wurde auf diese Vorgehensweise verzichtet.
Eine weitere Möglichkeit besteht darin, die Videoübertragung mit der Videobildrate zu übertragen, welche das EVS maximal liefert. Kann das EVS die Videobilder nicht mit einer ausreichenden Videobildrate liefern, so kann das letzte empfangene Videobild verwendet werden. Der Vorteil dieser Methode liegt darin, dass eine flüssigere Videowiedergabe ermöglicht wird. Die Funktionalität, dass immer Videobilder verfügbar sind, wird in der Klasse `CYUVVideoTCPSource` implementiert.

Ablauf der Videobildverarbeitung

Das Sequenzdiagramm in Abbildung 35 zeigt einerseits die Klassen, welche in den Videobildverarbeitungsprozess eingebunden sind und andererseits die Interaktion zwischen diesen Klassen. Die Abarbeitung der einzelnen Verarbeitungsschritte werden in der Klasse `CVideoPipeline` festgelegt und erfolgt Videobild für Videobild. Für die Anforderungen an diese Software sind folgende Verarbeitungsschritte notwendig.

1. Schritt: Ein Task in der Klasse `CYUVVideoTCPSource` versucht ständig, das aktuellste Videobild vom EVS einzulesen.
2. Eine Funktion in der Klasse `CVideoPipeline` wird aufgerufen, welche die einzelnen Verarbeitungsschritte der Videopipeline beinhaltet.

3. Schritt: Die Videopipeline liest das aktuellste Videobild durch eine Funktion der Klasse `CYUVVideoTCPSource` ein.
4. Schritt: Die Pufferrollen in der Klasse `CDoubleVideoBuffer` werden getauscht.
5. Schritt: Das eingelesene Videobild wird durch den MPEG-4/H.263-Encoder komprimiert.
6. Schritt: Die Pufferrollen in der Klasse `CDoubleVideoBuffer` werden ein weiteres Mal getauscht.
7. Schritt: Das komprimierte Videobild wird in den Ringpuffer geschrieben. Damit ist ein Verarbeitungszyklus beendet.

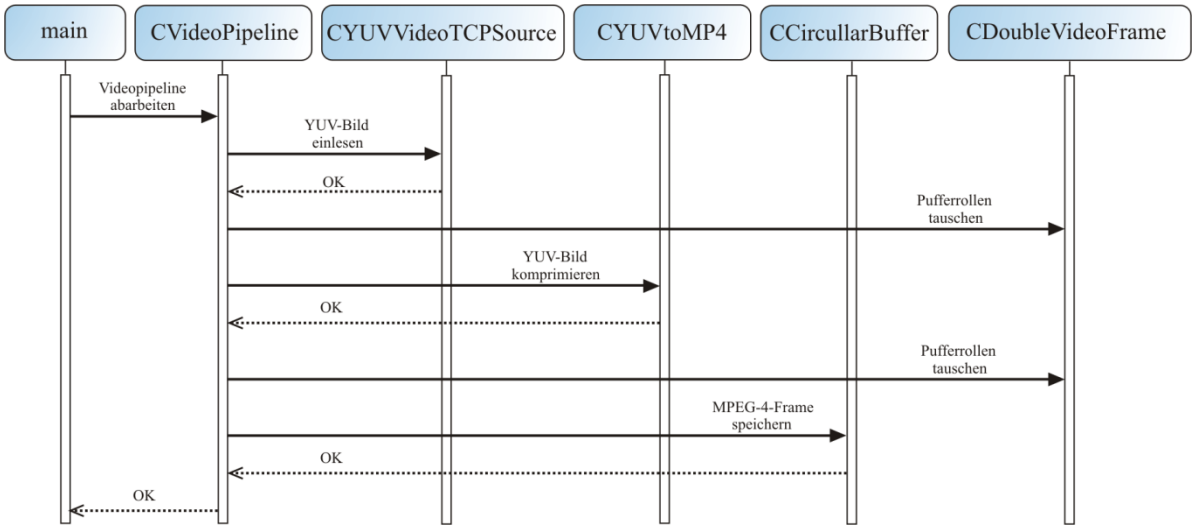


Abbildung 35: Sequenzdiagramm Videopipeline

5.2.5 Ringpuffer

Zirkularpuffer, beziehungsweise Ringpuffer, finden beim asynchronen Datenaustausch zwischen verschiedenen Prozessen Verwendung. Dabei schreibt ein Prozess Daten in den Puffer und ein andere liest sie aus. Im Speziellen werden Ringpuffer dann verwendet, wenn es zwischen den Prozessen Geschwindigkeitsunterschiede bei der Lese- beziehungsweise Schreibrate gibt [KS03, S. 271].
Die Realisierung des Ringpuffers wäre grundsätzlich, aufbauend auf den Standard Template Library (STL), möglich. Dazu würde sich eine Doppelschlange am besten eignen, da das Einfügen sowie Entfernen von Elementen am Beginn sowie am Ende effizient möglich ist [MDS96, S. 129, 148]. Jedoch hat die Implementierung des Ringpuffers mit STL den Nachteil, dass die Daten im Arbeitsspeicher abgelegt werden. Die Anforderungen an die VCSU sind jedoch mindestens 25 Minuten an Videodaten zu speichern. Jedoch ist es ungewiss, ob die Videodaten ausreichend komprimiert werden und damit genügend Arbeitsspeicher zur Verfügung steht. Durch die Speicherung in einer Datei ist es bei Bedarf möglich, die Datei auf einen Massenspeicher auszulagern. Um die Möglichkeit zu haben, die Ringpufferdaten in einer Datei zu speichern, wurde auf die Implementierung mittels STL verzichtet und ein eigener Ringpuffer entwickelt.

Für die Realisierung des Ringpuffers sind fünf Klassen vorgesehen. Dabei ist es notwendig, dass die Klasse `CCircularBuffer` intern aus zwei getrennten und unterschiedlich arbeitenden Ringpuf-

fern besteht. Bei diesen beiden Puffern handelt es sich um eine Instanz der Klasse `CCircular-BufferHDD` sowie `CCircularBufferRAM`. Die Klasse `CCircularBufferHDD` ist für die Speicherung der Videodaten im engeren Sinne zuständig. Die letzten drei Buchstaben dieser Klasse stehen bei dieser Klasse für "Hard Disk Drive" und soll andeuten, dass dieser Ringpuffer auf den Dateisystemfunktionen von Linux aufbaut.
Der zweite Puffer wird in der Klasse `CCircularBufferRAM` implementiert. Diesem Puffer kommt dabei die in vorherigem Unterkapitel beschriebene Funktion des Zwischenpuffers zu. Die letzten drei Buchstaben stehen für „Random Access Memory“ und sollen in diesem Fall andeuten, dass dieser Puffer im Arbeitsspeicher abgelegt wird.
Die Klasse `CCircularBufferHDD` sowie `CCircularBufferRAM` besitzen die grundsätzliche Struktur eines Ringpuffers. Der Unterschied macht sich beim Lesen von Daten aus dem Puffer ersichtlich. So ist das Auslesen eines Elementes aus der Klasse `CCircularBufferRAM` nur ein einziges Mal möglich, da im Anschluss der Speicherplatz freigeben wird. Im Gegensatz dazu besitzt die Klasse `CCircularBufferHDD` beim Lesen von Daten nicht diese Eigenschaft wie die Klasse `CCircularBufferRAM`. Hier ist es grundsätzlich möglich, beliebig oft Datenelemente aus dem Puffer zu lesen. Einzelne Videobilder werden hier erst dann frei gegeben, wenn der Speicherplatz für das Speichern eines neuen Videobildes benötigt wird.

Durch die Ähnlichkeiten der Funktionsweise dieser beiden Ringpuffer wurden diese als Spezialisierungen der Klasse `CCircularBufferHigh` modelliert. Die Klasse `CCircularBufferHigh` beinhaltet sämtliche höhere Ringpuffer-Funktion, auf die Klassen `CCircularBufferHDD` sowie `CCircularBufferRAM` aufbauen. Sie besitzt sozusagen die Grundmenge an allgemeinen höheren Ringpuffer-Funktionen. Die Klasse `CCircularBufferHigh` baut dabei wiederrum auf den Funktionen der Klasse `CCircularBufferLow` auf. Mit der Klasse `CCircularBufferLow` werden sämtliche abgeleitete Klassen gezwungen, bestimmte Ringpuffer-Grundfunktionen zu implementieren.

5.2.6 Nachrichtenverarbeitung

Für die Kommunikation zwischen VSUV und VCSU sind in der VCSU drei Klassen eingebunden. Dies sind die Klassen `CCommunication`, `CSocket` und die Protobuf Klassen (siehe Abbildung 29). Um Nachrichten zu empfangen müssen die Daten für eine Nachricht eingelesen werden. Für das Einlesen der Daten wurde am Anfang dieses Kapitel ein eigener Task festgelegt. Die Interaktion zwischen den Klassen beim Empfangen von Nachrichten wird in Abbildung 36 gezeigt.
Der Task in der Klasse `CCommunication` liest über die Instanz der Klasse `CSocket` Daten über die TCP/IP-Verbindung ein. Wurden eine bestimmte Menge von Bytes eingelesen, so wird versucht, mithilfe der Deserialisierungs-Funktion der Protobuf Klassen aus der Byte-Folge eine Nachricht zu decodieren. Konnte die Nachricht richtig decodiert werden, so wird diese für die weiter Verarbeitung in einen Nachrichtenpuffer zwischen gespeichert. Die Zwischenpufferung erlaubt es, dass ein anderer Task die Nachrichten zu einem späteren Zeitpunkt ausliest und diese weiter verarbeitet.
Neben dem Empfangen stellt die Klasse `CCommunication` auch Funktionen für das Senden von Nachrichten zu Verfügung.

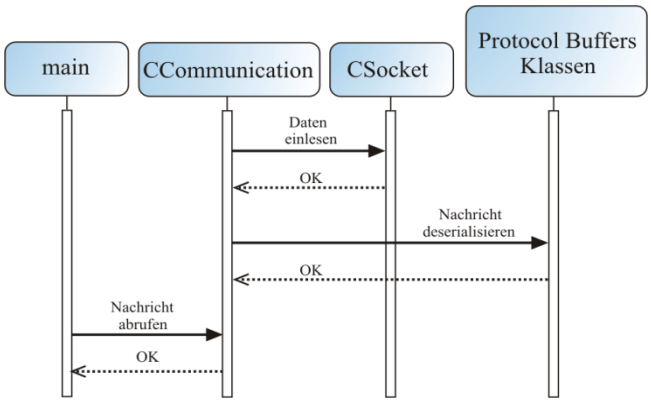


Abbildung 36: Sequenzdiagramm Nachrichtenverarbeitung

6. Implementierung der Video Streaming Unit Visualization

Ein Ziel bei der Realisierung der VSUV ist es, eine Referenzimplementierung für eine spätere umfangreichere Visualisierungssoftware bereitzulegen. Deswegen wird in den folgenden Unterkapiteln nur auf die wesentlichen Teile näher eingegangen. Diese sind vor allem die Einbindung des VLC Media Players und die Funktionsweise des Kommunikationsprotokolls.

6.1 Einbindung des VLC Media Players

Die Wiedergabe des Video-Streams erfolgt durch den VLC Media Player, welcher unter [10] kostenlos beziehbar ist. Um die ActiveX-Schnittstelle des VLC Media Players zu nutzen, muss bei der Installation darauf geachtet werden, dass diese mitinstalliert wird. Daraufhin muss unter Visual Studio unter dem Menüpunkt Extras → Toolboxelemente → Reiter „COM-Steuerelemente“ → VideoLAN ActiveX Plugin v1 ausgewählt werden. Dadurch kann der VLC Media Player über die Toolbox per Drag & Drop in die Anwendung übernommen werden.
Nach der Einbindung in die VSUV ist der Player über die Instanz axVLCPlugin1 der Klasse AxAXVLC ansprechbar. Durch den Aufruf der Funktionen aus dem Programmauszug 1 wird über RTSP eine RTP-Sitzung zwischen VSUV und VCSU aufgebaut.
Durch die Verwendung von RTSP wird es möglich, dass bei der On-Demand-Anforderung des RTP-Streams der VLC Media Player von Beginn der Sitzung an jedes übertragende RTP-Paket zugestellt bekommt und somit die Videosequenz von Anbeginn anzeigt wird. Hingegen würde ohne RTSP der Streaming-Server zum Streamen beginnen und der VLC Media Player mit einer Verzögerung der RTP-Sitzung teilnehmen. Das führt dazu, dass die gewünschte Videosequenz nicht von Beginn an angezeigt wird.

```
1 AxVLCPlugin1.addTarget(rtsp://<IP-Adresse des SENSE-Knoten>:8555/video, null,
  AXVLC.VLCPlaylistMode.VLCPlayListAppendAndGo, -666);
2 AxVLCPlugin1.play();
```

Programmauszug 1

An dieser Stelle wird auf zwei Probleme rund um das Thema Video-Streaming hingewiesen. So wurde bei Tests festgestellt, dass, wenn der Video-Stream über das Internet abgerufen wird, die Videowiedergabe nach circa 40 Sekunden einfriert. Durch Versuche mit unterschiedlichen VLC Media Player Versionen konnte jedoch das Wiedergabeproblem gelöst werden. Dabei konnte festgestellt werden, dass die Version des VLC Media Players Einfluss auf die Streaming-Funktion hat. Tabelle 5 zeigt die Resultate zum Test. Daraus ist erkennbar, dass nur die neueren Versionen des Players Prob-

leme machen. Durch die Verwendung einer älteren Version konnte erreicht werden, dass die Übertragung des Streams über das Internet ohne Abbrüche funktionierte.
Ähnlich wie beim Streamen über das Internet, kam es beim Streamen im lokalen Netzwerk zu Abbrüchen der Sitzung. Dieses zweite Problem konnte jedoch nicht durch den Wechsel auf eine ältere VLC Media Player Version beseitigt werden. Erst durch das Ersetzen des verwendeten Ethernet-Hubs durch einen Ethernet-Switch war es möglich Video-Streams im lokalen Netzwerk einwandfrei zu übertragen.

Tabelle 5: VLC Media Player Versionen

VLC Media Player Version	Resultat
1.0.1	Wiedergabe eingefroren
1.0.0	Wiedergabe eingefroren
0.9.9	Wiedergabe eingefroren
0.9.2	Wiedergabe eingefroren
0.8.6i	funktioniert
0.8.6	funktioniert

6.2 Kommunikationsprotokoll

Die erste Aufgabe bei der Erstellung eines Kommunikationsprotokolls mit Protobuf ist es, den Aufbau der Nachrichten festzulegen. Basierend auf dieser Festlegung werden die Protobuf-Klassen generiert. Der zweite Schritt umfasst die Einbindung der Protobuf-Klassen in die Anwendung. Im ursprünglichen Design der VSUV war vorgesehen, dass Protobuf -Net verwendet wird. Diese lag zum Zeitpunkt der Implementierung in der Version r282 vor und ist unter [7] beziehbar. Jedoch stellte sich bei den Softwaretests heraus, dass damit nur eine eingeschränkte Kommunikation möglich ist. So konnten ohne Probleme Protobuf Nachrichten an die VCSU geschickt und von der VCSU deserialisiert werden. Jedoch machte der Empfang von Protobuf-Nachrichten von der VCSU kommend Probleme. Dies äußerte sich dadurch, dass ein Großteil der empfangenen Nachrichten nicht deserialisiert werden konnte. Um dieser Problem zu umgehen, wurde eine andere Herangehensweise gewählt. Dazu wurde auf dieselbe Protobuf Version zurückgegriffen wie sie auch bei der VCSU verwendet wird. Dabei musste jedoch ein Weg gefunden werden, C++ Funktionen in das C# Programm zu integrieren.
Dies wurde gelöst, indem bestimmte Funktionen in eine Dynamic Link Library (DLL) ausgelagert wurden. Durch die DLL ist es möglich, die Funktionen in beliebigen Programmen zu verwenden [Sch00, S.689].

6.2.1 Strukturierung der Nachrichten

Um eine Kommunikation zwischen der VSUV und VCSU zu ermöglichen, muss eine Nachrichtenstruktur festgelegt werden, damit beide Teilnehmer die übertragenden Daten richtig interpretieren. Dafür muss eine .proto-Datei erstellen werden, welche diese Strukturbeschreibung der Nachrichten

enthält.
Eine Protobuf-Nachricht wird mit `message` und einem Nachrichten-Namen definiert. Jede dieser Nachrichten besteht aus einem oder mehreren Feldern, welche durchnummeriert werden. Jedes Feld besteht wiederum aus einem Namen-Datentyp-Paar. Protobuf unterstützt eine Reihe von Datentypen, wie zum Beispiel Kommazahlen, Zeichenketten aber auch andere Nachrichten. Jede dieser Datenfelder kann dabei als erforderlich oder optional spezifiziert werden.
Im Anhang wird die entwickelte uicom.proto Datei gezeigt. Diese enthält die Beschreibung der Nachrichten für die Kommunikation zwischen VSUV und VCSU, beziehungsweise zwischen den RDUs. Ausgangspunkt jeder Nachricht an einen Teilnehmer ist `mMessage`. Innerhalb von `mMessage` ist es zwingend erforderlich, dass der Nachrichtentyp (`mM_Type`) sowie SENSE-Knoten-ID (`mM_NodeID`) angegeben wird. Optional kann die Nachricht `mMessage` weitere Nachrichten enthalten. Wichtige optionale Nachrichten für die Kommunikation zwischen VSUV und VCSU sind die Nachrichten `mM_Video` und `mVSU_Debug`. Die Nachricht `mM_Video` enthält dabei Felder, welche für die Anforderung von Videos erforderlich sind. Hingegen wurde die Nachricht `mVSU_Debug` für den Austausch von Debug-Informationen während der Entwicklung verwendet. Schlussendlich wird `mVSU_Debug` dazu verwendet, um alle wichtigen Zustände der Ringpuffer, wie zum Beispiel die gespeicherte Anzahl von Videobildern, zu übertragen. Damit kann sich der Benutzer einen schnellen Überblick über sämtliche Zustände der Ringpuffer verschaffen.
Nachdem die Struktur durch die Datei uicom.proto definiert ist, werden durch den Protobuf-Compiler die Klassen für den Nachrichtenzugriff generieren. Dazu muss der plattformspezifische Protobuf-Compiler verwendet werden. Mit dem Aufruf des Befehles protoc --cpp_out=<Ausgabe Verzeichnis> <.proto Datei> werden aus der uicom.proto-Datei sämtliche Klassen für das Erstellen einer Protobuf-Nachricht generiert.

6.2.2 Protobuf_uicom Bibliothek

Als Entwicklungsumgebung für die Erstellung der DLL wurde Visual C++ Express Edition 2008 verwendet, welche unter [11] kostenlos verfügbar ist. Die Verwendung von C++ ist deswegen erforderlich, da Google Protobuf nicht in C# entwickelt wurde, sondern unter anderem in C++.
Der erste Schritt für die Implementierung der DLL ist die Portierung auf die vorliegende Windows Plattform. Zum Zeitpunkt der Implementierung wurde die Protobuf Version 2.3.0 verwendet, welche unter [6] beziehbar ist. Für die Portierung sind folgende Schritte notwendig:

1. Herunterladen von protobuf-2.3.0.tar.gz und in das <BASISVERZEICHNIS> entpacken.
2. Die Projektdatei <BASISVERZEICHNIS>\protobuf-2.3.0\vsprojects\protobuf.vcproj mit Visual C++ Express Edition 2008 öffnen.
3. Die Projektmappe neu erstellen.
4. Die Portierung ist abgeschlossen, sobald die Kompilierung erfolgreich beendet wurde.
5. Erstellen der Protobuf-Klassen (uicom.pb.cc und uicom.pb.h) mit protoc.exe --cpp_out= <Pfad> uicom.proto

Der zweite Schritt umfasst die Implementierung der DLL. Dazu ist erforderlich, ein neues Projekt unter Visual C++ anzulegen. Dabei ist es bei der Konfiguration des Projektes zu achten, dass eine

Klassenbibliothek erzeugt wird.
Aufbauend auf WinSocket wird durch die Protobuf-Funktionen die Serialisierung beziehungsweise Deserialisierung durchgeführt. Dazu wird das blockierende Verhalten des Socket verwendet. Ein Aufruf einer Socket-Funktion kehrt vom Funktionsaufruf erst dann zurück, wenn die Funktion vollständig abgearbeitet werden konnte. Das blockierende Verhalten zeichnet sich jedoch dadurch aus, dass beim Empfangen von Daten nicht ständig zyklisch abgefragt werden muss, ob Daten verfügbar sind. Damit die ganze Anwendung durch dieses blockierende Verhalten der Socket-Funktionen nicht blockiert, wird die Empfangsroutine in einen eigenen Thread ausgeführt [Sch00, S907].

Über verschiedene Schnittstellen der DLL kann das C#-Programm auf die DLL zugreifen. Diese werden in Folge näher beschrieben.

DLL int open_connection (char* ip_address, UINT16 port)

Durch diese Funktion wird innerhalb der DLL durch einen Socket eine Verbindung zur VCSU aufgebaut. Mit der Funktion `socket()` wird ein Socket erstellt, welcher einen Socket-Handle zurück gibt. Die Parameter `ip_address` und `port` beschreiben dabei den gewünschten Verbindungspartner, welche für den Verbindungsaufbau mit der Funktion `connect()` benötigt wird. Ein Rückgabewert von 0 gibt an, dass eine Verbindung erfolgreich aufgebaut werden konnte.

DLL void close_connection (void)

Durch den Aufruf dieser Funktion wird eine bestehende TCP/IP-Verbindung abgebaut. Dazu wird innerhalb dieser Funktion die Funktion `closesocket()` ausgeführt.

DLL void start_receiving (void)

Durch den Aufruf dieser Funktion wird mit Hilfe der Funktion `_beginthread()` der Thread `recv_thread` innerhalb der DLL gestartet. Dieser Thread ist für das Einlesen der Daten vom Socket sowie für die Verarbeitung der empfangen Daten zuständig. Die empfangenden Daten werden in Folge im Nachrichtenpuffer gespeichert, welche durch die folgende Funktion ausgelesen werden kann.

DLL int read_message (uicom_mVSU_Debug* message)

Durch diese Funktion kann eine Protobuf-Nachricht ausgelesen werden. Der Rückgabewert gibt Auskunft, ob eine Nachricht gelesen werden konnte oder nicht.

DLL int streaming_request_duration (UINT32 duration)

Durch diese Funktion wird in der DLL eine Videoübertragungsanforderung erstellt und an die VCSU gesendet. Der Parameter `duration` gibt abhängig von der Streaming-Videobildrate die Startposition an, wenn eine RTP-Sitzung aufgebaut wird.

DLL int start_pipeline (void) und DLL int stop_pipeline (void)

Durch diese beiden Funktionen werden jeweils unterschiedliche Protobuf-Nachrichten an die VCSU gesendet, welche bewirken, dass entweder die Verarbeitung von Videodaten in Videopipeline pausiert oder fortgesetzt wird.

DLL int quit_VCSU (void)

Diese Funktion erstellt und versendet eine Protobuf-Nachricht, welche der VCSU signalisiert, dass die VCSU beendet werden soll.

6.2.3 Ablauf der Kommunikation

Durch die Auslagerung der Socket- und Protobuf-Funktionen in die protobuf_uicom.dll kommt es zu einer Verschiebung bestimmter Abläufe. So wird nun der Verbindungsaufbau mit einem Socket sowie das Serialisieren und Deserialisieren der Nachrichten in der DLL implementiert.
Das Senden von Nachrichten erfolgt über Funktionen aus der DLL, welche vom C#-Programm aus aufgerufen werden. Durch den Aufruf dieser DLL-Funktion wird eine Protobuf-Nachricht erstellt, die Nachricht serialisiert und über den Socket verschickt.

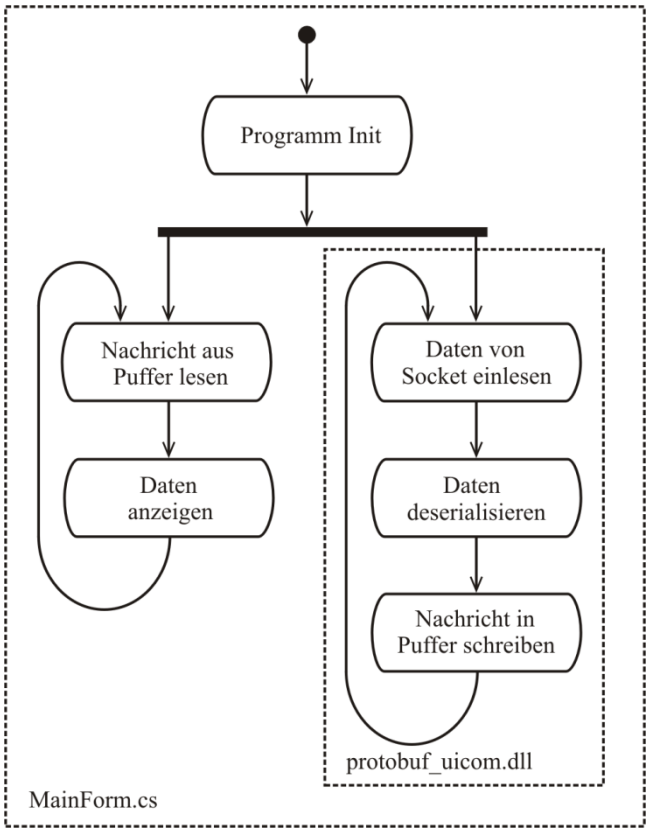


Abbildung 37: Protobuf-Nachrichten-Verarbeitung innerhalb der VSUV

Für das Empfangen von Protobuf-Nachrichten liest ein Thread innerhalb der DLL die empfangenen Daten über den Socket ein (siehe Abbildung 37). Konnten Daten eingelesen werden, so wird ver-

sucht, eine Protobuf-Nachricht richtig zu deserialisieren. Konnte eine Protobuf-Nachricht erfolgreich deserialisiert werden, wird diese im Nachrichtenpuffer gespeichert. Über eine Lese-Funktion kann das C#-Programm zu einem beliebigen Zeitpunkt auf die Nachricht zugreifen.

6.3 Programmoberfläche

Die Bedienung der VSUV ist auf drei Reiter aufgeteilt. Dies sind die Reiter Connection, Video Streaming und Circularbuffer Visual.

6.3.1 Reiter "Connection"

Der erste Reiter ermöglicht es die Verbindung zur VCSU zu konfigurieren. Dazu wird die IP-Adresse des SENSE-Knoten sowie die Port Nummer der VCSU benötigt. Durch die Schaltflächen „connect" und „disconnect" wird eine TCP/IP-Verbindung auf- beziehungsweise abgebaut. Der Verbindungsstatus wird in Folge über ein Statusfeld angezeigt.

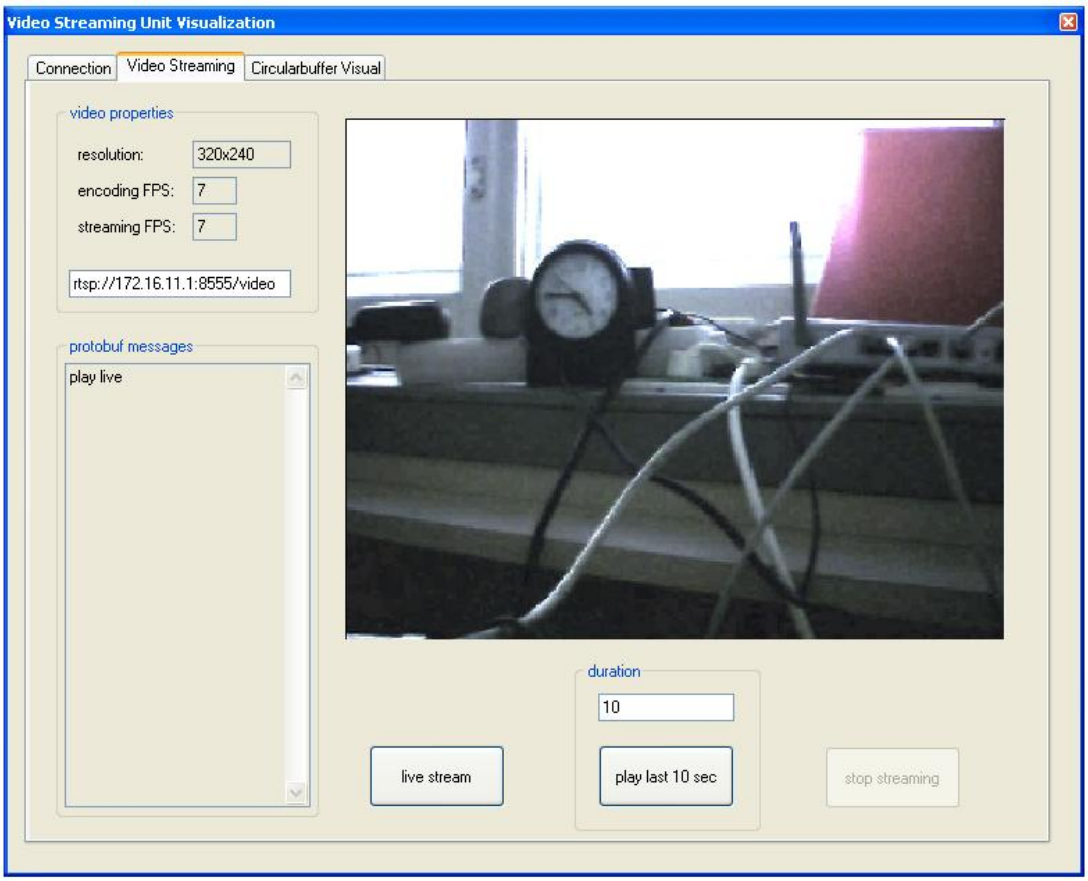


Abbildung 38: VSUV- Video Streaming

6.3.2 Reiter „Video Streaming“

Der Reiter Video Streaming ermöglicht dem Benutzer das Anfordern und Betrachten von Video-Streams. Dem Benutzer wird durch die Schaltfläche „live stream“ ermöglicht, Live-Videos von der VCSU anzufordern. Diese werden in Folge im Anzeigefenster angezeigt. Abbildung 38 zeigt den Reiter „Video Streaming“ bei der Übertragung eines Live-Streams.
Hingegen wird dem Benutzer über die Schaltfläche „play last <Zeitdauer> sek“ die Möglichkeit gegeben, On-Demand eine Videosequenz aus dem Ringpuffer zu betrachten. Dazu ist es erforderlich, die Wiedergabeposition im Ringpuffer anzugeben. Dies erfolgt durch die Zeitdauer, welche im Eingabefeld „duration“ eingegeben werden kann.

6.3.3 Reiter „Circularbuffer Visual“

Der dritte Reiter ermöglicht es, die Zustände des Zwischen- und Ringpuffers zu beobachten. Dazu werden alle Ringpuffer-relevanten Zustände einerseits textuell sowie über eine Ringpuffer-Visualisierungskomponente dargestellt (siehe Abbildung 39).

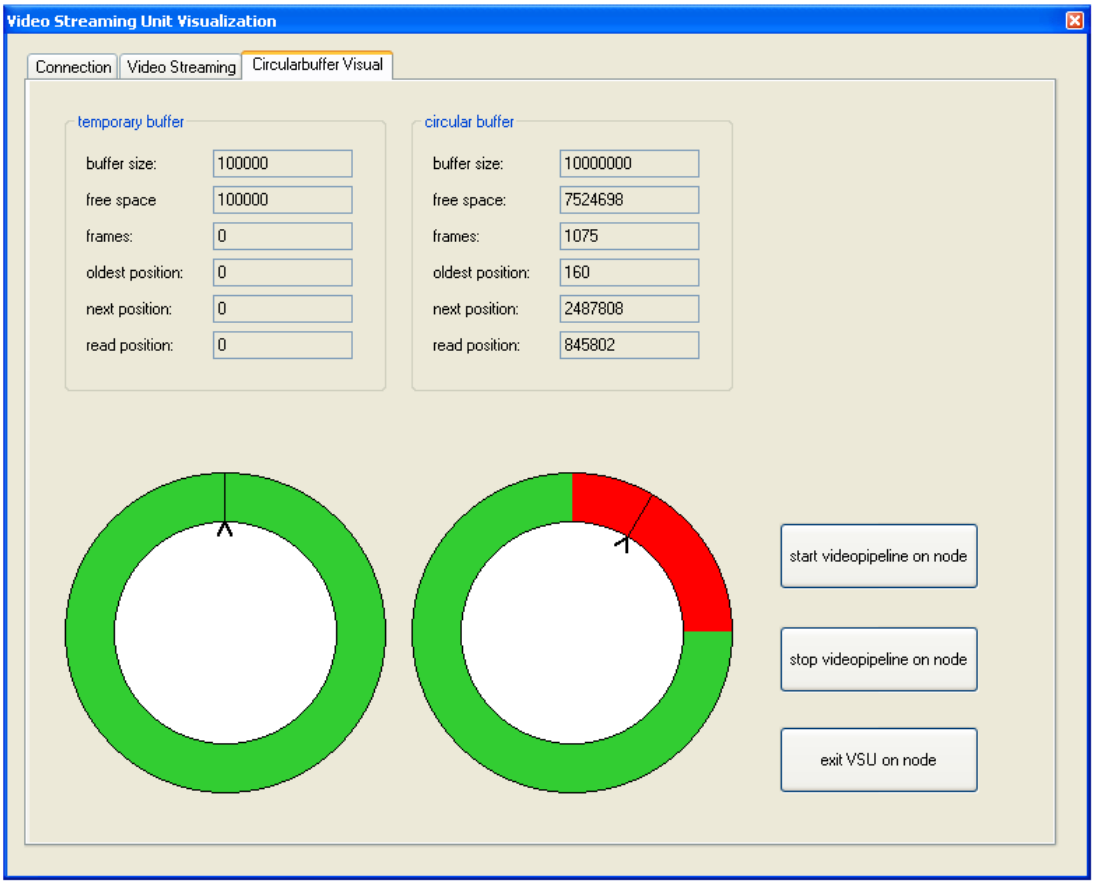


Abbildung 39: VSUV – Circularbuffer Visual

Dabei zeigt `oldest_position` auf jenes Ringpuffer-Frame, welches sich am längsten im Ringpuffer befindet. Hingegen wird mit der Position `newest_position` auf jenes Ringpuffer-Frame gezeigt, welches zuletzt im Ringpuffer gespeichert wurde. Der belegte Speicherbereich wird in der Ringpuffervisualisierung rot dargestellt, der freie hingegen grün. Die aktuelle Leseposition wird mit `read_position` angegeben, wobei in der Visualisierung dies mit dem nicht ausgefüllten Pfeil im Kreisringinneren angezeigt wird. Durch die Schaltflächen „stop videopipeline on node“ beziehungsweise „start videopipeline on node“ kann die Videoverarbeitung in der VCSU pausiert beziehungsweise fortgesetzt werden. Schlussendlich kann durch die Schaltfläche „exit VCSU on node“ die VCSU am Reasoning-Board beendet werden.

7. Implementierung der Video Compression and Streaming Unit

Ausgehend vom Design der VCSU wurde als erster Schritt die Zuordnung der Tasks auf verschiedene Prozesse beziehungsweise Threads durchgeführt. So wurde festgelegt, dass die Verarbeitung der Videobilder und Nachrichten im VCSU-Prozess durchgeführt werden. Die Tasks für das Einlesen der Videobilder und Nachrichten sowie das Video-Streaming werden jeweils in einem eigenen Thread ausgeführt. Die Vorteile von Threads sind, dass sie weniger Ressourcen benötigen, die einfachere Kommunikation zwischen den Threads und Prozessen sowie der einfachere Umgang während der Entwicklung [WWW99, S. 184].
Ursprünglich war es geplant, die Videoverarbeitung ebenfalls in einem eigenen Thread auszuführen. Jedoch musste dieser Weg verworfen werden, da der MPEG-4/H.263-Encoder innerhalb eines Threads nicht funktioniert.

7.1 Videopipeline

Ein wesentlicher Punkt bei der Implementierung der Videoverarbeitung durch die Videopipeline ist, diese mit einer möglichst gleichbleibenden Verarbeitungsrate auszuführen. Um dies zu erreichen, wird mit der Funktion `pthread_create()` der Thread `fps_const_thread` in der Datei main.c erstellt [SGD09, S. 270]. Mit der Funktion `sleep()` wird der Thread ständig für eine bestimmte Zeit pausiert [WWW99, S. 182]. Dadurch ergibt sich die gewünschte Verarbeitungsrate der Videopipeline durch die Dauer der Pause. Die Hauptverarbeitungsschleife wartet durch die Funktion `pthread_cond_wait()` solange, bis der Thread mit der Funktion `pthread_cond_signal()` einen weiteren Verarbeitungsdurchlauf erlaubt [SGD09, S. 272]. Durch diesen Ablauf kann sichergestellt werden, dass die Videopipeline immer genügend Daten erzeugt und es bei einem Live-Stream zu keiner Beendigung der RTP-Sitzung kommt.

Die folgenden drei Klassen bilden das Grundgerüst für die Videoverarbeitung. Dieses Grundgerüst dient für die Ausführung der einzelnen Videobildverarbeitungsschritte.

7.1.1 CVideoPipeline

Die Klasse `CVideoPipeline` fasst sämtliche videoverarbeitenden Klassen zusammen. Der Ablauf der Verarbeitungsschritte wird dabei innerhalb dieser Klasse festgelegt.

```
class CVideoPipeline
{
 private:
    CDoubleVideoFrame* mydoublevideoframe;
    CYUVVideoTCPSource* myyuvvideotcpsource;
    CYUVtoMP4* myyuvtomp4;
    pthread_t fps_thread;
    …
 public:
    CVideoPipeline (char* result);
    void next_frame (void);
    void start_pipeline (void);
    void stop_pipeline (void);
    unsigned long get_frames_in_buffer (void);
    void set_read_position (unsigned long frame);
    char get_framerate (unsigned char* frame_rate);
    …
};
```

CVideoPipeline (char* result)

Der Konstruktor der Klasse initialisiert sämtliche in der Klasse `CVideoPipeline` untergeordneten Klassen. Dazu zählen die Klassen `CDoubleVideoFrame`, `CYUVVideoTCPSource`, `CYUVtoMP4` und `CCircularBuffer`. Zu beachten ist, dass für die Initialisierung des MPEG-4/H.263-Encoders durch die Klasse `CYUVtoMP4` bereits eine Instanz der Klasse `CCircularBuffer` benötigt wird. Dies ist notwendig, da bei der Initialisierung des Encoders durch die Videoparameter ein MPEG-4-Start-Header generiert wird, der im Ringpuffer abgespeichert werden muss.
Schlussendlich wird im Konstruktor mit der Funktion `pthread_create()` ein Thread (`fps_thread`) gestartet, welcher die Videobildrate der Videopipeline ermittelt [SGD09, S. 270].

void next_frame (void)

Durch den Aufruf dieser Funktion wird die Abarbeitung der Videopipeline durchgeführt. Diese Umfasst das Einlesen eines YUV-Bildes durch die Klasse `CYUVVideoTCPSource`, der Komprimierung des YUV-Bildes durch die Klasse `CYUVtoMP4` und der Abspeicherung des MPEG-4-Frames in der Klasse `CCircularBuffer`. Die Funktion `next_frame` wird dabei aus dem VCSU-Prozess periodisch aufgerufen. Mögliche Erweiterungen in der Videoverarbeitung sind in dieser Funktion zu ergänzen.

void start_pipeline (void) und void stop_pipeline (void)

Durch diese beiden Funktionen steht eine Möglichkeit zur Verfügung, die Abarbeitung der Videopipeline zu steuern. So kann mit der Funktion `start_pipeline()` die Abarbeitung der Videopipeline bei einem Aufruf der Funktion `next_frame` erlaubt werden beziehungsweise durch die Funktion `stop_pipeline()` unterbunden werden.

void set_read_position (unsigned long frame)

Durch die `set_read_position()` Funktion wird der `read_position` Zeiger innerhalb des Ringpuffers positioniert. Dieser Zeiger gibt an, ab welchem Frame gelesen wird, wenn ein Lesezugriff auf den Ringpuffer erfolgt. Als Parameter wird eine Frame-Nummer übergeben. Die Frame-Nummer 0 gibt dabei das älteste Videobild im Ringpuffer an. Um die Frame-Nummer des zuletzt im Ringpufferpuffer geschrieben Frames zu erhalten, kann diese über die Funktion `get_frames_in_buffer()` abgerufen werden.

char get_framerate (unsigned char* frame_rate)

Der im Konstruktor gestartete Thread zählt die Anzahl der Abarbeitungsdurchläufe der Videopipeline pro Sekunde. Durch die Funktion `get_framerate()` kann die Anzahl der Aufrufe der letzten Sekunde abgerufen werden.

7.1.2 CVideoFrame

Diese Klasse enthält sämtliche Variablen und Funktionen für das Allokieren eines Videobildpuffers. Die Funktionen aus dieser Klasse greifen dabei auf einen speziellen Treiber zu, um für den MPEG-4/H.263-Encoder einen passenden Speicherbereich zu erstellen.

```
class CVideoFrame
{
 public:
    u32* buffer;
    u32 buffer_bus_address;
    u32 buffer_size;
    unsigned long frame_size;
    char frame_type;
 private:
    const char *memdev;
    int buffer_fd;
 public:
    CVideoFrame (char* result);
    ~CVideoFrame ();
};
```

CVideoFrame (char* result)

Der Konstruktor dieser Klasse greift auf den Treiber memalloc.ko zu und reserviert einen Speicherbereich. Dabei wird in der Variable `buffer` die virtuelle Speicheradresse des Puffers gespeichert sowie mit der Variable `buffer_bus_address` die physikalische Speicheradresse des Puffers. Die physikalische Adresse wird benötigt, damit der MPEG-4/H.263-Encoder die richtige Adresse für den Puffer bekommt. Mit `buffer` ist man zugleich in der Lage, unter Linux auf den Puffer zu zugreifen. Die Variable `buffer_size` gibt die Größe des reservierten Puffers an. Da der Puffer zur Laufzeit immer die gleiche Größe besitzt, wird mit der Variable `frame_size` der tatsächliche im Moment benutzte Speicherplatz angegeben. Der Konstruktor gibt über den Parameter `result` zurück, ob das Allokieren des Puffers erfolgreich war oder nicht.

~CVideoFrame ()

Für die ordnungsgemäße Freigabe des reservierten Speichers sorgt diese Destruktor-Funktion. Diese wird aufgerufen, sobald eine Instanz dieser Klasse freigegeben wird.

7.1.3 CDoubleVideoFrame

Mit dieser Klasse wird die Funktionalität für den Videobilddatenaustausch innerhalb der Videopipeline realisiert. Um dies zu realisieren, beinhaltet dieser Klasse zwei Instanzen der Klasse `CVideoFrame`.

```
class CDoubleVideoFrame
{
 public:
    CVideoFrame* input_frame;
    CVideoFrame* output_frame;
 public:
    CDoubleVideoFrame (char* result);
    void swap_input_output_video_frame (void);
    …
};
```

CDoubleVideoFrame (char* result)

Eine Instanz der Klasse `CDoubleVideoFrame` besteht aus zwei Instanzen der Klasse `CVideoFrame`. Ein Puffer stellt dabei einer Videoverarbeitungsfunktionen die Eingangsdaten (`input_frame`) bereit und der zweite Puffer speichert die Ausgangsdaten (`output_frame`). Mit dem Konstruktor dieser Klasse werden unter anderem diese beiden Puffer initialisiert. Damit ist es möglich, Videobilddaten effizient und einheitlich zwischen verschiedenen Funktionen auszutauschen.

void swap_input_output_video_frame (void)

Durch den Aufruf der Funktion `swap_input_output_video_frame()` wird die Rolle des Ausgangspuffer und des Eingangspuffer getauscht (siehe Abbildung 34).

7.2 Videobildbeschaffung und Kompression

Das Einlesen der YUV-Bilder erfolgt durch die Klasse `CYUVVideoTCPSource`. Diese dient als Videobildquelle für die Videopipeline. Die Videobilder werden in Folge dem On-Chip-Encoder zugeführt. Die Klasse `CYUVtoMP4` enthält die notwendigen Implementierungen für die Kompression der YUV-Bilder.

7.2.1 CYUVVideoTCPSource

In der Klasse `CYUVVideoTCPSource` wurden Funktionen implementiert, die für die Verwaltung der TCP/IP-Verbindung und zum Empfangen der YUV-Bilder zuständig sind. Dabei wird die

TCP/IP-Verbindung durch die Funktionen der Klasse `CSocket` verwaltet.
Beim Design der Software wurde ein Task für das Empfangen der Videobilddaten festgelegt. Dieser Task ist in jeder Instanz der Klasse `CYUVVideoTCPSource` vorhanden. Der Thread (`frame_capture_thread`) wird dabei durch die Funktion `pthread_create()` erstellt [SGD09, S. 270]. Die Aufgabe dieses Threads besteht darin, ständig Daten von Socket einzulesen. Durch die Verwendung eines eigenen Threads für das Einlesen der Videobilddaten muss beim Aufruf der Funktion `read_yuv_video_frame()` kein Videobild mehr über den Socket-Verbindung eingelesen werden, was dazu führt, dass es zu keinen undefinierten Verzögerungen bei der Videobildverarbeitung kommt. Innerhalb der Klasse wird immer das zuletzt empfangene Videobild gespeichert, um bei jedem Lesezugriff durch die Videopipeline sofort ein YUV-Bild liefern zu können.
Beim Testen der Software wurde beobachtet, dass die TCP/IP-Verbindung zum EVS des Öfteren keine Daten mehr an die VCSU sendet. Dieses Problem kann jedoch behoben werden, indem die TCP/IP-Verbindung neu aufgebaut wird.

```
class CYUVVideoTCPSource
{
 private:
    static const unsigned int YUV_FRAME_SIZE_320_240 = 115200;
    CSocket* client_socket;
    char newframe[YUV_FRAME_SIZE_320_240];
    char oldframe[YUV_FRAME_SIZE_320_240];
    pthread_t frame_capture_thread;
    …
 public:
    CYUVVideoTCPSource(char* result, unsigned short portnumber);
    char read_yuv_video_frame(CDoubleVideoFrame* doublevideoframe);
    …
 private:
    char connect_to_yuv_service(void);
    char get_videoboard_ip_address(char *filename);
    …
};
```

CYUVVideoTCPSource (char* result, unsigned short portnumber)

Der Konstruktor der Klasse `CYUVVideoTCPSource` liest in einem ersten Schritt mit der Funktion `get_videoboard_ip_address()` die Konfigurationsdatei jedes SENSE-Knoten aus, um die eigene IP-Adresse zu erhalten. Im zweiten Schritt instanziiert der Konstruktor eine Instanz der Klasse `CSocket`. Dabei wird die Instanz von `CSocket` als Client konfiguriert. In Folge wird mit der Funktion `connect_to_yuv_service()` und der im ersten Schritt erhaltenen IP-Adresse eine Verbindung zum EVS aufgebaut. Bei einem erfolgreichen Verbindungsaufbau zum EVS wird der `frame_capture_thread` gestartet, welcher die YUV-Bilder einliest.

char read_yuv_video_frame (CDoubleVideoFrame* doublevideoframe)

Über diese Funktion wird ein YUV-Bild vom internen Buffer der Klasse `CYUVVideoTCPSource` in den Buffer der Klasse `CDoubleVideoFrame` kopiert. Dabei wird beim Aufruf dieser Funktion

immer das aktuellste Videobild zurückgegeben.
Das Flussdiagramm in Abbildung 40 zeigt den Ablauf, der beim Auslesen eines YUV-Bildes implementiert wurde. Durch die erste Verzweigung wird entschieden, ob ein neues YUV-Bild übernommen werden kann. Liegt hingegen kein aktuelles Videobild vor, so wird das zuletzt empfangene Videobild verwendet. Jedes Mal, wenn ein altes YUV-Bild kopiert wurde, wird die Zählervariable `counter` erhöht, welche bei einem Zählerstand von Fünf den `frame_capture_thread` beendet, die TCP/IP-Verbindung abbaut und wieder aufbaut. Mit diesem Mechanismus wird sichergestellt, dass das EVS wieder Bilder and die VCSU verschickt.

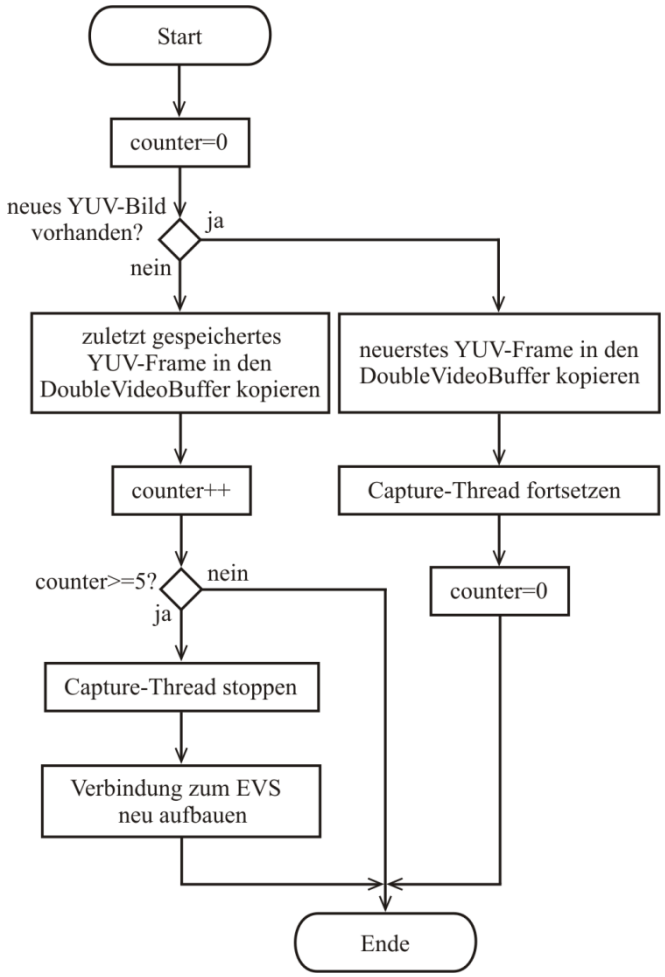


Abbildung 40: Flussdiagramm von read_yuv_video_frame

7.2.2 CYUVtoMP4

Die Klasse `CYUVtoMP4` kapselt sämtliche Funktionen, die für das Ansprechen des Hantro 5250 MPEG-4/H.263-Encoders notwendig sind. Die Funktionen dieser Klasse bauen dabei auf den Treiber des Encoders auf, der über eine API angesprochen werden kann. Der Treiber wurde von der Freescale Homepage [9] bezogen. Der Treiber des MPEG-4/H.263-Encoders ist für die Verwendung unter Linux konzipiert und lag zum Zeitpunkt der Implementierung in der Version 3.0 vor.

Der grundsätzliche Ablauf zum Encodieren von YUV-Bildern wird in Abbildung 41 gezeigt. Der Ablauf teilt sich in die Abschnitte Encoder-Initialisierung, Video-Stream-Erzeugung und Freigabe des Encoders auf. Die Video-Stream-Erzeugung unterteilt sich dabei wiederum in die Abschnitte Stream starten, Frame-Codierung und Stream beenden.
Diese Verarbeitungsschritte werden innerhalb der Klasse `CYUVtoMP4` in verschiedenen Funktionen implementiert, die nun näher erklärt werden.

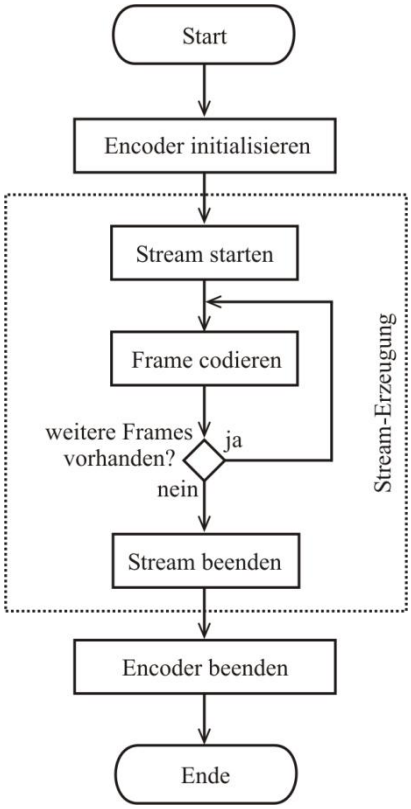


Abbildung 41: Flussdiagramm MPEG-4-Codier-Prozess (Quelle: [Han04, S. 6])

```
class CYUVtoMP4
{
 private:
    MP4EncInst encoder;
    …
 public:
    CYUVtoMP4(char* result, struct s_mp4_video_parameter*
    mp4_video_parameter, CDoubleVideoFrame* doublevideoframe);
    ~CYUVtoMP4();
    void encode_frame(CDoubleVideoFrame* doublevideoframe);
};
```

CYUVtoMP4 (char* result, struct s_mp4_video_parameter* mp4_video_parameter, CDoubleVideoFrame* doublevideoframe)

Die Initialisierung des Encoders wird im Konstruktor durchgeführt. Dazu wird mit der API-Funktion `MP4EncInit()` der Encoder mit den gewünschten Video-Parametern konfiguriert. Diese Parameter werden durch die Struktur `s_mp4_video_parameter` an den Konstruktor übergeben. Im Folgenden wird auf diese Parameter für die MPEG-4-Videoerzeugung näher eingegangen. Dabei konnten keine Probleme bei der Wiedergabe mit dem VLC Media Player beobachtet werden, wie diese in [KLW08 S. 33] berichtet wurden. Probleme konnten hingegen nur beim Versuch, einen H.263-Stream wiederzugeben, festgestellt werden. Deswegen wird in Folge auf die Beschreibung der Parameter für H.263 verzichtet.

Videobildauflösung

Bei der Initialisierung ist es notwendig, die Auflösung der Eingangsvideobilddaten genau zu spezifizieren. Dabei muss zwischen der Aufnahmeauflösung und der Encoder-Eingangsvideobildauflösung unterschieden werden. Diese können voneinander unterschiedlich sein, da die Videobildvorverarbeitung eine größere Auflösung unterstützt. Die Limitierung der Videobildauflösung wird durch die eingeschränkte Auswahl der unterstützten Profile & Levels hervorgerufen [Han04, S. 9].
Bei der Implementierung wurde auf die Verwendung der Videobildvorbearbeitung verzichtet, da die Videobilder bereits eine Auflösung von 320 x 240 Pixel besitzen.

Tabelle 6: Einige Profile & Level des Hantro 5250 Encoders (Quelle: [Han04, S. 8])

Profile & Level	Encoderbildgröße [Pixel]	Bildrate [FPS]	Birrate [kBits/s]
MPEG4_SIMPLE_PROFILE_LEVEL_0	128 x 96 (sub-QCIF)	15	64
	160 x 120 (QQVGA)	15	64
	176 x 144 (QCIF)	15	64
MPEG4_SIMPLE_PROFILE_LEVEL_0B	128 x 96 (sub-QCIF)	15	128
	160 x 120 (QQVGA)	15	128
	176 x 144 (QCIF)	15	128
MPEG4_SIMPLE_PROFILE_LEVEL_1	128 x 96 (sub-QCIF)	30	64
	160 x 120 (QQVGA)	15	64
	176 x 144 (QCIF)	15	64
MPEG4_SIMPLE_PROFILE_LEVEL_2	160 x 120 (QQVGA)	30	128
	176 x 144 (QCIF)	30	128
	352 x 288 (CIF)	15	128
MPEG4_SIMPLE_PROFILE_LEVEL_3	176 x 144 (QCIF)	30	384
	352 x 288 (CIF)	30	384

Videobildrate

Damit der Encoder in der Lage ist, die Zieldatenrate einzuhalten, ist es notwendig, eine Videobildrate zu konfigurieren. Die Videobildrate wird mit einem Videobildraten-Zähler und Videobildraten-Nenner dargestellt. Die Videobildrate ergibt sich dabei aus der Division von Videobildraten-Zähler

durch Videobildraten-Nenner. Damit lassen sich auch ungerade Videobildraten konfigurieren [Han04, S. 10].

Stream-Profile & Level

Dadurch, dass MPEG-4 so ausgelegt wurde, in den verschiedensten Anwendungen eingesetzt werden zu können, ist es notwendig, die vielen Methoden auf die gegebenen Anforderungen anzupassen [PE02, S. 52]. Dies kann mit Profilen und Levels festgelegt werden. Profile und Levels sind sowohl bei den H.26x-Standards als auch bei den MPEG-Standards zu finden. Dabei beschreibt ein Profil eine Teilmenge der Codiermethoden des Standards. Ein Level hingegen spezifiziert die Datenrate, welche sich aus der Videobildauflösung beziehungsweise Videobildrate des Videos zusammensetzt [Mil95, S. 51]. Einen Auszug aus den unterstützten Profilen und Levels des Hantro Encoders wird in Tabelle 6 gezeigt.

Stream-Typ

Der Encoder ist in der Lage, verschiedene Stream-Typen zu erzeugen. Tabelle 7 zeigt die zur Verfügung stehenden Konfigurationsmöglichkeiten des Encoders [Han04, S. 10].

Mit MPEG4_PLAIN_STRM erzeugt der Encoder einen gewöhnlichen MPEG-4-Stream. Mit diesem Stream-Typ werden keine Verfahren verwendet, welche die Robustheit bei Übertragungsfehlern erhöhen. Durch den Stream-Typ MPEG4_VP_STRM erzeugt der Encoder Synchronisationsmarkierungen, welche dem Decoder im Fehlerfall ermöglichen, den Anfang von Videopaketen einfacher wiederzufinden. MPEG4_VP_DP_STRM erweitert MPEG4_VP_STRM um die Eigenschaft, dass die Daten partitioniert werden. Dadurch erhöht sich die Fehlerbehebungsrate. MPEG4_VP_DP_RVLC_STRM baut auf den Eigenschaften von MPEG4_VP_DP_STRM auf und erweitert diese mit umkehrbaren VLC. Dadurch wird der Stream noch robuster gegenüber fehlerhaften Datenübertragungen. Mit MPEG4_SVH_STRM produziert der Encoder einen Stream der SVH (short video header) verwendet [Han04, S. 19].

Tabelle 7: Wesentliche Stream-Typen des Hantro 5250 Encoders (Quelle: [Han04, S. 10])

Stream-Typ	Video-pakete	Datenpar-tionierung	RVLC	AC/DC Vorhersage	4 BV	uneingesch-ränkte BV
MPEG4_PLAIN_STRM	AUS	AUS	AUS	EIN	EIN	EIN
MPEG4_VP_STRM	EIN	AUS	AUS	EIN	EIN	EIN
MPEG4_VP_DP_STRM	EIN	EIN	AUS	EIN	EIN	EIN
MPEG4_VP_DP_RVLC_STRM	EIN	EIN	EIN	EIN	EIN	EIN
MPEG4_SVH_STRM	AUS	AUS	AUS	AUS	AUS	AUS

Die Initialisierung des Encoders wird mit der Funktion `MP4EncInit()` durchgeführt. Diese gibt `ENC_OK` zurück, wenn die Initialisierung erfolgreich ist. Nachdem der Encoder initialisiert wurde, wird die Funktion `Mp4EncStrmStart()` aufgerufen. Durch diese Funktion wird der Stream-Start-Header erstellt [Han04, S. 7, 14]. Der Start-Header wird im Anschluss im Ringpuffer gespeichert.

void encode_frame (CDoubleVideoFrame* doublevideoframe)

Nach dem Erstellen des Start-Headers kann eine beliebig lange Sequenz von Videobildern codiert werden. Die Encodierung der YUV-Bilder in MPEG-4-Bilder funktioniert bildweise, wobei durch den Aufruf der Funktion `MP4EncStrmEncode()` eine Encodierung durchgeführt wird. Als Eingangsvideobildformat wird ausschließlich planar YCbCr 4:2:0 unterstützt. Der Eingangs Videopuffer kann dabei als einzelner, linearer Puffer oder aus drei einzelnen Puffern mit den jeweiligen Komponenten übergeben werden [Han04, S. 14-15]. Als Parameter der Funktion ist ein Zeiger auf eine Instanz der Klasse `CDoubleVideoFrame` notwendig, welche die Videobilddaten als einzelnen, linearen Puffer enthält.

~CYUVtoMP4 ()

Der MPEG-4-Stream wird mit der Funktion `MP4EncStrmEnd()` beendet. Die Funktion erstellt dabei den Stream-End-Header. Nachdem der End-Header erstellt wurde, werden mit der API-Funktion `MP4EncRelease()` sämtliche Ressourcen rund um den MPEG-4-Encoder wieder frei gegeben [Han04, S. 20].

7.3 Ringpuffer

Die Implementierung des Ringpuffers orientiert sich nach dem Design, welches in Kapitel 5.2.5 beschrieben wurde. Die Funktionen teilen sich dazu auf fünf Klassen auf, welche zum Teil in einer hierarchischen Struktur aufeinander aufbauen.

Auf der untersten Hierarchiestufe befindet sich die Klasse `CCircularBufferLow`. Sie enthält Funktionsprototypen, welche das byteweise Lesen und Schreiben in den Ringpuffern ermöglicht. Die Funktionen aus der Klasse `CCircularBufferHigh` ermöglichen, basierend auf den Funktionen der Klasse `CCircularBufferLow`, das Schreiben und Lesen von einzelnen Frames.
Die Klassen `CCircularBufferHDD` und `CCircularBufferRAM` enthalten unter anderem die Implementierungen aus der Klasse `CCircularBufferLow` und ergänzen die geerbten Funktionen der Klasse `CCircularBufferHigh` durch spezielle Funktionen, durch die sich die Ringpuffer auszeichnen. Die oberste Hierarchiestufe stellt die Klasse `CCircularBuffer` dar. Diese enthält die implementierten Funktionen, welche den Zugriff auf die Puffer-Instanzen der Klassen `CCircularBufferHDD` und `CCircularBufferRAM` regelt. Dabei ist die Instanz der Klasse `CCircularBufferHDD` für die Speicherung von mindestens der letzten 25 min zuständig, während die Instanz der Klasse `CCircularBufferRAM` als Zwischenpuffer verwendet wird.

7.3.1 CCircularBufferLow

Die Funktionen und Variablen, welche in dieser Klasse definiert sind, stellen eine allgemeine Funktionsbasis für den Ringpuffer dar. Die Implementierung der Funktionen erfolgt dabei nicht in dieser Klasse, sondern erst in den Klassen `CCircularBufferHDD` und `CCircularBufferRAM`. Diese grundlegenden Funktionen zeichnen durch den Typen-Qualifizierer `virtual` aus. Dieser signalisiert dem Compiler, dass die Implementierung nicht in dieser Klasse erfolgt.

```
class CCircularBufferLow
{
 public:
    unsigned long write_position;
    unsigned long read_position;
    …
 private:
    virtual unsigned int circular_buffer_write_low (void *ptr, un-
signed long size) = 0;
    virtual unsigned int circular_buffer_read_low (void *ptr, un-
signed long size) = 0;
    virtual char circular_buffer_fseek_write_low (long int offset,
int origin) = 0;
    virtual char circular_buffer_fseek_read_low (long int offset,
int origin) = 0;
};
```

virtual unsigned int circular_buffer_write_low (void *ptr, unsigned long size)

Mit dieser Funktion wird in einer abgeleiteten Klasse erzwungen, dass eine byteweiser Schreibfunktion implementiert wird. Im Speziellen soll durch die Implementierung dieser Funktion ermöglicht werden, dass Daten aus einem allgemeinen Speicherbereich in den Ringpuffer kopiert werden können. Der Parameter `ptr` beinhaltet einen Zeiger auf den Speicherberreich, wobei der Parameter `size` die Anzahl der zu kopierenden Bytes angibt. Die Funktion soll dabei die tatsächliche Anzahl der geschriebenen Bytes zurück geben.

virtual unsigned int circular_buffer_read_low (void *ptr, unsigned long size)

Diese Funktion stellt die Lesefunktion dar und ermöglicht das byteweise Lesen aus dem Ringpuffer. Im Gegensatz zur Schreibfunktion zeigt der Parameter `ptr` auf einen Speicherbereich, indem die Daten aus dem Ringpuffer geschrieben werden sollen. Der Parameter `size` gibt die Anzahl der Bytes an, die gelesen werden sollen. Die Funktion soll schlussendlich die tatsächliche Anzahl der gelesenen Bytes zurück geben.

virtual char circular_buffer_fseek_write_low (long int offset, int origin)

Durch die Implementierung dieser Funktion soll ermöglicht werden, die Position, an der mit der Funktion `circular_buffer_write_low()` geschrieben wird, zu ändern. Der Parameter `offset` gibt die Anzahl der Byte, ausgehend vom Parameter `origin,` an. Dabei sind für `origin` die Parameter `SEEK_SET`, `SEEK_CUR` und `SEEK_END` gültig. Bei `SEEK_SET` wird der `offset` ausgehend vom Dateibeginn addiert, bei `SEEK_CUR` von der momentanen Position und bei `SEEK_END` ausgehend von Dateiende. Die aktuelle Schreibposition soll dabei in der Variable `write_position` gespeichert werden. Wenn die Funktion erfolgreich ausgeführt wurde, soll die Implementierung eine null zurückgeben, oder keine null, wenn ein Fehler aufgetreten ist.

virtual char circular_buffer_fseek_read_low (long int offset, int origin)

Diese Funktion besitzt grundsätzlich die gleiche Funktion wie die Funktion `circular_buffer_fseek_write_low()`, jedoch erfolgt durch diese Funktion die Änderung der Leseposition, welche durch die Variable `read_position` gespeichert wird. Die Änderung der Leseposition wirkt sich dabei auf die Funktion `circular_buffer_read_low()` aus.

7.3.2 CCircularBufferHigh

Diese Klasse `CCircularBufferHigh` ist eine Spezialisierung der Klasse `CCircularBufferLow` und umfasst die Implementierungen für das Lesen und Speichern einzelner Frames. Durch die Implementierung von `CCircularBufferHigh` wird auch der grundsätzliche Aufbau der Datenstruktur des Ringpuffers festgelegt. Dieser ist in Abbildung 42 zu sehen und wird in Folge näher erklärt.

Der erste Bereich eines Ringpuffers umfasst die statischen Daten. Dieser besteht im Wesentlichen aus einem Ringpuffer-Header sowie dem MPEG-4-Start-Header. Der Ringpuffer-Header enthält allgemeine Informationen über den Ringpuffer, wie zum Beispiel die absolute Größe, Anzahl der Frames, Position des ersten Frames und so weiter. Der MPEG-4-Start-Header befindet sich ebenfalls im statischen Bereich des Ringpuffers. Dieser wird separat gespeichert, damit dieser nicht innerhalb des Ringpuffers überschrieben wird. Der MPEG-4-Start-Header wird zu Beginn jeder Videoübertragung benötigt, da dieser den MPEG-4-Stream beschreibt. Anhand des Start-Headers ist es dem VLC Media Player möglich, den Datenstrom dem richtigen Decoder zu zuführen.

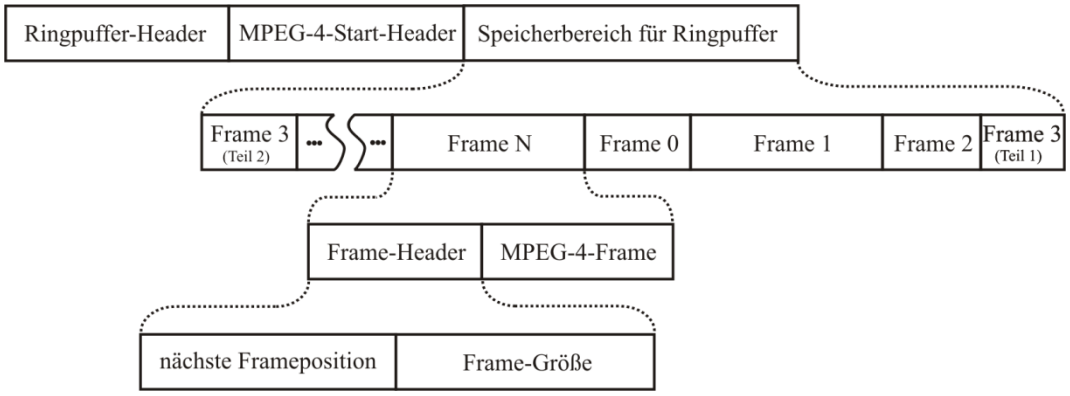


Abbildung 42: Grundsätzliche Aufbau der Ringpuffer-Datenstruktur

Der zweite Teil der Pufferdatei steht für die Speicherung der einzelnen sogenannten Frames zur Verfügung. Dabei ergibt sich dieser Speicherplatz aus der festgelegten Maximalgröße des Speichers, abzüglich der Größe des statischen Teiles. Innerhalb des Speicherbereiches werden die Nutzdaten mittels Frames abgelegt. Ein Frame besteht aus einem statischen Frame-Header und den Nutzdaten in Form eines MPEG-4-Frames. Dadurch, dass die Größe des MPEG-4-Frames nicht konstant ist, befinden sich die Nutzdaten nach dem statischen Teil des Frames. Die einzelnen Frames sind innerhalb des Speicherbereichs als Schlange verkettet und definieren damit die Abfolge der MPEG-4-Frames und somit der Videosequenz [KS03, S. 270]. Dabei zeigt die erste Variable im Frame-

Header auf das Startbyte der Speicherposition des nächsten Frames im Speicherbereich. Das Ende der Verkettung wird durch eine 0 definiert, da an der Speicherstelle 0 keine Frames gespeichert werden dürfen. Die zweite Variable im Frame-Header gibt über die Größe des MPEG-4-Frames Auskunft.

```
class CCircularBufferHigh : public CCircularBufferLow
{
 public:
    unsigned long max_buffer_size;
    unsigned long free_buffer_space;
    unsigned long oldest_frame_position;
    unsigned long newest_frame_position;
    unsigned long frames;
    …
 public:
    char write_mp4_frame (struct s_mp4_frame* mp4_frame);
    unsigned int read_mp4_frame (struct s_mp4_frame* mp4_frame);
 private:
    virtual unsigned int circular_buffer_write_low (void *ptr, un-
signed long size) = 0;
    virtual unsigned int circular_buffer_read_low (void *ptr, un-
signed long size) = 0;
    virtual char circular_buffer_fseek_write_low (long int offset,
int origin) = 0;
    virtual char circular_buffer_fseek_read_low (long int offset,
int origin) = 0;
};
```

char write_mp4_frame (struct s_mp4_frame* mp4_frame)

Diese Funktion schreibt ein MPEG-4-Frame in den Ringpuffer. Durch die Schreibfunktion wird ein Frame auf die Position geschrieben, auf die die Variable `newest_frame_position` zeigt. Des Weiteren werden noch die Puffer-Zustandsvariablen, wie die Anzahl der Frames im Puffer, sowie der noch freie Speicherplatz (`free_buffer_space`) im Ringpuffer, aktualisiert. Die Funktion gibt eine eins zurück, wenn die Speicherung des Frames erfolgreich war.

unsigned int read_mp4_frame (struct s_mp4_frame* mp4_frame)

Die Funktion `read_mp4_frame()` ermöglicht das Lesen eines MPEG-4-Frames. Dabei wird jenes Frame ausgelesen, auf das die Variable `read_position` zum Zeitpunkt des Funktionsaufrufes zeigt. Nachdem das Frame gelesen wurde, wird die Variable `read_position` auf das folgende Frame gesetzt. Der Rückgabewert der Funktion gibt Auskunft über die Anzahl der gelesenen Bytes.

7.3.3 CCircularBufferHDD

Die Klasse `CCircularBufferHDD` stellt eine von zwei Spezialisierung der Klasse `CCircularBufferHigh` dar. Diese Klasse implementiert dabei den Ringpuffer, welcher für die Speiche-

rung von den mindestens letzten 25 Minuten zuständig ist.
Die Implementierung der Klasse gliedert sich dabei in zwei Teile. Der erste Teil umfasst die Implementierung der Funktionen, welche in der Klasse `CCircularBufferLow` deklariert wurden. Die Implementierung dieser Funktionen basiert dabei auf Dateisystem-Funktionen des Betriebssystems, wie `fopen()`, `fclose()`, `fwrite()`, `fread()` und `fseek()`. Damit kann auf die Datei-Streams zugegriffen werden, welche unter anderem eine permanente Speicherung der Daten ermöglicht [WWW99, S. 162-170]. Durch die Änderung des Speicherpfads der Ringpufferdatei ist eine Anpassung an die vorhandenen Massenspeicher möglich.
Der zweite Teil der Implementierung umfasst die Vorgehensweise beim Schreiben und Lesen von Frames aus dem Ringpuffer. Dies wird in den folgenden Funktionen näher beschrieben.

```
class CCircularBufferHDD : public CCircularBufferHigh
{
   static const unsigned int OFFSET_MP4_START_HEADER = 100;
   FILE *buffer_file;
   pthread_mutex_t buffer_mutex;
   unsigned long position_index[POSITION_INDEX_SIZE];
   unsigned long position_index_oldest;
   unsigned long position_index_newest;
   …
 public:
    CCircularBufferHDD (char* result, char* buffer_file_name, un-
signed int buffer_file_name_lenght, unsigned long buffer_size);
    char write_mp4_frame (CDoubleVideoFrame* doublevideoframe);
    unsigned int read_mp4_block (unsigned char *pBuffer, unsigned
int BufferSize);
    void set_read_position (unsigned long frames);
    …
};
```

CCircularBufferHDD (char* result, char* buffer_file_name, unsigned int buffer_file_name_lenght, unsigned long buffer_size)

In der Konstruktor-Funktion wird durch die Funktion `fopen()` eine neue Pufferdatei (`buffer_file`) angelegt [WWW99, S. 163]. Der Speicherpfad beziehungsweise Dateiname der Pufferdatei wird der Funktion `fopen()` durch die Parameter `buffer_file_name` und `buffer_file_name_lenght` der Konstruktor-Funktion übergegeben. Liegt beim Aufruf der Funktion `fopen()` bereits eine Datei mit dem gleichen Speicherpfad sowie Dateinamen vor, so wird der Inhalt dieser Datei gelöscht. Der Parameter `buffer_size` gibt die maximale Größe an, welche die Pufferdatei erreichen darf. Des Weiteren werden im Konstruktor sämtliche Zustandsvariablen des Ringpuffers initialisiert. Damit der Zugriff durch mehrere Prozesse beziehungsweise Threads nicht zu inkonsistenten Daten führt, wird mit der Funktion `pthread_mutex_init()` ein Mutex erzeugt [Wall99, S. 191].

char write_mp4_frame (CDoubleVideoFrame* doublevideoframe)

Durch die Funktion `write_mp4_frame()` wird es ermöglicht, ein einzelnes MPEG-4-Frame in die Ringpufferdatei zu schreiben.

Als ersten Schritt wird mit der Funktion `pthread_mutex_trylock()` überprüft, ob sich ein Thread im kritischen Codebereich des Ringpuffers befindet [Wall99, S. 191]. Wurde das Mutex durch einen anderen Thread gesetzt, so kommt die Funktion `write_mp4_frame()` bereits zum Ende, indem sie eine 0 zurückgibt, welche angibt, dass das MPEG-4-Frame nicht gespeichert werden konnte. Ist das Mutex hingegen nicht gesetzt, so wird das MPEG-4-Frame gespeichert. Dafür ist ein gültiger Parameter `doublevideoframe` notwendig, welcher auf eine Instanz der Klasse `CDoubleVideoFrame` zeigt und das zu speichernde MPEG-4-Frame enthält. Das zu speichernde MPEG-4-Frame wird anhand des Frame-Typs unterschiedlich behandelt. So wird der MPEG-4-Start-Header in den statischen Teil des Ringpuffers gespeichert, während ein MPEG-4-Frame im Ringpufferspeicherbereich am vorderen Ende der Frame-Kette abgelegt wird.
Bevor ein Frame in den Ringpuffer geschrieben wird, wird geprüft, ob genügend freier Speicherplatz verfügbar ist. Ist dies der Fall, so wird das Frame an der Position `newest_frame_position` gespeichert. Steht hingegen nicht ausreichend Speicherplatz zur Verfügung, wird Speicherbereich freigegeben. Die Speicherfreigabe betrifft dabei die zeitlich am längsten gespeicherten Frames. Das älteste Frame wird durch den Zeiger `oldest_frame_position` in der Klasse `CCircular-BufferHigh` markiert. Beim Freigeben von Speicher werden so viele ältere Frames freigegeben, bis das neue Frame gespeichert werden kann. Durch das eventuelle Freigeben von Frames beziehungsweise Speichern eines neuen Frames werden auch die Pufferzustandsvariablen `free_buffer_space`, `newest_frame_position`, `oldest_frame_positon` sowie die Anzahl der gespeicherten Frames (`frames`) aktualisiert. Damit wird erreicht, dass immer die aktuellsten MPEG-4-Frames gespeichert werden.
Jedoch hat die Verkettung der Frames einen Nachteil. Dieser kommt zu tragen, wenn die Pufferdatei sehr groß ist und das letzte Frame, welches sich am vorderen Ende der Frame-Kette befindet, gelesen wird. Dadurch, dass die Verkettung nur in eine Richtung durchschritten werden kann, kann es durchaus länger dauern, bis der gewünschte Frame erreicht wird. Um diesen Nachteil zu umgehen, werden die einzelnen Frame-Positionen zusätzlich in einem Feld (`position_index`) gespeichert, welcher sich im Arbeitsspeicher befindet. Der Anfang und das Ende der Frame-Positionen in diesem Feld werden durch die Variablen `position_index_oldest` und `positi-on_index_newest` gespeichert. Durch dieses Feld wird es ermöglicht, in kürzester Zeit ein Frame im Ringpuffer auszulesen. Die Verwaltung dieses Feldes erfolgt dabei ebenfalls durch diese Schreibfunktion.

unsigned int read_mp4_block (unsigned char *pBuffer, unsigned int BufferSize)

Diese Funktion ermöglicht das blockweise Auslesen von Daten aus dem Ringpuffer. Der Parameter `pBuffer` zeigt auf einen Speicherbereich, in den die Daten geschrieben werden sollen. Der Parameter `BufferSize` gibt hingegen die Größe dieses Speicherbereichs an.
Beim Füllen der Speicherbereichs wird unterschieden, ob eine RTP-Sitzung neu aufgebaut wurde oder bereits eine Sitzung besteht. Liegt Ersteres vor, so muss zu Beginn der MP4-Start-Header in den Speicherbereich kopiert werden. Der restliche Speicherbereich wird schlussendlich mit MPEG-4-Frames gefüllt. Wird hingegen die Funktion `read_mp4_block` aufgerufen wenn keine RTP-Sitzung begonnen hat, so wird Speicherbereich auf den `pBuffer` zeigt ausschließlich mit MPEG-4-

Frames gefüllt. Als Rückgabewert wird die Anzahl der in den Speicherbereich geschriebenen Bytes zurück gegeben.

void set_read_position (unsigned long frames)

Durch diese Funktion kann die Start-Position für das Lesen (`read_position`) aus dem Ringpuffer geändert werden. Um die Start-Position eines Frames im Ringpuffer zu erhalten, wird auf das bereits erwähnte `position_index` Feld zurückgegriffen. Durch den Parameter `frames` kann die Position des gewünschten Frames angegeben werden. Dabei wird mit 0 das älteste Frame im Ringpuffer angesprochen.

7.3.4 CCircularBufferRAM

Die Klasse `CCircularBufferRAM` stellt die zweite Spezialisierung der Klasse `CCircularBufferHigh` dar. Die Instanz dieser Klasse kommt dabei für die Zwischenspeicherung der MPEG-4-Frames zum Einsatz. Der Zwischenspeicher arbeitet nach dem Prinzip, dass das zuletzt gespeicherte Frame beim nächsten Lesezugriff wieder ausgelesen wird (FIFO). Damit kann die Videopipeline mehrere MPEG-4-Frames speichern, auch wenn gerade kein Zugriff auf den eigentlichen Ringpuffer möglich ist. Die Anzahl der zu speichernden Frames hängen dabei von der konfigurierten Größe des Speicherbereichs in der Klasse `CCircularBufferRAM` ab.
Der erste Teil der Implementierung dieser Klasse umfasst wie in der Klasse `CCircularBufferHDD` die Implementierung der Funktionen aus der Klasse `CCircularBufferLow`. Die Speicherung der Frames erfolgt jedoch in dieser Klasse nicht in einer Datei, sondern in einem Speicherbereich im Arbeitsspeicher. Der Aufbau des Puffers unterscheidet sich dabei ebenfalls leicht. So besitzt der Ringpuffer keinen statischen Header-Teil sowie kein Feld, welches sämtliche Frame-Positionen des Puffers speichert. Diese ist auch nicht notwendig, da der Zwischenspeicher eine deutlich geringere Größe besitzt. Der zweite Teil der Implementierung umfasst auch hier wieder die Schreib- und Lesefunktion auf den Puffer, auf die nun näher eingegangen wird.

```
class CCircularBufferRAM : public CCircularBufferHigh
{
 public:
    char write_mp4_frame (CDoubleVideoFrame* doublevideoframe);
    unsigned int read_mp4_frame (struct s_mp4_frame* mp4_frame);
 private:
    unsigned int circular_buffer_write_low (void* ptr, unsigned
long size);
    unsigned int circular_buffer_read_low (void* ptr, unsigned long
size);
    char circular_buffer_fseek_write_low (long int offset, int ori-
gin);
    char circular_buffer_fseek_read_low (long int offset, int ori-
gin);
};
```

char write_mp4_frame (CDoubleVideoFrame* doublevideoframe)

Diese Funktion ist dafür zuständig, einzelne MPEG-4-Frames in den Zwischenpuffer zu schreiben. Dabei wird das zu speicherende Frame, wie auch bei der Funktion `write_mp4_frame` aus der Klasse `CCircularBufferHDD`, am vorderen Ende der Frame-Kette eingeordnet. Dabei unterscheiden sich diese Funktionen von der gleichnamigen Funktion aus der Klasse `CCircularBufferHDD` in einigen Punkten.
Die Speicherung der MPEG-4-Frames erfolgt wie auch in der Klasse `CCircularBufferHDD` durch verkettete Frames, wobei die einzelnen Frames im Arbeitsspeicher abgelegt werden und nicht auf Dateisystem-Funktionen zurückgreifen. Des Weiteren wird in dieser Klasse keinen Ringpuffer-Header benötigt, da die Speicherung unabhängig vom MPEG-4-Frame Typ erfolgt und somit der MPEG-4-Start-Header keine besondere Verarbeitung erfordert. Da der Zwischenpuffer nach dem FIFO-Prinzip arbeitet, ist es nicht notwendig jede einzelnen Frame-Positionen zu speichern. Hier ist es ausreichend die beiden Enden der Frame-Kette zu speichern, welche durch die Variablen `oldest_frame_position und newest_frame_position` markiert werden. Ein weiterer wesentlicher Unterschied kommt zu tragen, wenn kein freier Speicherplatz mehr vorhanden ist. So wird bei dieser Spezialisierung nicht das älteste Frames überschrieben, sondern das zu speicherende MPEG-4-Frame verworfen. Jedoch sollte diese Situation nie auftreten wenn die Größe des Zwischenspeichers groß genug gewählt wurde beziehungsweise der i.MX31 ausreichend Rechenzeit für die Verarbeitung der Videos besitzt. Deswegen es gleichgültig ob das zu speichernde MPEG-4-Frame verworfen wird oder ältere Frames im Zwischenpuffer überschrieben werden. Der Parameter beziehungsweise Rückgabewert dieser Funktion verhält sich sonst gleich wie die gleichnamige Funktion aus der Klasse `CCircularBufferHDD`.

unsigned int read_mp4_frame (struct s_mp4_frame* mp4_frame)

Um aus dem Zwischenpuffer Frames zu lesen, kommt die Funktion `read_mp4_frame()` zum Einsatz. Die Lesefunktion ist dabei nur in der Lage, einzelne Frames zu lesen. Beim Lesen von Frames unterscheidet sich diese Funktion von der Lesefunktion aus der Klasse `CCircularBufferHDD` dahingehend, dass die Lese-Position vorgegeben ist. So wird bei einem Lesezugriff immer das Frame gelesen, welches sich am hinteren Ende der Frame-Kette befindet (FIFO). Nachdem ein Frame ausgelesen wurde, wird der verwendete Speicherplatz freigegeben und kann durch ein anderes Frame überschrieben werden.

7.3.5 CCircularBuffer

Die Klasse CCircularBuffer verwaltet die Instanz `circular_buffer` der Klasse `CCircularBufferHDD` und die Instanz `temporary_buffer` der Klasse `CCircularBufferRAM`. Dadurch, dass der VCSU-Prozesses und der Streaming-Thread Zugriff auf den Ringpuffer haben, muss der Zugriff auf den Ringpuffer geregelt werden. Die Funktionalitäten dazu werden in der Lese- beziehungsweise Schreibfunktion implementiert.

```
class CCircularBuffer
{
```

```
 private:
    CCircularBufferHDD* circular_buffer;
    CCircularBufferRAM* temporary_buffer;
    …
 public:
    CCircularBuffer (char* result);
    char write_mp4_frame (CDoubleVideoFrame* doublevideoframe);
    unsigned int read_mp4_block (unsigned char *pBuffer, unsigned
int BufferSize);
    …
};
```

CCircularBuffer (char* result)

Der Konstruktor der Klasse `CCircularBuffer` instanziiert jeweils eine Instanz der Klasse `CCircularBufferHDD` und `CCircularBufferRAM`. Über den Parameter `result` wird zurückgeben, wenn die Puffer angelegt werden konnten.

char write_mp4_frame (CDoubleVideoFrame* doublevideoframe)

Die Schreibfunktion ist dafür zuständig, selbst während das Zeitintervalls in dem der Streaming-Thread auf den Ringpuffer zugreift, MPEG-4-Frames zu speichern. Dazu wird entschieden, ob das Frame im Ringpuffer oder im Zwischenpuffer gespeichert wird. Der implementierte Ablauf der Schreib-Funktion wird in Abbildung 43 gezeigt. Dabei wird in einer ersten Programmverzweigung geprüft, ob sich Daten im Zwischenpuffer befinden. Ist dies der Fall, so wird das MPEG-4-Frame in den Zwischenpuffer geschrieben, auch wenn ein Zugriff auf den Ringpuffer möglich wäre. Damit wird die Reihenfolge der Videosequenz beibehalten.

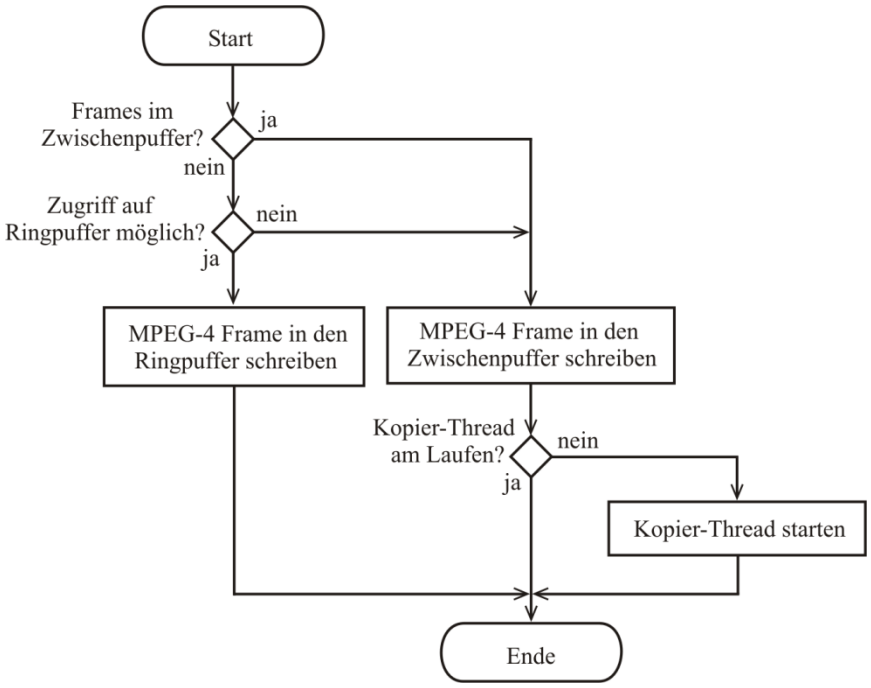


Abbildung 43: Flussdiagramm von write_mp4_frame

Sobald das Frame in den Zwischenpuffer gespeichert wurde, wird geprüft, ob der Kopier-Thread bereits ausgeführt wird. Wenn dies nicht der Fall ist, wird dieser mit der Funktion `pthread_create()` gestartet [SGD09, S. 270].
Ist der Zwischenpuffer hingegen leer, so wird geprüft, ob es möglich ist, das Frame in den Ringpuffer zu schreiben. Ist dies nicht der Fall, so wird das MPEG-4-Frame in den Zwischenpuffer geschrieben. Hingegen wird das MPEG-4-Frame im Ringpuffer gespeichert, wenn die Funktion `write_mp4_frame()` der Klasse `CCircularBufferHDD` nicht 0 zurück gibt.

unsigned int read_mp4_block (unsigned char *pBuffer, unsigned int BufferSize)

Die `read_mp4_block()` ruft ausschließlich die Member-Funktion `read_mp4_block()` der Klasse `CCircularBufferHDD` auf. Diesbezüglich verhalten sich die Parameter gleich wie in der Funktion aus der Klasse `CCircularBufferHDD`.

7.3.6 Speichermedium für den Ringpuffer

Ein wichtiger Punkt für die Erfüllung der Anforderung ist die Speicherung von mindestens 25 Minuten Videomaterial. Dadurch, dass die Funktionen der Klasse `CCircularBufferHDD` auf den Dateisystemfunktionen aufbauen, stehen am SENSE-Knoten folgende Speichermöglichkeiten zur Verfügung.

RAM-Disk

Jeder SENSE-Knoten verfügt über 32 MByte RAM, wovon 16 MByte für eine RAM-Disk verwendet werden. Diese RAM-Disk zeichnet sich dadurch aus, dass die Daten gelöscht werden, sobald die RAM-Bausteine von der Spannungsquelle getrennt werden. Will man Daten in der RAM-Disk speichern, so können diese im Verzeichnis „/var/sense/“ abgelegt werden.

Flash-Disk

Ein SENSE-Knoten verfügt über 128 MByte NAND Flash. Dieser wird unter anderem dazu verwendet, Anwendungen wie die VCSU und RDU zu speichern. Verglichen mit dem RAM gehen die Daten im Flash nicht verloren, wenn die Flash-Bausteine von der Spannungsversorgung getrennt werden. Jedoch haben Flash-Speicherbaust eine den Nachteil, dass diese nur eine begrenzte Anzahl von Schreib- und Löschzyklen besitzen. Will man Daten auf der Flash-Disk speichern, so sind diese im Pfad „/sense“ abzulegen.

SD-Karte

Jeder SENSE-Knoten besitzt einen Anschluss für eine SD-Karte. Jedoch konnte kein Test durchgeführt werden, da die Schnittstelle nicht erfolgreich in Betrieb genommen werden konnte.

USB-Stick

Nebenbei besitzt jeder SENSE-Knoten noch einen USB-Anschluss, mit dem USB-Clients angesprochen werden können. Um auf den Stick zugreifen zu können, sind jedoch folgende Schritte notwendig:

1. USB-Stick mit FAT oder FAT32 formatieren.
2. USB-Stick einstecken, SENSE-Knoten mit Spannung versorgen.
3. Mit einer Konsole zum SENSE-Knoten verbinden.
4. Mit dem Befehl **insmod /lib/modules/2.6.22.6/kernel/drivers/usb/host/ehci-hcd.ko** wird das USB Host Controller Modul geladen. Daraufhin wird auch das USB Mass Storage Device geladen und der USB-Stick wird unter den verfügbaren Geräten im Verzeichnis /dev/… hinzugefügt.
5. Mit dem Befehl **dmesg** kann man herausfinden, wie das hinzugefügte Gerät heißt (sda, sda1 oder so ähnlich).
6. Das hinzugekommene USB-Stick muss mit dem Befehl **mount /dev/<Gerätename aus Punkt 5)> /<Einhängepfad des USB-Sticks>** in den Verzeichnisbaum hinzugefügt werden.
7. Der USB-Stick ist nun unter dem Pfad <Einhängepfad des USB-Sticks> verfügbar.

7.4 Videostreaming

Die Basis für die Implementierung der Videoübertragung ist die Portierung von Live 555 Media Streaming auf die vorliegende Plattform. Aufbauend auf dem quelloffenen Programmcode wurde dieser modifiziert und in die VCSU eingebunden.

7.4.1 Portierung von Live 555 Media Streaming

Die Voraussetzung für die Portierung von Live 555 Media Streaming ist eine OpenSUSE 10.2 Installation auf einem PC. Diese ist unter [8] frei beziehbar. Aufbauend auf dieser wird das Board Support Package für dem i.MX31 installiert, welches unter [9] erhältlich ist. Details für die Installation dieses Board Support Package sind in [LLS+08] und [5] zu finden. Aufbauend auf dieser Installation wurde die Portierung durchgeführt. Der Programmcode für Live 555 Media Streaming wurde unter [4] bezogen. Zum Zeitpunkt der Portierung stand die Version 2009.06.02 zur Verfügung, welche auch verwendet wurde. Im Folgenden werden die einzelnen Schritte für die Portierung näher beschrieben.

1. Herunterladen von live555.2009.06.02.tar.gz.
2. Konsole öffnen und zu dem Speicherort von live555.2009.06.02.tar.gz wechseln.
3. Entpacken der Datei live555.2009.06.02.tar.gz in das <BASISVERZEICHNIS> mit dem Befehl **tar –zxvf live555.2009.06.02.tar.gz**.
4. In das <BASISVERZEICHNIS>/live/ wechseln.
5. Dieser Schritt umfasst die Konfiguration der makefiles, die für das Cross-Kompilieren benötigt werden. Dafür werden bereits mehrere Zielplattformen vordefiniert, darunter auch config.armlinux. Um eine erfolgreiche Portierung zu ermöglichen, müssen folgende Änderungen in der Datei config.armlinux durchgeführt werden:

- CROSS_COMPILE=arm-elf- ersetzen durch CROSS_COMPILE=arm-linux-
- LINK= $(CROSS_COMPILE)gcc -o ersetzen durch LINK= $(CROSS_COMPILE) g++ -o

6. Erstellung der makefiles mit dem Befehl **./genMakefiles armlinux**.
7. Kompilieren des Programmcodes mit dem Befehl **make**.
8. Die Portierung ist abgeschlossen, sobald die Kompilierung erfolgreich beendet wurde.

7.4.2 ByteStreamFileSource

Die Klasse `ByteStreamFileSource` ist ursprünglich dafür konzipiert worden, Medien, welche in einer Datei gespeichert sind, zu streamen. Da dies mit der Ringpufferdatei nicht direkt möglich ist, wurde die Klasse soweit umgeschrieben, dass diese über die Funktion `read_mp4_block()` der Klasse `CCircularBuffer` auf den Ringpuffer zugreifen kann.

7.4.3 Streaming-Thread

Die Funktionalitäten rund um Live 555 Media Streaming werden im Streaming-Thread ausgeführt. Dieser Thread wird in der Datei streaming.cpp durch die Funktion `pthread_create()` erstellt [SGD09, S. 270]. Die Abarbeitungen der Funktionen erfolgt innerhalb von Live 555 Media Streaming über eine Instanz der Klasse `BasicTaskScheduler`. Durch die Funktionen dieser Klasse kann man Funktionen anmelden und die zeitliche Abarbeitung festlegen. Basierend auf diesem Abarbeitungsgerüst wird ein RTSP-Server durch die Klasse `RTSPServer` instanziiert. Dem Server wird dabei eine Server-Media-Session hinzugefügt, welche es erlaubt, MPEG-4-Video-Streams zu übertragen. Schlussendlich wird im Streaming-Thread der Task-Scheduler gestartet, welche den Server ausführt.

7.5 Kommunikationsprotokoll

Die Kommunikation basiert, wie bei der VSUV auch bei der VCSU, auf Protobuf. Um die Funktionen von Protobuf zu nutzen, ist es notwendig, diese auf die Zielplattform zu portieren. Basierend auf der Portierung wurde die in Kapitel 6.2 entwickelte Nachrichtenstruktur verwendet, um die Protobuf-Klassen zu generieren. Diese Klassen werden in Folge innerhalb der Klasse `CCommunication` verwendet, um eine Kommunikation zwischen VSUV und VCSU zu ermöglichen.

7.5.1 Portierung von Protocol Buffers

Ausgangspunkt für die Portierung ist der Protobuf-Programmcode, welcher unter [6] bezogen werden kann. Dabei kommt die Version 2.3.0 in der VCSU zur Verwendung. Die folgenden Schritte zeigen die Portierung von Protobuf unter Linux.

1. Herunterladen von protobuf-2.3.0.tar.gz.
2. Konsole öffnen und zu dem Speicherort von protobuf-2.3.0.tar.gz navigieren.

3. Entpacken der Datei protobuf-2.3.0.tar.gz in das <BASISVERZEICHNIS> mit dem Befehl **tar –zxvf protobuf-2.3.0.tar.gz**.
4. In das <BASISVERZEICHNIS>/protobuf-2.3.0/ wechseln.
5. Ausführen des Befehls **./configure --prefix=/home/temp/install/**.
6. Ausführen des Befehls **make**.
7. Ausführen des Befehls **make install**.
8. Ausführen des Befehls **make distclean**.
9. Ausführen des Befehls **./configure --host=arm-linux --prefix=/home/temp/install/ARM --with-protoc=/home/temp/install/bin/protoc**.
10. Ausführen des Befehls **make**.
11. Ausführen des Befehls **make install**.
12. Die Portierung ist abgeschlossen, sobald die Kompilierung erfolgreich beendet wurde.

7.5.2 CCommunication

Die Klasse `CCommunication` ermöglicht das Senden und Empfangen von Protobuf-Nachrichten. Die Übertragung der Daten erfolgt über eine TCP/IP-Verbindung. Für die Verwaltung der Funktionen wird wiederrum auf die Funktionen der Klasse `CSocket` zurückgegriffen. Innerhalb der Klasse `CCommunication` erfüllt ein Thread den Verbindungsaufbau sowie ein weiterer Thread den Empfang und die Verarbeitung der Nachrichten.

```
class CCommunication
{
 private:
    CSocket* mysocket;
    char recv_buffer[BUFFER_SIZE];
    …
 public:
    CCommunication (char* result, unsigned short port);
    char write_protobuf_message (mMessage* message);
    char read_protobuf_message (mMessage* message);
    …
};
```

CCommunication (char* result, unsigned short port)

Der Konstruktor der Klasse `CCommunication` erstellt eine Instanz der Klasse `CSocket`. Aufbauend auf dieser Instanz wird der Socket als Server konfiguriert, um eingehende Client-Verbindungen entgegenzunehmen (siehe Abbildung 44).

Da bestimmte Socket-Funktionen ein blockierendes Verhalten besitzen, werden diese Funktionen in einem eigenen Thread ausgeführt [Sch00, S. 907]. Dieses blockierende Verhalten besitzt unter anderem die Funktion `socket_accept_connection()` aus der Klasse `CSocket`. Deswegen wird diese Funktion innerhalb des Threads `accept_thread` ausgeführt. Innerhalb dieses Threads wird an einem Port auf eingehende Verbindungen gewartet. Die Port-Nummer wird dabei durch den Parameter `port` festgelegt.
Konnte eine Verbindung aufgebaut werden, so wird der Thread `recv_thread` mit

`pthread_create()` erzeugt und der Thread `accept_thread` beendet [SGD09, S. 270]. Der Thread `recv_thread` liest die Daten über die Instanz der Klasse `CSocket` ein. Konnten Daten empfangen werden, so wird versucht, diese mit Hilfe der Protobuf-Funktion `ParseFromArray()` zu deserialisieren. Konnte eine Nachricht erfolgreich deserialisiert werden, wird diese in den Nachrichtenpuffer `recv_buffer` gespeichert.

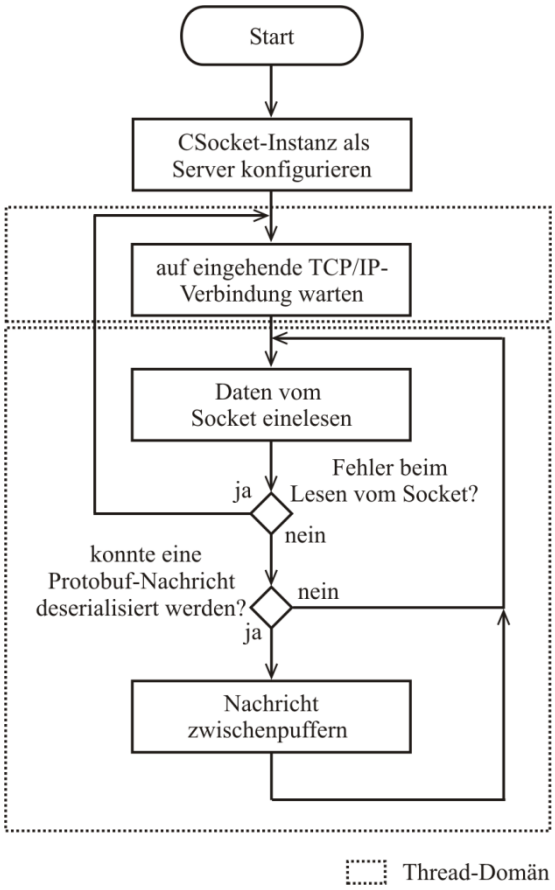


Abbildung 44: Flussdiagramm von CCommunication

char write_protobuf_message (mMessage* message)

Diese Funktion ermöglicht das Senden von Protobuf-Nachrichten. Die Protobuf-Nachricht wird innerhalb der Funktion mit `SerializeToArray()` serialisiert und über die Kommunikationsverbindung versendet. Der Parameter `message` ist dabei ein Zeiger auf eine Protobuf-Nachricht. Die Funktion gibt als Rückgabewert die Anzahl der gesendeten Bytes zurück.

char read_protobuf_message (mMessage* message)

Die Funktion `read_protobuf_message()` ermöglicht das Auslesen von Protobuf-Nachrichten aus dem klasseninternen Nachrichtenpuffer, in den der Thread `recv_thread` die empfangen Protobuf-Nachrichten schreibt. Durch den Rückgabewert kann festgestellt werden, ob eine Nachricht gelesen werden konnte.

7.5.3 CSocket

Die Klasse `CSocket` kapselt sämtlichen Funktionen für den Zugriff auf die Socket-Funktionen von Linux. Die Socket-Funktionen haben in ihrer Standardkonfiguration ein Prozess-blockierendes Verhalten. Dieses Verhalten hat den Vorteil, dass keine Rechenleistung durch zyklisches Abfragen verschwenden wird. Jedoch sollten diese Funktionen in einem eigenen Prozess oder Thread ausgeführt werden, damit die Anwendung andere Funktionen nicht blockiert [Sch00, S. 907]. Dabei kann eine Instanz entweder als Client oder Server konfiguriert werden. So wird die Client-Konfiguration in der Klasse `CYUVVideoTCPSource` verwendet, um auf das EVS zuzugreifen. Die Konfiguration als Server wird hingegen in der Klasse `CCommunication` verwendet.

```
class CSocket
{
 private:
    int client_socket;
    int server_socket;
    …
 public:
    char socket_as_client (char* ip_address, unsigned int port);
    char socket_as_server (unsigned int port);
    char socket_accept_connection ();
    ssize_t socket_recv (void* buf, ssize_t len);
    ssize_t socket_send (const void* buf, ssize_t len);
    void disconnect_all ();
    …
};
```

char socket_as_client (char* ip_address, unsigned int port)

Durch diese Funktion wird die Instanz der Klasse `CSocket` als Client konfiguriert und eine Verbindung zum Server aufgebaut. Dazu wird mit der Funktion `socket()` ein neuer Socket eingerichtet. Die durch die Funktion zurückgegebene Identifikationsnummer wird in der Variable `client_socket` gespeichert. Mit den Parameter `ip_address` und `port` wird über die Funktion `connect()` eine Verbindung zum Server hergestellt [Pol04, S. 47-53]. Ein Rückgabewert von nicht 0 signalisiert, wenn die Funktion nicht erfolgreich abgeschlossen werden konnte.

char socket_as_server (unsigned int port)

Die Funktion `socket_as_server()` konfiguriert die Instanz der Klasse `CSocket` als Server. Dazu wird mit der Funktion `socket()` ein neuer Socket eingerichtet. Die durch die Funktion zurückgegebene Identifikationsnummer wird in der Variable `server_socket` gespeichert. Der Parameter `port` gibt der Funktion `bind()` an, auf welchen Port der Server auf eingehende Verbindungen warten soll. Mit der Funktion `listen()` wird es dem Client ermöglicht, Verbindungen zum Server aufzubauen [Pol04, S. 47-53, 94]. Konnte die Funktion ordnungsgemäß ausgeführt werden, so gibt die Funktion eine null zurück.

char socket_accept_connection ()

Diese Funktion wird aufgerufen, sobald die Instanz von `CSocket` als Server konfiguriert wurde und auf eine eingehende Verbindung gewartet werden soll. Durch den Aufruf der Funktion `accept()` wird das Programm an dieser Stelle solange blockiert, bis eine Verbindung zu einem Client aufgebaut werden konnte [Pol04, S. 94]. Die von der Funktion zurückgegebene Identifikationsnummer wird in der Variable `client_socket` gespeichert. Bei einem erfolgreichen Verbindungsaufbau gibt die Funktion eine 0 zurück.

ssize_t socket_send (const void* buf, ssize_t len)

Die Funktion `socket_send()` erlaubt es beliebige Daten über eine TCP/IP-Verbindung zu schicken. Der Parameter `buf` zeigt auf einen Speicherbereich, der die zu sendenden Daten enthält. Die Anzahl der zu sendenden Bytes wird durch `len` angegeben. Das Senden der Daten erfolgt mit der Funktion `send()` [Pol04, S. 204]. Der Rückgabewert der Funktion gibt die Anzahl der tatsächlich gesendeten Bytes zurück.

ssize_t socket_recv (void* buf, ssize_t len)

Die Funktionalität, um Daten zu empfangen, wird durch die Funktion `socket_recv()` realisiert. Dazu wird die Funktion `recv()` verwendet, welche solange wartet, bis `len` Bytes empfangen wurden [Pol04, S. 201]. Die empfangen Bytes werden dabei auf den Speicherbereich, der durch den Zeiger `buf` festgelegt wird, geschrieben. Der Rückgabewert der Funktion gibt die Anzahl der empfangenen Bytes zurück.

void disconnect_all ()

Mit der Funktion `disconnect_all()` werden sämtliche verwendete Socket-Verbindungen mit der Funktion `close()` abgebaut.

8. Ergebnisse, mögliche Erweiterungen und Ausblick

Zu Beginn dieses Kapitels kann gesagt werden, dass alle Anforderungen, welche in dieser Arbeit gestellt wurden, zur Gänze erfüllt werden konnten. Um den Kreis zu schließen, werden in diesem Kapitel alle wichtigen Ergebnisse in zusammengefasster Form präsentiert. Neben den Ergebnissen werden einige Verbesserungsmöglichkeiten für eine zukünftige Version dieser Software aufgezeigt. Die Erweiterungsmöglichkeiten sowie der Ausblick schließen diese Arbeit ab.

8.1 Ergebnisse

Durch die Implementierung der am Beginn der Arbeit beschriebenen Anforderungen konnten weitere Funktionalitäten dem SENSE-Projekt erfolgreich hinzugefügt werden. Die Ergebnisse werden in den folgenden Unterkapiteln näher beschrieben.

8.1.1 Videokomprimierung

Die Videokamera des EVS verfügt über eine maximale Auflösung von 640 x 480 Pixel. Die Kompression mit dem MPEG-4-Encoder bei dieser Videobildauflösung und einer Videobildrate von 30 FPS würde eine Datenrate von 4000 kBit/s hervorrufen. Jedoch wäre es bei dieser Datenrate nicht möglich, eine Echtzeitübertragung mittels Funkverbindung durchzuführen. Die Datenrate des MPEG-4-Encoders könnte jedoch unter 1000 kBit/s liegen, wenn durch die Verringerung der Videobildrate auf 7 FPS die Datenrate des Video-Streams ebenfalls sinkt. Jedoch würde damit ein Großteil der Bandbreite der Funkmodule benötigt werden und somit keine Bandbreite für den Nachrichtenaustausch zwischen den Knoten zur Verfügung stehen. Um die Bandbreite der Funkmodule nicht voll auszulasten, wurde die Videobildauflösung von 640 x 480 Pixel auf 320 x 240 Pixel reduziert. Bei dieser Auflösung ist es weiterhin möglich Alarme, beziehungsweise Fehlalarme problemlos zu erkennen.

Für die Videokomprimierung wurde der Encoder mit folgenden Parametern initialisiert:

- Auflösung: 320 x 240 Pixel
- Videobildrate: 7 FPS
- Profile & Level: MPEG4_SIMPLE_PROFILE_LEVEL_2
- Stream-Typ: MPEG4_PLAIN_STRM

Durch die Verwendung dieser Parameter erzeugt der Encoder einen Video-Stream, welcher maximal eine Datenrate von 128 kBit/s besitzt. Mit dieser Datenrate ist es nun einerseits möglich, einen Video-Stream durch die Funkmodule zu übertragen und andererseits gibt es genügend Bandbreitenreserve für die Übertragung von anderen Daten.

8.1.2 Videospeicherung

Mit einer maximalen Video-Stream-Datenrate von 128 kBit/s wird für die Speicherung einer Videosequenz mit der Dauer von 1 Minute 960 kByte benötigt. Dieser Wert konnte bei praktischen Tests bestätigt werden. Um ein Video mit mindestens 25 Minuten speichern zu können, wird somit ungefähr 25 MByte an Speicherplatz benötigt. Dadurch ist die Speicherung auf der RAM-Disk nicht möglich, da diese nur eine Speichergröße von 16 MByte besitzt. Die Speicherung der Ringpuffer-Datei muss somit entweder im NAND-Flash-Speicher oder auf einen USB-Massenspeicher durchgeführt werden.
Dadurch, dass bei der Implementierung des Ringpuffers 32-Bit-Variablen verwendet wurden, ist es möglich, Videos mit einer Größe von über 4 GByte zu speichern. Dies entspricht bei den verwendeten Videobildeigenschaften einer Speicherdauer von circa 3 Tagen.

8.1.3 Video-Streaming

Das Streamen von Videos mittels RTP ist sowohl über die 100 MBit/s Ethernet-Verbindung möglich als auch über die 1 MBit/s Funkverbindung. Dabei wurde jedoch nur ein einzelner Videostream, von einem per Funkknoten angebunden SENSE-Konten, übertragen. Durch die im Vorfeld bekannte Übertragungsinstabilität der Funkknoten konnte jedoch keine dauerhaft stabile Videoübertragung erreicht werden. Hingegen ist das Streamen von Videos über Ethernet über längere Zeit möglich.
Charakteristische Werte bei der Videoübertragung durch Live 555 Media Streaming und der Wiedergabe mit dem VLC Media Player sind nun Folgende.
Beim Anfordern von Video-Streams durch die VSUV dauert es ungefähr 3 – 5 s, bis ein Videobild angezeigt wird. Bei der Anzeige eines Live-Video-Streams kommt es zu einer Verzögerung von ungefähr 5 s, bis die durch die Kamera aufgenommenen Videobilder im Anzeigefenster der VUSV angezeigt werden. Die Datenrate bei dieser Übertragung bewegt sich im Bereich von 82 - 176 kBit/s und wurde durch den VLC Media Player ermittelt.

8.1.4 Prozessorauslastung/Programmgröße

Eine Anforderung an die Software im SENSE-Knoten ist, eine möglichst geringe Prozessorauslastung zu erreichen. Die Resultate sind in der Tabelle 8 zu sehen, wobei die Prozessorauslastungen bei verschiedenen Szenarien bestimmt wurden. Die Auslastungswerte wurden unter Linux mit dem Konsolen-Kommando „top“ ermittelt.
Wie erwartet, ist zu erkennen, dass die Videobildverarbeitung durch die Videopipeline am rechenintensivsten ist und über eine längere Zeit betrachtet gleichbleibend bei circa 6 % liegt. Hingegen kommt es beim Video-Streaming, vor allem beim Aufbau einer RTP-Sitzung, zu kleineren Auslastungsspitzen. Die Auslastung des VCSU-Prozesses liegt im Bereich von 6,5 bis 8,5 %. Damit ist

sichergestellt, dass die Prozessorauslastung der VCSU nur eine geringe Einschränkung für die gleichzeitig arbeitende RDU bedeutet. Die Programmgröße der VCSU besitzt schlussendlich eine Gesamtgröße von unter 5,5 MByte und findet damit im Flash-Speicher leicht Platz.

Tabelle 8: Prozessorauslastung

Szenario	Prozessorauslastung
Leerlauf	< 1%
Videopipeline	~ 6%
Video-Streaming	< 2%
Videopipeline & Video-Streaming	< 9%

8.2 Verbesserungsvorschläge

- ➔ Durch die Verwendung von Protobuf konnte schnell ein Kommunikationsprotokoll realisiert werden, jedoch zeigten sich bei den ersten Softwaretests Probleme. Zwar konnte der Empfang von Protobuf-Nachrichten durch die VSUV deutlich verbessert werden, jedoch wurden immer wieder Nachrichten falsch deserialisiert. Eine Optimierung der Kommunikation könnte so aussehen, dass entweder die Einbindung von Protobuf weiter optimiert wird oder das Protobuf durch ein anderes Protokoll ersetzt wird.
- ➔ Die teilweise stark schwankende Videobildrate führt dazu, dass durch die Videobildratenstabilisierung redundante Videobildinformationen erzeugt werden. Diese redundanten Videobildinformationen belegen dabei unnötig Speicherplatz im Ringpuffer beziehungsweise benötigen zusätzliche Bandbreite bei der Übertragung. Eine Lösung für dieses Problem wäre, dem Reasoning-Board einen direkten Zugriff auf die Videokamerabilder zu ermöglichen. Damit müssten keine redundanten Videobilddaten mehr generiert werden. Durch die direkte Anbindung des Reasoning-Boards an die Kamera wäre es auch möglich, eine höhere Videobildrate zu verwenden, was dazu führt, dass eine flüssigere Wiedergabe möglich wäre.
- ➔ Bei RTP muss man in Kauf nehmen, dass es zu Übertragungsverzögerungen kommt. Die Gründe dafür liegen daran, dass die Daten sowohl am Server als auch am Client jeweils gepuffert werden. Um diese Verzögerungen zu minimieren, müsste der frei zugängliche Programmcode von Live 555 Streaming Media und des VLC Media Player optimiert werden, indem die Pufferung der Videodaten auf ein Minimum reduziert wird.

8.3 Erweiterungsmöglichkeiten

- ➔ Eine Erweiterungsmöglichkeit des Systems könnte so aussehen, dass des dem Benutzer ermöglicht wird, die SENSE-Knoten über ein Tablett oder Smartphone zu verwalten. Dafür wäre die Entwicklung eines zusätzlichen Programmes notwendig, welches auf das verwendete Betriebssystem optimiert ist. Durch die Verwendung von MPEG-4 und RTP sollte eine Entwicklung eines solchen Programmes keine größeren Probleme verursachen. Vor allem

deswegen, weil für alle derzeit gängigen Betriebssysteme eine Portierung des VLC Media Players zur Verfügung steht.

- ➔ Die Verwendung von RTP würde es erlauben, Video-Streams per Broadcasts zu streamen. Damit besteht die Möglichkeit, bei einem Alarm, sämtliche Benutzer mit den gleichen Live-Videobildern beziehungsweise einer Videosequenz aus einer vergangenen Aufzeichnung per Broadcast an eine bestimmte Gruppe von Benutzer zu streamen. Voraussetzung dafür ist jedoch, dass die Bitübertragungsschicht Broadcasts unterstützt. Diese Eigenschaft erfüllt zwar Ethernet, jedoch müsste untersucht werden, ob die Funkmodule Broadcasts erlauben.
- ➔ Als Kommunikationsinfrastruktur kommt im SENSE-Projekt Ethernet als auch ein Funkkommunikationssystem basierend auf NanoNet zum Einsatz. Bei der Entwicklung der Software wurde darauf geachtet, dass eine Videobildübertragung von jedem SENSE-Knoten möglich ist. Um dies zu ermöglichen, wurde die Videobildauflösung verringert, damit die Funkverbindungen genügend Bandbreitenreserve besitzen. Andererseits wäre diese Reduzierung der Videobildauflösung bei den SENSE-Knoten, welche eine Ethernet-Verbindung mit 100 MBit/s besitzen, nicht notwendig. Eine Verbesserungsmöglichkeit wäre, eine adaptive Anpassung der Videobildauflösung an die verfügbare Bandbreite durchzuführen. Um dies zu realisieren, müsste jedoch eine effizientere JPEG zu YUV Transcodierung implementiert werden. Dadurch könnten die Videobilddaten mithilfe des MPEG-4-Encoders komprimiert werden und in Folge mittels RTP an den Client gesendet werden. Eine andere Möglichkeit wäre, die einzelnen JPEG-Bilder mittels RTP an den Client zu übertragen.
- ➔ Zu Beginn des SENSE-Projektes wurde bewusst auf die Verwendung für WLAN verzichtet, da zu diesem Zeitpunkt WLAN als zu unsicher eingestuft wurde. Deswegen wurde ein proprietäres, kabelloses Funksystem entwickelt. Jedoch besitzt dies den Nachteil, dass es nur eine maximale Datenrate von 1 MBit/s besitzt. Die Erweiterungsmöglichkeit, welche in diesem Punkt aufgezeigt wird, könnte so aussehen, dass die bestehenden Funkmodule durch leistungsfähigere Funkmodule ersetzt werden. Dabei könnte auch durchaus die Verwendung von WLAN herangezogen werden, da der anfangs erwähnte Sicherheitsaspekt heute nicht mehr gültig ist.
- ➔ Beim Design der VCSU wurde darauf geachtet, die Videopipeline flexibel zu gestalten, um eventuell weitere bildverarbeitende Klassen zu integrieren. So wäre es möglich, eine Klasse zu implementieren, welche erlaubt, Informationen über das eigentliche Videobild zu legen. Dadurch wäre es möglich, dem Benutzer Informationen, in Form von Texten oder einfache geometrische Grundformen wie Rechtecke und Kreise, einzublenden. Die einfachste Möglichkeit, dies zu implementieren, wäre, die einzublendenden Informationen im YUV-Bildpuffer einfließen zu lassen.
- ➔ Um den MPEG-4-Video-Stream fehlertoleranter gegenüber Übertragungsfehlern zu machen, wäre es zu empfehlen, den Encoder auf einen anderen Stream-Typ zu konfigurieren, wie zum Beispiel MPEG4_VP_DP_RVLC_STRM. Die verwendete Version der VLC Media Players verweigerte zwar die Wiedergabe des MPEG-4-Stream mit diesem Typ, jedoch wäre zu prüfen, ob eine neuere Version des VLC Media Players doch in der Lage wäre, diesen Stream-Typ wiederzugeben.

8.4 Ausblick

Im SENSE-Projekt werden mehrere technische Bereiche behandelt, denen einerseits wissenschaftlich große Aufmerksamkeit gewidmet werden, andererseits diese Bereiche auch in der Industrie und in Endverbraucherprodukten immer mehr Einzug finden. Einer dieser Bereiche ist Bildverarbeitung, welche in den letzten Jahren starke Umsatzzuwächse aufweist. Die Gründe dafür sind unter anderem darin zu finden, dass die Prozessoren immer leistungsfähiger werden. Diese Leistungsfähigkeit ist notwendig, um die Berechnung der Algorithmen durchzuführen. Dabei ist zu beobachten, dass der Trend bei den Prozessoren im Embedded-Bereich bereits in Richtung Quad-Core Prozessoren geht, welche einen Takt von über 1 GHz aufweisen. Da in der Bildverarbeitung oft parallel Berechnungen durchgeführt werden müssen, kann die Mehrkernarchitektur meist einen enormen Zuwachs an Rechengeschwindigkeit bewirken.
Ein zweiter wichtiger Bereich, welcher ebenfalls im SENSE-Projekt behandelt wird, ist das Gebiet der vernetzen, intelligenten Sensornetzwerke. Dieser Bereich ist vor allem aus wissenschaftlicher Sicht sehr interessant. Zwar kann man schon überall die steigende Vernetzung der Geräte beobachten, jedoch fehlt es bei solchen Systemen am intelligenten Verhalten.
Das entwickelte intelligente Überwachungssystem im SENSE-Projekt könnte ein erster Schritt sein, welches für die Gebäudeüberwachung eingesetzt werden kann. Das Problem dabei ist noch, dass die entwickelten Algorithmen noch nicht mit dem Auffassungsvermögen des Menschen mithalten können. Mit der Implementierung der VCSU und VSUV konnte somit ein weiterer Grundstein für die Entwicklung einer zentralen Visualisierungs- und Verwaltungssoftware gelegt werden. Durch diese Erweiterung würde die Verwaltung von einer größeren Anzahl von SENSE-Knoten ermöglicht werden. Der Benutzer dieser Software könnte dabei sämtliche Informationen der SENSE-Knoten dargestellt bekommen, angefangen von den verschiedenen Alarmen bis hin zu Video-Streams [Bru10, S. 5]. Durch die Möglichkeit, Video-Streams betrachten zu können, steht nun eine effektive Möglichkeit bereit, die für das Auslösen eines Alarmes verursachenden Videobilddaten zu betrachten. Damit können vorhandene und neu entwickelte Algorithmen im Bereich der Bildverarbeitung sowie künstlichen Intelligenz verbessert werden. Neben einer zentralen Verwaltungssoftware wäre auch eine Software für mobile Endgeräte wie Tablets oder Mobiltelefonen möglich. Damit könnte das Sicherheitspersonal in Echtzeit mit Informationen über Alarme versorgt werden und somit schnellstmöglich auf Gefahren aufmerksam gemacht werden.

Literatur

[Ado09a] Adobe Systems Incorporated: *Real Time Messaging Protocol (RTMP) Message Formats*, 2009.

[Ado09b] Adobe Systems Incorporated: *Real Time Messaging Protocol Chunk Stream*, 2009.

[Ado09c] Adobe Systems Incorporated: *RTMP Commands Messages*, 2009.

[AAC+10] Aravind V., Archan Pratap Mishra, Ch. Arjun Kumar Reddy, Sneha S., Sumit Kishore: *Enhancement of Live555 Media Server to support MPEG4 streaming*, International Institute of Information Technology Bangalore, 2010.

[Bru10] Bruckner D.: *Implementation of Algorithms for User Communication/Diagnosis - Final Version*, Confidential Project Report, 2010.

[BH10] Bruckner D., Herzner W.: *Publishable Activity Report*, Confidential Project Report, 2010.

[BZP+09] Bruckner D., Zucker G, Picus C., Cambrini L., Herzner W.: *Implementation of Information Exchange Algorithms - Final Version*, Confidential Project Report, 2009.

[ES98] Effelsberg W., Steinmetz R.: *Video Compression Techniques*, dpunkt, 1998.

[Fre07] Freescale Semiconductor Incorporated: *i.MX31 Fact Sheet*, 2007.

[Fre09] Freescale Semiconductor Incorporated: *i.MX31 PDK 1.5 Linux – Reference Manual*, 2009.

[Haf99] Hafner W.: *Segmentierung von Video-Bildfolgen durch Adaptive Farbklassifikation*, Herbert Utz Verlag, 1999.

[Han04] Hantro Products Oy: *Application Programming Interface 3.1 - MPEG-4/H.263 VGA Encoder 5250*, 2004.

[HCS07] Hanzo L. L., Cherriman P., Streit J.: *Video Compression and Communications: From Basics to H.261, H.263, H.264, MPEG4 for DVB and HSDPA-Style Adaptive Turbo-Transceivers*, John Wiley & Sons, Second Edition, 2007.

[ISO04] ISO/IEC 14496-2: Information *technology - Coding of audio-visual objects - Part 2: Visual*, ISO, Third Edition, 2004.

[ISO09] ISO/IEC 14496-10: *Information technology - Coding of audio-visual objects - Part 10: Advanced Video Coding*, ISO, Fifth Edition, 2009.

[ITU05] ITU: *Video coding for low bit rate communication (ITU-T Recommendation H.263)*, International Telecommunications Union, 2005.

[ITU10] ITU: *Advanced video coding for generic audiovisual services (ITU-T Recommendation H.264)*, International Telecommunications Union, 2010.

[ITU93] ITU: *Video Codec for Audiovisual Services at p × 64 kbits (ITU-T Recommendation H.261)*, International Telecommunications Union, 1993.

[KS03] Klima R. Selberherr S.: *Programmieren in C*, Springer Wien/NewYork, 2003.

[KS09] Kaumanns R., Sigenheim V.: *Die Google-Ökonomie – Wie der Gigant das Internet beherrschen will*, Books on Deman, 2009.

[Kof08] Kofler M.: *Linux – Installation, Konfiguration, Anwendung*, Addison-Wesley, 8. Auflage, 2008.

[KLW08] Köck J., Linder T., Wernig M.: *Alarm Definition Framework for an Airport Security System*, Technische Universität Wien, Institut für Computertechnik, Computertechnik Vertiefung, 2008.

[Kün01] Künkel T.: *Streaming Media – Technologien, Standards, Anwendungen*, Addison-Wesley, 2001.

[Lon11] Longolius N.: *Web-TV – AV-Streaming im Internet – Echtzeitübertragung von Ton & Bild im Internet*, O'Reilly Verlag, 2011.

[LLS+08] Lassl C., Leder N., Schmaus M., Tschida B., Winkler D., Zeman C.: *Videoapplikationen für einen Mikroprozessor*, Technische Universität Wien, Institut für Computertechnik, Bachelorarbeit, 2008.

[LX08] Liberty J., Xie D.: *Programming C# 3.0*, O'Reilly Media, Fifth Edition, 2008.

[MDS96] Musser D. R., Derge J. G., Saini A.: *STL Tutorial and Reference Guide – C++ Programming with the Standard Template Library*, Addison-Wesley, 1996.

[Mil95] Milde T.: *Videokompressionsverfahren im Vergleich – JPEG, MPEG, H.261, XCCC, Wavelets, Fraktale*, dpunkt Verlag, 1995.

[PE02] Pereira F., Ebrahimi T.: *The MPEG-4 Book*, Prentice Hall PTR, 2002.

[Per03] Perkins C.: *RTP - Audio and Video for the Internet*, Addison-Wesley, 2003.

[Pol04] Pollakowski M.: *Grundkurs Socketprogrammierung mit C unter Linux – So entwickeln Sie schlanke Web-Applikationen*, Friedr. Vieweg & Sohn Verlag/GWV Fachverlage Gmbh, 2004.

[Ric03] Richardson E. G. I.: *H.264 and MPEG-4 Video Compression – Video Coding for Next-generation Multimedia*, John Wiley & Sons, 2003.

[SC03] Schulzrinne H, Casner S.: RTP Profile for Audio and Video Conferences with Minimal Control (IETF RFC 3551), The Internet Society, 2003.

[Sch00] Scheibl H. J.: *Visual C++ 6.0 für Einsteiger und Fortgeschrittene*, Carl Hansa Verlag München Wien, 2000.

[Sch05] Schwichtenberg H.: *Microsoft .net 2.0 Crashkurs – Schnelleinstieg in neue Technologien und Tools*, Microsoft Press Deutschland, 2005.

[SCF+03] Schulzrinne H., Casner S., Frederick R., Jacobson V.: RTP: A Transport Protocol for Real-Time Applications (IETF RFC 3550), The Internet Society, 2003.

[SGD09] Schröder J., Gockel T., Dillmann R.: *Embedded Linux – Das Praxisbuch*, Springer Dordrecht Heidelberg London New York, 2009.

[SRL98] Schulzrinne H., Rao A., Lanphier R.: Real Time Streaming Protocol (IETF RFC 2326), The Internet Society, 1998.

[Sta04] Stallings W.: *Information Operating Systems - Internals and Design Principles*, Prentice Hall International, Fifth Edition , 2004.

[Str09] Strutz T.: *Bilddatenkompression - Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264*, Vieweg+Teubner, 4. Auflage, 2009.

[Sym04] Symes P.: *Digital Video Compression*, The Mc Graw Hill Companies Inc., 2004.

[TH07] Teich J., Haubelt C.: *Digitale Hardware/Software-Systeme: Synthese und Optimierung*, Springer-Verlag Berlin Heidelberg, 2. Auflage, 2007.

[TK96] Torres L., Kunt M.:*Video Coding – The second generation approach*, Kluwer Academic Publisher Boston, 1996.

[WWW99] Wall K., Watson M., Whitis M.: *Linux Programming*, Sams, 1999.

Internet Referenzen

[1] www.ffmpeg.org
Projekthomepage von FFmpeg
[aufgerufen am 17.09.2009]

[2] www.saillard.org
Projekthomepage von Tiny JPEG Decoder
[aufgerufen am 10.09.2010]

[3] www.gstreamer.net
Projekthomepage von GStreamer
[aufgerufen am 12.09.2009]

[4] www.live555.com/liveMedia
Projekthomepage von Live 555 Streaming Media
[aufgerufen am 07.06.2009]

[5] www.bluetechnix.com
Homepage von Bluetechnix Mechatronische Systeme GmbH
[aufgerufen am 11.08.2011]

[6] code.google.com/p/protobuf/
Projekthomepage von Google Protocol Buffers
[aufgerufen am 13.03.2010]

[7] code.google.com/p/protobuf-net/
Projekthomepage von Protobuf-NET
[aufgerufen am 15.05.2010]

[8] software.opensuse.org
Homepage von OpenSUSE
[aufgerufen am 11.08.2011]

[9] www.freescale.com
Homepage von Freescale Semiconductor Inc.
[aufgerufen am 11.08.2011]

[10] www.videolan.org/vlc/
Projekthomepage der VideoLAN Organization

[aufgerufen am 17.09.2009]

[11] msdn.microsoft.com/en-us/express/future/bb421473
Homepage von Visual Studio 2008 Express Edition
[aufgerufen am 15.09.2009]

[12] www.adobe.com
Homepage von Adobe Systems Inc.
[aufgerufen am 25.01.2012]

[13] www.adobe.com/de/flashplatform
Homepage von Adobe Flash Plattform
[aufgerufen am 25.01.2012]

[14] www.adobe.com/de/products/flashmediaserver
Homepage von Adobe Flash Media Server
[aufgerufen am 25.01.2012]

[15] www.adobe.com/de/products/flashplayer
Homepage von Adobe Flash Player
[aufgerufen am 25.01.2012]

[16] www.analog.com
Homepage von Analog Devices Inc.
[aufgerufen am 25.01.2012]

Anhang: Nachrichtenstruktur

```
message mMessage
{
	required eMessageType mM_Type = 1 [default = EMT_UNKNOWN];
	required fixed32 mM_NodeID = 2;
	optional string mM_TS = 3;
	optional mAlarm mM_Alarm = 4;
	optional mInfo mM_Info = 5;
	optional mDebug mM_Debug = 6;
	optional mReasoning mM_Reasoning = 7;
	optional mVideo mM_Video = 8;
	optional mVSU_Debug mMVSU_Debug = 9;
}
enum eMessageType
{
	EMT_UNKNOWN = 0;
	eMT_Alarm = 1;
	eMT_Info = 2;
	eMT_Debug = 3;
	eMT_Reasoning = 4;
	eMT_Video = 5;
	eMT_VCSU_Debug = 6;
}
message mAlarm
{
	optional eAlarmType mA_Type = 1 [default = EAT_UNKNOWN];
	optional eAlarmSeverity mA_Severity = 2 [default = EAS_UNKNOWN];
}
message mInfo
{
	optional eInfoType mI_Type = 1 [default = EIT_UNKNOWN];
	optional string mI_Content = 2;
}
message mDebug
{
	optional eDebugType mD_Type = 1 [default = EDT_UNKNOWN];
	optional string mD_Content = 2;
}
message mReasoning
{
```

```
        required eReasoningCommand mR_Command = 1 [default = ERC_UNKNOWN];
        optional uint32 mR_ParamUInt = 2;
}
message mVideo
{
        required eVideoCommand mV_Command = 1 [default = EVC_UNKNOWN];
        optional uint32 mV_Duration = 2;
        optional string mV_Info = 3;
}
message mVSU_Debug
{
        optional bool circular_buffer_new_value = 1 [default = false];
        optional uint32 circular_buffer_oldest_position = 2;
        optional uint32 circular_buffer_next_write_position = 3;
        optional uint32 circular_buffer_current_read_position = 4;
        optional uint32 circular_buffer_frames_inside = 5;
        optional uint32 circular_buffer_size = 6;
        optional uint32 circular_buffer_free = 7;
        optional bool temporary_buffer_new_value = 8 [default = false];
        optional uint32 temporary_buffer_oldest_position = 9;
        optional uint32 temporary_buffer_next_write_position = 10;
        optional uint32 temporary_buffer_current_read_position = 11;
        optional uint32 temporary_buffer_frames_inside = 12;
        optional uint32 temporary_buffer_size = 13;
        optional uint32 temporary_buffer_free = 14;
        optional bool encoding_frame_rate_new_value = 15 [default = false];
        optional uint32 encoding_frame_rate = 16;
        optional bool streaming_frame_rate_new_value = 17 [default = false];
        optional uint32 streaming_frame_rate = 18;
        optional bool stop_videopipeline = 19;
        optional bool start_videopipeline = 20;
        optional bool reset_buffer = 21;
        optional bool quit_VCSU = 22;
}
enum eAlarmType
{
  EAT_UNKNOWN = 0;
  eAT_Bagh = 1;
  eAT_Bagl = 2;
  eAT_Carsp = 3;
  eAT_Lurkl = 4;
  eAT_Missl = 5;
  eAT_Screamh = 6;
  eAT_Screaml = 7;
  eAT_Glassh = 8;
  eAT_Glassl = 9;
  eAT_Gunh = 10;
  eAT_Gunl = 11;
  eAT_Fencerz = 12;
  eAT_Undh = 13;
  eAT_Undl = 14;
```

```
  eAT_Runningpl = 15;
  eAT_Runningph = 16;
}
enum eAlarmSeverity
{
  EAS_UNKNOWN = 0;
  eAS_Important = 1;
  eAS_Medium = 2;
  eAS_Informal = 3;
}
enum eInfoType
{
        EIT_UNKNOWN = 0;
        eIT_Emergency = 1;
        eIT_Alert = 2;
        eIT_Critical = 3;
        eIT_Error = 4;
        eIT_Warning = 5;
        eIT_Notice = 6;
        eIT_Informational = 7;
        eIT_Debug = 8;
}
enum eDebugType
{
        EDT_UNKNOWN = 0;
        eDT_Core = 1;
        eDT_VUReader = 2;
        eDT_Decoder = 3;
        eDT_Tracker = 4;
        eDT_Nodecom = 5;
        eDT_UICom = 6;
}
enum eReasoningCommand
{
        ERC_UNKNOWN = 0;
        eRC_Reset = 1;
        eRC_DeleteNeighbors = 2;
}
enum eVideoCommand
{
        EVC_UNKNOWN = 0;
        eVC_Request = 1;
        eVC_Ready = 2;
        eVC_Stop = 3;
}
```