

Institut für Informatik
Universität Hannover

Entwicklung einer Software zur Echtzeitübertragung von
Videodaten zwischen einer PCI-Videosignalprozessorkarte und
einem PC

Studienarbeit

Michael Hunold
31. Oktober 2000

Betreuer: Dipl.-Ing. Jörg Hilgenstock

Inhaltsverzeichnis

1	Einführung	3
1.1	Der Videosignalprozessor AxPe1280V	3
1.2	Die Multimedia-PCI-Bridge SAA7146A	4
1.3	Die AxPe1280V-Demonstrationskarte	5
1.4	Applikationen für die AxPe1280V-Prozessoren und den PC	6
1.5	Architektur des Treibers	7
2	Ideen für die Erweiterung des Treibers	8
3	Technische Aspekte der Erweiterung	10
3.1	Speicherbehandlung	10
3.2	Synchronisation	12
4	Der Capture-Vorgang	14
4.1	Schreiben eines Videodatenstroms in den Hauptspeicher	14
4.2	Implementierung aus der Sicht einer Applikation	14
4.3	Implementierung aus der Sicht des Treibers	16
5	Der Upload-Vorgang	19
5.1	Lesen von Videodaten aus dem Hauptspeicher	19
5.2	Synchronisation des Upload-Vorgangs	19
5.3	Implementierung aus der Sicht einer Applikation	20
5.4	Implementierung aus der Sicht des Treibers	22
6	Demonstrationssoftware „UplCap“	25
6.1	Capture-Vorgang	25
6.2	Upload-Vorgang	26
7	Ergebnisse und Ausblick	28
7.1	Integration in den bestehenden Treiber	28
7.2	Robustheit der Erweiterungen	29
7.3	Echtzeitfähigkeit und Prozessorlast	29
7.4	Einschränkungen	29
7.5	Demonstrationssoftware „UplCap“	30
8	Zusammenfassung	31
	Abbildungsverzeichnis	32
	Literaturverzeichnis	32
A	Anhang	33
A.1	Symbolische Fehlercodes	33
A.2	Funktionsaufrufe	33

1 Einführung

Die digitale Videosignalverarbeitung wird in Zukunft in immer mehr Bereichen des täglichen Lebens zu finden sein. Beispiele für eine sinnvolle Nutzung sind Bildtelefone, Videokonferenzsysteme und digitales Fernsehen.

Obwohl bereits heutige Prozessoren genug Rechenleistung bereitstellen, um Echtzeitmanipulationen an Videosignalen durchzuführen, macht es Sinn, spezielle Videosignalprozessoren für diese Aufgabe einzusetzen. Der Einsatz von für die Signalverarbeitung optimierten Prozessoren bringt Vorteile durch eine oftmals wesentlich geringere Leistungsaufnahme und die Entlastung des Hauptprozessors des Systems.

Am Laboratorium für Informationstechnologie der Universität Hannover wurde mit Blick auf die erwähnten Anwendungsgebiete der Videosignalprozessor AxPe1280V entwickelt. Um die Leistungsfähigkeit eines AxPe1280V-Multiprozessorsystems zu demonstrieren, wurde zusätzlich eine PCI-Einsteckkarte für den PC entwickelt.

Da der AxPe1280V für den Einsatz auf einer Steckkarte in einem PC alleine nicht die nötigen Anschlußmöglichkeiten besitzt, basiert die Einsteckkarte auf der Multimedia-PCI-Bridge SAA7146A von *Philips Semiconductors*, die den AxPe1280V-Prozessoren eine geeignete Infrastruktur mit einer Anbindung an den in PCs gängigen PCI-Bus bietet.

Bei der ursprünglichen Entwicklung eines Windows 98-Treibers für die AxPe1280V-Demonstrationskarte kam das Gerätetreiber-Toolkit *WinDriver* der Firma *Jungo* zum Einsatz.

In der bisherigen Form ist es nur möglich, die durch die AxPe1280V-Prozessoren erzeugten Videodaten auf dem Monitor darstellen zu lassen. Die Videodaten stehen nicht zur Weiterverarbeitung zur Verfügung. Es ist aber aus vielfältigen Gründen wünschenswert, diese Videodaten auch abspeichern zu können, z. B. für Vergleichszwecke, Archivierungen oder um daraus ein Demonstrationsvideo erstellen zu können.

Weiterhin ist der analoge Videosignaleingang die einzige Möglichkeit, Videodaten in die beiden AxPe1280V-Prozessoren einzuspeisen. Es besteht keine Möglichkeit, bereits vorliegendes, digitales Bildmaterial zu verwenden. Für die Entwicklung von Videoenkodern ist es aber notwendig, synthetische oder bereits vorgefertigte Videosequenzen zu den beiden AxPe1280V-Prozessoren transportieren zu können. Nur so ist es möglich, die Güte der darauf laufenden Programme und Algorithmen vergleichen und verbessern zu können.

Ziel dieser Studienarbeit ist die Erweiterung des bestehenden Treibers um die Möglichkeit, die Videodaten der AxPe1280V-Prozessoren für eine spätere Weiterverarbeitung im Hauptspeicher abzulegen und bereits vorliegende, digitale Videodaten aus dem Hauptspeicher zu den AxPe1280V-Prozessoren zu transportieren.

Neben der rein programmiertechnischen Realisierung ist die Definition einer geeigneten Abstraktionsschicht zwischen Treiber und Applikation ein wichtiger Aspekt dieser Studienarbeit. Ziel dieser Definition ist es, einer Applikation einen sicheren und komfortablen Zugang zu den neuen Fähigkeiten des Treibers zu verschaffen.

1.1 Der Videosignalprozessor AxPe1280V

Der AxPe1280V ist ein universell einsetzbarer und frei programmierbarer Videosignalprozessor, der eine für blockbasierte Verfahren der Videokodierung- und Dekodierung (z. B. ISO-MPEG1 und ISO-MPEG2) optimierte Architektur mit hoher Rechenleistung besitzt.

Ein einzelner AxPe1280V besteht aus 1,3 Millionen Transistoren und verbraucht nur 0,7W bei eine Taktgeschwindigkeit von 66 Mhz. Der AxPe1280V wurde in einer Koprozessorarchitektur realisiert, d. h. er besteht aus einem RISC-Prozessor, dem ein Koprozessor zur Seite gestellt ist[1].

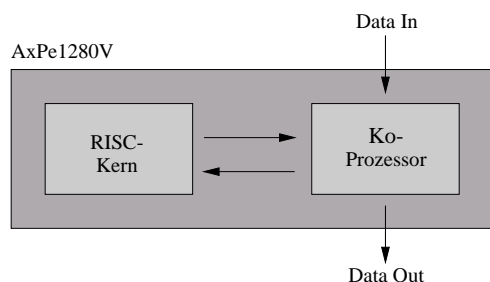


Abbildung 1: Koprozessorarchitektur des AxPe1280V

Bei seiner Standardtaktfrequenz von 66 Mhz erreicht der RISC-Prozessor eine Spitzenleistung von 66 MIPS (MIPS = Mega Instructions Per Second), der Koprozessor eine Spitzenleistung von 1 GOPS (GOPS = Giga Arithmetic Operations per Second).

Sollte für eine Anwendung diese Leistung nicht ausreichend sein, so besteht die Möglichkeit, AxPe1280V-Prozessoren zu einem Multiprozessorsystem zusammenzuschalten.

1.2 Die Multimedia-PCI-Bridge SAA7146A

Der SAA7146A-Baustein stellt ein PCI-Bus-Interface zur Verfügung, das den „PCI local bus“ Spezifikationen in der Revision 2.1 genügt[2]. Ein entsprechender PC erkennt somit PCI-Steckkarten, die mit dem SAA7146A ausgerüstet sind, automatisch nach dem Systemstart und weist ihnen die benötigten Systemressourcen zu.

Der SAA7146A blendet einen 512 Byte großen Adreßraum mit Registern in den Adreßraum des PCs ein und ermöglicht dem Prozessor den Lese- und Schreibzugriff auf den SAA7146A und somit die vollständige Kontrolle über alle Funktionen des Bausteins.

Der SAA7146A besitzt zwei bidirektionale, 8 bit breite Videointerfaces, die jeweils einen nach CCIR656 definierten Datenstrom transportieren können. Im folgenden werden diese beiden Interfaces Port A bzw. Port B genannt. Der Transport erfolgt durch zwei unabhängige Video-DMA-Einheiten. Beide Einheiten können sowohl lesend auf die Interfaces als auch schreibend auf den Hauptspeicher des PCs zugreifen. Eine Einheit kann zusätzlich noch lesend auf den Hauptspeicher und schreibend auf die Videointerfaces zugreifen.

Des weiteren besitzt der SAA7146A zwei Busse, um mit anderen Bausteinen auf der Karte Daten austauschen zu können. Zum einen ist dies der I²C-Bus, ein zweiadriger Bus zum einfachen, byteorientierten Datenaustausch mit niedriger Bandbreite. Dieser Bus wird hauptsächlich für Steuerungs- und Initialisierungszwecke benutzt. Zum anderen existiert das „Data Expansion Bus Interface“ (kurz: DEBI), ein 16 bit breiter, paralleler Bus mit einer Bandbreite von 23 MByte/s und Anschluß an das PCI-Interface.

Die Hauptanwendung des SAA7146A sieht im allgemeinen wie folgt aus: Zunächst werden die diversen Hilfsbausteine über den I²C-Bus initialisiert, danach ist der SAA7146A für den Datenzu- und -abtransport zuständig. Oftmals ist es erwünscht, ein an einem der beiden Videointerfaces anliegendes digitales Videosignal in den Speicher der Grafikkarte zu schreiben und damit auf dem Monitor darzustellen. Man sagt auch, daß das Videosignal „als Overlay“ dargestellt wird.



Abbildung 2: Overlaydarstellung eines Videosignals

Der SAA7146A ist ein busmasterfähiges PCI-Gerät, d. h. alle Datenübertragungen über den PCI-Bus erfolgen ohne Eingriffe des Hauptprozessors. Dadurch wird lediglich Bandbreite auf dem PCI-Bus verbraucht, aber keine Prozessorrechenzeit.

Bei der Overlaydarstellung eines Videosignals kopiert der SAA7146A die Bildinformationen über den PCI-Bus direkt in den Speicher der Grafikkarte. Die übrigen Systemkomponenten werden durch diesen Vorgang nicht beeinflusst und auch nicht darüber benachrichtigt. Daher müssen durch den Treiber geeignete Maßnahmen getroffen werden, damit dieser Vorgang z. B. Fenster auf dem Desktop nicht überschreibt und zerstört.

1.3 Die AxPe1280V-Demonstrationskarte

Abbildung 3 zeigt eine stark vereinfachte schematische Darstellung der Videodatenströme auf der AxPe1280V-Demonstrationskarte und die Anbindung an den PC[3].

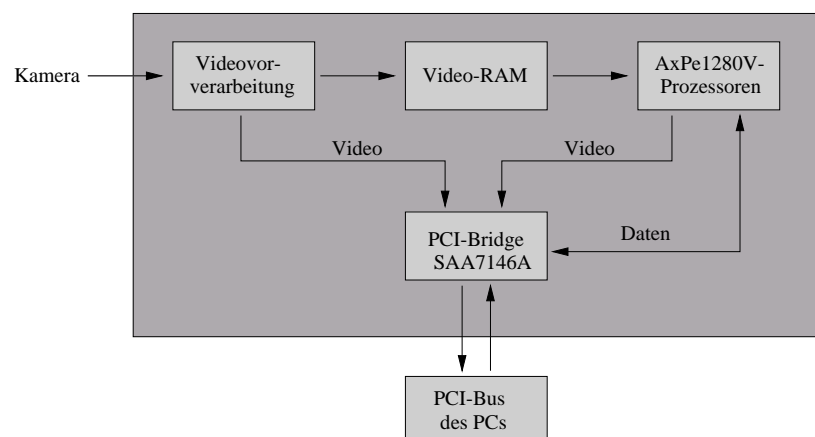


Abbildung 3: Schematischer Aufbau der AxPe1280V-Demonstrationskarte

Zunächst wird das analoge Videosignal z. B. einer Kamera in zwei Schritten der Videovorverarbeitung digitalisiert und skaliert.

Danach wird dieser Datenstrom sowohl an den Port A des SAA7146A geliefert als auch in ein 2 MB großes Video-RAM zur späteren Verarbeitung geschrieben. Die beiden AxPe 1280V-Prozessoren können die Bilddaten aus dem Video-RAM entnehmen und bearbeiten. Den bearbeiteten Videodatenstrom geben die beiden AxPe1280V-Prozessoren an den Port B des SAA7146A. Der SAA7146A kann nun entweder das Originalvideosignal von Port A oder das bearbeitete Videosignal von Port B auf dem Monitor als Overlay darstellen.

Alternativ können die beiden AxPe1280V-Prozessoren auch Daten über das DEBI mit dem SAA7146A bzw. dem PC austauschen.

1.4 Applikationen für die AxPe1280V-Prozessoren und den PC

Um die Leistungsfähigkeit der AxPe1280V-Prozessoren zu demonstrieren wurden bereits mehrere Programme entworfen.

- *vector*: Die AxPe1280V-Prozessoren vergleichen so schnell wie möglich aufeinanderfolgende Bilder und blenden auf Grundlage einer Bewegungsschätzung darin Bewegungsvektoren ein.
- *diff*: Die AxPe1280V-Prozessoren „subtrahieren“ zwei aufeinanderfolgende Bilder und stellen das Ergebnis dieser Operation dar.
- *h263cod*: Die AxPe1280V-Prozessoren führen eine Kodierung des Videodatenstroms nach ITU-T H.261 durch, einem Kompressionsverfahren mit dem Ziel, Bewegtbilder über Telefonleitungen übertragen zu können. Die komprimierten Daten können über das DEBI ausgelesen werden.
- *h263dec*: Die AxPe1280V-Prozessoren führen eine Dekodierung eines nach ITU-T H.261 kodierten Videodatenstroms durch. Die komprimierten Daten müssen über das DEBI zugeführt werden. Das Ergebnis der Dekodierung wird an Port B ausgegeben.

Zu Demonstrationszwecken existieren für die AxPe1280V-Demonstrationskarte im wesentlichen zwei Applikationen.

- *control*: Diese Applikation kann alle wichtigen Einstellungen an der AxPe1280V-Demonstrationskarte vornehmen (Abbildung 4). Sie bringt die Karte in einen definierten Anfangszustand und erlaubt das Laden von Programmen in die AxPe1280V-Prozessoren.

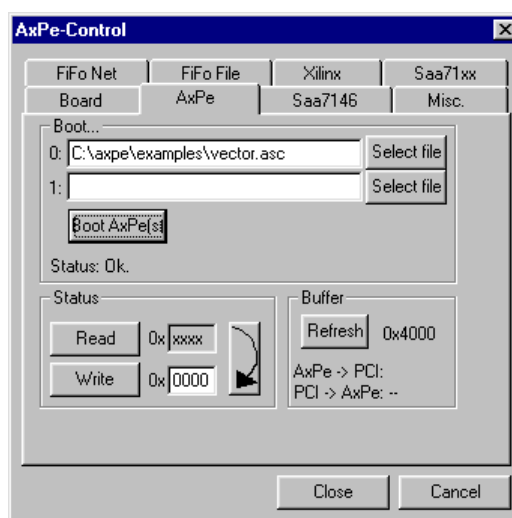


Abbildung 4: „control“-Applikation für die AxPe1280V-Demonstrationskarte

- *viewer*: Diese Applikation ermöglicht es, die Videodaten eines der beiden Videointerfaces des SAA7146A als Overlay darzustellen.

1.5 Architektur des Treibers

Bei der ursprünglichen Entwicklung eines Treibers für die AxPe1280V-Demonstrationskarte kam das Gerätetreiber-Toolkit *WinDriver* der Firma *Jungo* zum Einsatz, das Zugriff auf Geräte auf dem PCI-Bus auch „normalen“ Windows 98-Applikationen erlaubt[4].

Ergebnis dieser Entwicklung sind die beiden Bibliotheken *uh7146.dll* und *axpe_ut.dll*. Applikationen, die auf eine AxPe1280V-Demonstrationskarte zugreifen möchten, müssen diese beiden Bibliotheken in geeigneter Weise zu ihrer ausführbaren Datei hinzubinden.

Abbildung 5 zeigt die Treiberarchitektur in Form eines Schichtenmodells.

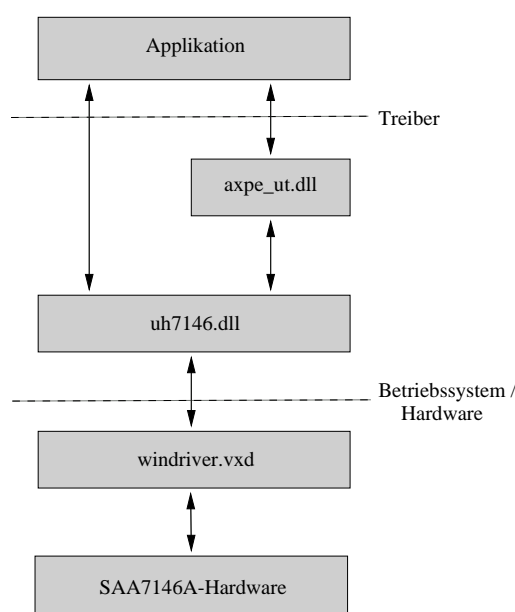


Abbildung 5: Bestehende Treiberarchitektur

In der Bibliothek *axpe_ut.dll* sind alle Funktionen gekapselt, die zwar Zugriff auf den SAA 7146A benötigen, aber spezifisch für die AxPe1280V-Demonstrationskarte sind.

uh7146.dll stellt eine Vielzahl von Funktionen für die verschiedenen Fähigkeiten des SAA 7146A bereit, die auch für andere SAA7146A-basierende Karten von Nutzen sein könnten. Für den physikalischen Zugriff auf den SAA7146A nutzt *uh7146.dll* den virtuellen Gerätetreiber *windriver.vxd* des *WinDriver*-Toolkits.

windriver.vxd schließlich stellt Funktionen für den physikalischen Zugriff auf die eigentliche SAA7146A-Hardware bereit.

2 Ideen für die Erweiterung des Treibers

In diesem Abschnitt soll ein grober Überblick über die Erweiterung des Treibers gegeben werden, der dann in den Abschnitten 4 und 5 verfeinert wird.

Neben der rein programmiertechnischen Realisierung ist die Definition einer geeigneten Abstraktionsschicht zwischen Treiber und Applikation ebenfalls sehr wichtig, damit ein sicherer und bequemer Zugang zu den neuen Fähigkeiten des Treibers möglich wird.

Der SAA7146A arbeitet auf der Basis von Bildern, die für ihn die kleinste, abgeschlossene Informationseinheit darstellen. Aus diesem Grund sollte die Verarbeitung bzw. Bereitstellung von Daten durch eine Applikation ebenfalls in Form von vollständigen Bildern geschehen. Ist eine Applikation im Besitz eines Bildes, steht es ihr natürlich frei, beliebige Manipulationen (z. B. eine Kompression) daran vorzunehmen.

Der Treiber benötigt zunächst die Information, wie viele Pufferspeicher angelegt werden sollen sowie die Breite, Höhe und das Format der zu verarbeitenden Bilder. Dadurch kann eine optimale Speicherausnutzung angestrebt werden. Der SAA7146A kann Bilder mit einer maximalen Breite von 768 Pixel und einer maximalen Höhe von 576 Pixel verarbeiten. Kleinere Bildgrößen sind ohne Einschränkungen möglich.

Die gebräuchlichsten Formate sind `RGB16_COMPOSED`, `RGB24_COMPOSED` und `RGB32_COMPOSED`. Hierbei werden die Farbinformationen jedes Pixel in den Rot-, Grün- und Blauanteil aufgespalten und dann nacheinander in den Pufferspeicher geschrieben. Die Formate unterscheiden sich nur in der pro Farbinformation verfügbaren Anzahl der Bits und damit in der Anzahl der darstellbaren Farben. Bei `RGB16_COMPOSED` sind dies 5 Bits für den Rotanteil, 6 Bits für den Grünanteil und 5 Bits für den Blauanteil, also insgesamt $2^5 * 2^6 * 2^5 = 65536$ verschiedene Farben. Bei den beiden anderen Formaten sind es jeweils 8 Pixel pro Farbinformation, also $2^{8^3} = 16,777216$ Millionen Farben.

Ein weiteres wichtiges Format ist `YUV422_COMPOSED`, dem ein anderes Farbmodell zugrunde liegt. Hierbei wird die Helligkeitsinformation (Luminanz, Y) getrennt von einem Farbinformationsvektor mit zwei Elementen (Chrominanz, UV) übertragen. Gleichzeitig wird eine Informationsverringering vorgenommen, in dem für zwei Helligkeitsinformationen nur ein Farbinformationsvektor übertragen wird. Diese Informationsverringering ist für das menschliche Auge nicht sichtbar, da es Helligkeitsinformationen doppelt so hoch auflösen kann wie Farbinformationen.

Danach ist es die Aufgabe der Applikation dem Treiber explizit mitzuteilen, wann ein Pufferspeicher zur Bearbeitung zur Verfügung steht. Die Applikation soll durch einen geeigneten Mechanismus benachrichtigt werden können, wenn der Treiber einen Pufferspeicher erfolgreich bearbeitet hat. Dieser Pufferspeicher muß dann wiederum explizit beim Treiber abgemeldet werden, bevor die darin enthaltenen Daten verwendet werden können.

Alle Parameter, die sich auf das Format der Pufferspeicher beziehen, werden durch das Anfordern der Pufferspeicher festgelegt. Eine Änderung dieser Parameter ist nur durch das Freigeben und Wiederanlegen mit anderen Parametern möglich. Dies stellt im allgemeinen keine Einschränkung der Funktionsfähigkeit dar, weil das Bearbeiten eines Videodatenstroms eigentlich immer mit einem festen Format geschieht, das sich während der Bearbeitung einer bestimmten Aufgabe nicht dynamisch ändert.

Alle Parameter werden möglichst typgerecht übergeben. Die Breite eines Bildes wird z. B. als `unsigned_int` deklariert, da eine negative Breite eines Bildes keinen Sinn macht. Eine geeignete Typdefinition sind Zeiger auf Pufferspeicher, die aus historischen Gründen des Treibers als `u32*` deklariert werden. Dadurch sollte wohl darauf hingewiesen werden, daß es

sich dabei um nicht weiter spezifizierte Speicherbereiche handelt, die gegebenenfalls durch die Applikation geeignet umdeklariert werden müssen.

Alle Funktionen geben im Erfolgsfall den Wert 0 zurück. Im Fehlerfall wird ein detaillierter, negativer Fehlercode in Form eines symbolischen Fehlernamens zurückgegeben. Eine Beschreibung über die vorhandenen Fehlercodes kann im Abschnitt A.1 nachgelesen werden.

3 Technische Aspekte der Erweiterung

Bisher war der Treiber lediglich für die Initialisierung der AxPe1280V-Demonstrationskarte und zum Laden von Programmen in die AxPe1280V-Prozessoren zuständig. Nachdem die Overlaydarstellung gestartet wurde, arbeitete die AxPe1280V-Demonstrationskarte autonom ohne Eingriffe des Treibers.

Der SAA7146A soll nun sowohl einen Videodatenstrom im Hauptspeicher des PCs ablegen können, als auch einen Videodatenstrom aus dem Hauptspeicher zu den AxPe1280V-Prozessoren transportieren können.

Wenn der SAA7146A einen dieser Videodatenströme überträgt, so erfolgt die Übertragung synchron mit der Frequenz des anliegenden Videosignals. Für einen Datenstrom, der ein PAL-Signal mit einer Bildwiederholfrequenz von 25Hz beinhaltet, transportiert der SAA7146A 25 mal in der Sekunde ein neues Bild über den PCI-Bus. Bei einer Bildgröße von 384*288 Pixel im Format RGB16_COMPOSED mit einer Farbtiefe von 2 Bytes pro Pixel benötigt dies bereits eine Bandbreite von

$$384 * 288 \text{ Pixel} * 2 \text{ Bytes} / \text{Pixel} * 25 \text{ Bilder} / \text{Sekunde} = 5,27 \text{ Mbyte} / \text{Sekunde}$$

Für den gleichzeitigen Transport von zwei Videodatenströmen verdoppelt sich die erforderliche Bandbreite entsprechend. Für den PCI-Bus stellen diese Anforderungen im normalen Betrieb kein Problem dar, da er eine theoretische Bandbreite von 133 Mbyte / Sekunde besitzt.

Da der Treiber lediglich für die Übertragung zwischen dem Hauptspeicher und der AxPe 1280V-Demonstrationskarte zuständig ist, liegt der Transport der Daten von einem Massenspeicher in den Hauptspeicher des PCs vollständig in der Hand der Applikation. Der verwendete Massenspeicher muß bei unkomprimierten Videodaten also ebenfalls die obige Datenrate liefern können.

Bereits aufgrund der obigen Rechnung können mehrere Forderungen an die Erweiterungen gestellt werden: Robustheit, Echtzeitfähigkeit und geringe Prozessorlast, damit dem PC möglichst viel Zeit für die Verarbeitung und Bereitstellung der Daten bleibt.

Natürlich ist für die Erweiterungen der SAA7146A entsprechend aufwendig zu programmieren, damit der digitale Videodatenstrom korrekt verarbeitet wird. Auf die konkrete Programmierung wird in dieser schriftlichen Ausarbeitung der Studienarbeit allerdings nicht detailliert eingegangen.

3.1 Speicherbehandlung

Schreibt der SAA7146A einen Videodatenstrom direkt in den Speicher der Grafikkarte, so stellt dies keine besonderen Anforderungen an das System. Der Speicher der Grafikkarte ist im allgemeinen linear in den Adreßraum des PCs eingeblendet, d. h. er erscheint dort z. B. als 64 Megabyte zusammenhängender Speicher.

Die Programmierung des SAA7146A ist in diesem Fall sehr einfach. Man benötigt lediglich eine Startadresse, eine Formatangabe, die Anzahl der zu schreibenden Bytes und den Abstand zweier aufeinanderfolgender Zeilen in Bytes. Auf Grundlage dieser Informationen schreibt der SAA7146A dann die Videodaten in den Speicher der Grafikkarte.

Der SAA7146A ist aber nicht an den Speicher der Grafikkarte als Ziel einer Schreiboperation gebunden. Durch entsprechende Programmierung kann der Videodatenstrom auch in den

Hauptspeicher des PCs geschrieben werden. Außerdem kann eine der beiden Video-DMA-Einheiten Videodaten nicht nur direkt von den Videointerfaces entnehmen, sondern auch aus dem Hauptspeicher des PCs lesen. Natürlich ist vorher dafür zu sorgen, daß entsprechend große Speicherbereiche für diese Zwecke zur Verfügung stehen und diese beim Betriebssystem korrekt reserviert worden sind.

Ein Problem bei modernen 32-bit Betriebssystemen für PCs ist deren Adreßverwaltung. Diese stellt im allgemeinen jeder Applikationen unabhängig vom tatsächlich verfügbaren Speicherausbau des PCs den maximal adressierbaren Speicherbereich von 4 Gigabyte als virtuellen Speicher zur Verfügung.

„Virtuell“ bedeutet in diesem Zusammenhang, daß der Speicher natürlich nicht von Anfang an für jeden Prozeß vorhanden ist, sondern sowohl der Speicher selbst als auch etwaige Adressen in diesen Speicher zur Laufzeit vom Betriebssystem und dem Prozessor mit Hilfe einer „memory management unit“ (MMU) erzeugt und verwaltet werden. Gerade von Applikationen nicht benötigte Speicherbereiche werden dabei häufig auf einem Massenspeicher ausgelagert.

Auf physikalischer Ebene in den Speicherbausteinen hingegen erfolgt bei den PCs das Ablegen der Daten in 4 Kilobyte großen Blöcken, den sogenannten „pages“. Es erfolgt also eine Umsetzung von virtuellen Adressen des Betriebssystems auf physikalische Adressen der Speicherbausteine.

Dies bedeutet insbesondere, daß Speicherbereiche, die für die Applikation als zusammenhängend erscheinen, auf der untersten physikalischen Ebene im allgemeinen nicht zusammenhängend sind (Abbildung 6). Es besteht für Applikationen normalerweise keine Möglichkeit, einen auf physikalischer Ebene zusammenhängenden Speicherbereich direkt anzufordern.

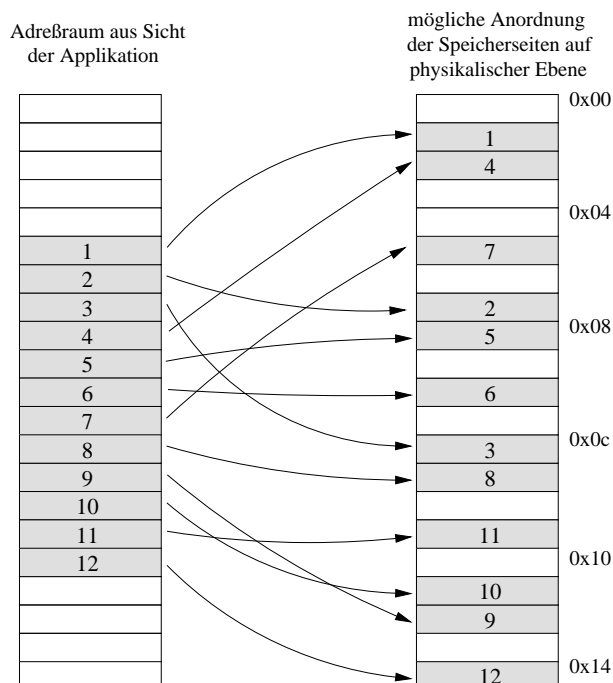


Abbildung 6: Speicher aus Sicht einer Applikation und der Hardware

Dies läßt die Programmierung des SAA7146A auf den ersten Blick unmöglich erscheinen, da er ausschließlich mit physikalischen Adressen arbeitet, um unabhängig vom Betriebssystem

zu sein. Da aber der durch eine Applikation angeforderte, virtuell lineare Speicherbereich auf physikalischer Ebene nicht linear und zusammenhängend ist, kann er ohne eine spezielle Programmierung nicht als Basis einer linearen Schreib- oder Leseoperation des SAA7146A dienen.

Glücklicherweise unterstützt sowohl der SAA7146A durch eine eigene MMU als auch das *WinDriver*-Toolkit diesen Umstand moderner Betriebssysteme der PC-Architektur.

Das *WinDriver*-Toolkit kann für einen durch eine Applikation angeforderten Speicherbereich die physikalischen Adressen der darunterliegenden Speicherseiten ermitteln. Für den SAA7146A muß dann für jeden Speicherbereich aus dessen physikalischen Adressen eine sogenannte „pagetable“ aufgebaut werden.

1	2	3	4
0x01	0x07	0x0c	0x02
5	6	7	8
0x08	0x0a	0x05	0x0d
9	10	11	12
0x12	0x11	0x0f	0x14

Abbildung 7: Pagetable zum Beispiel aus Abbildung 6

Abbildung 7 zeigt die „pagetable“ für das Beispiel aus Abbildung 6. Die physikalischen Adressen der Speicherseiten werden einfach entsprechend ihrer linearen Reihenfolge aus Sicht der Applikation darin abgelegt.

Während des Zugriffs auf den Hauptspeicher durch den SAA7146A werden durch die MMU die physikalischen Adressen aus der „pagetable“ entnommen und damit die Zieladresse im Hauptspeicher berechnet. Dadurch werden die Zugriffe des SAA7146A so angeordnet, daß sie zu einem aus der Sicht der Applikation linearen Zugriff in dessen Adreßraum führen, und somit die Videodaten korrekt in den Speicherbereichen geschrieben werden.

3.2 Synchronisation

In Abschnitt 3 wurde bereits geschildert, daß die Verarbeitung der Bilder auf der AxPe 1280V-Demonstrationskarte synchron zum Takt der Videovorverarbeitung erfolgt. Daher muß einer Applikation z. B. immer zu Beginn der Übertragung eines neuen Bildes (also bei einer Übertragung mit 25 Bildern pro Sekunde alle 40 Millisekunden) die Möglichkeit zur Synchronisation mit den Daten in den Pufferspeichern zu geben.

Der SAA7146A kann beim Eintreten einer Vielzahl von Ereignissen einen Interrupt auslösen. Ein Interrupt ist ein Signal einer Hardware, das den Prozessor des PCs veranlasst, die aktuelle Bearbeitung zu unterbrechen. Es wird dann eine spezielle, privilegierte Interruptserviceroutine ausgeführt, um auf das Ereignis so schnell wie möglich reagieren zu können.

Der SAA7146A kann einerseits so programmiert werden, daß er einen Interrupt auslöst, wenn auf einem der beiden Videointerfaces der Anfang eines neuen Bildes signalisiert wird. Das *WinDriver*-Toolkit ermöglicht es andererseits, eine Interruptserviceroutine im Betriebssystem zu verankern. Damit ist es möglich, eine geeignete Logik für die Synchronisation zu entwickeln.

Löst der SAA7146A einen Interrupt aus, so wird die auslösende Bedingung im Interruptstatusregister (ISR) des SAA7146A vermerkt. Der Prozessor unterbricht daraufhin seine aktuelle Arbeit, verhindert das Auslösen weiterer Interrupts durch andere Komponenten des

PCs und verzweigt in die Interruptserviceroutine des Treibers. Diese führt nun die nötigen Verarbeitungsschritte durch und bestätigt abschließend die entsprechende Bedingung im ISR. Danach fährt der PC mit seiner vorher unterbrochenen Bearbeitung fort.

Über einen geeigneten Mechanismus des Betriebssystems wird die Applikation über die Bearbeitung eines Interrupts durch den Treiber informiert, damit eine eventuell zeitintensivere Vor- bzw. Nachbearbeitung der Daten durch die Applikation erfolgen kann.

4 Der Capture-Vorgang

4.1 Schreiben eines Videodatenstroms in den Hauptspeicher

Der Vorgang des Schreibens eines digitalen Videodatenstroms in einen durch eine Applikation angeforderten Pufferspeicher wird im folgenden als Capture-Vorgang bezeichnet.

Die in Abschnitt 2 grob beschriebene Idee zur Erweiterung soll im folgenden sowohl durch die Beschreibung der Abstraktionsschicht des Treibers zur Applikation als auch der konkreten Implementierung innerhalb des Treibers erklärt werden.

Dabei wurde mehr Wert auf die Verständlichkeit und den Umfang der Erklärungen gelegt, als beispielsweise auf die Nennung aller möglichen Fehlerbedingungen. Diese können bei Bedarf im Abschnitt A.2 nachgelesen werden.

4.2 Implementierung aus der Sicht einer Applikation

Durch den Aufruf der Funktion `saa7146_capture_buffers_request` kann eine Applikation den Treiber veranlassen, ihr die Anzahl von `number` Puffern zur Verfügung zu stellen, um Bilder mit der Höhe `height`, der Breite `width`, im Format `format` vom Interface `port` abgreifen zu können.

```
int saa7146_capture_buffers_request(struct saa7146* saa,
    unsigned int* number, unsigned int* width,
    unsigned int* height, unsigned int* format,
    unsigned int* port, u32* ptrs[], unsigned long* sizeimage);
```

Der Parameter `format` beschreibt die Information, in welchem Format der SAA7146A den Videodatenstrom in die Puffer schreiben soll. Dieser Parameter umfaßt die Information über die Farbtiefe des Bildes und wirkt sich deshalb unter anderem auch auf die Anzahl der Bytes aus, die pro Pixel geschrieben werden. Der Capture-Vorgang unterstützt die bereits in Abschnitt 2 erwähnten Formate `RGB16_COMPOSED`, `RGB24_COMPOSED`, `RGB32_COMPOSED` und `YUV422_COMPOSED`.

Der Parameter `saa` bezeichnet im folgenden ein Handle für den gewünschten SAA7146A, der über eine entsprechende Funktion des Treibers (`saa7146_choose_saa7146`) bekommen werden kann.

Die Funktion belegt den Speicher für die Puffer im Adreßraum der aufrufenden Applikation und kopiert Zeiger, die auf den Beginn der jeweiligen Pufferspeicher zeigen, in das Array `ptrs`. Im Parameter `sizeimage` wird schließlich die Größe eines Bildes in Bytes zurückgegeben.

Der Treiber prüft zunächst, ob die Vorgaben der Applikation erfüllt werden können. Ist dies für einzelne Parameter nicht möglich (z. B. weil die Breite des Bildes außerhalb des gültigen Bereiches liegt), so werden diese entsprechend korrigiert und dann erst die Puffer angelegt. Da die Applikation die Parameter „by reference“ als Zeiger übergeben hat, muß sie nach der Rückkehr der Funktion überprüfen, inwiefern die Vorgaben erfüllt werden konnten.

Wie bereits in Abschnitt 2 erwähnt, ist der Rückgabewert dieser und aller anderen Funktion 0, falls kein Fehler aufgetreten ist. Ansonsten wird ein symbolischer Fehlercode zurückgegeben, der den Fehler eingrenzt (vgl. Abschnitt A.1).

Über einen Aufruf der Funktion

```
int saa7146_capture_buffers_release(struct saa7146* saa);
```

kann dem Treiber mitgeteilt werden, daß kein Bedarf mehr an den Pufferspeichern besteht. Selbstverständlich werden dadurch die Zeiger auf die Pufferspeicher ungültig.

Die übergebenen Parameter bei `saa7146_capture_buffers_request` wirken sich bis zur Freigabe der Puffer mit `saa7146_capture_buffers_release` aus. Es besteht keine Möglichkeit, diese Parameter während des Capture-Vorgang zu ändern. Dadurch kann die Hardware optimal programmiert und der Speicherverbrauch effizient gehandhabt werden. Wäre dies nicht der Fall, so müßten Sicherheitsmaßnahmen für den Fall vorgesehen werden, daß Bilder nach geänderten Parametern nicht mehr in die Pufferspeicher passen. Dadurch würde die gesamte Programmierung wesentlich komplizierter werden, ohne Vorteile bei der Benutzung zu bringen.

Durch den Aufruf von `saa7146_capture_buffers_request` hat die Applikation eine bestimmte Anzahl von Puffern reserviert, die intern in aufsteigender Reihenfolge durchnummeriert sind. Durch den Aufruf der Funktion

```
int saa7146_capture_queue_buffer(struct saa7146* saa, unsigned int buffer);
```

kann der jeweilige Puffer `buffer` der Capture-Logik für einen zukünftigen Capture-Vorgang zur Verfügung gestellt werden. Auf den dazugehörigen Speicherbereich sollte dann natürlich nicht mehr schreibend zugegriffen werden.

Die folgenden zwei Funktionen sind für das Starten bzw. Stoppen der Capture-Logik zuständig und haben neben dem Handle für den SAA7146A keine Parameter:

```
int saa7146_capture_stream_on(struct saa7146* saa);  
int saa7146_capture_stream_off(struct saa7146* saa);
```

Durch einen Aufruf der Funktion

```
int saa7146_capture_wait_frame(struct saa7146* saa, unsigned long timeout);
```

kann die Applikation in einen Wartezustand übergehen, in dem keine Rechenzeit des Prozessors vergeudet wird. Der Parameter `timeout` gibt eine Maximalwartezeit in Millisekunden an. Ein Wert von 0 bedeutet eine unendliche Zeitspanne. Die Funktion kehrt erst wieder zurück, wenn das nächste Bild von der Capture-Logik verarbeitet wurde, ein Fehler aufgetreten ist, oder aber ein Timeout aufgetreten ist.

Der Aufruf dieser Funktion gibt keine Auskunft darüber, wie viele Bilder in der Zwischenzeit schon erfolgreich durch die Capture-Logik bearbeitet wurden.

Die Funktion

```
int saa7146_capture_dequeue_buffer(struct saa7146* saa);
```

schließlich gibt als Rückgabewert die Nummer des Puffers zurück, in dem sich das nächste fertige Bild befindet. Zu beachten ist: in diesem Fall bedeutet der Wert 0, daß sich ein fertiges Bild im ersten Pufferspeicher befindet.

Schließlich existiert noch eine Funktion, mit der jederzeit eine Statistik über die Capture-Logik abgerufen werden kann:

```
int saa7146_capture_get_statistics(struct saa7146* saa,  
    struct statistics* stat);
```

Die Struktur `statistics` enthält zwei Zählvariablen:

```
struct statistics {
    unsigned long frames_lost;
    unsigned long frames_processed;
};
```

Die beiden Zählvariablen geben Auskunft über die verlorengegangen bzw. erfolgreich verarbeiteten Bilder seit dem letzten Start der Capture-Logik. Als verlorengegangen gilt ein Bild, wenn die Capture-Logik beim Anfang eines neuen Bildes keinen freien Puffer vorfinden konnte (siehe Abschnitt 4.3, Abbildung 8).

4.3 Implementierung aus der Sicht des Treibers

In Abschnitt 3.2 wurde bereits erläutert, daß die Programmierung der Hardware synchron zum Auftreten der Synchronisationsimpulse an den beiden Videointerfaces in einer speziellen Interruptserviceroutine geschehen sollte.

Die Interruptserviceroutine des Treibers beinhaltet die eigentliche Intelligenz des Capture-Vorgangs und benötigt einige Variablen, um ihre Informationen und den Ablauf zu organisieren.

Zur Verwaltung der Pufferspeicher besitzt der Treiber zwei Queues. In der Queue mit dem Namen `todo` werden die durch einen Aufruf von `saa7146_capture_queue_buffer` gewünschten Puffer eingefügt und damit der Capture-Logik zur Verfügung gestellt. Die Queue mit dem Namen `done` enthält alle Puffer, die bereits verarbeitet wurden und durch einen Funktionsaufruf von `saa7146_capture_dequeue_buffer` aus der Capture-Verarbeitung entfernt werden können.

Es handelt sich um eine simple Zeiger-Queue, für die lediglich die benötigten Funktionen zum Initialisieren (`init_queue()`), zum Einfügen von Elementen am Ende (`AddTail()`), Löschen von Elementen am Anfang (`RemoveHead()`) und zur Abfrage, ob die Queue leer ist (`IsEmpty()`) vorhanden sind.

Ferner verwaltet der Treiber noch einen Zeiger `current`, der auf den aktuell verwendeten Puffer zeigt und zwei Zählvariablen `frames_processed` und `frames_lost` für Statistikzwecke.

Die Abbildung 8 verdeutlicht den schematischen Ablauf der Vorgänge innerhalb der Upload-Interruptserviceroutine, der nun erläutert werden soll.

Zu Anfang werden die oben erwähnten Variablen initialisiert.

Die Interruptserviceroutine wird beim Auftreten eines Synchronisationsimpulses an einem der beiden Videointerfaces geweckt. Zunächst wird überprüft, ob dieser Impuls für die weitere Bearbeitung überhaupt relevant ist, d. h. der Impuls von dem Videointerface kam, von welchem der Capture-Vorgang durchgeführt werden soll. Ist das der Fall so wird geprüft, ob dies der erste Interrupt nach einem Aufruf von `saa7146_capture_stream_on` ist. Falls ja, so muß der Capture-Vorgang erst gestartet werden und der folgende Schritt kann übersprungen werden. Falls nicht, so ist die Capture-Logik bereits aktiv.

Nun wird der Zustand des `current`-Zeigers überprüft. Wenn dieser `NULL` ist, so ist kein aktiver Puffer vorhanden und das letzte Bild ist verlorengegangen; der Zähler `frames_lost` wird erhöht. Falls `current` aber auf einen Puffer zeigt, so wurde das letzte Bild erfolgreich in diesen Puffer geschrieben. Der Zähler `frames_processed` wird erhöht und der Puffer `current` wird der Queue `done` hinzugefügt.

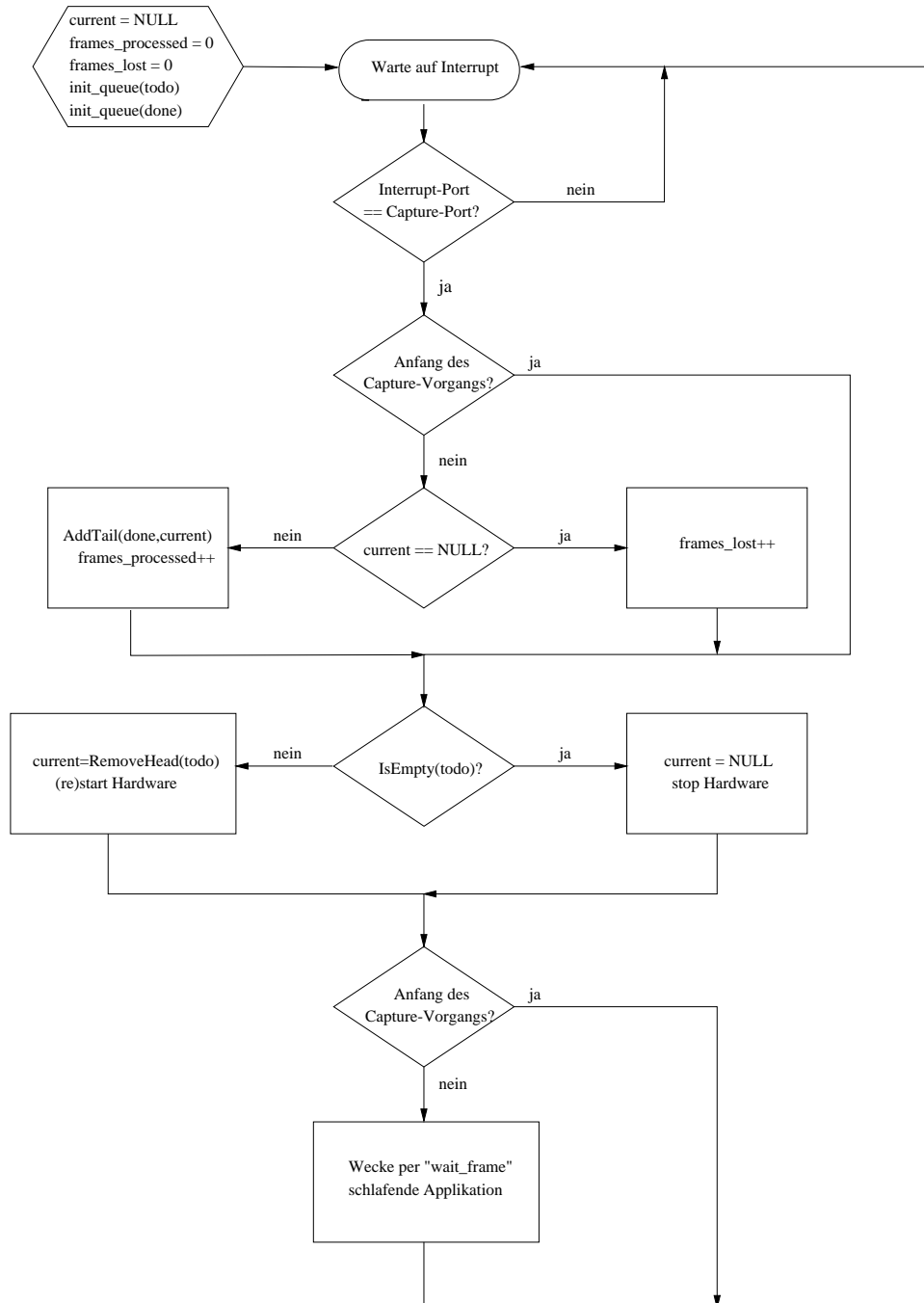


Abbildung 8: Schematischer Ablauf der Capture-Interruptserviceroutine

Nun muß die Hardware auf den nächsten Capture-Vorgang programmiert werden. Dazu wird geprüft, ob in der `todo`-Queue noch Puffer vorhanden sind. In diesem Fall wird die Hardware entsprechend programmiert (und eventuell neu gestartet) und der `current`-Zeiger auf den aus der `todo`-Queue entfernten Puffer gesetzt. Ist die `todo`-Queue leer, so wird der `current`-Zeiger auf `NULL` gesetzt und die Hardware gestoppt.

Zum Schluß wird noch ein mit `saa7146_capture_wait_frame` wartende Applikation geweckt, falls der Capture-Vorgang nicht gerade gestartet wurde. Danach wird die Interruptservice-routine beendet.

5 Der Upload-Vorgang

Auf der AxPe1280V-Demonstrationskarte erhält der SAA7146A einen gültigen digitalen Videodatenstrom, der durch die Digitalisierung des Signals einer externen Videosignalquelle durch einen Hilfsbaustein erzeugt wurde.

Der SAA7146A ist aber auch in der Lage, Videodaten aus dem Hauptspeicher des PCs zu lesen und daraus einen für sich gültigen Datenstrom zu erzeugen. Zur Synchronisation werden allerdings Synchronisationsimpulse einer externen Quelle benötigt, da diese Information nicht aus den Videodaten generiert werden kann.

Damit läßt sich der in Abbildung 9 dargestellte Videodatenfluß innerhalb der AxPe1280V-Demonstrationskarte erzeugen.

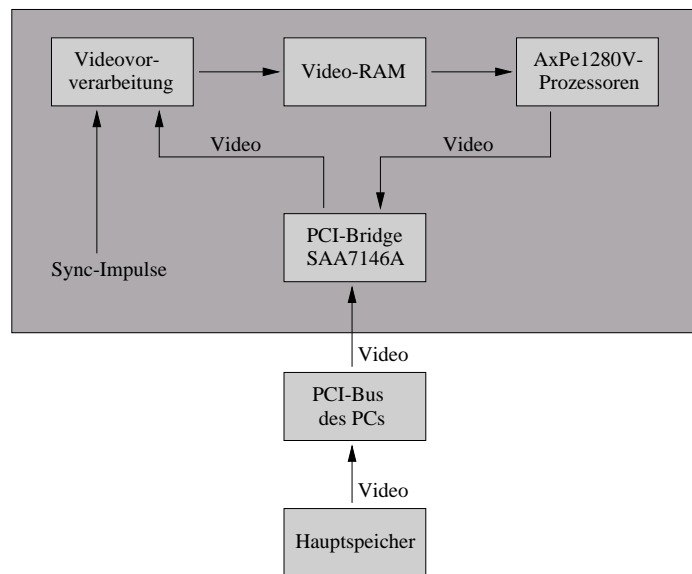


Abbildung 9: Videodatenfluß beim Upload-Vorgang

5.1 Lesen von Videodaten aus dem Hauptspeicher

Der Transport eines digitalen Videodatenstroms aus einem durch eine Applikation angeforderten Pufferspeicher zu einem der beiden Videointerfaces des SAA7146A wird im folgenden als Upload-Vorgang bezeichnet.

Die in Abschnitt 2 grob beschriebene Idee zur Erweiterung soll im folgenden sowohl durch die Beschreibung der Abstraktionsschicht des Treibers zur Applikation als auch der konkreten Implementierung innerhalb des Treibers erklärt werden.

Auch hier wurde mehr Wert auf die Verständlichkeit und den Umfang der Erklärungen gelegt, als beispielsweise auf die Nennung aller möglichen Fehlerbedingungen. Diese können bei Bedarf in Abschnitt A.2 nachgelesen werden.

5.2 Synchronisation des Upload-Vorgangs

Wie bereits in Abschnitt 5 erwähnt, kann der SAA7146A aus den Videodaten allein kein Synchronisationssignal erzeugen, sondern ist auf eine externe Synchronisationsquelle angewiesen.

Für die Anwendung auf der AxPe1280V-Demonstrationskarte ist es notwendig, Videodaten an den Port A des SAA7146A zu senden. Dort liegen idealerweise schon die Synchronisationsimpulse der Videosignalvorverarbeitung an. Daher erfolgt der Upload-Vorgang der Videodaten ebenfalls mit dem Takt der Videosignalvorverarbeitung.

Wie beim Capture-Vorgang wird der SAA7146A so programmiert, daß er beim Auftreten des Synchronisationsimpulses einen Interrupt auslöst. Die Interruptserviceroutine wird um die Behandlung des Upload-Interrupts erweitert.

5.3 Implementierung aus der Sicht einer Applikation

Die Implementierung und Logik des Upload-Vorgangs ist der des Capture-Vorgangs in weiten Teilen ähnlich. Daher wird darauf nicht mehr ganz so ausführlich eingegangen. Alle Details sind aber aus den detaillierten Funktionsbeschreibungen in Abschnitt A.2 ersichtlich.

```
int saa7146_upload_buffers_request(struct saa7146* saa,
    unsigned int* number, unsigned int* width, unsigned int* height,
    unsigned int* port_sync, unsigned int* port_upload,
    u32* ptrs[], unsigned long* sizeimage);
```

Durch den Aufruf der Funktion `saa7146_upload_buffers_request` kann eine Applikation den Treiber veranlassen, ihr die Anzahl von `number` Puffern zur Verfügung zu stellen, um Bilder mit der Höhe `height` und der Breite `width` zu verarbeiten. Zusätzlich werden die Angaben benötigt, von welchem Videointerface die Synchronisationsimpulse genommen werden sollen (`port_sync`) und zu welchem Port schließlich die Daten geschickt werden sollen (`port_upload`). Es wird keine Formatangabe benötigt, da der SAA7146A in diesem Modus nur Daten im Format `YUV422_COMPOSED` handhaben kann.

Für die AxPe1280V-Demonstrationskarte sind die letzten beiden Angaben eigentlich irrelevant, da die Videodaten sowieso nur an den Port A gesendet werden können. Da es sich aber um eine allgemeine Erweiterung des SAA7146A-Treibers handelt, wurden diese Optionen mit aufgenommen.

Die Funktion belegt den Speicher für die Puffer im Adreßraum der aufrufenden Applikation und kopiert Zeiger, die auf den Anfang der jeweiligen Puffer zeigen, in das Array `ptrs`. Im Parameter `sizeimage` wird schließlich die Größe eines Bildes in Bytes zurückgegeben.

Auch hier prüft der Treiber zunächst, ob die Vorgaben der Applikation erfüllt werden können. Falls dies nicht möglich ist, werden die Parameter entsprechend korrigiert. Eine Applikation sollte daher die Werte nach der Rückkehr des Funktionsaufrufs überprüfen, um zu sehen inwieweit die Anforderung erfüllt werden konnte.

Über einen Aufruf der Funktion

```
int saa7146_upload_buffers_release(struct saa7146* saa);
```

kann dem Treiber mitgeteilt werden, daß kein Bedarf mehr an den Puffern besteht. Selbstverständlich werden dadurch die Zeiger auf die Puffer ungültig.

Analog zum Capture-Vorgang können über die beiden Funktionen `saa7146_upload_queue_buffer` und `saa7146_upload_dequeue_buffer` Puffer der Upload-Logik zur Verfügung gestellt werden bzw. die Information abgerufen werden, welcher Puffer bereits erfolgreich bearbeitet wurde. Natürlich sollte sich das zu übertragene Bild bereits vor einem Aufruf von `saa7146_upload_queue_buffer` in dem entsprechenden Speicherbereich befinden.

```
int saa7146_upload_queue_buffer(struct saa7146* saa, unsigned int buffer);
```

```
int saa7146_upload_dequeue_buffer(struct saa7146* saa);
```

Weiterhin existieren auch hier zwei Funktionen zum Starten bzw. Stoppen der Upload-Logik:

```
int saa7146_upload_stream_on(struct saa7146* saa,
    unsigned int repeat_send, unsigned int repeat_count);
```

```
int saa7146_upload_stream_off(struct saa7146* saa);
```

Die Funktion `saa7146_upload_stream_on` kennt zwei zusätzliche Parameter. `repeat_send` steuert das Verhalten der Logik, wenn die Upload-Queue leerlaufen sollte. Dabei sind die Werte `UPLOAD_REPEAT_SEND_NORMAL` und `UPLOAD_REPEAT_SEND_KEEP_LAST` erlaubt. Bei `UPLOAD_REPEAT_SEND_NORMAL` wird die Hardware gestoppt und der Upload-Vorgang unterbrochen. Bei `UPLOAD_REPEAT_SEND_KEEP_LAST` wird der letzte Puffer in der Upload-Queue so lange übertragen, bis ein neuer Puffer hinzugefügt wird.

Diese Wahlmöglichkeit ist wichtig, da einige auf den AxPe1280V-Prozessoren laufenden Programmen einen zu jeder Zeit gültigen Daten erwarten. Bei einem zwischenzeitlichen Stoppen des Upload-Vorgangs ist dies dann natürlich nicht gewährleistet. Selbst wenn die Programme fehlertolerant arbeiten ist es meistens zweckmäßig, die Option `UPLOAD_REPEAT_SEND_KEEP_LAST` zu wählen, damit sich Verzögerungen und Aussetzer beim Upload-Vorgang später nicht als Bildaussetzer bemerkbar machen.

Der zweite Parameter `repeat_count` enthält die Information, wie oft jeder Puffer bei der Übertragung wiederholt werden soll. Der Wert 0 weißt den Treiber an, jeden Puffer nur einmal zu übertragen.

Diese Wahlmöglichkeit ist im Hinblick auf die allgemeine Belastung des Rechners wichtig. Einige auf den AxPe1280V-Prozessoren laufende Programme können beispielsweise nicht mehr als 5 Bilder in der Sekunde verarbeiten. Bei einer Bildwiederholfrequenz von 25 Hz ist es dann zweckmäßig, jedes Bild 5 mal wiederholen zu lassen. Dadurch wird zwar keine Bandbreite auf dem PCI-Bus eingespart, aber der Hauptprozessor des PCs kann sich anderen Dingen widmen.

Soll die Applikation einen kontinuierlichen Upload von Bilddaten durchführen, ist die Funktion

```
int saa7146_upload_wait_frame(struct saa7146* saa, unsigned long timeout);
```

nützlich.

Durch einen Aufruf dieser Funktion kann die Applikation in einen Wartezustand übergehen, in dem kein Rechenzeit des Prozessors vergeudet wird. Der Parameter `timeout` gibt eine Maximalwartezeit in Millisekunden an. Ein Wert von 0 bedeutet eine unendliche Zeitspanne. Die Funktion kehrt erst wieder zurück, wenn ein Puffer frei geworden, ein Fehler aufgetreten, oder aber der Timeoutwert erreicht ist.

Der Aufruf dieser Funktion gibt keine Auskunft darüber, wie viele Bilder in der Zwischenzeit von der Upload-Logik verarbeitet wurden.

Schließlich existiert noch eine Funktion, mit der jederzeit eine Statistik über die Upload-Logik abgerufen werden kann:

```
int saa7146_upload_get_statistics(struct saa7146* saa,
    struct statistics* stat);
```

Die beiden Zählvariablen in der Struktur `stat` geben auch hier Auskunft über die verlorengangenen bzw. erfolgreich verarbeiteten Bilder seit dem letzten Start der Upload-Logik.

Als verlorengegangen gilt ein Bild, wenn die Upload-Logik beim Anfang eines neuen Bildes keinen neuen Puffer mit Bilddaten vorfinden konnte.

5.4 Implementierung aus der Sicht des Treibers

Um die in Abschnitt 5.3 beschriebene Funktionalität zu implementieren, muß auch für den Upload-Vorgang die Interruptserviceroutine modifiziert werden, da auch hier die Programmierung der Hardware synchron zum Auftreten der Synchronisationsimpulse an den beiden Videointerfaces erfolgen muß.

Die Upload-Interruptserviceroutine bedient sich bei der internen Organisation der Pufferspeicher einem ähnlichen Satz an Variablen wie die Capture-Interruptserviceroutine.

Zur Verwaltung der Pufferspeicher besitzt der Treiber ebenfalls zwei Queues. In der Queue mit dem Namen `todo` werden die durch einen Aufruf von `saa7146_upload_queue_buffer` gewünschten Puffer eingefügt und damit der Upload-Logik zur Verfügung gestellt. Die Queue mit dem Namen `done` enthält alle Puffer, aus denen bereits erfolgreich ein Upload durchgeführt wurde. Diese Puffer können durch einen Funktionsaufruf von `saa7146_upload_delete_buffer` entfernt und wiederverwendet werden.

Beim Upload-Vorgang enthält jeder Pufferspeicher zusätzlich die Information `buf_count`, in der festgehalten wird, wie oft der jeweilige Puffer noch wiederholt übertragen werden muß.

Ferner verwaltet der Treiber ebenfalls einen Zeiger `current`, der auf den aktuell verwendeten Puffer zeigt und zwei Zählvariablen `frames_processed` und `frames_lost` für Statistikzwecke.

Die Abbildung 10 verdeutlicht den schematischen Ablauf der Vorgänge innerhalb der Interruptserviceroutine, der nun erläutert werden soll.

Zu Anfang werden die oben bereits erwähnten Variablen initialisiert.

Die Interruptserviceroutine wird beim Auftreten eines Synchronisationsimpulses an einem der beiden Videointerfaces geweckt. Zunächst wird überprüft, ob dieser Impuls für die weitere Bearbeitung überhaupt relevant ist, d. h. der Impuls von dem Videointerface kam, von welchem die Synchronisationsimpulse für den Upload-Vorgang erwartet werden. Ist das nicht der Fall, kehrt die Routine sofort zurück.

Ansonsten wird geprüft, ob der `current`-Zeiger auf einen gültigen Pufferspeicher zeigt. Ist dies nicht der Fall, ist entscheidend, ob dies der erste Interrupt nach einem Aufruf von `saa7146_upload_stream_on` ist. Falls nicht, wurde der letzte Upload verpasst und die Variable `frames_lost` erhöht.

In jedem Fall muß nun versucht werden, die Hardware für den nächsten Upload-Vorgang zu programmieren. Dazu wird geprüft, ob in der `todo`-Queue Puffer vorhanden sind. Ist dies der Fall, so wird die Hardware entsprechend programmiert (und eventuell neu gestartet) und der `current`-Zeiger auf den aus der `todo`-Queue entfernten Puffer gesetzt. Ist die `todo`-Queue leer, so wird die Routine beendet.

Behandeln wir nun den Fall, daß der `current`-Zeiger auf einen gültigen Puffer zeigt. Es wird zunächst der Inhalt der Variable `buf_count` ausgewertet und geprüft, ob der aktuelle Puffer nochmals wiederholt werden muß. Falls ja, so wird einfach die Variable `buf_count` erniedrigt und die Routine beendet.

Falls nicht, so muß überprüft werden, ob die Variable `buf_count` genau 0 ist. In diesem Fall wurde der Upload-Vorgang für den aktuellen Puffer erfolgreich durchgeführt. Die Variable `buf_count` wird nochmals (auf -1) dekrementiert und `frames_processed` erhöht.

Falls `buf_count` echt kleiner als 0 ist, so muß der Upload-Vorgang mit der Option `UPLOAD_REPEAT_SEND_KEEP_LAST` gestartet worden sein und der Treiber wiederholt nun den letzten Puffer aufgrund der gewählten Option. Da aber der Upload-Vorgang eigentlich mit einem neuen Puffer hätte durchgeführt werden müssen, wird in diesem Fall der Zähler `frames_lost` erhöht.

Nun muß die Hardware auf den nächsten Upload-Vorgang programmiert werden. Dazu wird geprüft, ob in der `todo`-Queue Puffer vorhanden sind. Ist dies der Fall, so wird der aktuelle Puffer der `done`-Queue hinzugefügt, der erste Puffer aus der `todo`-Queue entfernt und zum aktuellen Puffer erklärt und die Hardware entsprechend programmiert (und eventuell neu gestartet). Natürlich wird dann noch eine mittels `saa7146_upload_wait_frame` wartende Applikation geweckt.

Ist die `todo`-Queue leer, so ist entscheidend, mit welchem Parameter `repeat_send` die Funktion `saa7146_upload_stream_on` aufgerufen wurde. War dies `UPLOAD_REPEAT_SEND_KEEP_LAST`, so muß die Hardware den letzten Puffer wiederholen. Daher passiert nichts weiter und die Routine kehrt zurück. Wurde hingegen die Option `UPLOAD_REPEAT_SEND_NORMAL` gewählt, wird der `current`-Zeiger auf `NULL` gesetzt und die Hardware gestoppt. Danach wird die Interruptserviceroutine beendet.

6 Demonstrationssoftware „UplCap“

Um die Funktionsfähigkeit in ansprechender Weise zu zeigen, wurde die Demonstrationssoftware „UplCap“ geschrieben. Die Implementierung erfolgte unter *Visual C++ 6.0* von *Microsoft* auf Basis der *Microsoft Foundation Classes 6.0 (MFC)* Klassenbibliothek.

„UplCap“ präsentiert auf zwei Dialogseiten die Möglichkeiten, sowohl den Capture-Vorgang von Bildern als auch den Upload-Vorgang von Bildern durchführen zu können. Die Dialoge wurden passend zum Rest des Treibers in englischer Sprache gehalten.

6.1 Capture-Vorgang

Abbildung 11 zeigt die Dialogseite „Capture“ des Programms „UplCap“.

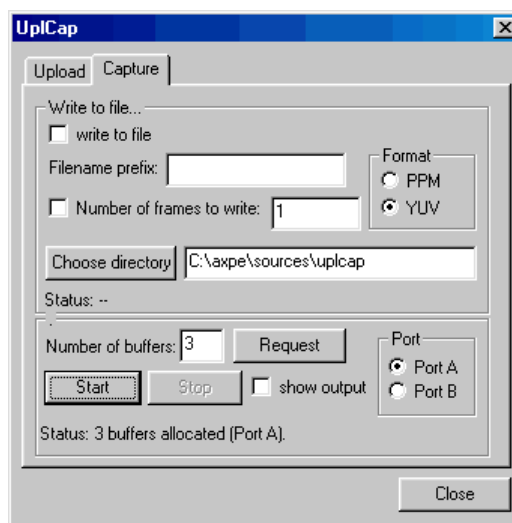


Abbildung 11: Dialogseite „Capture“

Mit Hilfe dieses Dialoges ist es möglich, Bilder von einem der beiden Videointerfaces des SAA7146A abzugreifen, darzustellen und bildweise auf einem Massenspeicher abzulegen.

Am unteren Rand des Dialoges befindet sich die Statuszeile, die Auskunft über das Ergebnis einer Aktion oder über den eventuell laufenden Capture-Vorgang gibt.

Darüber befindet sich das Eingabefeld „Number of Buffers“, in dem die Anzahl der verwendeten Pufferspeicher verändert werden kann. Mit Hilfe der Radiobuttons kann der für den Capture-Vorgang verwendete Port gewählt werden. Durch einen Druck auf den Button „Request“ werden die Änderungen durchgeführt und der Erfolg oder Mißerfolg in der Statuszeile bekanntgegeben.

Danach kann durch die Buttons „Start“ und „Stop“ der Capture-Vorgang gestartet bzw. gestoppt werden.

Mit Hilfe der Checkbox „show output“ kann dynamisch ein Fenster eingeblendet werden, in dem der Inhalt der Pufferspeicher dargestellt wird.

Im oberen Teil des Dialog finden sich die Einstellungen zum Speichern der Inhalte der Pufferspeicher auf einem Massenspeicher. Durch Aktivieren der Checkbox „Number of buffers to write“ kann von vornherein die Anzahl der zu schreibenden Pufferspeicher begrenzt werden. Als Speicherformate stehen das „portable pixmap“-Format (PPM) und das proprietäre

„YUV“-Format zur Verfügung. Bilder im PPM-Format können mit nahezu allen Bildbearbeitungsprogrammen weiterverarbeitet werden, Bilder im YUV-Format können ohne Konvertierung sofort wieder für den Upload-Vorgang weiterverwendet werden. Die Wahl des Speicherformates erfolgt über die zwei Radiobuttons.

Die Wahl eines Zielverzeichnisses kann über einen Druck auf den Button „Choose directory“ graphisch erfolgen oder aber direkt in dem Eingabefeld vorgenommen werden.

Schließlich kann man einen Präfix für die Bildung der Dateinamen im Eingabefeld „filename prefix“ bestimmen. Die Dateien werden dann in aufsteigender Reihenfolge nach dem Schema „[Präfix]-[Bildnummer].[Formatkürzel]“ benannt.

Das Speichern der Puffer sowie die dazugehörigen Optionen können während eines laufenden Capture-Vorgangs verändert werden.

6.2 Upload-Vorgang

Abbildung 12 zeigt die Dialogseite „Upload“ des Programms „UplCap“.

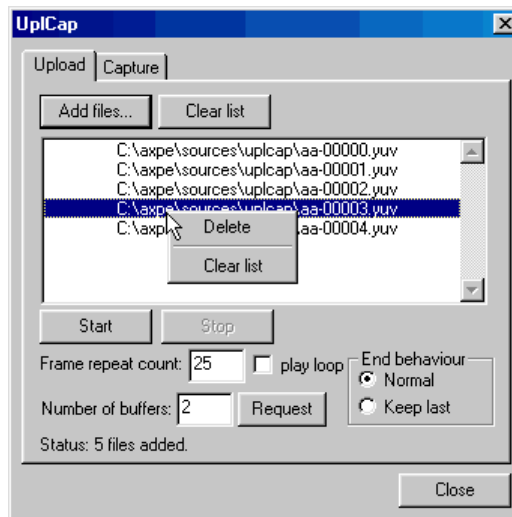


Abbildung 12: Dialogseite „Upload“

Dieser Dialog ermöglicht es, Bilder von einem Massenspeicher der Upload-Logik des Treibers zur Verfügung zu stellen und die diversen Parameter graphisch zu verändern.

Nach einem Druck auf den Button „Add files“ öffnet sich ein Dialog, der das Hinzufügen von einem oder mehreren Bilddateien ermöglicht. Eine Übersicht über die bereits erfassten Dateien bietet die Listbox in der Mitte des Dialoges.

Die Reihenfolge der Einträge innerhalb der Listbox kann per „drag and drop“ verändert werden. Während sich der Mauszeiger über einem Eintrag in der Listbox befindet, öffnet sich durch den Druck auf die rechte Maustaste ein Kontextmenü mit den jeweils passenden Optionen. Ein Druck auf den Button „Clear list“ löscht alle Einträge in der Listbox.

Das Eingabefeld „Frame repeat count“ gibt die Möglichkeit, die Anzahl der Wiederholungen eines Bildes zu verändern. Über die Radiobuttons der Gruppe „End behaviour“ kann das Verhalten der Upload-Logik bei einem Mangel an Upload-Puffern gesteuert werden (vgl. Abschnitt 5.3). Die Anzahl der verwendeten Upload-Puffer kann über das Eingabefeld „Number of buffers“ verändert werden. Alle diese Einstellungen werden durch einen Druck auf den Button „Request“ überprüft und wenn möglich ausgeführt.

Am unteren Rand des Dialoges befindet sich die Statuszeile, die Auskunft über das Ergebnis dieser Aktion und später über den laufenden Upload-Vorgang gibt.

Mit Hilfe der Buttons „Start“ und „Stop“ kann der Upload-Prozeß gestartet bzw. gestoppt werden. Durch einen Doppelklick auf einen Eintrag in der Listbox wird das entsprechende Bild in den nächsten freien Puffer geladen und dieser Puffer dann der Upload-Logik zur Verfügung gestellt.

Ist beim Starten des Upload-Prozesses hingegen die Checkbox „play loop“ aktiviert, werden die Puffer in aufsteigender Reihenfolge gefüllt und der Upload-Logik immer wieder nacheinander zur Verfügung gestellt. Dann ist ein interaktives Eingreifen in den Upload-Prozeß nicht möglich und die Listbox ist deaktiviert.

7 Ergebnisse und Ausblick

7.1 Integration in den bestehenden Treiber

Um die neue Funktionalität in den bestehenden Treiber zu integrieren, war eine Ergänzung und Änderung der Treiberarchitektur notwendig. Die in Abbildung 13 dunkel markierten Teile der Treiberarchitektur wurden für die Implementierung verändert bzw. hinzugefügt. Diese Veränderungen werden im folgenden kurz erläutert.

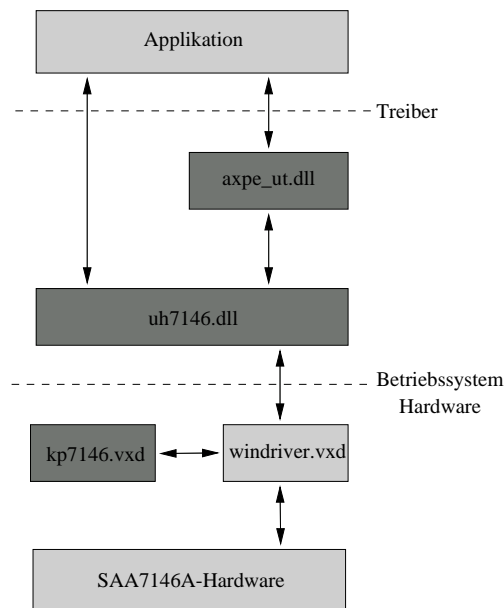


Abbildung 13: Neue Treiberarchitektur

Das *WinDriver*-Toolkit ermöglicht es, ein sogenanntes „Kernel-Plugin“ zu installieren, daß auf Windows 98-Kernelebene die Interruptverarbeitung durchführt. Damit war es möglich, die Interruptserviceroutinen für den Capture- und Upload-Vorgang tief im System zu verankern, um eine unverzügliche Behandlung auftretender Interrupts zu garantieren.

Aus diesem Grund wurde für alle zeitkritischen Funktionen das Kernel-Plugin `kp7146.vxd` geschaffen. Darin enthalten ist zusätzlich die Implementierung einer einfachen Zeiger-Queue mit den grundlegenden Queue-Operationen zum Einfügen von Elementen am Ende und Löschen von Elementen am Anfang der Queue.

Da auf die Funktionen von `kp7146.vxd` kein direkter Zugriff aus einer Applikation möglich ist, wurden Rumpffunktionen in `uh7146.dll` eingefügt, die die Parameter einer Applikation an das Kernel-Plugin weiterleiten und eventuelle Rückgabewerte zurückzugeben.

Weiterhin wurden die Initialisierungs- und Aufräumfunktionen von `uh7146.dll` um die Behandlung der Pufferspeicher für das den Capture- bzw. Upload-Vorgang erweitert.

In `axpe_ut.dll` wurde die Funktion `boot_yuv_upload` hinzugefügt, die die peripheren Komponenten auf der AxPe1280V-Demonstrationskarte so umprogrammiert, daß die Daten des Upload-Vorgangs entsprechend fließen können.

7.2 Robustheit der Erweiterungen

Die Erweiterungen sind so programmiert worden, daß benötigte Ressourcen vom Treiber selbst belegt werden und nicht von der Applikation bereitgestellt werden müssen. Dies gilt insbesondere für den Speicherplatz für die Puffer, der im Adreßraum der Applikation angefordert wird.

Falls es möglich ist, geben die Funktionen Hilfen in Form von zusätzlichen Variablen. Die Funktionen `saa7146_upload_buffers_request` bzw. `saa7146_capture_buffers_request` geben beispielsweise den Parameter `sizeimage` zurück, damit die Applikation die Größe der einzelnen Pufferspeicher genau kennt. Diese Information könnte die Applikation zwar auch alleine aus den übergebenen Parametern Höhe, Breite und Format berechnen, doch um Fehler zu vermeiden bzw. aufzudecken, gibt der Treiber diese Information explizit bekannt. In diesem Fall wird der Programmierer der Applikation darauf hingewiesen, sich Gedanken über die Größe der Speicherbereiche zu machen.

Sollte eine Applikation auf Grund eines eigenen Fehlers abstürzen, so hat dies keine Auswirkungen auf den Treiber. Da das Betriebssystem in jedem Fall das korrekte Entladen der verwendeten Bibliotheken vornimmt, kann in einer geeigneten Routine das korrekte Abschalten der Capture- und Upload-Logik selbst nach einem Programmabsturz erfolgen.

7.3 Echtzeitfähigkeit und Prozessorlast

Nachdem Pufferspeicher angefordert und die Capture- und Upload-Logik gestartet wurde, verbraucht die Verarbeitung durch den Treiber so gut wie keine Prozessorzeit. Das Umprogrammieren der Hardware zum Umschalten zwischen den verschiedenen Speichern erfolgt in der Interruptserviceroutine und benötigt nur wenige Taktzyklen.

Die Auslastung des Systems bei laufendem Capture- und Upload-Vorgang unterscheidet sich bei der Messung mit dem *Systemmonitor* von Windows 98 nicht von der Auslastung des Systems ohne die laufende Applikation. Die Echtzeitfähigkeit des gesamten Systems hängt damit komplett von der Verarbeitung der Daten durch die Applikation ab.

Sofern die periphere Hardware die Daten schnell genug verarbeiten, laden und speichern kann, ist eine Echtzeitverarbeitung möglich. Wie die einfache Rechnung aus Abschnitt 3 aber bereits zeigt, ist dies für anspruchsvollere Applikationen eigentlich nur durch geeignete Kompression der Daten möglich.

7.4 Einschränkungen

Die Struktur des Treibers in Zusammenhang mit dem *WinDriver*-Toolkit hat zur Folge, daß ein Capture-Prozeß bzw. Upload-Prozeß immer nur durch eine Applikation gesteuert werden kann. Es ist nicht möglich, daß zwei Applikationen getrennt auf die Capture- und Upload-Funktionalität des Treibers zugreifen.

Der Grund dafür liegt darin, daß das Kernel-Plugin `kp7146.vxd` nicht einmalig für jede AxPe 1280V-Demonstrationskarte im System gestartet wird, sondern jede aufrufende Applikation eine eigene Instanz davon erhält. Dieses hat zur Folge, daß auch jede Instanz einen eigenen Interrupthandler in die Bearbeitung einklinkt. Ein Interrupthandler ist aber verpflichtet, beim Auftreten eines Interrupts schnellstmöglich die Auslösebedingung in der Hardware zu bestätigen. Bei mehreren Interrupt-Handlern, die in einer nicht definierten Reihenfolge

vom Betriebssystem abgearbeitet werden, müsste also der „letzte“ Interrupthandler in dieser Kette die Bedingung bestätigen oder aber der „erste“ Interrupthandler alle weiteren benachrichtigen.

Da es weder möglich ist, den „letzten“ Interrupthandler zu bestimmen noch eine verlässliche Kommunikation zwischen den verschiedenen Instanzen des Kernel-Plugins herzustellen, bleibt nur die Möglichkeit, die gesamte Interruptbehandlung auf eine Instanz des Kernel-Plugins zu beschränken. Daher kann immer nur eine Applikation einen interruptbasierten Capture- oder Upload-Vorgang durchführen.

Zur Organisation der Pufferspeicher verwendet das Kernel-Plugin eine einfache Queue. Diese Queue wird sowohl durch Funktionsaufrufe der Applikation (z. B. `saa7146_capture_queue_buffer`) als auch durch den Interrupthandler manipuliert. Da der Interrupthandler privilegiert ausgeführt wird, ist es denkbar, daß ein Aufruf von `saa7146_capture_queue_buffer` innerhalb einer Queue-Manipulation vom Interrupthandler unterbrochen und in Folge dessen die Queue irregulär modifiziert wird. In der jetzigen Version des Treibers wird eine solche irreguläre Modifikation zwar bemerkt, aber nicht verhindert.

7.5 Demonstrationssoftware „UplCap“

Obwohl der Treiber durch die Bereitstellung von Mehrfachpuffern für den Capture- bzw. Upload-Prozeß eine gewisse Latenz bei der Weiterverarbeitung der Daten erlaubt, sollte sie doch so schnell wie möglich erfolgen.

Als praktikable Lösung wurde für „UplCap“ der Einsatz parallel arbeitender Threads gewählt. Je nach Anwendungsfall wird sowohl für den Capture- als auch den Upload-Prozeß ein eigener Verarbeitungsthread gestartet, der dann via `saa7146_capture_wait_frame` bzw. `saa7146_upload_wait_frame` wartet. Nachdem die Verarbeitung eines Puffers erfolgt ist und der Treiber den Thread wieder aufgeweckt hat, wird die parallel dazu laufende Applikation benachrichtigt. Diese ist dann für die eigentliche Verarbeitung der Daten zuständig.

Aber selbst wenn entweder nur die Capture-Funktionalität oder nur die Upload-Funktionalität des Treibers durch „UplCap“ benutzt wird, kann auf dem für die Entwicklung benutzten PC keinen Echtzeitdurchsatz für eine größere Anzahl von Bildern erreicht werden. Das System ist mit dem Lesen von bzw. Schreiben auf den Massenspeicher bereits vollständig ausgelastet. Die einzige Lösung ist eine geschickte Vor- bzw. Nachbearbeitung der Daten.

8 Zusammenfassung

Ziel dieser Studienarbeit war die Erweiterung des bestehenden Treibers um die Möglichkeit, die Videodaten der AxPe1280V-Prozessoren im Hauptspeicher des PCs ablegen zu können und bereits vorliegende, digitale Videodaten zu den AxPe1280V-Prozessoren transportieren zu können.

Es ist nun möglich, die früher nur per Overlay verfügbaren Videodaten in benutzerdefinierten Pufferspeichern im Hauptspeicher abzulegen. Des weiteren können im richtigen Format vorliegende Bilddaten an die beiden AxPe1280V-Prozessoren durchgereicht und damit weiterverarbeitet werden.

Mögliche Anwendungen für diese neuen Funktionen sind das Abspeichern der Videodaten für Vergleichszwecke, Archivierungen oder um daraus ein Demonstrationsvideo erstellen zu können. Das Einspeisen von bereits vorliegendem, digitalen Bildmaterial kann z. B. die Entwicklung bzw. Verbesserung der in den Videokodern benutzten Algorithmen ermöglichen.

Es wurde insbesondere darauf geachtet, eine klare und saubere Abstraktionsschicht zu definieren, um Applikationen einen sicheren und komfortablen Zugang zu den neuen Funktionen zur Verfügung zu stellen. Außerdem wurden die Erweiterungen so entworfen, daß nur ein möglichst geringer Verwaltungsaufwand und Verarbeitungsverlust auftritt.

Applikationen erhalten über die definierte Schnittstelle eine umfassende Kontrolle über die neuen Funktionen des Treibers.

Um die Funktionsfähigkeit in ansprechender Weise zu zeigen, wurde die Demonstrationssoftware „UplCap“ entwickelt. Die Implementierung des Programs und der graphischen Benutzeroberfläche erfolgte auf Basis der *Microsoft Foundation Classes 6.0* Klassenbibliothek.

„UplCap“ kann sowohl den Originaldatenstrom von der analogen Videosignalquelle als auch die durch die AxPe1280V-Prozessoren modifizierten Videodaten bildweise auf einem Massenspeicher ablegen. Zur Verifikation der Ergebnisse können die Videodaten in einem zusätzlichen Ausgabefenster auch visualisiert werden. Des weiteren können durch „UplCap“ bereits vorliegende, digitale Videodaten bildweise zu den AxPe1280V-Prozessoren transportiert werden. Vielfältige Wahlmöglichkeiten ermöglichen es, alle Optionen der neuen Funktionen des Treibers auszuprobieren.

Durch den Einsatz parallel arbeitender Threads erfolgt die Verarbeitung der eigentlichen Aufgaben unabhängig von der Bedienung der graphischen Benutzeroberfläche. „UplCap“ kann daher gut als Vorlage für andere Implementierungen auf Basis der *Microsoft Foundation Classes 6.0* Klassenbibliothek dienen.

Abbildungsverzeichnis

1	Koprozessorarchitektur des AxPe1280V	4
2	Overlaydarstellung eines Videosignals	5
3	Schematischer Aufbau der AxPe1280V-Demonstrationskarte	5
4	„control“-Applikation für die AxPe1280V-Demonstrationskarte	6
5	Bestehende Treiberarchitektur	7
6	Speicher aus Sicht einer Applikation und der Hardware	11
7	Pagetable zum Beispiel aus Abbildung 6	12
8	Schematischer Ablauf der Capture-Interruptserviceroutine	17
9	Videodatenfluß beim Upload-Vorgang	19
10	Schematischer Ablauf der Upload-Interruptserviceroutine	23
11	Dialogseite „Capture“	25
12	Dialogseite „Upload“	26
13	Neue Treiberarchitektur	28

Literatur

- [1] RALF KÖHLER. Entwicklung einer PCI-Einsteckkarte für den Videosignalprozessor AxPe1280V und Software-Anbindung an eine Simulationsumgebung. *Diplomarbeit am Laboratorium für Informationstechnologie der Universität Hannover* (1998).
- [2] PHILIPS SEMICONDUCTORS. SAA7146A - Multimedia bridge, high performance scaler and PCI circuit. *Product specification and data sheet* (1998).
- [3] JÖRG HILGENSTOCK. AxPe1280V Evaluation Board. *PDF-Dokumentation zur AxPe1280V-Demonstrationskarte* (1998).
- [4] JUNGO. WinDriver - a device driver developing toolkit. <http://www.jungo.com>.
- [5] BURKHART SCHNEIDERHEINZE. Funktionaler Entwurf und Verifikation eines DRAM-Controllers für den Einsatz in einem Videosignalprozessor. *Diplomarbeit am Laboratorium für Informationstechnologie der Universität Hannover* (1997).

Windows und *Windows 98*, *Visual C++* und *Microsoft Foundation Classes (MFC)* sind eingetragene Warenzeichen der *Microsoft Corporation*.

WinDriver ist ein eingetragenes Warenzeichen von *Jungo*.

A Anhang

A.1 Symbolische Fehlercodes

- -EACCES

Die Applikation darf nicht auf die Capture- oder Upload-Fähigkeiten des Treibers zugreifen. Dies deutet darauf hin, daß der SAA7146A-Handle nicht im richtigen Modus angefordert wurde.

- -EBUSY

Die Applikation versucht Pufferspeicher anzufordern oder freizugeben, obwohl die Verarbeitung noch nicht gestoppt wurde.

- -ENOMEM

Der Treiber hat nicht genug Speicher für das Anlegen der geforderten Anzahl an Pufferspeichern.

- -EDEADLOCK

Die Applikation versucht, entweder Pufferspeicher freizugeben, die nicht angefordert wurden oder neue Pufferspeicher anzufordern, obwohl die „alten“ Pufferspeicher noch nicht freigegeben wurden.

- -EINVAL

Die Pufferspeichernummer ist ungültig oder befindet sich außerhalb des zulässigen Bereichs.

- -EINUSE

Der Pufferspeicher ist bereits in Gebrauch.

- -NOBUFFERS

Es sind keine Pufferspeicher angefordert worden.

- -ETIMEOUT

Ein Timeout ist aufgetreten.

- -ENOTSTREAMING

Der Aufruf ist fehlgeschlagen, weil die Capture- oder Upload-Logik nicht aktiviert ist.

A.2 Funktionsaufrufe

A.2.1 Capture-Vorgang

```

/*
function: saa7146_capture_buffers_request
=====
this functions allocates buffers for the capture process.

parameters:
-----
*saa:          handle to a saa7146 to be used.
*number:       pointer to an unsigned integer, specifying how many buffers are going to be used.
                range: between 1 and SAA7146_MAX_BUFFERS.
*width:        pointer to an unsigned integer, specifying the width of the requested capture buffers.

```

```

range: between 0 and 768 (hardware limitation).
*height: pointer to an unsigned integer, specifying the height of the requested capture buffers.
range: between 0 and 576 (hardware limitation).
*format: pointer to an unsigned int, specifying the format to be used for capture.
possible values: RGB16_COMPOSED, RGB24_COMPOSED, RGB32_COMPOSED, YUV422_COMPOSED
*port: pointer to an unsigned integer, specifying the port to be used for capturing.
possible values: 0 (= port a) and 1 (= port b)
*ptrs: pointer to an array of u32*s of at least '*number' elements. this array will receive
the user-space adresses of the beginning of each capture-buffer
*sizeimage: pointer to an unsigned long. this value will receive the size of a picture in bytes.

return values: (see above)
-----
0, -EACCES, -EBUSY, -ENOMEM, -EDEADLOCK

remarks:
-----
invalid values for 'number', 'width', 'height', 'format' and 'port' do not lead to an error.
instead, the values are set to the minumum or maximum of the corresponding setting and are
passed back to the application.
*/
int saa7146_capture_buffers_request(struct saa7146* saa, unsigned int* number,
    unsigned int* width, unsigned int* height,
    unsigned int* format, unsigned int* port,
    u32* ptrs[], unsigned long* sizeimage);

/*
function: saa7146_capture_buffers_release
=====
this function releases any buffers that have been requested.

parameters:
-----
*saa: handle to a saa7146 to be used.

return values: (see above)
-----
0, -EACCES, -EBUSY

remarks:
-----
calling this function without having allocated any capture buffers before does not produce an error.
*/
int saa7146_capture_buffers_release(struct saa7146* saa);

/*
function: saa7146_capture_queue_buffer
=====
this function queues the specified buffer into the driver's capture logic.

parameters:
-----
*saa: handle to a saa7146 to be used.

buffer: unsigned integer, specifying the buffer to be used for capturing.
range: between 0 and SAA7146_MAX_BUFFERS-1.

return values: (see above)
-----
0, -EACCES, -EBUSY, -EINVAL, -EINUSE, -ENOBUFFERS

remarks:
-----
none.
*/
int saa7146_capture_queue_buffer(struct saa7146* saa, unsigned int buffer);

/*
function: saa7146_capture_dequeue_buffer
=====
this function dequeues the next ready buffer from the driver's capture logic.

parameters:

```

```
-----
*saa:   handle to a saa7146 to be used.

return values:
-----
1) 0 to SAA7146_MAX_BUFFERS-1: the corresponding buffer has been successfully dequeued
2) -EACCES, -ENOBUFFERS, -EINVAL

remarks:
-----
none.
*/
int saa7146_capture_dequeue_buffer(struct saa7146* saa);

/*
function: saa7146_capture_stream_on
=====
this function turns the streaming capture on.

parameters:
-----
*saa:   handle to a saa7146 to be used.

return values:
-----
0, -EACCES, -ENOBUFFERS

remarks:
-----
calling this function while streaming capture is already in progress does not produce an error.
*/
int saa7146_capture_stream_on(struct saa7146* saa);

/*
function: saa7146_capture_stream_off
=====
this function turns the streaming capture off.

parameters:
-----
*saa:   handle to a saa7146 to be used.

return values:
-----
0, -EACCES, -ENOBUFFERS

remarks:
-----
calling this function while streaming capture is already off does not produce an error.
*/
int saa7146_capture_stream_off(struct saa7146* saa);

/*
function: saa7146_capture_wait_frame
=====
this function will go to sleep until a capture buffer is ready or a timeout occurs.

parameters:
-----
*saa:           handle to a saa7146 to be used.

timeout:       unsigned long, specifying an timeout value in milliseconds.
                possible values: any, 0 means unlimited.

return values:
-----
0, -EACCES, -ETIMEOUT, -ENOTSTREAMING

remarks:
-----
none.
*/
int saa7146_capture_wait_frame(struct saa7146* saa, unsigned long timeout);
```

```

/*
function: saa7146_capture_get_statistics
=====
this function provides some statistics for the capture logic.

parameters:
-----
*saa:   handle to a saa7146 to be used.

*stat:  pointer to a struct statistics.

return values:
-----
0, -EACCES

remarks:
-----
the statistic is resetted when "saa7146_capture_stream_on_is" called.
*/
int     saa7146_capture_get_statistics(struct saa7146* saa, struct statistics* stat);

```

A.2.2 Upload-Vorgang

```

/*
function: saa7146_upload_buffers_request
=====
this functions allocates buffers for the upload process.

parameters:
-----
*saa:           handle to a saa7146 to be used.
*number:        pointer to an unsigned integer, specifying how many buffers are going to be used.
                 range: between 1 and SAA7146_MAX_BUFFERS.
*width:         pointer to an unsigned integer, specifying the width of the requested upload buffers.
                 range: between 0 and 768 (hardware limitation).
*height:        pointer to an unsigned integer, specifying the height of the requested upload buffers.
                 range: between 0 and 576 (hardware limitation).
*port_sync:     pointer to an unsigned integer, specifying the port that provides the sync information.
                 possible values: 0 (= port a) and 1 (= port b)
*port_upload:   pointer to an unsigned integer, specifying the port to which the video data is send.
                 possible values: 0 (= port a) and 1 (= port b)
*ptrs:          pointer to an array of u32*s of at least '*number' elements. this array will receive
                 the user-space adresses of the beginning of each caputure-buffer
*sizeimage:     pointer to an unsigned long. this value will receive the size of a picture in bytes.

return values: (see above)
-----
0, -EACCES, -EBUSY, -ENOMEM, -EDEADLOCK

remarks:
-----
invalid values for 'number', 'width', 'height', 'format' and 'port' do not lead to an error.
instead, the values are set to the minumum or maximum of the corresponding setting and are
passed back to the application.
*/
int saa7146_upload_buffers_request(struct saa7146* saa,
                                   unsigned int* number, unsigned int* width,
                                   unsigned int* height, unsigned int* port_sync,
                                   unsigned int* port_upload, u32* ptrs[], unsigned long* sizeimage);

/*
function: saa7146_upload_buffers_release
=====
this function releases any upload buffers that have been requested.

parameters:
-----
*saa:   handle to a saa7146 to be used.

return values: (see above)

```

```

-----
0, -EACCES, -EBUSY

remarks:
-----
calling this function without having allocated any upload buffers before does not generate an error.
*/
int saa7146_upload_buffers_release(struct saa7146* saa);

/*
function: saa7146_upload_queue_buffer
=====
this function queues the specified buffer into the driver's upload logic.

parameters:
-----
*saa:   handle to a saa7146 to be used.

buffer: unsigned integer, specifying the buffer to be used for uploading.
        range: between 0 and SAA7146_MAX_BUFFERS-1.

return values: (see above)
-----
        0, -EACCES, -EBUSY, -EINVAL, -EINUSE, -ENOBUFFERS

remarks:
-----
none.
*/
int saa7146_upload_queue_buffer(struct saa7146* saa, unsigned int buffer);

/*
function: saa7146_upload_dequeue_buffer
=====
this function dequeues the next ready buffer from the driver's upload logic.

parameters:
-----
*saa:   handle to a saa7146 to be used.

return values:
-----
1) 0 to SAA7146_MAX_BUFFERS-1: the corresponding buffer has been successfully dequeued
2) -EACCES, -ENOBUFFERS, -EINVAL

remarks:
-----
none.
*/
int saa7146_upload_dequeue_buffer(struct saa7146* saa);

/*
function: saa7146_upload_stream_on
=====
this function turns the streaming upload on.

parameters:
-----
*saa:           handle to a saa7146 to be used.
repeat_send:   unsigned integer, specifying the driver's behaviour when the upload queues gets empty.
                possible values: UPLOAD_REPEAT_SEND_NORMAL, UPLOAD_REPEAT_SEND_KEEP_LAST
repeat_count:  unsigned integer, specifying how often each buffer will be repeated for upload.
                0 means no repetition, e.g. only one upload per buffer.

return values:
-----
0, -EACCES, -ENOBUFFERS

remarks:
-----
calling this function while streaming upload is already in progress does not produce an error,
but the new values for "repeat_send" and "repeat_count" are ignored, though.
*/

```

```
int saa7146_upload_stream_on(struct saa7146* saa, unsigned int repeat_send, unsigned int repeat_count);

/*
function: saa7146_upload_stream_off
=====
this function turns the streaming upload off.

parameters:
-----
*saa:   handle to a saa7146 to be used.

return values:
-----
0, -EACCES, -ENOBUFFERS

remarks:
-----
calling this function while streaming upload is already off does not produce an error.
*/
int saa7146_upload_stream_off(struct saa7146* saa);

/*
function: saa7146_upload_wait_frame
=====
this function will go to sleep until an upload buffer is ready or a timeout occurs.

parameters:
-----
*saa:           handle to a saa7146 to be used.

timeout:        unsinged long, specifying an timeout value in milliseconds.
                 possible values: any, 0 means unlimited.

return values:
-----
0, -EACCES, -ETIMEOUT, -ENOTSTREAMING

remarks:
-----
none.
*/
int saa7146_upload_wait_frame(struct saa7146* saa, unsigned long timeout);

/*
function: saa7146_upload_get_statistics
=====
this function provides some statistics for the upload logic.

parameters:
-----
*saa:   handle to a saa7146 to be used.

*stat:  pointer to a struct statistics.

return values:
-----
0, -EACCES

remarks:
-----
the statistic is resetted when "saa7146_upload_stream_on_is" called.
*/
int   saa7146_upload_get_statistics(struct saa7146* saa, struct statistics* stat);
```