Entwurf und Implementierung eines Framegrabber-Treibers für das Java Betriebssystem JX

Diplomarbeit im Fach Informatik

vorgelegt von

Andreas Heiduk

geboren am 28. September 1972 in Schwandorf

Angefertigt am

Lehrstuhl 4 der Informatik,

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	*Prof. Dr. F. Hofmann*
	Dipl.-Inf. Michael Golm
Beginn der Arbeit:	*1. Oktober 2000*
Abgabe der Arbeit:	*2. April 2001*

Ich erkläre hiermit, daß ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebener Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 2. April 2001

// Kurzfassung

Im Rahmen dieser Arbeit wird ein Treiber für die Framegrabber-Karte WinTV-PCI-FM der Firma Hauppauge für das Java-Betriebssystem JX entworfen und implementiert. Das JX-System eignet sich u.A. für die schnelle Entwicklung von "Embedded"-Geräten, da es auch auf kleineren Rechner läuft. Der auf der WinTV-Karte verwendete BT878 kommt dem entgegen, indem er die Videodaten ohne Hilfe des Prozessors in den Speicher schreiben kann und so keinen leistungsfähigen Prozessor benötigt.

Die Anwendungs-Schnittstelle des Treibers ist nicht auf den BT878 der WinTV-Karte festgelegt, sondern ist so flexibel gehalten, daß auch Produkte anderer Hersteller unterstützt werden können.

Durch die Implementierung des Treibers in Java wird außerdem gezeigt, daß die Anwendung objektorientierter Techniken auch bei Gerätetreibern sinnvoll ist und zu einem flexiblen, leicht erweiterbaren Treiberkonzept führt.

Um die Funktionalität des Treibers bzw. der entworfenen Schnittstelle zu verifizieren, werden einige typische Anwendungsfälle untersucht und prototypisch implementiert.

Abtract

This work describes the design and implementation of a device driver for the framegrabber card WinTV-PCI-FM from Hauppauge running with the Java operating system JX. The JX system can be used - among other things - to rapidly develop small embedded systems. The chip used on the WinTV card, a BT878 from Brooktree, supports this by writing the captured video data without needing support by the processor. Therefore high performing processors are not necessary for such a system.

The developed interface is not restricted to the BT878, it is flexible enough to support products from other vendors.

The implementation also shows that the use of object oriented techniques makes sense even in the development of a low level device driver and results in a highly flexible and extensible driver concept.

In order to verify the driver and it's interface some typical applications of a framegrabber card are examined and implemented in a prototypical manner

Inhaltsverzeichnis

Verzeichnis der Bilder

Verzeichnis der Tabellen

1 Einleitung

Normalerweise verbindet man mit dem Stichwort "Multimedia" sehr rechenintensive Anwendungen, die große Datenmengen bewegen müssen und dementsprechend auch sehr rechenstarke Prozessoren benötigen. Diese Anwendungen werden daher in der Regel mit hochoptimierten Routinen zur Bearbeitung der Datenmengen ausgestattet.

Doch es geht auch anders.

Moderne Hardware kann mit spezialisierten Prozessoren die CPU in vielen Bereichen entlasten, so daß die CPU hauptsächlich nur zur Steuerung der Spezial-Prozessoren benötigt wird. Folglich ist es dann auch nicht mehr notwendig, sehr viel Entwicklungsaufwand in die Optimierung des Programmes zu investieren und man kann stattdessen zur Entwicklung eine Hochsprache verwenden, die es im Gegenzug erlaubt, sehr schnell Anwendungen zu entwickeln.

Java bietet sich für diesen Zweck in vielen Bereichen an, da es sehr - abgesehen von der Ausführungsgeschwindigkeit - viele Vorteile hat, die einen schnellen Entwicklungszyklus erlauben. So ist Java sehr einfach zu erlernen, und durch seine Objektorientierung flexibel erweiterbar. Außerdem ist es stark standardisiert und daher auch sehr portabel.

Im Rahmen dieser Arbeit wird daher untersucht, ob es tatsächlich möglich und praktikabel ist, Java und speziell das Java-Betriebssystem JX [JXOS] für zeitkritische Multimedia-Anwendungen zu verwenden. Es soll ein Treiber entworfen und implementiert werden, der eine Hauppauge WinTV-PCI Framegrabberkarte dazu verwendet, analoge Videobilder zu digitalisieren und für weitere Anwendungen verfügbar zu machen. Mittels prototypischer Anwendungen sollen einige Anwendungsfälle implementiert werden, um die praktische Verwertbarkeit sicherzustellen.

Der so entwickelte Treiber könnte dann z.B. zusammen mit der Framegrabber-Hardware und beispielsweise einer Netzwerkkarte dazu, dienen ein kompaktes Gerät herzustellen, das auf einfache Art und Weise Videokonferenzen über ein Computernetzwerk erlaubt.

Struktur der Arbeit

In Kapitel 2 werden die Grundlagen analoger Videoübertragung erklärt, die für das Verständnis des [BT878], der das Herz der WinTV Karte bildet, notwendig sind. Dieses Kapitel soll daher nicht als generelle Einführung in die Fernsehtechnik verstanden werden.

Im Anschluß daran wird als weiteres Grundlagen-Kapitel in Kapitel 3 die Arbeitsweise des BT878 dargelegt. Die Arbeitsweise dieses Chips prägt naturgemäß den Entwurf des Treibers und auch die später zugängliche Schnittstelle zu den Anwendungsprogrammen.

Nachdem die Grundlagen in den beiden vorherigen Kapiteln gelegt wurden, kann in Kapitel 4 mit dem Entwurf des Treibers begonnen werden. Nach der Definition der Entwurfsziele wird in diesem Kapitel der Digitalisierungsprozeß des BT878 so abstrahiert, daß die entstehende Schnittstelle auch für andere Videocapture-Produkte anwendbar ist.

In Kapitel 5 wird schließlich die interne Struktur des Treibers entworfen, welche die Verbindung der Anwendungs-Schnittstelle aus dem vorherigen Kapitel mit der zugrundeliegenden Hardware herstellt.

Nach dem Entwurf des Treibers werden in Kapitel 6 prototypische Anwendungen des Treibers entwickelt und Besonderheiten in der jeweiligen Implementierung besprochen. Es wird eine Anwendung entwickelt, die das Videobild auf dem Monitor in einem Fenster darstellen kann, wobei das Videobild von weiteren Fenstern überlappt werden kann. Außerdem wird die Übertragung der Videodaten über ein Netzwerk besprochen.

Anschließend wird in Kapitel 7 erläutert, wie der Treiber erweitert werden kann. Dies geschieht u.a. am Beispiel von Framegrabber-Karten anderer Hersteller.

Kapitel 8 schließlich faßt die Ergebnisse der Arbeit zusammen.

2 Grundlagen und Begriffe für analoge Videoübertragung

In diesem Kapitel sollen einige Grundbegriffe der analogen Videoübertragung dargestellt werden, die für die Programmierung der Framegrabber-Karte notwendig sind.

2.1 Analoge Videosignale im Allgemeinen

Das Format eines analogen Videosignales wird weitgehend durch die Darstellungsweise im Endverbraucher, dem TV-Gerät, bestimmt. Im TV-Gerät werden mit einem Elektronenstrahl die Farbstoffe der Bildfläche zum Leuchten gebracht, um so das Bild darzustellen. Der Elektronenstrahl fährt dabei Zeile für Zeile ab, wobei er am Ende einer Zeile "ausgetastet" wird und an den Anfang der nächsten Zeile springt. Beim **Austasten** wird der Elektronenstrahl ausgeschaltet, um beim Rücklauf die Farbschicht auf der Bildröhre nicht weiter zum Leuchten anzuregen. Wurde auf diese Wiese ein einzelnes Bild Zeile für Zeile dargestellt, wird der Elektronenstrahl wieder ausgetastet und an den Anfang der ersten Zeile zurückgestellt.

Obwohl also immer von einem analogen Videosignal gesprochen wird, ist das Signal aus technischen Gründen nicht mehr in allen Dimensionen (Höhe, Breite und Zeit) analog. Vielmehr wurde es bereits in einzelne Zeilen diskretisiert. Auch entlang der Zeitachse erfolgt eine Diskretisierung, da die einzelnen Bilder nacheinander übertragen werden, um so den Anschein eines flüssig bewegten Bildes zu erzeugen. Der einzige analoge Anteil ist und bleibt also der Inhalt einer Bildzeile, den der Elektronenstrahl ohne Austastung darstellen kann.

Bei der Übertragung eines Videosignales muß also die Information über die Austastlücken und den Beginn eines neuen Bildes in das Bildsignal mit integriert werden (s. Bild 1). Dies erreicht man, indem man die Spannung des Videosignales unter einen bestimmten Schwellenwert, den **Schwarzwert** (Black Level) fallen läßt, wodurch die Austastung des

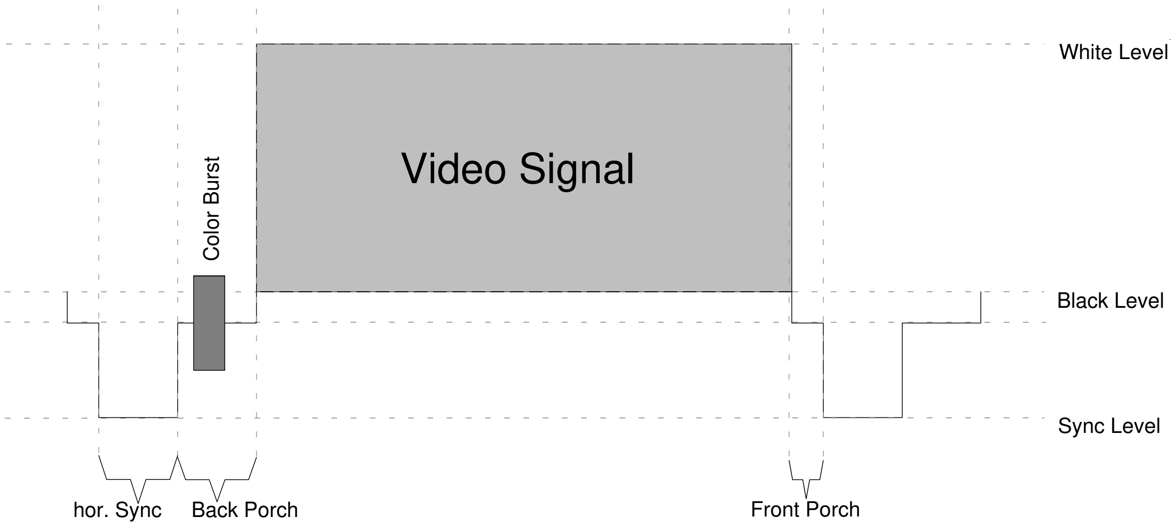

Bild 1: Zeile eines analogen Videosignales

Elektronenstrahls ausgelöst wird. Eine weitere Absenkung des Signales veranlaßt den Zeilenrücksprung des Elektronenstrahls. Dieser Vorgang wird als **Zeilensynchronisation** bezeichnet. Nachdem das Signal vom Synchronisierungswert auf den Schwarzwert zurückgeht und bevor das analoge Zeilensignal beginnt, wird bei einem farbigen Videosignal ein sogenannter **Color Burst** eingefügt, mit dem die Empfängerlogik ein Graustufensignal von einem Farbsignal unterscheiden kann. Die Zeit zwischen dem Ende des Synchronisierungssignales und dem Beginn des Bildsignales wird **hintere Schwarzschulter** (Back-Porch) genannt. Analog dazu bezeichnet die **vordere Schwarzschulter** (Front-Porch) die Zeit nach dem analogen Bildsignal und dem Beginn des Synchronisierungssignales.

Die bisherige Darstellung ging, abgesehen vom Color-Burst, davon aus, daß das Videosignal kein Farbsignal, sondern nur ein Graustufensignal ist. Um Farbe darstellen zu können, benötigt man aber drei unabhängige Signale. Die Bedeutung der drei Farbsignale hängt dabei vom gewählten Farbmodell ab. So können die drei Kanäle die Rot-, Grün- und Blau-Intensität eines Bildpunktes angeben. Diese im Computerbereich sehr verbreitete Darstellung wird aber nicht im Videobereich verwendet. Dort werden den drei Farbkanälen meistens Helligkeit, Kontrast und Farbe zugeordnet. Wie diese drei Kanäle in das Rot-,Grün-Blau-Farbmodell umgerechnet werden, kann man ([Fol91], S. 574ff) entnehmen.

Das verwendete Farbmodell hat den Vorteil, daß das Signal auch mit Schwarz-Weiß Geräten noch dargestellt werden kann, da diese nur den Helligkeitsanteil verwenden. Um nun die übrigen zwei Farbkanäle neben dem Helligkeitskanal im analogen Videosignal unterzubringen, werden diese auf das Helligkeitssignal mit Hilfe einer Trägerfrequenz aufmoduliert (s. Bild 2). Diese Trägerfrequenz wird **Farbträger** (Subcarrier) genannt und mit $\mathbf{f_{SC}}$ bezeichnet. Da die beiden Farbsignale mit nur einer Frequenz aufmoduliert sind, muß natürlich die Bandbreite eines Signales größer sein als die des zweiten Farbsignales.

Eine weitere Eigenschaft aller gängigen Videoformate ist die Übertragung von **Halbbildern**, d.h. es werden nicht in jedem Bild (von einem vertikalen Synchronisierungsimpuls bis zum nächsten) alle Bildzeilen übertragen. Stattdessen werden im ersten Halbbild alle ungeraden Bildzeilen (Zeile 1, 3, 5, ...) gesendet und im zweiten Halbbild alle geraden Bildzeilen (Zeile 2, 4, 6, ...). Dadurch wird die benötigte Bandbreite des Videosignales bei konstanter Bildwiederholrate halbiert. Leider hat dieses Vorgehen zudem den Nachteil, daß auch bei der Abtastung des Bildes zwischen erstem und zweitem Halbbild ein kleiner zeitlicher Unterschied auftritt. Dadurch ist es nicht möglich, beide Halbbilder auf einfache Art und Weise zu kombinieren, wenn man Standbilder betrachten will. Bei

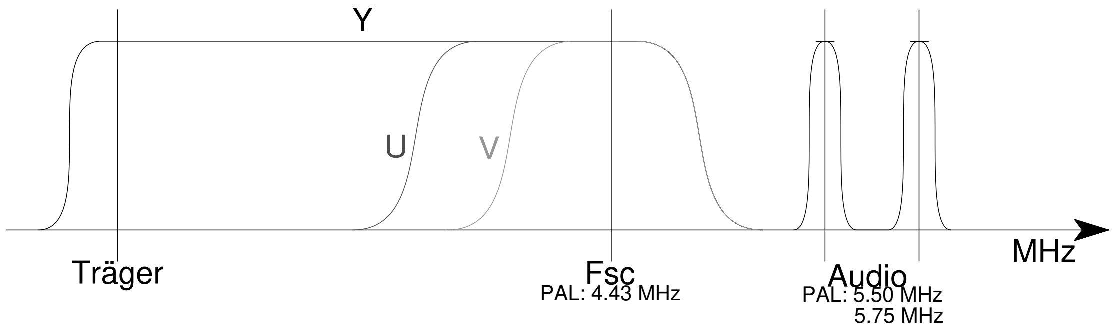

Bild 2: Frequenzzerlegung einer Videobildzeile

bewegten Bildern ist das ohne weiteres möglich, da das Auge den kleinen Unterschied so schnell nicht wahrnehmen kann.

2.2 TV-Normen

Die bisherige Darstellung von Videoformaten ist sehr generell. Leider gibt es in der Praxis eine Vielzahl von Videoformaten, die sich an das dargestellte Muster anlehnen, die sich aber in der konkreten Wahl der Parameter und auch in einigen fundamentalen Details unterscheiden. Dies hat nicht nur technische, sondern durchaus auch politische Gründe.

Die erste TV-Norm, die auch Farbe ermöglichte, war das amerikanische **NTSC**-Format (National Television System Comittee), welches 1953 eingeführt wurde. Eines der inhärenten Probleme dieses Formates ist, daß durch leichte Phasenverschiebungen der Farbträgerfrequenz falsche Farben angezeigt werden. Dies brachte dem System auch den Spitznamen "**N**ever **T**he **S**ame **C**olor" ein.

Das deutsche **PAL**-System (Phase Alternating Line) wurde 1963 so entworfen, daß Farbfehler, wie sie beim NTSC-System auftreten, nicht vorkommen können. Im weiteren wurde gegenüber NTSC die Anzahl der Zeilen etwas erhöht und gleichzeitig die Bildwiederholfrequenz von 60 auf 50 Hz erniedrigt.

Das französische **SECAM**-System wurde in etwa zeitgleich mit dem PAL-System entwickelt und wollte ebenfalls die Fehler von NTSC ausbügeln. Um Farbfehler zu vermeiden, wurde nicht ein Farbträger verwendet, sondern zwei, die jeweils separat die beiden Farbkanäle modulieren. Die Auflösung und die Bildwiederholfrequenz ist mit dem PAL-System identisch.

Ausgehend von diesen drei Grundverfahren wurden weltweit etliche Variationen eingeführt, die sich hauptsächlich durch die Auflösung, die Wahl des Farbträgers und die Bildwiederholfrequenz unterscheiden. Weitere Unterschiede treten in der Hochfrequenz-Modulation auf, die benutzt wird, um mehrere Videosignale gleichzeitig mit Funkwellen oder über Breitbandkabel zu übertragen.

2.3 Relevante Parameter eines Videosignales

Aus den beiden vorherigen Abschnitten ist klar ersichtlich, daß zur Beschreibung eines Videoformates etliche Parameter notwendig sind. Zum Glück schränkt sich die Anzahl der Parameter in etlichen Details stark ein, da der Decoder-Chip viele Dinge automatisch erkennen und verarbeiten kann. Welche Parameter müssen nun berücksichtigt werden?

- Frequenz des Farbträgers $\mathbf{f_{SC}}$ [MHz]
- Anzahl der sichtbaren und unsichtbaren Bildzeilen
- Anzahl der Bildzeilen vor dem sichtbaren Bildbereich
- Zeitdauer einer Bildzeile [µs]
- Zeitdauer des sichtbaren Signales innerhalb einer Bildzeile [µs]

- Zeitdauer vom Anfang einer Bildzeile bis zum Anfang des sichbaren Bereiches (Back Porch) [µs]
- Bildwiederholfrequenz [Hz]

In folgender Tabelle sind die erforderlichen Werte für die Video-Normen PAL und NTSC aufgeführt.

	NTSC-M	**PAL-B/G/H**	
Farbträger	3.58	4.43	MHz
Zeilenanzahl gesamt	525	625	
sichtbar	480	576	
vor dem Bild	24	32	
Dauer einer Zeile	63.5	64	µs
Signaldauer	52	52	µs
Back Porch	10.8	12	µs

Tabelle 1: Parameter von NTSC und PAL

Der Vollständigkeit halber sei noch erwähnt, daß die oben aufgeführten Parameter natürlich nicht ausreichen, ein Videosignal formal vollständig zu beschreiben. Zu den weiteren Parametern eines Videosignales zählen:

- Einbettung der vertikalen Synchronisierungssignale in das Videosignal.
- Unterscheidung der Halbbilder anhand von eingebetteten Phasensignalen.
  So werden z.B. im PAL System 8 unterschiedliche Phasen verwendet.
- Anpassung unterschiedlicher Farbmodelle.
- Modulierung als HF-Signal zur Übertragung über Funk und Kabel

Auf diese Größen soll nicht weiter eingegangen werden, da sie der Decoderchip mit Hilfe der in der Tabelle aufgeführten Größen selbständig decodieren und weiterverarbeiten kann. Eine detailliertere Beschreibung dieses Prozesses erfolgt im nächsten Kapitel.

2.4 Übertragung von Videosignalen

Unabhängig von der verwendeten Videonorm gibt es verschiedene Möglichkeiten, Videosignale zu übertragen.

Das **Composite**-Signal entspricht einer direkten Übertragung des zuvor beschriebenen Videosignales. Zur Übertragung reicht ein einadriges Kabel, welches meist mit Cinch-Steckern versehen ist.

Dagegen sind beim **Component**-Signal zwei Adern zur Übertragung notwendig, weil hier der Farbanteil unabhängig vom Helligkeitssignal übertragen wird, wodurch vor allem bei mehreren hintereinander folgenden Bearbeitungsschritten die Qualitätsverluste nicht so gravierend sind. Entsprechende Kabel sind entweder zweiadrige Cinch-Kabel oder mehradrige Kabel mit **SVideo**-Stecker.

Soll das Videosignal über längere Strecken transportiert oder an mehrere Empfänger verteilt werden, so wird das Composite-Signal noch mit einem **Hochfrequenz-Träger** moduliert und anschließend entweder in ein Breitbandkabel eingespeist oder über Rundfunk gesendet. Dieses **HF-**Signal hat den Vorteil, daß man mehrere unabhängige Übertragungskanäle auf dem gleichen Medium hat.

Diese kleine Einführung in die Grundlagen analoger Videodarstellung soll genügen, um die Arbeitsweise des BT878-Chips zu verstehen, ohne allerdings zu tief in die Materie einzusteigen.

3 Der BT878 -Chip

Nachdem im vorherigen Kapitel die Grundlagen für analoges Video erklärt wurden, soll in diesem Kapitel die Arbeitsweise des Framegrabber-Chips BT878 dargelegt werden.

Ziel dieser Arbeit ist die Entwicklung eines Treibers für die Videocapture-Karte WinTV-FM der Firma Hauppauge. Diese Karten haben als Herzstück den Capture-Chip BT878, der von der Firma Brooktree entwickelt wurde[1].

Der Framegrabber-Chip BT878 (vgl. [BT878]) und seine Verwandten haben sich inzwischen zum Standardchip bei der Digitalisierung von analogen Videosignalen entwickelt. Dies verdankt er seiner Flexibilität und seinen zahlreichen Fähigkeiten, von denen die wichtigsten hier erwähnt werden sollen:

- **Hochintegrierter Chip:** Der BT878 Chip enthält sämtliche Logik, um ein analoges Videosignal zu digitalisieren und in den Hauptspeicher zu schreiben. Daher werden neben dem BT878 nur wenige weitere Bauteile benötigt, um eine Framegrabber-Karte herzustellen[2].
- **Unterstützung zahlreicher Videonormen:** Der BT878 unterstützt neben den "Grundnormen" wie NTSC, PAL und SECAM auch einige Abarten und Mischformen.
- **PCI Busmaster-Zugriff:** Die digitalisierten Daten werden über das integrierte PCI Interface in den Speicher geschrieben, ohne die CPU dabei zu belasten.
- **Video-Overlay mit Clipping**: Die digitalisierten Daten können direkt in den Speicher der Grafikkarte geschrieben werden. Dabei ist es möglich, Fenster oder Popup-Menüs, die das Videobild ganz oder teilweise überdecken, nicht zu überschreiben (Clipping). Auch dazu ist keine CPU-Unterstützung notwendig.
- **Integrierter I^2C-Bus:** Mit dem I^2C-Bus von Philips (vgl. [Phil00]) ist es sehr einfach möglich, weitere Komponenten auf der Framegrabber-Karte anzusteuern. Dies wird z.B. häufig für einen TV-Tuner verwendet, der das HF-Signal von der Antenne demoduliert und in ein Videosignal umwandelt.
- **Zahlreiche Modifikationen des Bildes:** Neben den üblichen Einstellungen wie Helligkeit und Kontrast können auch viele weitere Modifikationen und Filter eingeschaltet werden wie z.B. Skalierung des Bildes, die Entfernung der Gamma-Korrektur und die Interpolation benachbarter Bildpunkte.

---

[1] Brooktree wurde von Rockwell aufgekauft und inzwischen an Conexant weiterverkauft.

[2] Dadurch ergibt sich auch der Vorteil, daß sich die Karten der verschiedenen Hersteller im Videobereich recht wenig unterscheiden. Um eine neue Karte zu unterstützen sind daher nur wenige Modifikationen am Treiber notwendig.

Natürlich ist die Liste der Features des BT878 bedeutend länger, aber die hier erwähnten Möglichkeiten sind die wichtigsten und werden auch später bei der Implementierung des Treibers berücksichtigt.

Nach diesem groben Überblick über die Möglichkeiten des Chips soll im Detail erklärt werden, wie die Digitalisierung funktioniert, da es erst danach möglich ist, mit dem Entwurf des Treibers zu beginnen.

Registerzugriff

Wie jedes PCI-Gerät haben auch Karten mit dem BT878-Chip einen sogenannten "PCI-Header", dessen Aufbau in der PCI-Spezifikation genormt ist und auch in ([Shan00], S. 351ff) beschrieben wurde. Der PCI-Header liegt in einem besonderen Adressraum und kann über den PCI-Bus nur mit speziellen Konfigurations-Zugriffen gelesen oder beschrieben werden (vgl. [Shan00], S. 317ff).

Ein Register im PC-Header beinhaltet die Basisadresse für die übrigen Register des BT878. Die Register, die den Vorgang des Video-Capture kontrollieren, sind also von der CPU aus mit einfachen Speicherzugriffen zu erreichen. Für den üblichen Betrieb des Chips sind die Register im PCI-Header nicht mehr notwendig.

Auswahl der Videoquelle

Der erste Schritt bei der Digitalisierung ist die die Auswahl der richtigen Videoquelle. Der BT878 bietet einen integrierten Dreiwege-Multiplexer an. Wie die einzelnen Eingänge belegt sind, hängt natürlich stark von der jeweiligen Karte ab. Normalerweise ist an einem der Eingänge ein TV-Tuner angeschlossen, der über den integrierten I^2C-Bus gesteuert wird. An einem anderen Eingang ist üblicherweise eine SVideo Buchse angeschlossen. Mit Hilfe eines kleinen Adapters kann an diesem SVideo-Eingang auch ein normales Composite-Videosignal eingespeist werden.

Videodecoder

Nachdem die Videoquelle eingestellt wurde, ist es notwenig, ein wenig zu warten, bis sich der Videodecoder auf die horizontalen und vertikalen Sychronisierungssignale eingestellt hat. Danach wird das analoge Signal sofort digitalisiert, so daß alle folgenden Schritte ausschließlich auf den Digitaldaten erfolgen. Der Videodecoder muß dazu entsprechend dem gewünschten Videoformat konfiguriert werden. Da der Chip die meisten Einstellungen automatisch erkennt, genügt es, einen passenden Wert in das FORMAT-Register zu schreiben. Außerdem muß man die Frequenz des zur Videonorm passenden Farbträgers Fsc einstellen.

Farbseparation

Nach der Digitalisierung des Videosignales erfolgt die Aufspaltung des Signales in die drei Farbkanäle. Dazu wird zuerst mit Hilfe der schon zuvor eingestellten Farbträgerfrequenz Fsc das Farbsignal aus dem Schwarz-Weiß-Signal entfernt. Die beiden Datenströme werden mit "Luma" (Helligkeit, kurz Y) und "Chroma" (Farbanteil, kurz UV) bezeichnet. In einem zweiten Schritt wird je nach eingestellter Videonorm das Chroma-Signal in die beiden Farbkanäle U und V aufgespaltet. Der so entstehende Pixelstrom steht jetzt für die weiteren Schritte zur Verfügung.

Bildmodifikationen

Nach der Farbseparation werden die meisten Bildmodifikationen auf die Daten des kompletten Bildes angewendet. Dazu gehören:

- Helligkeit und Kontrast
- Entfernung der Gamma-Korrektur
- Sättigung der beiden Farbkanäle
- Anpassung der Farbphase (Hue, nur für NTSC)
- Interpolationsfilter

Bis auf die Interpolationsfilter haben die meisten dieser Anpassungen keine Auswirkungen auf die nachfolgenden Schritte. Bei der Interpolation ist zu beachten, daß sie unmittelbaren Einfluß auf die Menge der Daten hat, die später in den Hauptspeicher geschrieben werden.

Skalierung und Ausschneiden

Nach den Bildmodifikationen erfolgt die Skalierung des Bildes und die Wahl des Bildausschnittes. Dies ist nicht mit einem nachfolgenden Clipping-Schritt zu verwechseln, mit dem Popup-Menüs und Fenster das Videobild überdecken können.

Für den BT878 reicht eine Zeile des Videobild einfach von einem horizontalen Synchronisierungsimpuls zum nächsten, d.h. die vordere und hintere Schwarzschulter und der Color-Burst (vgl. Kap. 2.1) würden nicht ausgeblendet sondern mit angezeigt werden. Daher bietet der BT878 die Möglichkeit, nur einen Teil der Bildzeile zu bearbeiten, indem man festlegt, ab welchem Zeitpunkt nach einem Synchronisierungsimpuls das später sichtbare Signal beginnt, und wie lange es anhält.

Analog dazu verfährt der BT878 mit den einzelnen Bildzeilen bzw. dem vertikalen Synchronisierungsimpuls.

Außerdem kann der BT878 den so festgelegten sichtbaren Bereich skalieren, so daß es möglich ist, auch kleinere Videobilder zu bearbeiten.

Die Werte für den Bildausschnitt und die Skalierung (im folgenden als **Bildgeometrie** bezeichnet) hängen dabei von der jeweils verwendeten Videonorm ab. Speziell die in Kapitel 2.3 aufgeführten Parameter lassen sich in die für den Bildausschnitt notwendigen Parameter umrechnen. Die Skalierung ist insofern auch von der Videonorm abhängig, da durch die Abtastrate bei der Digitalisierung einer Videozeile eine unskalierte Bildzeile sichbar zu breit ist, also die Breite auf jeden Fall skaliert werden muß, andererseits aber durch die Norm ein konkretes Seiten-Höhen-Verhältnis des Bildes vorgegeben ist.

Farbraumkonvertierung

Nach der Skalierung erfolgt eine Konvertierung des Pixelstroms in das Farbformat und die Farbtiefe des Zielgerätes. Dabei wird grob zwischen zwei Gruppen von Modi unterschieden:

- **Packed Mode**
  In diesem Modus werden die Daten der drei Farbkanäle gemäß dem gewünschten Format hintereinander in den Speicher geschrieben. Es stehen folgende Formate zur Verfügung: RGB 32, RGB 24, RGB 15, RGB 16, YUV 4:2:2, YUV 4:1:1, Y8 und RGB8.

- **Planar Mode**
  Hier werden die drei Farbkanäle getrennt an verschiedene, wählbare Speicherbereiche geschrieben. Jedem Farbkanal ist also eine eigene Bildebene zugeordnet. In dieser Gruppe gibt es nur folgende Varianten des YUV Formates: YUV 4:2:2, YUV 12, YUV 4:1:1 und YUV 9.

Zur Anzeige des Videobildes auf dem Bildschirm eignet sich einer der gepackten Modi am besten, da Grafikkarten diese meist direkt unterstützen. Will man dagegen auf dem Videobild noch weitere Berechnungen durchführen, so empfiehlt sich eher einer der planaren Modi, da die Farbkanäle sowieso schon getrennt sind.

Zusätzlich zur Wahl des Farbraumes gibt es die Möglichkeit, die einzelnen Bytes des Datenstromes zu vertauschen, um sich der Zahlendarstellung der Grafikkarte oder der CPU anzupassen.

FIFO-Puffer

Schließlich wird der Pixelstrom in einen FIFO-Puffer gesteckt, um beim Zugriff auf den PCI Bus leichte Verzögerungen besser handhaben zu können (vgl. [BT878], S. 42ff). Der Puffer ist, je nachdem, ob beim Farbmodell ein Packed Mode oder ein Planar Mode gewählt wurde, in einen großen FIFO mit 140 Worten oder in drei kleinere Puffer mit 70 Worten für die Luminanz-Daten und je 35 Worte für die beiden Chrominanz-Datenströme aufgeteilt. Ein Wort besteht dabei aus 36 Bit, welche in 32 Bit Daten und 4 Bit Status- und Steuerinformation aufgeteilt sind. Die Steuerinformationen dienen dazu, um z.B. den Anfang eines Halbbildes, den Anfang oder das Ende einer Bildzeile zu markieren.

DMA-Controller

Nachdem der Pixelstrom in den FIFO-Puffer geschrieben wurde, behandelt der Chip die Daten nicht mehr als Pixel, sondern nur noch als einzelne Bytes, welche der DMA-Controller mit Hilfe eines im Hauptspeicher stehenden RISC-Programmes einfach in den Speicher schreibt. Neben dem Schreiben der Bytes unterstützt der Befehlssatz auch das Überspringen einer Anzahl von Bytes im Datenstrom, wodurch es möglich ist, bestimmte Speicherbereiche nicht zu überschreiben. In diesen Bereichen können sich z.B. Popup-Menüs oder überlappende Fenster befinden.

Der Befehlssatz des DMA-Controllers ist ausschließlich auf das Schreiben bzw. Überspringen von Datenströmen spezialisiert (vgl. Tabelle 2). Außerdem unterstützt er noch absolute

Sprünge und Synchronisierungsbefehle. Der DMA-Controller unterstützt keine Adressierungsarten außer "Speicher direkt" und auch keine Variablen.

Mit den Befehlen des DMA-Controllers ist es also nicht möglich, irgendwelche weiteren Vorkodierungen oder Transformationen durchzuführen.

Packed Mode	
Write	#bytes, @addr
WriteC	#bytes
Skip	#bytes

Planar Mode	
Write123	#bytes1, @addr1, #bytes2, @addr2, #bytes3, @addr3
Write1S23	#bytes1, @addr1,#bytes2, #bytes3
Skip123	#bytes1, #bytes2, #bytes3

Allgemein	
Jump	@addr
Sync	Odd/Even, FM1 (Packed Mode)/FM3 (PlanarMode)

Tabelle 2: Befehlssatz des DMA-Controllers

Abhängigkeiten des RISC Codes

Der RISC-Code, der den DMA-Controller steuert, stellt die Verbindung zwischen dem digitalisierten Videosignal und dem Zielgerät, beispielsweise dem Speicher der Grafikkarte, dar. Die Erstellung eines RISC-Programmes ist also insofern eine schwierige Aufgabe, da in den Code eine Vielzahl von Faktoren mit einfließen müssen.

- TV Norm, insbesondere
  - Breite des Bildes in Pixel
  - Höhe des Bildes in Pixel
- Skalierung
- Bildausschnitt (Cropping)
- eventuelle Interpolationsschritte
- gewünschtes Farbmodell bzw. Farbtiefe
- Packed / Planar Mode
- Organisation des Bildspeichers, insbesondere
  - Adresse des Grafikspeichers
  - Farbtiefe und Pixeldarstellung
  - Breite und Höhe des Bildschirms in Pixeln
  - Position des Videobildes auf dem Bildschirm
  - Adressierungsversatz zwischen den Bildschirmzeilen

- Clipping von Popup-Menüs und Fenstern

Wie man sieht, fließen in die Erzeugung des Codes fast alle Parameter mit ein, die in den vorherigen Stufen der Pipeline verwendet wurden. Dazu kommen etliche Parameter, um den Grafikspeicher zu beschreiben.

In diesem Kapitel wurde die Arbeitsweise des BT878 Capture-Chips erklärt. Dadurch wurden auch schon viele der Einflußgrößen aufgezeigt, die im folgenden Schritt, dem Entwurf des Treibers, zu berücksichtigen sind.

4 Entwurf eines Treibers unter JX

Nachdem im vorherigen Kapitel der BT878-Chip vorgestellt wurde, soll in diesem Kapitel ein Treiber für das Java-Betriebssystem JX ([JXOS]) entworfen werden. Die Schnittstelle dieses Treibers soll dabei aber so flexibel sein, daß es nicht nur für Chips der BT8x8-Familie anwendbar ist.

Da das JX-Betriebssystem ein recht außergewöhnliches System ist, werden die Besonderheiten, die die Entwicklung eines Hardware-Treibers tangieren, kurz vorgestellt. Für einen Vergleich ist es aber sinnvoll, zuerst kurz die Treiberentwicklung bei einem konventionellen Betriebssystem zu betrachten.

4.1 Treiberentwicklung unter konventionellen Betriebssystemen

Die Entwicklung eines Gerätetreibers für das Java-Betriebssystem JX unterscheidet sich stark von dem Vorgehen bei konventionellen Betriebssystemen. Das liegt zum einen daran, daß diese Betriebssysteme auf ganz anderen Schutzmechanismen aufbauen, als dies beim JX-System der Fall ist. Zum anderen ist JX bis auf einen sehr kleinen Kern ausschließlich in Java geschrieben, so daß selbst Gerätetreiber und andere der Hardware nahe Komponenten objektorientiert entworfen und implementiert werden können.

Speicherschutz

Einer der üblichen Schutzmechanismen von Betriebssystemen ist der sogenannte Speicherschutz. Mit Hilfe der Hardware laufen die einzelnen Programme in je einem eigenen Adressraum. Auch der Betriebssystemkern ist von den Anwendungsprogrammen nicht über Speicherzugriffe zu erreichen. Die Kommunikation der Anwendungsprogramme mit dem Kern erfolgt über Systemaufrufe. Auch die Kommunikation der Anwendungsprogramme untereinander erfordert die Mithilfe des Kerns.

Auf diese Weise können sich die Programme nur mit Hilfe definierter Mechanismen gegenseitig beeinflußen. So ist es auch nicht möglich, daß ein Programm ein anderes zum Absturz bringt, indem es fremden Speicher überschreibt.

Dieses an sich sinnvolle Vorgehen hat aber einige Nachteile, die sich direkt auf die Programmierung von Treibern auswirken:

1. Um den Speicherschutz zu gewährleisten, muß jeder Gerätetreiber im Kern integriert sein, denn die meisten Geräte wären in der Lage, den Speicherschutz auf Hardware-Ebene zu umgehen. Ein Gerät unter der direkten Kontrolle eines Anwenderprogrammes ist daher nicht möglich.
2. Für die Kommunikation zwischen Kern und Anwenderprogrammen werden nicht einfache Funktionsaufrufe, sondern spezielle Prozessorbefehle verwendet, um den Prozessor in einen Betriebsmodus zu bringen, in dem der Speicherschutz aufgehoben werden kann. Diese Schnittstelle ist sehr "eng", d.h. es können keine komplexen Datentypen übergeben werden. Außerdem kann diese Schnittstelle nicht erweitert werden.

3. Durch die festgelegte Semantik der Systemaufrufe ergibt sich auch, daß es für Gerätetreiber sehr schwer ist, dem Anwenderprogramm Funktionen des Gerätes zugänglich zu machen, die nicht in die vorgesehene Schnittstelle für die jeweilige Geräteklasse passen.

Objektorientierung? - Nein!

Ein weiteres Merkmal bei herkömmlichen Betriebssystemen ist, daß es kein systemweit verbindlich unterstütztes Objektmodell gibt. Insbesondere gibt es im Kern selbst keine Unterstützung für Objekte. Stattdessen ist jede Applikation selbständig für ihr Objektmodell verantwortlich. Für die Kommunikation zum Kern muß aber in jedem Fall auf die Standard-Datentypen zurückgegriffen werden.

Zusammenfassung

Da in konventionellen Betriebssystemen die Gerätetreiber im Kern integriert sein müssen und die Systemschnittstelle in der Regel schon fest vorgegeben ist, fällt es einem Treiber sehr schwer, erweiterte Funktionalität auf einfache und elegante Weise dem Anwendungsprogramm zugänglich zu machen. Hierfür würde sich ein objektorientierter Ansatz anbieten, da die Bereitstellung der erweiterten Funktionalität über Vererbung eine einfache und elegante Lösung wäre. Aber auch dies ist wegen des fehlenden Objektmodells in der Regel nicht möglich.

Stattdessen ist es üblich, die Erweiterungen über die Hintertür generischer Systemaufrufe anzubieten. Ein gutes Beispiel dafür ist unter den Unix-Derivaten die `ioctl(2)` Schnittstelle. Mit ihr können einem Gerätetreiber nur einfache Werte oder Speicherbereiche übergeben werden, deren Inhalt dann einer bestimmen Struktur und Semantik genügen müssen. Ändert sich die Semantik eines solchen Parameterblocks, muß jede Anwendung einzeln angepaßt werden.

Treiberentwicklung unter JX

Ganz anders stellt sich die Lage bei JX dar. Das JX-System hat einen sehr kleinen Kern, der in Assembler und C geschrieben wurde. Er stellt nur die Grundlage für die in Java geschriebenen Klassen dar, durch welche die eigentliche Funktionalität implementiert wird.

JX arbeitet als Sprache nur mit dem Java Bytecode. Das Java Laufzeitsystem kann daher durch Überprüfungen zur Laufzeit garantieren, daß keine unerlaubten Speicherzugriffe stattfinden. Daher ist ein strenger, durch Hardware unterstützter Speicherschutz nicht nötig. Applikationen können sich also nur dann gegenseitig beeinflußen, wenn zuvor irgendwie Objektreferenzen ausgetauscht wurden.

Als zusätzliche Schutzmaßnahme ist es bei JX möglich, das System in mehrere "Domains" zu unterteilen. Die Kommunikation über Domaingrenzen hinweg erfolgt über "Portale". Außer Referenzen auf Portale werden keine anderen Objektreferenzen über Domain-Grenzen ausgetauscht. Stattdessen werden diese Objekte kopiert.

Ein Portal besteht aus zwei Komponenten: Erstens einem speziell markiertem Interface, das alle Methoden des Portals beinhaltet und zweitens einer Klasse, die dieses Interface implementiert. Nachdem das Portal initialisiert wurde, kann die Portalreferenz frei an andere Komponenten verteilt werden, so z.B. über einen Namensdienst. Die Portalreferenz ist dabei nicht die Referenz auf das eigentliche Objekt, sondern auf einen Platzhalter (Proxy), der automatisch aus dem offiziellen Interface des Portals erzeugt wurde. Der Platzhalter kümmert sich z.B. darum, daß bei den Portalaufrufen die Parameter des Aufrufs kopiert werden und daß der aufrufende Thread blockiert bzw. der Thread des Portals fortgesetzt wird, um den eigentlichen Aufruf abzuarbeiten.

JX bietet durch diese Mechanismen also die einfache und elegante Möglichkeit, durch Ableitung die erweiterte Funktionalität eines Gerätes anzubieten. Eine Anwendung, die diese Funktionalität nicht benötigt, kann einfach weiterhin die Methoden der Basisklasse verwenden.

Wie aber erfolgt nun der Zugriff auf die Hardware? Java ist eine Hochsprache und wurde ja gerade mit dem Ziel entwickelt, Programme in einem geschützen "Sandkasten" ausführen zu können, der vom Rest des Systems völlig unabhängig ist. Das beinhaltet, daß es in Java selbst keine Möglichkeit gibt, auf die Hardware direkt zuzugreifen. Bei JX ist das Problem so gelöst, daß von der Kerndomäne, auch "DomainZero" genannt, zwei Portale, "`Ports`" und "`Memory`" zur Verfügung gestellt werden. Diese Portale ermöglichen den Zugriff auf die Hardware über die IO-Ports bzw. auf Speicherbereiche, womit man auch Geräte ansprechen kann, die ihre Register in einen Speicherbereich einblenden (Memory Mapped IO). Mit der letzteren Methode können auch die Register des BT878 angesprochen werden.

4.2 Ziele des Entwurfes

Bevor der Entwurf des Treibers erläutert wird, ist es ratsam, die Zielsetzungen bei der Entwicklung des Treibers zu bestimmen.

Universalität: Eines des Hauptziele bei der Entwicklung des Treibers war es, nicht nur einen Treiber für die Hauppauge WinTV-Karte zu entwickeln und zu implementieren. Vielmehr sollte ein möglichst universeller Treiber entstehen, dessen Schnittstelle auch für andere Framegrabber-Karten verwendet werden kann, selbst wenn diese keinen Chip der BT878 Familie verwenden.

Flexibilität: Auch wenn die Schnittstelle des Treiber nicht auf BT878-Karten festgelegt ist, soll sie doch so flexibel sein, daß dennoch alle Möglichkeiten des BT878 damit kontrollierbar sind. Dabei soll die Schnittstelle vom Benutzer elegant zu benutzen sein. Insbesondere sind daher solche Verrenkungen zu vermeiden, wie sie bei konventionellen Betriebsystemen z.B. bei der `ioctl(2)` Schnittstelle notwendig sind.

Sicherheit: Beim Entwurf der Schnittstelle muß auch darauf geachtet werden, daß es nicht möglich sein soll, die Sicherheitsmechanismen des Betriebssystems zu unterwandern. Dies hat auch Auswirkungen auf die vorher erwähnten Punkte. So ist es schon aus Gründen der Sicherheit nicht möglich, den RISC-Code des DMA-Controllers vom Anwendungsprogramm erstellen zu lassen, um ihn dann einfach zu verwenden. Natürlich schränkt man dadurch die Flexibilität des BT878 stark ein.

Modularität: Ein weiterer sehr wichtiger Punkt ist die leichte Anpassbarkeit und Wiederverwendbarkeit des Treibers selbst für andere Karten aus der BT878 Familie. Die Grundidee ist, daß die Treiber für neue Karten nicht den vorhandenen Treiber modifizieren, sondern durch Ableitung der vorhandenen Klassen einen neuen Treiber zu erstellen. Auf diese Weise sollen die Klassen für die einzelnen Kartentypen sehr übersichtlich bleiben, vorausgesetzt, der Treiber ist auch modular aufgebaut. Der Vorteil ist neben der leichten Lesbarkeit der einzelnen Treiber auch die Flexibilität. Treiber in konventionellen Betriebssystemen, die eine Vielzahl von ähnlichen Geräten unterstützen sollen, haben meist eine integrierte Tabelle, in der für jedes unterstütze Produkt die möglichen Optionen festgehalten sind. Leider ist dann der Treiber selbst ziemlich schwer zu verstehen, da ja an vielen Stellen im Treiber auf die Tabelle verwiesen wird und daher der Kontrollfluß bei einem konkreten Gerät schwer nachvollziehbar ist.

4.3 Grobmodell des Treibers

Nachdem die Entwurfsziele definiert wurden, soll nun ein grober Überblick über den prinzipiellen Aufbau des Treibers gegeben werden. Die genaue Erklärung der einzelnen Aspekte folgt dann in den darauffolgenden Kapiteln.

Der Treiber gliedert sich in zwei große Teile (s. Bild 3): den Teil, den das Benutzerprogramm sieht und einen zweiten Teil, der aus Sicherheitsgründen vollständig gekapselt ist.

Zu den Klassen und Schnittstellen, die dem Anwendungsprogramm sichtbar sind, gehören in erster Linie die Portal-Interfaces, über welche die Anwendung mit dem geschützten Teil des Treibers kommuniziert. Außerdem stehen dem Anwendungsprogramm noch einige Hilfsklassen zur Verfügung, die das Anwendungsprogramm selbst nach Belieben erzeugen und dem Treiber übergeben kann. Beispiele hierfür sind die Klassen für Clipping--Rechtecke, Übersetzungsklassen für globale Videoeinstellungen wie Helligkeit und Kontrast sowie Klassen für die Skalierung des Videobildes. Diese Klassen werden in den entsprechenden Kapiteln genauer behandelt.

Der zweite Teil der Klassen, die den Anwendungsprogrammen nicht direkt zugänglich sind, unterteilt sich wiederum in drei Gruppen:

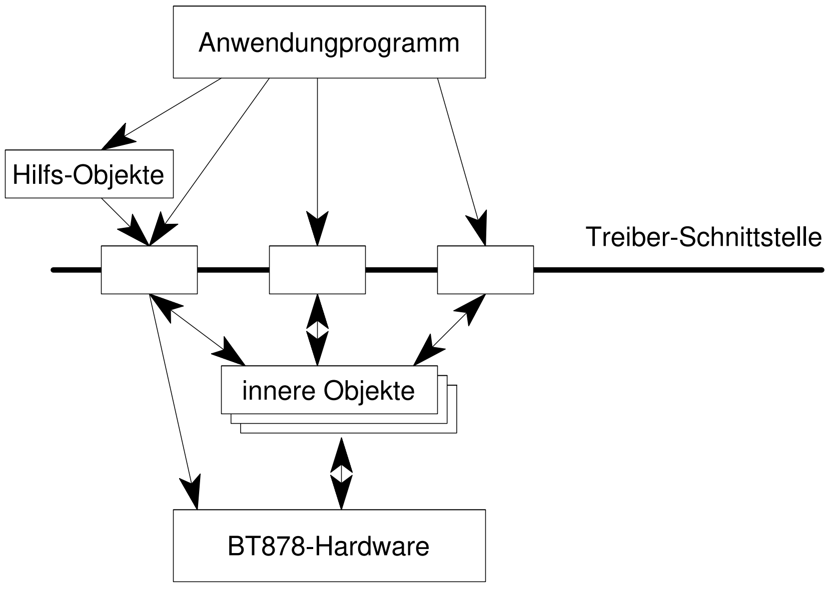

Bild 3: Modell des Treibers

1. Implementierungen der Portal-Interfaces
2. innere Verwaltungsklassen
3. Klassen für die Kapselung von Hardwarekomponenten.

Diese Klassen implementieren größtenteil entweder den direkten Zugriff auf die Hardware oder sind für die Erstellung oder Verwaltung des RISC-Codes zuständig. Da diese Teile sicherheitsrelevant sind, können sie auf keinen Fall in das Anwenderprogramm verlagert oder über mehrere Domänen verteilt werden.

Durch die innere Strukturierung des Treibers soll aber dennoch gewährleistet werden, daß Erweiterungen wie neue Karten oder neue Treiberfunktionalität, wie z.B. Fernbedienungen, leicht eingefügt werden können.

Die Klassen und Interfaces, die dem Anwendungsprogramm sichtbar sind, stellen eine universelle Abstraktion des Capture-Prozesses im Sinne der oben definierten Ziele dar. Sie werden direkt im Anschluß an dieses Kapitel erläutert. Erst danach wird auf den internen Aufbau des Treibers eingegangen, da dieser verständlicherweise stark von den öffentlichen Klassen abhängig ist.

4.4 Abstraktion des Capture-Prozesses

Um den Prozeß der Video-Digitalisierung in geeignete Klassen einteilen zu können, ist es zuerst einmal notwendig, die einzelnen Stationen des in Kapitel 2 beschriebenen Digitalisierungsprozesses geeignet zusammenzufassen, um so die notwendigen Abstraktionen zu erhalten. Eine passende Einteilung der Stufen wäre z.B. folgende:

1. Auswahl und Manipulation der Videoquelle

   Dazu gehören im einzelnen:

   - Initialisierung und Einstellungen an externen Geräten. So muß z.B. der auf der Karte integrierte Tuner initialisiert und eine dem gewünschten Programm entsprechende Empfangsfrequenz eingestellt werden.
   - Aktivierung des entsprechenden Videoeinganges am BT878
   - Einstellung des Chroma-Pfades (z.B. bei Component-Signalen (s. Kap. 1.4))

2. Einstellung der verwendeten Videonorm

   Hier ist im wesentlichen der Videodecoder entsprechend dem anliegenden Videosignal zu konfigurieren, d.h. die Kennung der Videonorm und die Frequenz des Farbträgers sind einzustellen. Außerdem empfiehlt es sich, zuallererst eine Bildgeometrie, also Bildausschnitt und Skalierung, für ein volles Bild der jeweiligen Videonorm einzustellen.

   Leider kann diese Einstellung nicht automatisch zusammen mit der Auswahl der Videoquelle erfolgen, da z.B. beim Cinch-Eingang ein Videogerät mit beliebiger Videonorm anschließbar und auch jederzeit austauschbar ist.

3. Einstellung globaler Bildmanipulationen

Hier werden alle Einstellungen vorgenommen, die das Bild als Ganzes beeinflußen, wie z.B. Helligkeit, Kontrast und Entfernung der Gammakorrektur. Bei der Abstraktion und Implementierung ist darauf zu achten, daß gerade dieser Bereich bei anderen Videocapture-Produkten sehr unterschiedlich ausfallen wird. Um die Capture-Schnittstelle möglichst produktunabhängig zu halten, ist also eine sehr flexible Schnittstelle notwendig, um alle Modifikationen anbieten zu können, ohne sich auf ein Produkt festzulegen.

4. Einstellung der Bildgeometrie

   Diese Einstellung beinhaltet die Wahl des gewünschten Bildausschnittes und der gewünschten Skalierung. Beide Werte haben bereits Einfluß auf die Größe des erzeugten Bildes und damit auf den RISC-Code, der später erzeugt werden soll.

5. Einstellung des Zielgerätes

   Dazu gehört neben der Wahl des Farbmodelles auch die Erzeugung eines passenden RISC-Codes, mit dem die Daten dann mit PCI-Busmaster-Transfers in den Zielspeicher geschrieben werden. Für die Erzeugung des Codes ist daher neben der Bildgeometrie auch die Kenntnis über Organisation des Zielspeichers notwendig. Es verbietet sich aus Gründen der Systemsicherheit, den Code vom Anwenderprogramm erzeugen zu lassen, um ihn dann einfach zu übernehmen, da der RISC-Code dann beliebige Speicherbereiche überschreiben könnte.

6. Starten des Capture-Prozesses

Nachdem nun der Prozeß der Videodigitalisierung in einzelne Stationen zusammengefaßt wurde, ist es möglich, diese Stationen relativ unabhängig voneinander zu betrachten, um sie anschließend in ein geeignetes Klassenmodell überzuführen.

4.5 Einstellung der Videoquelle

Für die Auswahl der Videoquelle ist ein flexibles System notwendig, da sich erstens die Belegung der Videoeingänge von Karte zu Karte ändert und es zweitens in der Regel auch nicht ausreicht, einfach einen Eingang einzuschalten. So ist es zum Beispiel für die Verwendung des Tuners notwendig, die Sendefrequenz des gewünschten TV-Senders einzustellen, bevor dann am BT878 der entsprechende Videoeingang aktiviert werden kann.

Es bietet sich an, bei der Initialisierung der Karte für jeden Videoeingang ein Portal zu definieren und diese Portale mit Hilfe eines kleinen Namensdienstes für die Anwendungsprogramme verfügbar zu machen. Auf diese Weise können die spezifischen Einstellungen der jeweiligen Videoquelle vorgenommen werden. Wie in Bild 4 zu sehen ist, gibt es als dem Anwendungsprogramm zugängliche Interfaces `InputSource`, `Tuner` und `SVideo`, durch die neben der allgemeinen Aktivierung der Videoquelle auch weitere spezifische Einstellungen vorgenommen werden können. Die jeweiligen Implementierungen dieser Interfaces (`MUXPin`, `ColorBars`, `FM1216` und `SVideoImpl`) sind dem Anwendungsprogramm natürlich nicht direkt zugänglich, sie werden bei der Initialisierung des Treibers gemäß den Gegebenheiten der jeweiligen Karte erzeugt und nur als Portalreferenz dem Anwendungsprogramm ausgehändigt.

Am Beispiel des auf der Hauppauge-Karte vorhandenen Philips-Tuners (vgl. [Phil97]) soll erklärt werden, welche Schritte zur Aktivierung des Tuners notwendig sind (s. Bild 5).

Zuerst muß vom Gerätetreiber die Portalreferenz der Tuner-Videoquelle erfragt werden. Im zweiten Schritt werden die Frequenzbereiche erfragt, innerhalb derer der Tuner eingestellt werden kann. Anschließend kann eine Frequenz ausgewählt und eingestellt werden. Dieser Schritt erfordert die Kommunikation des Tuner-Treibers mit der Tuner-Hardware. Dies geschieht in diesem Falle über den im BT878 integrierten I^2C-Bus, an den der Tuner angeschlossen ist. Nachdem der Tuner eingestellt ist, kann das Anwendungsprogramm im vierten und letzten Schritt den Tuner als Bildquelle aktivieren. Da der Tuner an den am BT878 integrierten Multiplexer angeschlossen ist, muß der entsprechende Eingang ausgewählt werden.

Wie schon erwähnt sind alle analogen Videoquellen beim BT878 an einen internen Multiplexer angeschlossen. Es bietet sich daher an, diese Funktionalität in der Klasse `MUXPin` zusammenzufassen. Im einfachsten Falle entspricht einem dieser Eingänge eine

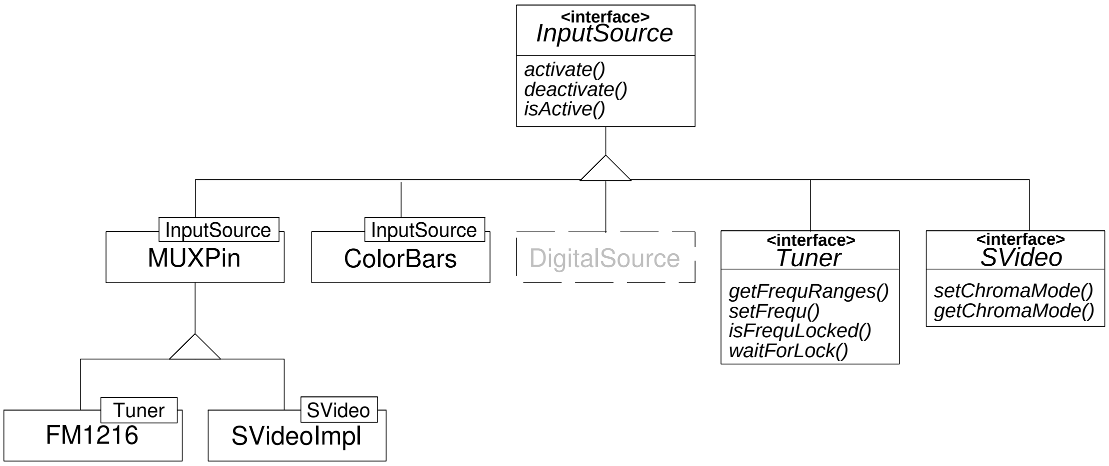

Bild 4: Klassendiagramm InputSource

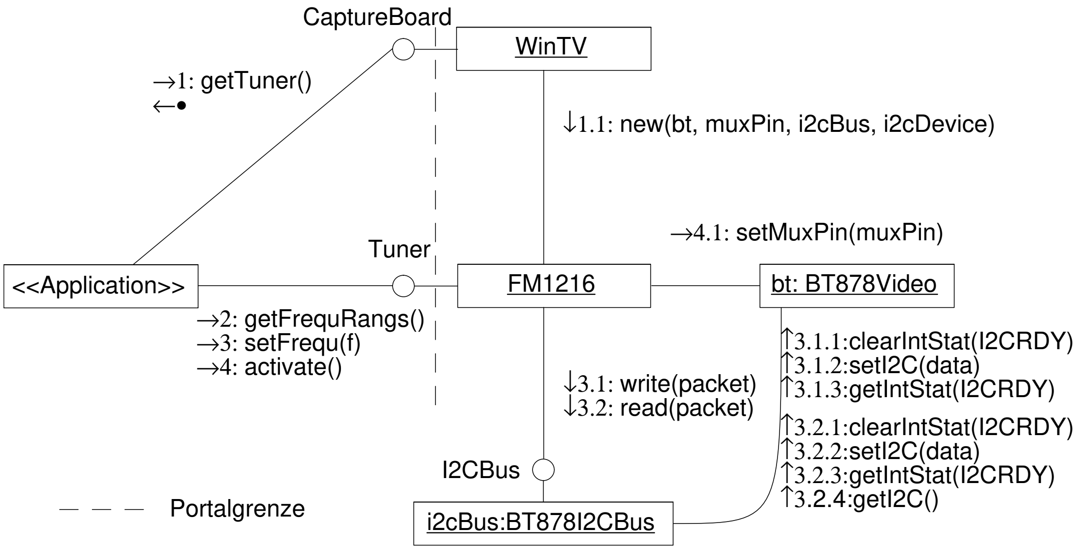

Bild 5: Kollaborationsdiagramm "Tuner aktivieren"

Cinch- oder SVideo-Buchse auf der Rückseite der Karte. Im Gegensatz dazu stellt `ColorBars` ein simples Testbild zur Verfügung, welches aber nicht über einen der Videoeingänge eingespeist wird, sondern vom BT878 intern erzeugt wird. Als weitere Quelle von Videodaten könnte der BT878 digitale Videoformate unterstützen. Diese Option ist auf der Hauppauge Karte aber nicht verdrahtet, weshalb auch keinerlei Implementierung für diese Möglichkeit vorliegt. Daher ist diese Klasse im Diagramm (Bild 4) auch grau dargestellt.

Mit diesem Grundgerüst können auf einfache Art und Weise durch Ableitung neue "Mini-Treiber" für Videogeräte implementiert werden. Es bietet ausreichende Stabilität bei den Interfaces, sowohl für den Gerätetreiber der Capture-Karte, als auch für die Anwendungsprogramme. So kann z.B. ein neuer Typ von Tunern völlig unabhängig von der Karte, auf der er montiert ist, implementiert werden und umgekehrt kann jede neue Karte aus alle bisher implementierten Tuner zurückgreifen.

4.6 Einstellung der Videonorm

Obwohl es bei der Einstellung der Videonorm nötig ist, etliche Parameter zu übergeben (vgl. Kap. 1.3), so erfordert dies doch wenig Aufwand beim Entwurf, denn die notwendigen Parameter sind erstens konstant und zweitens werden durch sie keine Bereiche des Treibers berührt, durch welche die Systemsicherheit beeinträchtigt werden könnte. Daher ist es am einfachsten, dem Anwendungsprogramm eine Tabelle mit den verschiedenen unterstützen Videonormen zu geben, aus der es sich die gewünschte Norm heraussuchen und über eine Portalmethode dem Treiber übergeben kann. Dadurch ist es ohne Änderung am Treiber möglich, neue Videonormen zu unterstützen oder auf den vorhandenen Videonormen aufbauende Varianten zu verwenden.

Falls der Videodecoder des BT878 die Wahl der Parameter nicht unterstützt, so werden beim Capture-Prozeß höchstens Zufallsdaten in den Speicher des Zielgerätes geschrieben - der Anwender sieht also nur Flimmern. Es ist aber aufgrund des RISC-Codes auf diese Weise nicht möglich, andere Speicherbereiche zu überfluten.

4.7 Globale Bildmanipulationen

Der BT878 unterstützt eine Vielzahl von Möglichkeiten, das digitalisierte Videobild zu manipulieren. Die meisten dieser Manipulationen werden auf das komplette Bild angewendet und verändern zumeist auch nicht das entstehende Datenvolumen, d.h. sie können unabhängig von den übrigen Einstellungen, insbesondere vom RISC-Code, betrachtet werden.

Die Einstellungsmöglichkeiten bei den einzelnen Optionen sind auch sehr breit gestreut. Einige der Optionen kann man nur an- oder ausschalten, so z.B. die Entfernung der Gammakorrektur. Bei anderen kann man aus einer Handvoll Möglichkeiten auswählen, und bei wieder anderen können fast stufenlose Einstellungen erfolgen.

Um nun alle dieser Möglichkeiten zu unterstützen, wäre es im Prinzip möglich, für jede der Optionen im Portal-Interface Methoden zum Lesen und Schreiben anzubieten. Das würde aber erstens das Interface gewaltig aufblähen. Zweitens wäre es so nur sehr schwer möglich, andere Videocapture-Produkte zu unterstützen, ohne eine komplett neue Schnittstelle zu definieren, da diese dieselben Möglichkeiten wahrscheinlich nicht oder in ganz anderer Form anbieten. Drittens würde ein großes Interface nicht das Problem lösen, wie denn nun die jeweiligen Werte zu interpretieren wären. So bietet der BT878 z.B. die Möglichkeit an, die Helligkeit um bis zu 50% zu senken oder um bis zu 49.6% anzuheben und zwar in Schritten von 0.39% (vgl. [BT878], S.100). Dies sind Grenzen, die wohl bei keinem anderen Hersteller Gültigkeit haben werden.

Die Lösung besteht darin, für jede gewünschte Manipulation eine Klasse einzuführen (s. Bild 6). Objekte dieser Klasse können über einen Namensdienst beim Portal angefragt werden, wobei der jeweils aktuelle Wert im Objekt kodiert ist. Alternativ übergibt man dem Portal ein passendes Objekt, so daß der Treiber durch den Typ des Objektes feststellen kann, welche Modifikation erwünscht ist.

Außerdem beinhalten die Objekte Informationen über die Möglichkeiten, die mit dieser Option einstellbar sind. So beinhaltet z.B. ein Objekt der Klasse `BrightnessAdjustment` Informationen über die obere und untere Schranke der Einstellungsmöglichkeit sowie die Angabe, daß die Einstellung in Prozent erfolgt.

Durch die Methoden des Objektes kann der geräteabhängige Wert in menschenverständliche Werte konvertiert werden, wobei natürlich auch die Rückrichtung möglich ist.

Da der Typ des jeweiligen Objektes entscheidet, welche Modifikation stattfindet, genügt beim Portal-Interface eine Methode zum Lesen und eine Methode zum Schreiben des Wertes. Das Portal-Interface bleibt also dadurch sehr schlank und doch gleichzeitig sehr flexibel, wie folgendes Beispiel zum Abfragen und Entfernen der Gamma-Korrektur zeigt.

```
GammaCorrectionRemovalAdjustment oldGamma =
     (GammaCorrectionRemovalAdjustment)
     captureBoard.getVideoAdjustment("GammaCorrectionRemoval");
System.out.println("gamma correction removal enabled:
"+oldgamma)

GammaCorrectionRemovalAdjustment newGamma =
     new GammaCorrectionRemovalAdjustment(false);
captureBoard.setVideoAdjustment(newGamma)
```

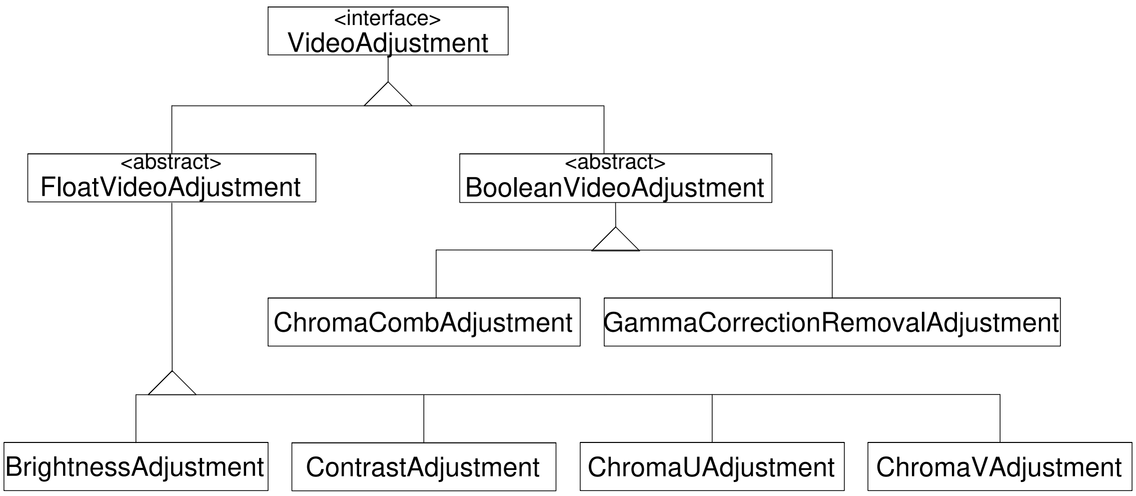

Bild 6: Klassendiagramm `VideoAdjustment`

4.8 Einstellung der Bildgeometrie

Nachdem die globalen Bildmanipulationen eingestellt wurden, kann jetzt die Bildgeometrie angepaßt werden. Schon bei der Einstellung der Videonorm wurde eine Geometrie eingestellt, mit der ein Vollbild mit der jeweils maximalen Auflösung erzeugt wird. Nun aber soll das Bild an die gewünschte Größe angepaßt und der gewünschte Bildausschnitt gewählt werden.

Es gibt nun mehrere Möglichkeiten, den Bildausschnitt und die Skalierung zu wählen:

- Der Bildausschnitt wird mittels unskalierter Bildpunkte angegeben.
- Der Bildausschnitt wird in skalierten Bildpunkten angegeben.
- Die Skalierung wird als Faktor angegeben.
- Die Skalierung wird indirekt durch das Verhältnis von gewünschter Bildbreite zur Breite des Vollbildes angegeben.

Der letzte Punkt stellt dann einen Unterschied zur Angabe als Faktor dar, wenn keine Fließkommazahlen zur Verfügung stehen oder die Skalierung nur in diskreten Schritten erfolgen kann, so daß der Treiber aus den Wünschen der Anwendung selbst den geeignetsten Faktor berechnen kann. Beides trifft auf das JX-System bzw. den BT878 zu.

Natürlich sind auch Mischformen der oben aufgeführten Punkte möglich. So kann es sinnvoll sein, den Startpunkt des Bildausschnittes in unskalierten Punkten und die Breite in skalierten Punkten anzugeben, wobei die Skalierung so gewählt werden soll, daß das Bild in ein vorhandenes Fenster paßt.

Aus diesen Gründen bietet der Treiber keine Schnittstelle zur Angabe der Geometrie an, die vom Anwendungsprogramm direkt benutzt werden soll. Vielmehr werden über das Portal direkt die vom BT878 benötigten Werte übergeben. Hilfsklassen auf Seiten des Anwendungsprogrammes verwenden diese Schnittstelle, um die gewünschten Skalierungs- und Ausschnitts-Parameter zu errechnen.

Als Beispiel für dieses Vorgehen wurde in der Klasse `FramebufferScaler` ein Modul entwickelt, welches das Vollbild einer Videonorm so skaliert, daß es in einen vorgegebenen Framebuffer hineinpaßt. Details zum Framebuffer-Konzept folgen im nächsten Kapitel. Im Anwendungsprogramm wird der `FramebufferScaler` folgendermaßen verwendet:

```
FramebufferScaler scaler = new FramebufferScaler(captureBoard);
scaler.setScalingOdd(tvnorm, framebufferOdd);
scaler.setScalingEven(tvnorm, framebufferEven);
```

4.9 Einstellung des Speicherzieles

Die letzte Station der Capture-Pipeline stellt die Erzeugung des RISC-Codes dar. Da es, wie schon erwähnt, aus Gründen der Systemsicherheit nicht möglich ist, den RISC-Code vom Benutzer erzeugen zu lassen, muß es das Ziel dieser Stufe sein, die vielfältigen Möglichkeiten, die der RISC-Code bietet, möglichst ohne Einschränkung zu unterstützen. Eine alternative Vorgehensweise besteht darin, einige Anwendungsfälle zu untersuchen, um die dafür notwendige Funktionalität zu ermitteln. Diese Methode hat zudem den Vorteil, daß sie bedeutend leichter zu realisieren ist.

Darstellung am Bildschirm

Der am häufigsten auftretende Anwendungsfall ist natürlich die Darstellung des Videobildes auf dem Bildschirm. Dazu muß der Datenstrom vom DMA-Controller in den Speicher des Grafikadapters geschrieben werden.

Wie ist nun der Grafikspeicher organisiert? Moderne Grafikadapter blenden ihren gesamten Speicher über den PCI- oder AGP-Bus in den Adressraum der CPU ein, so daß aus Sicht der CPU der Speicherzugriff sehr einfach ist. Auch der BT878 kann über den PCI-Bus direkt in diesen Speicher schreiben.

Die meisten Grafikadapter arbeiten in einem Modus, in dem der Grafikspeicher so organisiert ist, wie es der "Packed Mode" des BT878-RISC-Codes verlangt. Um nun die Organisation eines solchen Grafikspeicher zu beschreiben, sind folgende Angaben ausreichend:

- Adresse des ersten Bildpunktes
- Breite und Höhe des sichtbaren Bildes in Punkten
- Versatz zwischen den einzelnen Bildpunkten innerhalb einer Zeile
  Dies ist natürlich von der Farbtiefe abhängig, mit der der Grafikadapter arbeitet. Dennoch ist es möglich, daß verschiedene Farbtiefen denselben Versatz zwischen zwei Bildpunkten aufweisen. Dies ist z.B. bei 16 Bit Farbtiefe und 15 Bit Farbtiefe aber oft auch bei 24 Bit und 32 Bit Farbtiefe der Fall.
- Versatz zwischen den ersten Bildpunkten einer jeden Zeile
  Dieser Wert kann nicht aus der Breite des Bildes und dem Versatz zwischen den Bildpunkten errechnet werden, da es zulässig ist, hinter den angezeigten Punkten noch weitere, unsichtbare Bildpunkte zu haben.
- Farbformat eines Bildpunktes innerhalb einer Speicherzelle
  Auch dieser Wert ist natürlich von der Farbtiefe abhängig.

Außerdem ist noch speziell für JX das `Memory`-Objekt nötig, in dem der Bildschirmspeicher liegt. All dies wird zu einem Objekt des Typs `PackedFramebuffer` zusammengefaßt. Der Treiber des Grafikadapters sollte also eine Methode bereitstellen, die ein Objekt dieses Types zurückliefert.

Ausgehend vom `PackedFramebuffer` Objekt des Grafikspeichers ist es prinzipiell schon möglich, einen RISC-Code zu erzeugen, der das Videobild in den Grafikspeicher schreibt. Da aber das Videobild wohl eher selten in der linken oberen Ecke des Bildschirms eingeblendet werden soll, muß nun der Bereich spezifiziert werden, in dem das Videobild erscheinen soll.

Es wäre möglich, dies einfach durch ein Tupel (X-Position, Y-Position, Breite, Höhe) zu erreichen, das zusätzlich zum `PackedFramebuffer`-Objekt dem Capture-Treiber übergeben wird. Andererseits sollen ja noch andere Modifikation in Spiel kommen. Wie soll z.B. spezifiziert werden, daß der RISC-Code zwischen zwei Zeilen eines Halbbildes immer eine Zeile freilassen soll, in die dann das zweite Halbbild geschrieben werden soll?

Ein anderes Vorgehen, das flexibler und erweiterbar ist, also ganz den Entwicklungszielen entspricht, verwendet das bereits eingeführte `PackedFramebuffer` Interface und erweitert es zu einem Baukasten-System.

Soll z.B. ein Unterbereich eines schon vorhandenen `PackedFramebuffer`-Objektes erzeugt werden, so reicht simple Adress-Arithmetik aus, um einen neuen, passenden `PackedFramebuffer` zu erzeugen (s. Bild 7). So muß bei einem Ausschnitt nur die Adresse des ersten Bildpunktes sowie die Höhe und die Breite angepaßt werden. Dies wird durch die Klasse `SubFramebuffer` erreicht, welche bei der Erzeugung eines Objektes aus dem `PackedFramebuffer` und den übergebenen Ausschnittinformationen einen neuen `PackedFramebuffer` erstellt. Dieses neue Objekt ist völlig unabhängig vom alten Framebuffer- Objekt und kann anstelle des alten Objektes überall verwendet werden.

Ganz ähnlich kann man vorgehen, wenn man auf dem Bildschirm die beiden Halbbilder ineinander verschachtelt darstellen möchte: Durch die Klasse `InterlacedFramebuffer` können die einzelnen Halbbilder mit je einer Leerzeile geschrieben werden. Auch hier ist simple Adress-Arithmetik ausreichend (vgl. Bild 7).

Da ja die so entstehenden Objekte ebenfalls das Interface `PackedFramebuffer` implementieren, kann man `SubFramebuffer` und `InterlacedFramebuffer` beliebig ineinander schachteln. Insbesondere kommt es oft vor, daß man mehrere `SubFramebuffer` inneinander legt, um nicht nur Videobilder einblenden zu können. So kann man sich einen Teil des Bildschirmes reservieren, um ihn z.B. für Statusausgaben zu nutzen, während der Rest einfach weiter für die Videobilder unterteilt wird.

Das obige Beispiel mit dem Bildausschnitt, in dem die beiden Videobilder verschachtelt dargestellt werden sollen, würde mit den neuen Möglichkeiten im Code folgendermaßen aussehen:

```
PackedFramebuffer framebuffer;         // from graphic driver
PackedFramebuffer subsection =
     new SubFramebuffer(framebuffer, 100, 100, 796, 576);
PackedFramebuffer oddField =
     new InterlacedFramebuffer(subsection, 0);
PackedFramebuffer evenField =
     new InterlacedFramebuffer(subsection, 1);
captureBoard.setFramebufferOdd(frambufferOdd);
captureBoard.setFramebufferEven(framebufferEven);
```

Wie man sieht, wird die Schnittstelle zum Treiber durch die neuen Möglichkeiten nicht aufgebläht und bleibt dennoch sehr flexibel.

Ein weiterer Punkt, der noch bei der Darstellung von Videobildern am Bildschirm fehlt, ist das sogenannte Clipping. Beim Clipping werden Bereiche des eingeblendeten Videobildes ausgeschnitten und nicht neu in den Grafikspeicher geschrieben, damit an diesen Stellen Fenster oder Popup-Menüs, die das Videobild überdecken, nicht zerstört werden.

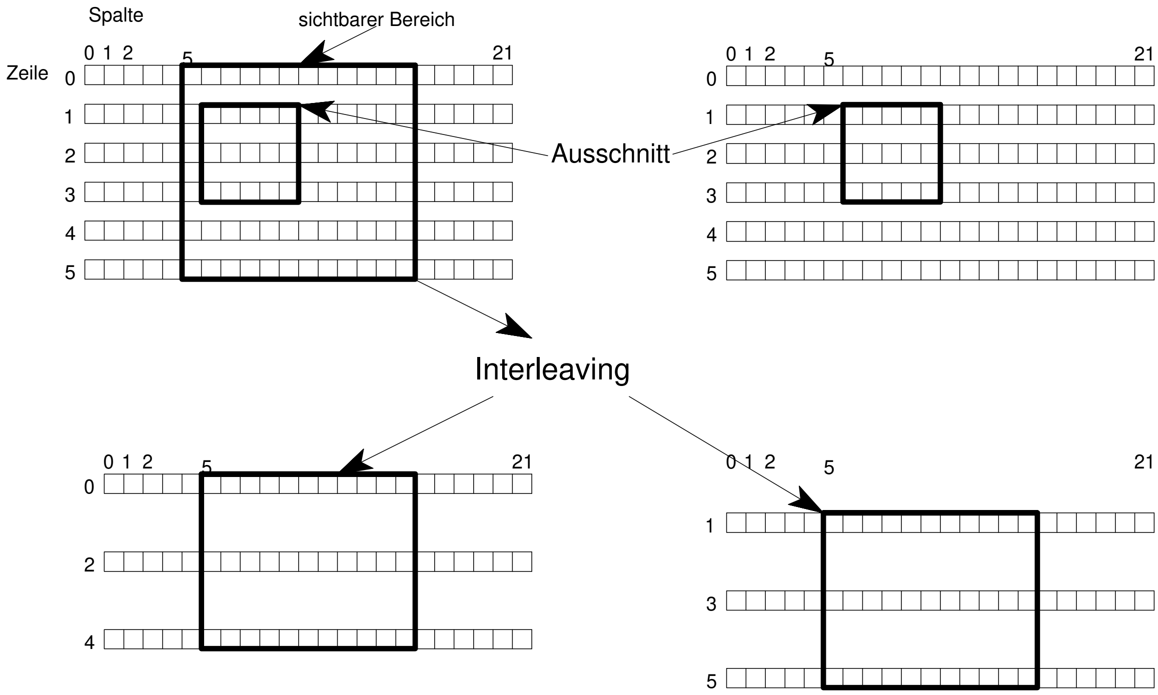

Bild 7: Manipulation von Framebuffer Objekten

Beim BT878 erfolgt das Clipping durch den RISC-Code. Durch den `skip` Befehl (s. Tabelle 2, S. 21) wird einfach eine bestimmte Anzahl von Bytes aus dem Datenstrom "vergessen". Die für das Clipping notwendigen Informationen könnten natürlich als weitere Information in das Framebuffer Interface integriert werden. Aber dadurch würde verhindert, daß das Framebuffer-Konzept auch für andere Anwendungsfälle verwendet werden könnte, bei denen das Clipping stören könnte. Daher wurde stattdessen eine Methode direkt ins Treiber-Interface eingefügt, mit der dem Treiber eine Liste von auszuschneidenden Rechtecken für jedes Halbbild übergeben werden kann. Auf diese Weise ist es auch einfacher, die Clipping-Rechtecke zu ändern, da es so nicht notwendig ist, die verschiedenen, ineinander verschachtelten Framebuffer-Klassen erneut auf dieselbe Weise, aber mit geänderten Clipping- Informationen, zu verschachteln.

Speichern in eine Datei

Der zweite große Anwendungsfall nach der Darstellung am Bildschirm ist die Sicherung der Videodaten in einer Datei. Dazu ist ein viel größerer Aufwand notwendig, als es bei der bloßen Darstellung der Videodaten der Fall ist. Der Grund liegt in den Beschränkungen des DMA-Prozessors. Er kann den Datenstrom nur an konstante Adressen schreiben. Dadurch ist es nicht möglich, die Bilddaten als Strom einfach kontinuierlich in den Speicher zu schreiben.

Um dies dennoch zu erreichen, ist es notwendig, den aktiven RISC-Code für jedes Halbbild zu verändern, sobald es vollständig in den Speicher geschrieben wurde. Auf diese Weise kann das kontinuierliche Schreiben simuliert werden. Es gibt zwei prinzipielle Verfahren, den RISC-Code entsprechend zu ändern.

Die erste Möglichkeit wäre, einfach ein neues Framebuffer-Objekt zu erzeugen, das im Gegensatz zum vorher aktiviertem Framebuffer nur einen anderen Speicher benutzt. Dies hat den Vorteil, daß es nahezu mit den schon vorhandenen Möglichkeiten auskommt. Es muß nur ein Verfahren eingeführt werden, mit dem das Anwendungsprogramm feststellen kann, wann ein Halbbild vollständig im Speicher abgelegt wurde. Der Nachteil ist natürlich, daß alle Schritte, die zur Erzeugung der RISC-Codes notwendig sind, immer wieder ausgeführt werden müssen.

Als zweite Möglichkeit kann man den vorhandenen RISC-Code direkt verändern, indem man die alten Adressen im Code mit neuen Werten überschreibt. Dazu ist es natürlich notwendig, daß sich der Treiber merkt, an welchen Stellen des erzeugten RISC-Codes Adressen stehen und welche die Befehle beinhalten. Auch diese Methode erfordert eine Rückmeldung an das Anwendungsprogramm.

Durch die Verwaltung der Information über die Lage der Speicheradressen und die zusätzlichen Sicherheitsüberprüfungen, die das Überschreiben von Speicher jenseits der Framebuffer-Objekte verhindern sollen, entstehen natürlich ebenfalls Kosten. Ein konkreter Vergleich der Kosten der beiden Methoden soll aber auf einen späteren Abschnitt verschoben werden, in dem die Interna des Treibers erklärt werden.

Beide Möglichkeiten erfordern, wie schon erwähnt, daß das Anwendungsprogramm benachrichtigt wird, wenn ein Halbbild vollständig in den Speicher geschrieben wurde. Der BT878 bietet mehrere Möglichkeiten, dies mittels eines Interrupts zu signalisieren ([BT878], S.49ff, ,S.130f). Als Schnittstelle zum Anwendungsprogramm bietet sich das Observer-Notifier-Pattern ([Gam97] S.293ff) an. Die Anwendung muß dazu ein Portal implementieren, welches dem Treiber bekannt gemacht wird. Dieser kann die Anwendung dann entsprechend benachrichtigen.

An beiden Methoden ist bemerkenswert, daß im Treiber selbst keine Strategie festgelegt wird, wann und auf welche Weise ein neues Framebuffer-Objekt aktiviert wird. Das heißt, es ist Aufgabe des Anwendungsprogrammes, diese Strategie zu implementieren. Dies ist auch notwendig, da für die Speicherung von PAL-Vollbildern bei 16 Bit Farbtiefe eine garantierte Transferrate von ca. 22 MByte/s notwendig ist. Für das Speichern von Videodaten in Dateien gibt es verschiedene Strategien, die je nach Anwendungsfall unterschiedliche Vorteile haben:

- Verwendung nur eines Framebuffer Objektes: Der BT878 schreibt die Videodaten kontinuierlich in das Framebuffer-Objekt und überschreibt dabei ältere Videobilder. Die Anwendung liest die Daten kontinuierlich aus dem Speicher und schreibt sie in die Datei.

  Dieses Verfahren hat den Vorteil der einfachsten Implementierung. Der Nachteil ist, daß je nach Geschwindigkeit des Massenspeichersystems die Bilder aus einem etwas neueren und einem etwas älteren Halbbild bestehen. Je nach Anwendungsfall mag dieses Verfahren ausreichend sein.

- Verwendung mehrerer Framebuffer-Objekte, die als Ringpuffer verwendet werden: Der BT878 schreibt jedes Halbbild in einen neuen Framebuffer, wobei der Wechsel der Framebuffer durch den Observer des Anwendungsprogrammes erfolgt. Das Anwendungsprogramm schreibt die Framebuffer nacheinander in die Datei.

  Dies ist eine Verallgemeinerung des "Double-Buffer"-Verfahrens mit beliebig vielen Puffern. Dadurch können Schwankungen der Transferrate besser abgefedert werden. Durch das Verfahren kann aber nur dann jedes Einzelbild abgespeichert werden, wenn der Massenspeicher im Durchschnitt schneller speichern kann als der BT878 die Daten anliefert. Anderenfalls gibt es einen Pufferüberlauf, wodurch Bilder verloren gehen.

Der Wechsel der Framebuffer-Objekte kann sowohl durch die Übergabe eines neuen Framebuffers, als auf durch die Adress-Modifikation des vorhandenen RISC-Codes erfolgen.

Als Vorteile des Verfahrens ist die korrekte Speicherung der Videobilder zu nennen. Andererseits ist die Verwaltung der einzelnen Puffer aufwendiger, vor allem, wenn man Überläufe garantiert verhindern muß.

Nachdem nun auch die Strategien zum Wechsel der Framebuffer besprochen wurde, bleibt nur noch eine Kleinigkeit übrig: die überall erwähnten Framebuffer-Objekte.

Man könnte natürlich einen Teil des Grafikspeichers für diesen Zweck verwenden, da er ja genauso anzusprechen ist wie der normale Hauptspeicher. Will man aber eine große Anzahl von Framebuffer-Objekten verwalten, kommt man um die Verwendung des Hauptspeichers nicht herum. Auch hier bewährt sich das Framebuffer-Konzept: Statt nun vom Treiber der Grafikkarte ein Framebuffer-Objekt erzeugen zu lassen, kann sich das Anwendungsprogramm selbst ein Objekt mit passendem Typ implementieren. Als Beispiel dazu kann die Klasse `MemoryFramebuffer` dienen, die sich von der Speicherverwaltung einen großen Speicherblock allokiert und ihn als `PackedFramebuffer`-Objekt behandelt.

Videokonferenz

Ein weiterer Anwendungsfall für einen Video-Capture-Treiber ist die Übertragung der Videodaten über ein Netzwerk. Entweder sollen auf dem Empfangsrechner die Videodaten angezeigt oder gespeichert werden. In beiden Fällen unterscheidet sich das prinzipielle Vorgehen nicht von den Verfahren, die bei der Speicherung von Videodaten in eine Datei angewendet werden. Nur ist bei einem Netzwerk verstärkt damit zu rechnen, daß die verfügbare Bandbreite erstens sehr stark schwankt und zweitens bedeutend geringer ist als bei der Speicherung auf die lokale Festplatte. Je nach verwendetem Protokoll sind außerdem Paketverluste zu erwarten. Aber all dies berührt nicht den Capture-Treiber an sich, sondern ausschließlich das Anwendungsprogramm.

Weitere Anwendungsfälle

Die meisten anderen Anwendungsfälle für den Treiber lassen sich sehr einfach auf einen der schon behandelten Fälle zurückführen. So erfordert z.B. die Bearbeitung von Videos oder das Vorschalten einer Video-Kompression vor die Speicherung oder Übertragung der Videodaten keine weitere Unterstützung des Treibers.

Nachdem nun die Schnittstelle des Treibers zum Anwendungsprogramm festgelegt und erläutert wurde, werden im folgenden Kapitel die "Innereien" des Treibers entworfen und erklärt.

5 Interner Aufbau des Treibers

Nachdem die Schnittstelle des Treibers zum Anwendungsprogramm hin erläutert wurde, soll nun der Aufbau und die Arbeitsweise im Inneren des Treibers erklärt werden. Dabei werden zuerst die Teile des Treibers erklärt, die für den direkten Zugriff auf die Hardware zuständig sind. Dies umfaßt folgende Gebiete:

- allgemeine PCI-Konfigurationsregister
- Video-Registerblock des BT878
- Audio-Registerblock des BT878
- I^2C-Geräte am I^2C-Bus des BT878
- Interrupt-Behandlung

Im weitesten Sinne noch zur Hardware gehört auch die Erzeugung und Verwaltung des RISC-Codes.

Diese Punkte werden in den nächsten Abschnitten behandelt. Anschließend werden die Bereiche des Treibers beleuchtet, welche die Verbindung der Anwendungsschnittstelle zu den Hardwareklassen herstellen.

Doch nun zuerst zu den grundlegenden Operationen eines PCI-Gerätes.

5.1 PCI Konfigurations-Zugriffe

Bei der Initialisierung des Treibers muß dieser den BT878 am PCI-Bus finden und ihn identifizieren um ihn anschließend initialisieren zu können. Dazu stellt jedes PCI-Gerät einen PCI-Header bereit, mit dem auf standardisierte Weise Grundeinstellungen und Ressourcen-Anforderungen eingestellt bzw. ermittelt werden können (vgl. [Shan00]). Auf der Seite des PCI-Busses erfolgt der Zugriff auf diese Register mittels spezieller Konfigurationszyklen und nicht mittels normaler Speicher- oder Port-Zugriffe. Dadurch sind die PCI-Header der einzelnen Geräte in einem speziellen Adressraum untergebracht (vgl. [Shan00], S.309ff).

Die CPU kann auf den Konfigurations-Adressraum zugreifen, indem sie an der PCI-Hostbridge, die den PCI-Bus mit der CPU und dem Speicher verbindet, bestimmte Port-Zugriffe tätigt (vgl. [Shan00], S317ff).

Es bietet sich also an, ein Portal zu erstellen, welches den Zugriff auf die PCI-Header erlaubt und auch geeignete Methoden für den Zugriff auf die einzelnen Register der PCI-Header zur Verfügung stellt. Da man aber in der Regel nicht mit dem PCI-Bus an sich, sondern mit dem PCI-Header eines einzelnen Gerätes arbeiten will, wurde als weitere Klasse `PCIDevice` eingeführt, die über das Portal auf die Register zugreift und diese mittels geeigneter Methoden dekodiert. Dadurch ist es möglich, das Portal-Interface minimal und deshalb sehr übersichtlich zu gestalten. Außerdem kann durch die Aufteilung die Systemsicherheit erhöht werden, indem ein spezieller Verwalter eingesetzt wird, der die Verteilung von `PCIDevice` Objekten überwacht.

Wie ein Treiber auf einen Wert im PCI-Header zugreift, ist in Bild 8 als Kollaborationsdiagramm dargestellt. Bei seiner Initialisierung läßt sich ein Treiber, z.B. der des BT878, über das Portal `PCIAccess` alle vorhandenen PCI-Geräte zurückgegeben. An dem zurückgegebenen `PCIDevice` ruft er nun die Identifikation des Gerätes ab, um festzustellen, ob das Gerät einen BT878 enthält. Wenn dem so ist, dann aktiviert er das Gerät, indem er das PCI-Command Register setzt. Auf die gleiche Weise kann der Treiber die Adresse des Speichers abfragen oder setzen, an der die Register des Gerätes in den Hauptspeicher eingeblendet werden.

Die Klasse `PCIDevice` stellt also Methoden zur Verfügung, um die einzelnen Felder der Register bequem zugänglich zu machen. Die Klasse `PCIAccess` erhält nur noch komplette Register, die ja durchaus mehrere unabhängige Werte enthalten können und oft noch speziell kodiert sind.

Eines dieser Register enthält, wie schon erwähnt, die Adresse des Speicherblockes, der die eigentlichen Register des BT878 enthält. Um diesen Block geht es im folgenden Abschnitt.

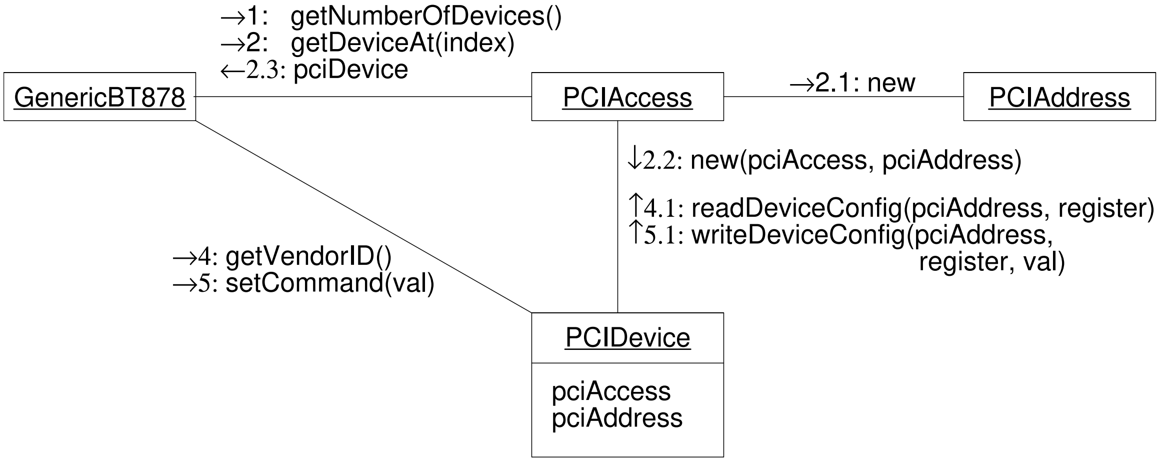

Bild 8: Kollaborationsdiagramm "PCI-Konfigurationszugriff"

5.2 Registerblock des BT878

Nachdem nun der Treiber die PCI-Konfigurationsregister des BT878 gefunden hat, kann er aus einem dieser Register die Adresse auslesen, an der der eigentliche Registersatz des BT878 in den Hauptspeicher der CPU eingeblendet wird. Diese Adresse wurde vom BIOS beim Starten des Rechners eingetragen.

Mithilfe dieser Adresse wird nun ein Objekt der Klasse `BT878Video` erzeugt. Diese Klasse hat die Aufgabe, die einzelnen Register des BT878 auf komfortable Weise zur Verfügung zu stellen. Dafür gibt es für jedes Register Methoden zum Lesen und Schreiben der Register. Auf diese Weise sind sämtliche Bitmanipulationen zum Lesen und Schreiben innerhalb der Klasse gekapselt. Außerdem ist so ein atomarer Zugriff auf ein Register möglich, das über mehrere Adressen verteilt ist[3]. Die Klasse `BT878Video` "implementiert" außerdem das Interface `BT878Defines`, welches ähnlich einer Header-Datei in C

[3]Von dieser Möglichkeit wird allerdings im Moment kein Gebrauch gemacht.

die für die Bitmanipulationen notwendigen Konstanten enthält. Dies ist möglich, da laut der Java-Sprachdefinition [JLS00], Kap.9.3 Definitionen von Variablen innerhalb eines Interface automatisch `public,final,static` sind und sich daher für die Definition von Konstanten hervorragend eignen.

Analog zu `BT878Video` dazu wäre es möglich, die Audio-Funktionalität des BT878 zu kapseln. Diese Funktionalität des BT878 ist in einem eigenen PCI-Gerät auf demselben Chip untergebracht und hat so neben einem eigenen PCI-Header einen eigenen Registersatz. Allerdings wird ja vom vorliegenden Treiber keine Verarbeitung des Audio-Signales unterstützt und so fehlen auch die entsprechenden Klassen.

Wie aber erfolgt nun letztendlich der Zugriff auf die Register von Java aus? Das JX-System bietet hierfür ein spezielles Portal `DeviceMemory` an, welches ein Speicherobjekt über einen bestimmten Speicherbereich legt. Mittels der Methoden des Portals kann nun auf die einzelnen Speicherstellen und damit auf die Register zugegriffen werden.

Die Methoden, die `BT878Video` anbietet, ermöglichen es nun den höheren Funktionen des Treibers, bequem und sicher auf die zugrundeliegende Hardware zuzugreifen.

5.3 I²C-Geräte am BT878

Der BT878 hat, wie schon erwähnt, einen integrierten I²C-Bus, durch den die anderen Chips gesteuert werden, die auf den meisten Karten vorhanden sind. Zu den Geräten, die auf diese Weise heute üblicherweise auf den Karten integriert sind, zählen:

- **TV-Tuner:** Er wandelt das HF-Antennensignal in das vom BT878 benötigte Composite-Signal um (vgl. Kap. 2.4).
- **EEPROM:** Hier werden meist einige Konfigurationsdaten gespeichert. Der Treiber unterstützt zwar das Auslesen des EEPROM, benötigt aber selbst keine Information daraus.
- **Infrarot-Empfänger:** Eine nette Zugabe vieler Hersteller erlaubt es, mittels eines Chips und eines Sensors eine Fernbedienung abzufragen, um damit z.B. die Kanäle wechseln zu können.
- **Audio-DSP:** Oft ist auch noch ein digitaler Signalprozessor auf den Karten zu finden, der das anliegende Audio-Signal auf vielfältige Weise manipulieren kann.

Damit die am I²C-Bus angeschlossenen Geräte einfach angesprochen werden können, wurde in der Klasse `BT878I2CBus` die Funktionalität zum Lesen und Schreiben von Daten gekapselt. Die Klasse verwendet dabei die durch `BT878Video` gekapselten Register. Damit Geräte, die am Bus des BT878 angeschlossen sind, auch unabhängig vom BT878-Treiber implementiert werden können, wurde zusätzlich noch das Interface `I2CBus` eingeführt, welches natürlich von `BT878I2CBus` ebenfalls angeboten wird.

Beispiele für die Implementierung von I²C-Gerätetreibern finden sich in den Klassen `FM1216`, `I2CEEPROM` und `IRReceiverImpl`. Dabei steuert `FM1216` einen Philips-TV-Tuner (vgl. Kap. 4.5), `I2CEEPROM` ist für das Lesen und Schreiben des EEPROM zuständig und mit `IRReceiverImpl` kann die Fernbedienung der Hauppauge-Karte abgefragt werden. Während der EEPROM-Treiber für das Anwenderprogramm überhaupt nicht sichtbar ist, bieten `IRReceiverImpl` und `FM1216` Portal-Interfaces an, mit denen das Anwenderprogramm direkt Einstellungen vornehmen kann.

Die Modellierung der I^{2}C-Geräte und des I^{2}C-Buses erfolgte also spiegelbildlich zur vorhandenen Hardware, wie man in Bild 9 sehr schön erkennen kann.

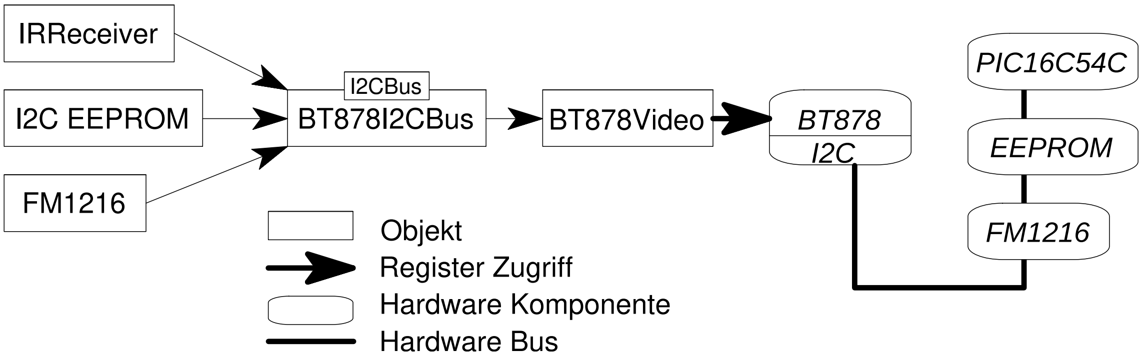

Bild 9: Klassen und Geräte des I^{2}C-Buses

5.4 Interrupt-Behandlung

Ein weiterer relativ hardwarenaher Aspekt des Treibers ist die Behandlung von Interrupts, die der BT878 erzeugt. Er kann aus bis zu 17 verschiedenen Gründen einen Interrupt generieren (vgl. [BT878], S.131ff.), aber glücklicherweise braucht man sich um die meisten davon nicht zu kümmern, da sie lediglich unwichtige Statuswechsel anzeigen. Diese Interrupt-Quellen können daher selektiv ausgeschaltet werden. Für den Treiber sind vor allem zwei Interrupts interessant: die Anzeige, daß das Halbbild wechselt und die Anzeige, daß im RISC-Code ein Interrupt angefordert wurde. Diese Ereignisse können verwendet werden, um den in Kapitel 4.9 geforderten Mechanismus zu implementieren, mit dem das Anwendungsprogramm benachrichtigt wird, wenn ein Halbbild vollständig in den Speicher geschrieben wurde, so daß die Framebuffer umgeschaltet werden können.

Wie kann nun unter JX eine Interrupt-Routine implementiert werden? Wie üblich erfolgt die hardwarenahe Programmierung des PIC (Programmable Interrupt Controller) (vgl. [Intel97], S.74ff) innerhalb der Implementierung des Portals `IRQ` von `DomainZero`. Das `IRQ`-Portal erlaubt es, einen bestimmten Interrupt der CPU an ein Portal der Typs `FirstLevelIrqHandler` zu binden. Dieses zweite Portal wird aufgerufen, sobald die CPU einen entsprechenden Interrupt erhält.

Das entsprechende Portal im Treiber ist in der Klasse `InterruptHandler` implementiert worden. Bei der Bearbeitung eines Interrupts wird zuerst geprüft, ob der Interrupt vom BT878 oder von einem anderen PCI-Gerät mit demselben Interrupt stammt, denn auf dem PCI-Bus muß jedes Gerät damit rechnen, daß es sich den Interrupt mit anderen Geräten teilen muß. Anschließend werden die Rückruf-Portale der registrierten Observer (vgl. Kapitel 4.9) aufgerufen.

Die Behandlung von Interrupts ist beim BT878 also relativ einfach und unspektakulär. Im folgenden Abschnitt wird es aber wieder interessanter, wenn es um die Erzeugung des RISC-Codes geht.

5.5 Erzeugung von einfachem RISC-Code

Die Erzeugung von RISC-Codes für den DMA-Controller des BT878 ([BT878], S.46ff) ist ein zentraler Punkt beim Entwurf der Interna des Treibers. Im Vordergrund steht dabei natürlich die Flexibilität, so daß Erweiterungen modular implementiert werden können.

Die zum RISC-Befehlssatz gehörenden Konstanten wurden im Interface `RISC` verpackt, welches nur Konstanten, aber keinerlei Methoden anbietet. Wie schon in Kapitel 5.2 erwähnt, sind solche Definitionen automatisch `public,final` und `static`.

Damit man sich bei der Erzeugung von RISC-Code nicht immer mit den Konstanten aus dem `RISC`-Interface und der Bitkodierung der Befehle herumschlagen muß, gibt es die Klasse `RISCGenerator`, die auf unterster Ebene RISC-Code auf einfache Art erzeugt. Diese Klasse stellt für jeden RISC-Befehl eine oder mehrere Methoden bereit, um den Befehl in ein `Memory`-Objekt zu schreiben. Das `Memory`-Objekt kann dann anschließend dem BT878 übergeben werden. Ein einfaches Beispiel demonstriert das prinzipielle Vorgehen:

```
RISCGenerator code = new RISCGenerator(1000)
code.sync(RISC.FIFO_FM1);       //wait for packed mode data
for(int y=0; y<height; ++y)
     code.write(scanlineLength,
framebufferStart+scanlineOffset*y)
code.sync(RISC.FIFO_VRE);       //wait for end of even field
code.inval();                   //invalid cmd stops
DMA-Controller
bt878.setRISCStartAddress(code.getStartAddress());
```

Auf diese Weise wird der Code erzeugt, um ein Halbbild in einen normalen Framebuffer zu schreiben und dieser Code wird anschließend gleich dem BT878 bekannt gemacht. Das Beispiel zeigt auch die Verwendung der Konstanten aus dem Interface `RISC`.

Im einfachsten Fall könnte man also ein RISC-Programm erzeugen, das innerhalb einer Schleife[4] nacheinander die beiden Halbbilder in den Speicher schreibt. Schaltet nun das Anwendungsprogramm ein Halbbild aus, so muß der gesamte Code neu geschrieben werden. Noch schlimmer wird es, wenn das Clipping ins Spiel kommt, da dann die Berechnung des RISC-Codes nicht mehr so simpel wie im Beispiel ist. Es ist also eine bessere Strukturierung des RISC-Codes wünschenswert, damit die einzelnen Halbbilder im RISC-Code unabhängig zu behandeln sind.

Da der DMA-Controller nicht das Konzept von Prozeduren unterstützt, sondern nur Sprungbefehle, muß eine entsprechende Funktionalität mit den Sprungbefehlen simuliert werden. Bild 10 zeigt eine Aufteilung des RISC-Codes in drei Einzelblöcke, die durch Sprungbefehle verbunden sind. Soll nun z.B. das zweite Habbild ausgeschaltet werden, genügt es, den zweiten Sprungbefehl im linken Block so umzuschreiben, daß er nicht auf den rechten unteren Block verweist, sondern direkt hinter sich selbst, also auf die Marke `main_loop`. Soll anschließend das zweite Halbbild einen anderen RISC-Block erhalten, weil sich z.B. die Clipping-Informationen geändert haben, muß man zuerst im neuen

[4]Jump-Befehle werden vom BT878 unterstützt (vgl. Tabelle 2, S. 21).

RISC-Block die Rücksprung-Adresse richtig setzen, was entweder schon bei der Erzeugung des Blocks oder auch erst nachträglich geschehen kann, um anschließend im rechten "Sprungblock" die Adresse des neuen RISC-Blocks in den Sprungbefehl bei der Marke `main_field2` einzusetzen.

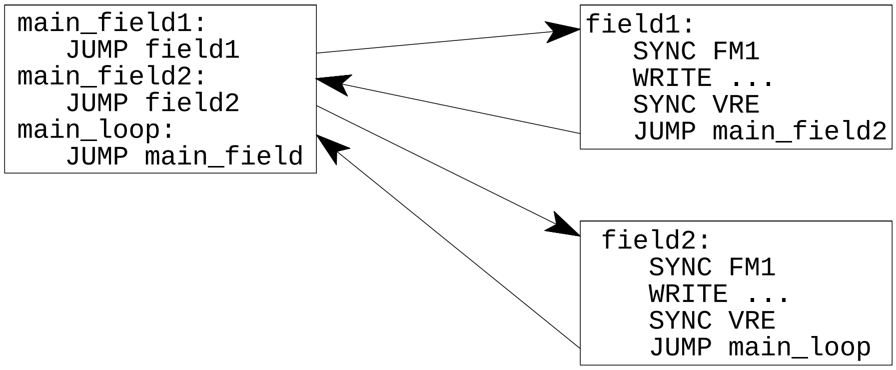

Bild 10: Unterprogramme im RISC-Code

Diese Funktionalität wird mittels der Klassen `RISCJumpBlock` und `RISCFieldBlock` angeboten, die sich von `RISCGenerator` ableiten. In der Klasse `RISCFieldBlock` werden die Methoden, die in den RISC-Code einen Sprungbefehl einfügen, so erweitert, daß sich das Objekt die Stelle des Sprungbefehles merkt. Mit neuen Methoden kann nun dieser Sprungbefehl manipuliert werden. In der Klasse `RISCJumpBlock` ist die Funktionalität enthalten, mehrere `RISCFieldBlock`-Objekte mittels Sprungbefehlen zu verketten, einzelne Unterblöcke zu überspringen oder durch neue zu ersetzen.

Die Klassen `RISCFieldBlock` und `RISCJumpBlock` ergeben zusammen eine sehr komfortable und mächtige Schnittstelle zu Verwaltung des RISC-Codes. Es ist so möglich, den RISC-Code zu strukturieren um z.B. den RISC-Code für einzelne Halbbilder separat auszutauschen oder zu verändern. Gleichzeitig bieten diese Klassen die Grundlage für Erweiterungen, die es erlauben, den RISC-Code auf komplexere Weise zu modifizieren, so wie es im nächsten Abschnitt beschrieben wird.

5.6 Erzeugung von verschiebbarem RISC-Code

Das im vorherigen Abschnitt behandelte Verfahren zu Erzeugung von RISC-Code stellt natürlich nur das absolute Minimum an Funktionalität zur Verfügung - jedenfalls aus Sicht des Anwenderprogrammes. Denn mit dem so erzeugten Code können die Videobilder immer nur an eine feste Speicherposition geschrieben werden, wobei neuere Bilder die alten einfach überschreiben. Soll aber das Bild, wie schon in Kapitel 4.9 dargelegt, in eine Datei geschrieben oder über ein Netzwerk übertragen werden, muß mit jedem Halbbild ein neuer RISC-Code erzeugt und installiert werden.

Wenn man aber davon ausgeht, daß sich die Speicherorganisation selbst zwischen den Halbbildern nicht ändert, das nächste Halbbild also nur im Speicher verschoben werden soll, so bietet sich ein anderes Verfahren an, das zudem vielleicht auch schneller ist.

Man könnte nämlich den schon vorhandenen RISC-Code einfach so abändern, daß die darin gespeicherten Zieladressen auf einen neuen Speicherbereich verweisen. Wenn im

Treiber nachgeprüft wird, daß der neue Speicherbereich ebenfalls durch ein `Memory`-Objekt abgedeckt ist, ergeben sich daraus auch keine Sicherheitsprobleme.

Da man die Zieladressen innerhalb eines schon existierenden `RISCFieldBlock` nicht auf einfache und damit schnelle Art und Weise herausfinden kann, müssen bereits bei der Erzeugung des Codes entsprechende Maßnahmen getroffen werden.

Konkret wird bei der Erzeugung von verschiebbarem RISC-Code[5] statt der Klasse `RISCFieldBlock` die Klasse `RISCFieldBlockReloc` verwendet. Diese abgeleitete Klasse überlädt in erster Linie die Methoden, die im RISC-Code Schreibbefehle erzeugen und merkt sich die Position der darin verwendeten Adressen in einem Array.

Später kann diese Information sehr einfach dazu verwendet werden, den bereits vorhandenen RISC-Code an neue Speicheradressen schreiben zu lassen.

Da `RISCFieldBlockReloc` von der bisher verwendeten Klasse `RISCFieldBlock` abgeleitet ist, kann der Algorithmus zur Erzeugung des RISC-Codes unverändert auch auf ein Objekt der abgeleiteten Klasse angewendet werden. Das einzige Problem dabei ist, daß der Algorithmus kein Objekt übergeben bekommt, sondern dieses erst innerhalb des Algorithmus erzeugt wird. Denn bei der Erzeugung des jeweiligen Objektes muß die Größe des entstehenden RISC-Programmes bekannt sein. Diese Größe hängt aber erstens vom jeweiligen Framebuffer-Objekt und zweitens vom verwendeten Erzeugungsalgorithmus ab. Denn z.B. der im nächsten Abschnitt eingeführte Clipping-Algorithmus braucht mehr RISC-Befehle als der Code für ein einfaches Rechteck, da ja beim Clipping eine Zeile des Videobildes mit mehreren Schreib- bzw. Überspring-Befehlen behandelt werden muß.

Daher wird dem Algorithmus nicht ein schon erzeugtes `RISCFieldBlock` oder `RISCFieldBlockReloc` Objekt übergeben, sondern ein Objekt mit dem Interface `FieldBlockCreator`, welches der Algorithmus verwendet, um das eigentliche RISC-Code Objekt zu erzeugen. Der Vorgang ist in Bild 11 dargestellt, wobei die Gerätetreiberklasse `WinTV` den Algorithmus zum Erzeugen eines rechteckigen Videobildes aufruft. Dabei wird, je nach Anwendungsfall, ein Objekt der Klasse `RISCFieldBlocCreator` oder `RISCFieldBlockRelocCreator` übergeben. Der Algorithmus berechnet zuerst aus dem ebenfalls übergebenen Framebuffer-Objekt die Größe des zu erzeugenden RISC-Codes, um anschließend mit dem Erzeuger-Objekt den Code-Block anzufordern. Danach wird der RISC-Code erzeugt, indem die entsprechenden Methoden an dem soeben erzeugten Objekt aufgerufen werden. Anschließend wird der Code-Block an den Aufrufer zurückgegeben.

Nachdem nun die Erzeugung des verschiebbaren RISC-Codes erläutert wurde, stellt sich natürlich die Frage, ob sich denn der ganze Aufwand lohnt und ob es nicht doch einfacher wäre, einen einfache `RISCFieldBlock` immer wieder mit neuen Zieladressen zu erzeugen.

Um dies zu untersuchen, wurde gemessen, wie schnell die einzelnen Varianten von RISC-Code erzeugt werden können. Die Messungen erfolgten dabei auf einem P2 mit 500MHz.

[5] Verschiebbarer RISC-Code meint im Folgenden nicht die Verschiebung des Codes sondern eine Verschiebung des entsprechenden Halbbildes im Speicher.

- Die Erzeugung mittels `RISCFieldBlock` benötigt etwa 0.21ms, das entspricht einer Bildrate von etwa 4800 Halbbildern pro Sekunde.
- Die Erzeugung mittels `RISCFieldBlockReloc` benötigt etwa 0.25ms, was einer Bildrate von etwa 4000 Halbbildern pro Sekunde entspricht.
- Die Verschiebung des Zielspeichers benötigt etwa 0.11ms und entspricht einer Bildrate von etwa 9000 Halbbildern pro Sekunde.

Wie zu erwarten war, ist durch die zusätzliche Ableitungsstufe und das Überschreiben der write-Methoden die Erzeugung von verschiebbarem RISC-Code etwas langsamer als die "pure" Code-Erzeugung. Es zeigt sich auch, daß beide Arten der Code-Erzeugung wohl kaum einen Flaschenhals darstellen, da die Bildwiederholfrequenz sowieso nur 50 bis 60 Hertz beträgt. Andererseits ist die anschließende Verschiebung des Bildspeichers doch deutlich billiger als die Erzeugung eines komplett neuen RISC-Codes.

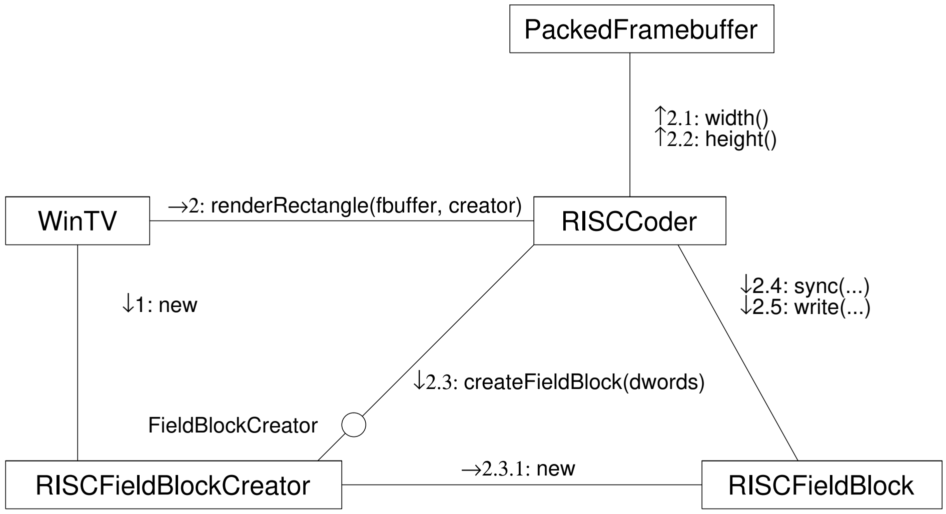

Bild 11: Erzeugung eines `RISCFieldBlock`

Das Anwendungsfeld für die Verschiebung der Zieladressen im RISC-Code ist die Speicherung oder Übertragung der Videodaten als kontinuierlicher Strom. Ein Anwendungsfall mit ganz anderen Anforderungen an den RISC-Code, nämlich die Darstellung auf dem Bildschirm, wird im folgenden Abschnitt erläutert.

5.7 Erzeugung von RISC-Code mit Clipping-Informationen

Soll das Videobild in einem Fenster am Bildschirm dargestellt werden, so kann es vorkommen, daß andere Fenster oder Popup-Menüs das Videobild ganz oder teilweise verdecken. Trifft man für diesen Fall keine Vorkehrungen, so überschreibt der BT878 die Fenster oder Menüs, sobald er das nächste Halbbild in den Speicher schreibt. Um dies zu vermeiden, muß man den RISC-Code des BT878 so anpassen, daß die entsprechenden Speicherbereiche beim Schreiben ausgespart werden. Dies erreicht man, indem man den DMA-Controller ein bestimmtes Stück aus dem Datenstrom vergessen läßt und dann an einer entsprechend höheren Speicherstelle mit dem Schreiben fortfährt. Dies muß für jede Bildzeile des überlappenden Fensters geschehen. Folgendes einfache Beispiel erzeugt den erforderlichen RISC-Code für eine Bildzeile mit einem Clipping-Rechteck:

```
PackedFramebuffer fb;
RISCFieldBlock rfb;
ClippingRectangle r = new ClippingRectanlge(10, 10, 100, 100);
rfb.write(RISC.SOL, r.xmin()*fb.pixelOffset(), scanlineAddr);
rfb.skip((r.xmax()-r.xmin()) * fb.pixelOffset());
rfb.write(RISC.EOL,
        (fb.width()-r.xmax()) * fb.pixelOffset(),
        scanlineAddr + r.xmax()* fb.pixelOffset());
```

Kompliziert wird es allerdings dadurch, daß man ja nicht nur mit einem überlappenden Fenster rechnen muß, sondern mit beliebig vielen, die sich zudem auch noch teilweise gegenseitig überlappen können, wodurch beliebig komplexe Clipping-Bereiche entstehen können.

Im Rahmen dieser Arbeit wurde dafür ein Algorithmus entwickelt, der sich an das Zeichnen von gefüllten Polygonen anlehnt. Um die Effizienz dieses Algorithmus bewerten zu können, wurde zum Vergleich das Clipping-Verfahren implementiert, das der Treiber des BT878 unter Linux verwendet.

Clipping mit Linux

Beim Linux-Treiber des BT878 kommt ein sehr einfaches, aber sicheres Verfahren zum Einsatz, das aus beliebigen Clipping-Rechtecken den passenden RISC-Code erzeugt. Dabei wird im Kern eine Bitmap angelegt, deren Größe dem größten möglichen Videobild entspricht. In diese etwa 80 kByte große Bitmap wird nun jedes Clipping-Rechteck hineingezeichnet. Anschließend wird die Bitmap wieder Zeile für Zeile untersucht, um so aus zusammenhängenden Bereichen in jeder Zeile der Bitmap den RISC-Code zu erzeugen.

Das Verfahren ist extrem einfach und sicher. Aber es benötigt einen relativ großen Speicherbereich für die Bitmap. Auch ist der Umweg über eine Bitmap nicht sehr elegant.

Um diese Nachteile zu mindern, wird im folgenden Abschnitt ein zweiter Algorithmus dargelegt.

Clipping mit dem Scanline-Algorithmus

Ein völlig anderer Ansatz lehnt sich an ein Verfahren aus der Computergrafik an, das Foley zum Rastern von gefüllten Polygonen verwendet (vgl. [Fol91],S.92ff).

Dabei wird für jede Zeile (Scanline) des Polygons geprüft, welche Kanten des Polygons diese Zeile schneidet, wobei horizontale Kanten ausgelassen werden. Die Schnittkanten werden anhand der Schnittpunkte von links nach rechts sortiert. Anschließend wird zwischen der ersten Kante und der zweiten das Füllmuster eingetragen, zwischen der zweiten und der dritten Kante nichts und zwischen der dritten und vierten Kante wieder das Füllmuster. Dieses Verfahren funktioniert auch bei entarteten Polygonen (Bild 12).

Das Verfahren ist durch geschickte Verwaltung der Kanten und Schnittpunkte sehr schnell.

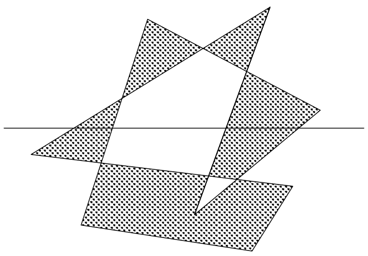

Bild 12: entartetes Polygon

Auch die Erzeugung des Clipping-Codes hat Ähnlichkeiten mit dem Füllen von Polygonen, wenn man das Videobild als Füllmuster begreift. Allerdings wird nicht ein einziges Polygon gefüllt, sondern es müssen mehrere unabhängige Rechtecke ausgespart werden. Das macht einige Modifikationen des Verfahrens notwendig, da sich unabhängige Rechtecke nicht in ein Polygon umrechnen lassen.

Die Grundidee, sich mittels einer Scanline über alle Zeilen des Videobilder zu iterieren und sich die schneidenden Kanten anzusehen, bleibt unverändert bestehen. Aber es ist notwendig, in einer Vorstufe die vertikalen Kanten der Rechtecke gegen alle anderen Rechtecke zu clippen. Die horizontalen Kanten werden komplett ausgelassen. Außerdem kommen zwei weitere, feste Kanten hinzu, durch die die Grenzen des Framebuffer-Objektes dargestellt werden (s. Beispiel Bild 13). Weil nur vertikale Kanten behandelt werden müssen, ist dies ohne großen Aufwand machbar, da derartige Schnittpunktberechnungen viel einfacher sind als bei schrägen Kanten. Auf die Implementierung eines komplexes Clipping-Verfahren für Linien (vgl. [Fol91], S.111ff) kann daher verzichtet werden.

Nachdem die wirklich verwendeten Kanten in der Vorstufe ermittelt wurden, kann im zweiten Schritt das eigentliche Scanline-Verfahren starten.

Es gibt zu Anfang eine Liste aller Kanten, die nach y-Koordinaten sortiert sein muß. Eine weitere Liste, die Liste der aktiven Kanten, ist zu Anfang leer, sie wird aber später immer nach x-Koordinaten sortiert sein.

Nun wird für jede Zeile überprüft, ob gegenüber der vorherigen Zeile neue Schnittkanten hinzukommen und gegebenenfalls werden diese Kanten in die Liste der aktiven Kanten einsortiert. Da die Liste aller Kanten nach y-Koordinaten sortiert ist, fällt das Erkennen neuer Kanten sehr leicht.

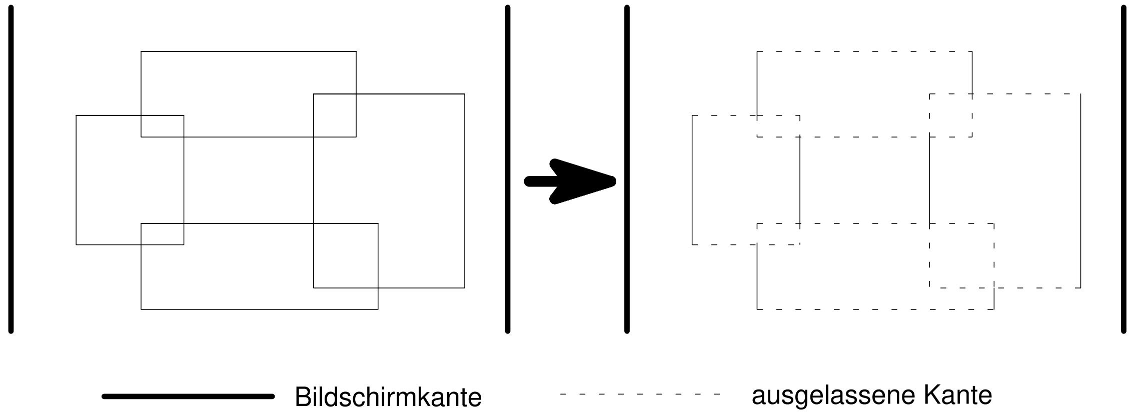

Bild 13: Erzeugung der Clipping-Kanten

Anschließend wird überprüft, ob die Liste aktiver Kanten alte Kanten enthält, um diese gegbenenfalls zu entfernen.

Mit der so aktualisierten Liste der aktiven Kanten wird nun die Zeile gerastert. Abwechselnd wird zwischen den Kanten das Videobild geschrieben oder ausgelassen (s. Bild 14).

Da das Schreiben des Videobildes mit der ersten Kante beginnt, ist es notwendig, bei Clipping-Rechtecken, welche die linke Seite des umgebenden Framebuffer-Objektes berühren, am Rand zwei Kanten an derselben Position einzufügen, so daß bei der Code-Erzeugung ein Bildbereich der Breite Null geschrieben werden kann.

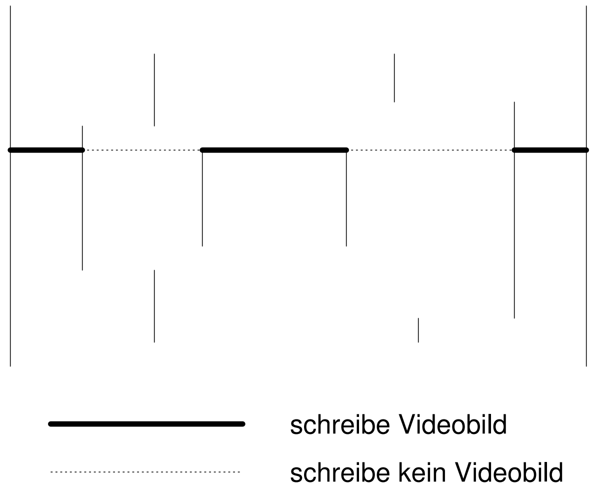

Bild 14: Codeerzeugung einer Bildzeile

Abschließend noch ein paar Bemerkungen zur Implementierung:

Der Algorithmus ist in der Klasse `ScanlineClippingRasterizer` in der statischen Methode `rasterizeField` implementiert. Die statische Methode bietet sich an, da der Algorithmus keine Informationen benötigt, außer dem Framebuffer-Objekt und der Clipping-Liste. Diese Informationen ändern sich aber bei jedem Aufruf, weshalb es unsinnig wäre, jedesmal ein neues Objekt zu erstellen.

Der Algorithmus wurde so allgemein implementiert, daß er nur die Höhe und die Breite des darzustellenden Videobildes sowie natürlich die Clipping-Rechtecke benötigt. Insbesondere verwendet der Algorithmus keinerlei Informationen über die Organisation des Zielspeichers, der verwendeten Farbtiefe und auch über die RISC-Befehle weiss er nichts. Die Verbindung zwischen den rein geometrischen Informationen, den Gegebenheiten des Zielspeichers und den RISC-Befehlen wird stattdessen durch das Interface `ClippingRasterizerOutput` hergestellt. Dies erlaubt es, den Algorithmus sehr schnell auf neue Ausgabegeräte zu portieren.

Darüberhinaus ist durch die Semantik des Interfaces festgelegt, daß die aktuelle Schreibposition des DMA-Controllers intern mitgeführt wird - auch über `skip`-Befehle hinweg. Dadurch ist es für den Clipping-Algorithmus sehr einfach, die Schreib- und Überspring-Befehle zu verwenden, ohne viel Adressarithmetik verwenden zu müssen. Das einführende Beispiel würde dann so aussehen:

```
ClippingRasterizerOutput out;
ClippingRectangle r = new ClippingRectanlge(10, 10, 100, 100);
```

```
out.write(r.xmin(), true, false);                    // only SOL
flag
out.skip(r.xmax()-r.xmin(), false, false);
out.write(fb.width() - r.xmax(), false, true);     // only EOL
flag
```

Wie man sieht, kapselt dieses Vorgehen alle von der Hardware abhängigen Informationen, so daß der Clipping-Algorithmus für alle Farbtiefen und sowohl für den Packed-Mode als auch den Planar-Mode des DMA-Controllers unverändert verwendbar ist.

Vergleich der Clipping-Algorithmen

Die beiden implementierten Clipping-Verfahren unterscheiden sich nicht nur in dem Komplexität des Algorithmus, vielmehr haben beide auch ein sehr unterschiedliches Zeitverhalten. Zur Untersuchung dieses Verhaltens wurden in einer Messreihe sowohl die Größe des Videobildes als auch die Anzahl der verwendeten Clipping-Rechtecke variiert.

Es wurden bei jeder Bildgröße drei verschiedene Clipping-Fälle untersucht:

- **kein Clipping-Rechteck**

  Damit sollen die Fixkosten des Algorithmus bestimmt werden.

- **komplexes Clipping** mit acht sich teilweise überlappenden Rechtecken

  Dieser Testfall stellt einen normalen Anwendungsfall dar.

- **extremes Clipping** mit 100 sich nicht überschneidenden Rechtecken, die etwa ein Viertel des Bildes bedecken.

  Mit diesem Fall soll geprüft werden, wie stark sich die Anzahl der Rechtecke auf den Algorithmus auswirken.

Die Bildgröße geht von einem 1000x526 Punkte großen Videobild aus, ein Halbbild hat also 263 Zeilen. Dies ist in etwa das größte Bild, das der BT878 aus einem PAL-Signal erzeugen kann.

	Scanline-Clipping			Bitmap-Clipping		
Bildgröße	**Einfach**	**Komplex**	**Extrem**	**Einfach**	**Komplex**	**Extrem**
1000x525	310	186	18	17	16	11
500x525	310	187	18	30	28	16
500x260	531	246	20	48	43	21
250x260	531	285	20	69	59	25
250x130	819	354	22	88	73	27
125x130	819	359	22	101	82	29
125x65	1111	362	22	111	88	30
32x10	1470	320	735	120	95	31

Tabelle 3: Meßergebnisse der Clipping-Algorithmen (Bilder/sec)

Gemessen wurde die Erzeugung des RISC-Codes für zwei Halbbilder. Das Ergebnis in wurde anschließend in (Voll-) Bilder pro Sekunde umgerechnet.

Die Interpretation der Meßergebnisse aus Tabelle 3 lassen einige sehr interessante Schlußfolgerungen zu.

Zuerst einmal ist auffällig, wie langsam das Bitmap-Verfahren im Gegensatz zum Scanline-Verfahren ist. Der Grund dafür könnte sein, daß die zahlreichen Bitmanipulationen des Bitmap-Verfahrens in Java nicht sehr effizient ausgeführt werden können, da bei jeder Operation auch sehr viele Sicherheitsabfragen, wie z.B. Referenz-Checks oder Array-Bound-Checks, durchgeführt werden. Das Scanline-Verfahren hat hier deutliche Vorteile, da es im günstigsten Fall mit einer Operation gleich mehrere Bildpunkte bearbeitet statt nur einen einzigen.

Neben dem direkten Vergleich der Zeiten fällt allerdings auch auf, daß die beiden Algorithmen unterschiedlich auf die Variationen reagieren.

So ist z.B. das Scanline-Verfahren sehr empfindlich gegenüber der Anzahl der verwendeten Clipping-Rechtecke, während diese Größe das Bitmap-Verfahren fast nicht berührt. Dagegen ist die Größe des Videobildes beim Bitmap-Verfahren sehr maßgebend. Hier ist das Scanline-Verfahren deutlich stabiler, es wird vor allem durch die Anzahl der Bildzeilen beeinflußt, aber nicht durch die Breite des Bildes. Auch bei der Fläche des überdeckten Bereiches zeigt das Scanline-Verfahren deutliche Vorteile, während beim Bitmap--Verfahren für jeden überdeckenden Bildpunkt ein Mehraufwand notwendig ist.

Das Scanline-Verfahren zum Clipping hat also in Java deutliche Vorteile gegenüber dem Bitmap-Verfahren. Es ist nicht nur deutlich schneller, darüber hinaus ist es deutlich robuster gegenüber Änderungen der Bildgröße. Der Mehraufwand für die erhöhte Komplexität des Verfahrens hat sich also gelohnt.

5.8 Integration durch `WinTV`

Nachdem nun sowohl die Schnittstelle zum Anwendungsprogramm, als auch die Abstraktionen von der Hardware durch geeignete Klassen besprochen wurde, fehlt für einen voll funktionsfähigen Treiber nur noch eine Instanz, die all diese Teile zusammenbringt. Die Aufgaben dieses Objektes sind also:

- Initialisierung der Hardware-Komponenten
- Implementierung und Initialisierung des Treiber-Portals
- Integration der einzelnen Hardware-Klassen
- Realisierung einer "benutzerfreundlichen" Semantik

Die Klasse, die dies für die Hauppauge-Karte WinTV-PCI-FM erledigt, ist, wen wundert es, in der Klasse `WinTV` realisiert. Um die Unterstützung für weitere TV-Karten auf der Basis des BT878 zu erleichtern, wird diese Klasse von der abstrakten Klasse `GenericBT878` abgeleitet, mit der die allgemeine Unterstützung von BT878-Karten realisiert wird. In der Klasse `WinTV` wird daher hauptsächlich die Belegung der Videoeingänge des BT878 und die Initialisierung der angeschlossenen I^2C-Geräte erledigt. In `GenericBT878` wird dagegen, bis auf die Implementierung der `InputSource`-Portale (vgl. Kap. 4.5), die gesamte restliche Schnittstelle zum Anwendungsprogramm realisiert.

Bis jetzt wurde nur sehr allgemein über die Schnittstelle zum Anwendungsprogramm gesprochen. So wurde in Kapitel 4 zwar die notwendige Funktionalität ausgearbeitet und in einigen Fällen (vgl. Kap. 4.5) schon die notwendigen Methoden und die zugrundeliegende

Semantik erklärt. Aber es ist immer noch offen, wie nun die Semantik der meisten Teile der Treiberschnittstelle aussehen soll.

So wäre es beispielsweise möglich, daß durch einen Portalaufruf, der neue Clipping-Informationen bekannt macht, sofort neuer RISC-Code erzeugt und auch aktiviert wird. Diese Semantik wäre sehr leicht zu implementieren, da in diesem Fall in der entsprechenden Portal-Methode nur wenige Zeilen Code genügen, um die Klassen `ScanlineClippingRasterizer`, `RISCJumpBlock` und `BT878Video` miteinander zu verknüpfen.

Andererseits kann die Erzeugung von RISC-Code, wie in Kapitel 5.7 dargelegt, sehr rechenintensiv sein. Daher sollte die unnötige Erzeugung von RISC-Code solange verzögert werden, bis alle gewünschten Informationen vorliegen, um dann den Code nur einmal erzeugen zu müssen. Eine zweite bedenkenswerte Konsequenz ist die Notwendigkeit, den RISC-Code sehr schnell aktivieren zu können. Dies ist dann wichtig, wenn die Anwendung die Einzelbilder in Dateien speichern oder über das Netzwerk übertragen will (vgl. Kap. 4.9). Da der Treiber das Anwendungsprogramm benachrichtigt, wenn ein Halbbild fertig geschrieben wurde, wird dieses wohl als erstes einen neuen Speicherplatz für das nächste Halbbild anfordern. Es wäre sehr fatal, wenn die Berechnung des RISC-Codes zu diesem Zeitpunkt soviel Zeit in Anspruch nähme, daß erstens die Anwendung nicht mehr genug Zeit hat, das alte Halbbild zu verarbeiten und zweitens die Berechnung erst dann fertig ist, wenn das nächste Halbbild schon gespeichert wird.

Es ist also notwendig, eine komplexere Semantik zu implementieren, die es erlaubt,

- alle Informationen über den Zielspeicher zu sammeln, bevor der RISC-Code berechnet wird,
- den neu berechneten RISC-Code nicht sofort, sondern erst zu einem späteren Zeitpunkt zu aktivieren und
- die Aktivierung des RISC-Codes sehr schnell erfolgen zu lassen.

Die Realisierung dieser Semantik erfolgt innhalb der Klasse `GenericBT878`, indem für jedes Halbbild neben den aktiven Informationen auch die bisher gesammelten Informationen vermerkt werden. Diese passiven Informationen können dann mittels der Methode `prepareField` in RISC-Code umgewandelt werden und mit `setFieldActive` wird schließlich der RISC-Code aktiviert. Zur Speicherung der angesammelten Informationen innerhalb von `GenericBT878` dient die Klasse `FieldDescr`. Durch die Zusammenfassung der Informationen in einer Klasse ist es sehr viel einfacher, die Informationen für jedes Halbbild getrennt und in zweifacher Ausführung, aktiv und passiv, zu verwalten bzw. den bisher passiven Zustand in den aktiven Zustand zu überführen.

Nachdem alle internen Klassen und Abläufe des Treibers beschrieben wurden, wird im nächsten Kapitel gezeigt, wie der Treiber in den Anwendungsprogrammen verwendet werden kann.

6 Beispiel-Anwendungen

Um die Tauglichkeit der in den vorherigen Kapiteln entworfenen Schnittstellen zu prüfen, wurden neben dem Treiber auch einige prototypische Anwendungen implementiert. Die Besonderheiten und einige interessante Implementierungsdetails werden in den folgenden Abschnitten beschrieben.

6.1 Anzeige auf dem Bildschirm

Als erster Anwendungsfall wurde natürlich die Anzeige des digitalisierten Videobildes auf dem Bildschirm implementiert. Als Grafikadapter stand eine Matrox-Karte mit einem MGA-G200 Grafikchip zur Verfügung (vgl. [Matrox98]).

Da noch keine Unterstützung für diesen Chip unter JX zur Verfügung stand, wurde in der Klasse `MGA200Impl` ein rudimentärer Treiber implementiert. Dieser Treiber initialisiert die Karte und schaltet sie in einen Grafikmodus mit einer Auflösung von 800x600 Punkten bei einer Farbtiefe von 16 Bit. Die Klasse implementiert auch das Portal-Interface `MGA200`. Dieses Interface stellt eine Methode zum Testen des Bildschirmspeichers zur Verfügung, wobei der Speicher mit verschiedenen Farbmustern gefüllt wird. Eine weitere Methode des Portals stellt den sichtbaren Grafikspeicher als ein `PackedFramebuffer` Objekt zur Verfügung.

Bei der Implementierung von `MGA200Impl` wurde ein ähnliches Schema verwendet, wie es bei den hardwarenahen Klassen des BT878-Treibers zum Einsatz kam: Die zur Bit- und Registermanipulation notwendigen Konstanten wurden in das Interface `MGA200Defines` ausgelagert, welches durch die Treiberklasse "implementiert" wird.

Da nun ein Grafikspeicher zur Verfügung steht, ist es ein Leichtes, eine einfache Applikation zur Anzeige eines digitalisierten Videobildes zu schreiben. Folgender Code z.B. zeigt das Videobild auf dem Bildschirm so an, daß die Zeilen der beiden Halbbilder miteinander verschränkt werden.

```
// get hardware parts
MGA200 mga = (MGA200)domainZero.lookup("MGA200");
CaptureBoard cBoard =
(CaptureBoard)domainZero.lookup("Framegrabber");

// select & activate video source
InputSource source = cBoard.getSVideo();
source.activate();
cBoard.setNorm(TVNorms.PAL);

// setup & activate framebuffer using both fields
cBoard.setFramebufferOdd(
     new InterlacedFramebuffer(framebuffer, 0));
cBoard.setFramebufferEven(
     new InterlacedFramebuffer(framebuffer, 1));
cBoard.setFieldsActive(true, true);
```

```
// start capturing
cBoard.captureOn();
```

Dieses einfache Beispiel kann nun z.B. so erweitert werden, daß Clipping-Informationen verwendet werden. Dazu ist es lediglich nötig, nach der Übergabe der Framebuffer-Objekte und vor deren Aktivierung `setClippingOdd` bzw. `setClippingEven`-Aufrufe mit einer Liste der Clipping-Rechtecke einzusetzen.

Weitere Möglichkeiten sind die Verwendung der globalen Bildmanipulationen aus Kapitel 4.7, die Skalierung des Videobildes aus Kapitel 4.8 oder die Abfrage einer eventuell vorhandenen Fernbedienung.

Die Möglichkeiten zur Erweiterung sind also zahlreich und vielfältig. Es bleibt daher festzustellen, daß dieser Anwendungsfall mit den vorhandenen Schnittstellen bequem handzuhaben ist.

6.2 Netzwerkübertragung

Ein zweiter Anwendungsfall ist die Übertragung der Videodaten über ein Netzwerk. Er steht auch exemplarisch für die Speicherung der Videodaten in eine Datei, da in beiden Fällen der Speicher der älteren Videobilder nicht durch neuere überschrieben werden soll.

Die Behandlung der Videodaten als Strom statt als Block erfordert es also, daß nach jedem geschriebenen Halbbild der RISC-Code so verändert wird, daß das nächste Halbbild an eine andere Stelle geschrieben wird (vgl. Kap. 4.9). Am einfachsten erreicht man dies, indem man den Framebuffer als "relozierbar" markiert und die Verschiebung dann durchführt, wenn der Treiber durch seinen Benachrichtigungsmechanismus anzeigt, daß das Halbbild fertig geschrieben wurde.

Andererseits werden wegen der geringen Bandbreite des Netzwerkes die Bilder sowieso nicht in Echtzeit übertragen werden können, da eine PAL-Vollbild-Übertragung etwa 11 MPixel/sec, bei 16 Bit Farbtiefe also etwa 22 MByte/sec, Bandbreite benötigt. Im Falle einer Videokonferenz ist es daher eher sinnvoll, mit drei Bildspeichern zu arbeiten.

- In den ersten Speicher schreibt der BT878 das aktuelle Videobild.
- Nachdem der erste Speicher einmal geschrieben wurde, wechselt der BT878 auf den zweiten Speicher und schreibt in diesen. Dies geschieht mittels einer Benachrichtigung vom Treiber.
- Den dritten Speicher liest das Übertragungsprogramm aus. Wurde der Speicher komplett übertragen, so tauscht er den bisherigen Sende-Speicher gegen den Puffer aus, der vom BT878 gerade nicht beschrieben wird.

Mit diesem Vorgehen ist sichergestellt, daß dem Übertragungsprogramm immer ein neues Videobild zur Verfügung steht. Werden die Daten vom Netzwerk zu langsam übertragen, so gehen zwar Bilder vorloren, da der BT878 immer zwischen zwei Bildspeichern wechselt, aber die Videobilder, die zur Übertragung verfügbar sind, sind aktuelle Videobilder. Auch ist mit diesem Verfahren ausgeschlossen, daß der BT878 und das

Netzwerkprogramm zugleich mit demselben Bildspeicher arbeiten, so daß ältere und neuere Videobilder nicht gemischt werden.

Würde man nur mit zwei Bildspeichern mit dem üblichen Doppelpuffer-Verfahren arbeiten, so müßte die Netzwerkkarte nach der Übertragung eines Bildes entweder einige Zeit warten, bis das nächste Halbbild komplett in den Speicher geschrieben wurde und der BT878 den nächsten Puffer beschreibt, oder es besteht die Gefahr, daß alte und neue Bilder gemischt werden.

Es ist auch möglich, mit nur einem Bildspeicher zu arbeiten. Der BT878 überschreibt die alten Videobilder immer wieder, das Übertragungsprogramm liest die Daten einfach möglichst schnell aus und überträgt sie. Der Vorteil ist natürlich, daß dieses Verfahren extrem simpel ist, andererseits muß man mit "interessanten" Effekten bei der Darstellung rechnen, wenn die Schreibposition des BT878 die Leseposition des Übertragungsprogrammes überholt.

Welches dieser Verfahren auch immer verwendet wird, so ist es bei allen notwendig, den Bildspeicher für die Übertragung nacheinander auszulesen. Im Allgemeinen wird aber für den Bildspeicher eine Organisation verwendet, bei der die einzelnen Bildzeilen nicht unmittelbar hintereinander im Bildspeicher liegen, wie es ja z.B. beim Speicher des Grafik-Adapters der Fall ist. Daher ist es sinnvoll, einen Iterator über ein Framebuffer-Objekt zu legen, der erstens von der aktuellen Leseposition und zweitens von der Speicherorganisation des Framebuffer-Objektes abstrahiert. Die Klasse `PictureIterator` ist die Implementierung dieses Konzeptes.

Nun können auf einfache Art und Weise die Videodaten über das Netzwerk übertragen werden, indem man mit Hilfe des `PictureIterator` ein Videobild in Pakete zerlegt, von denen jedes mit einem einzigen Netzwerk-Paket übertragen werden kann. Ist ein Bild vollständig übertragen, so erfolgt ein Umschalten des Bildspeichers und das nächste Bild wird übertragen.

Da je nach verwendetem Übertragungsprotokoll auch mit Paketverlusten zu rechnen ist, empfiehlt es sich, zusätzliche Informationen in jedes Paket zu integrieren. Im Beispielprogramm sind dies:

- X- und Y-Position des übertragenen Zeilenausschnittes
- Länge des Datenanteils in Byte
- Nummer des zugehörigen Halbbildes

Mit Hilfe dieser Informationen kann der Empfänger auch bei Paketverlusten oder Paketvertauschungen feststellen, wohin jedes Paket auf dem Bildschirm gehört und ob es ein altes Paket ist. Zusätzlich kann mit Hilfe der Bildnummer festgestellt werden, ob die Übertragung des Datenstromes das Netzwerk überlastet, da in diesem Fall die Nummer plötzliche Sprünge aufweist.

Der Empfänger muß Paketverluste im Falle einer Videokonferenz nicht besonders behandeln, da auf dem Bildschirm an der entsprechenden Stelle ja noch das alte Bild zu sehen ist. Üblicherweise unterscheiden sich die Einzelbilder nicht allzu stark voneinander, so daß der Fehler nicht so stark auffällt, wie es beispielsweise ein weißer Fleck tun würde.

Da leider zum Zeitpunkt der Implementierung der Netzwerk-Code von JX noch nicht stabil lief, konnte eine "richtige" Netzwerkübertragung noch nicht getestet werden. Daher wurde das Netzwerk durch ein entsprechendes Portal simuliert, sodaß die Übertragung der Videodaten von einer Domain in eine andere stattfand, wobei das eben beschriebene paketorientierte Verfahren zum Einsatz kam. Die Implementierung des Senders liegt in der Klasse `LocalSender` und die des Empfängers in `LocalReceiver` vor.

Nach diesem Ausflug in die Anwendungsprogrammierung wird nun besprochen, wie der Treiber an neue Geräte angepaßt werden kann.

7 Erweiterungen und Anpassungen an neue Hardware

In diesem Kapitel wird erklärt, wie das vorhandene Klassenkonzept erweitert werden kann. Dies bezieht sich in erster Linie auf die Unterstützung von Karten anderer Hersteller. Aber auch neue Konzepte wie die Unterstützung einer Fernbedienung werden erläutert.

7.1 Unterstützung neuer Framegrabber-Karten

Karten anderer Hersteller, die ebenfalls einen Chip der BT878-Familie verwenden, sind in der Regel relativ einfach zu unterstützen, da sich die Videocapture-Teile nur recht wenig voneinander unterscheiden. Zudem wurde, wie schon in Kapitel 5.8 angemerkt, bereits beim Entwurf des Treibers auf eine leichte Anpaßbarkeit Wert gelegt. So findet sich der Code, der unabhängig von der jeweiligen Karte arbeitet, in der Klasse `GenericBT878`, von der sich dann die verschiedenen Karten-Klassen ableiten lassen. In der abgeleiteten Klasse wird im wesentlichen Folgendes festgelegt:

- Identifikation der Karte bei der Initialisierung
- Initialisierung der angeschlossenen Geräte (z.B. TV-Tuner)
- Beschaltung der Videoeingänge
- Festlegung der auf der Karte verwendeten Quarze

Die Identifikation der Karte erfolgt in der Regel über die im PCI-Header vermerkten "VendorID" und "DeviceID"-Einträge. Leider läßt sich anhand dieser Merkmale nur feststellen, daß eine Karte einen BT878-Chip enthält. Zum Glück sind in den PCI-Spezifikationen ab Version 2.1 weitere Einträge spezifiziert (vgl. [Shan00], S.354ff), durch welche die Identifikation des jeweiligen Kartenherstellers möglich wird. Dies sind die Register "SubsystemVendor" und "SubsystemID". Nur einige sehr alte Karten mit dem BT848 verwenden diese Register nicht, so daß bei diesen andere Identifikationsmerkmale, wie z.B. ein eventuell vorhandenes EEPROM, verwendet werden müssen.

Die Initialisierung der Karte erfolgt im Konstruktor der jeweiligen Treiberklasse. Hier ist wohl der größte Teil der notwendigen Anpassungen vorzunehmen. Neben der Initialisierung des I^2C-Busses und der daran angeschlossenen Geräte ist hier auch die Initialisierung und Zuordnung der in Kapitel 4.5 eingeführten `InputSource` Portale zu den jeweiligen Video-Eingängen des BT878 notwendig.

Damit die `InputSource`-Portale später auch vom Anwendungsprogramm erfragt werden können, muß die abgeleitete Klasse auch die Methoden `getInputSources`, `getCurrentInputSource`, `getTuner`, `getCinch`, `getSVideo` und `getColorBars` implementieren. Wird eine entsprechende Videoquelle von der Karte nicht unterstützt, sollte eine `NotImpl`-Exception geworfen werden. Der Aufwand, diese Methoden zu implementieren, hält sich übrigens in Grenzen, da die meisten dieser Methoden nur Dreizeiler sind.

Als letzter Punkt ist in der abgeleiteten Klasse die Methode `getXT0Frequ` zu implementieren, welche die Frequenz des angeschlossenen Quarzes in Hertz zurückgibt.

Diese Frequenz wird von `GenericBT878` zur Einstellung der Videonorm benötigt, da aus ihr die Frequenz einer "Phase Locked Loop" (PLL) ableitet wird, die ihrerseits der Frequenz des verwendeten Farbträgers entspricht (vgl. Kap. 2 und [BT878], S.73ff). Normalerweise ist ein Quarz eingebaut, der für die NTSC-Norm geeignet ist. Ein Gegenbeispiel stellt die Karte TerraTV+ von Terratec dar, denn sie verwendet einen PAL-Quarz.

Die Unterstützung neuer Karten durch das eben beschriebene Vorgehen wurde anhand von zwei Karten getestet.

Zuerst wurde eine ältere MiroTV-Karte mit einem etwas älteren BT848A unterstützt. Der auf ihr eingesetzte Tuner ist nicht kompatibel zum Tuner der WinTV-PCI Karte und Informationen über ihn konnten nicht beschafft werden, weshalb bei dieser Karte nur die Cinch- und SVideo-Eingänge unterstützt werden. Ansonsten wird der volle Funktionsumfang der WinTV-Karte unterstützt. Die Implementierung liegt in der Klasse `MiroTV` vor.

Eine zweite Karte, die nachträglich unterstützt werden sollte, ist die schon erwähnte TerraTV+ von Terratec. Ihr Tuner wird unterstützt, da er kompatibel mit dem auf der WinTV-Karte ist. Als Besonderheit hat diese Karte, wie schon erwähnt, statt eines NTSC-Quarzes einen PAL-Quarz als Taktgeber integriert. Auch bei dieser Karte wird der volle Funktionsumfang des Treibers unterstützt. Die Implementierung dieses Treibers liegt in der Klasse `FlyTV` vor, da es sich laut den Informationen aus dem PCI-Header um eine FlyVideo98-Karte handelt.

7.2 Einbindung der Fernbedienung

Ein weiteres Beispiel für die leichte Erweiterbarkeit des Treibers soll an der Unterstützung der Fernbedienung, die der WinTV-Karte beiliegt, demonstriert werden.

Das Signal der Fernbedienung wird von einem auf der Karte integrierten Microcontroller PIC16C54C mittels eines externen Sensors empfangen und dekodiert. Das Ergebnis kann über den I^2C-Bus mittels einer Lese-Operation abgefragt werden. Dabei erhält man ein drei Byte großes Paket, in welchem die gedrückte Taste sowie die aktuelle "Phase" vermerkt ist. Die Phase ändert sich dabei bei jedem neuen Drücken einer Taste. Mit der Phaseninformation kann z.B. unterschieden werden, ob eine Taste länger gedrückt wird (Phase bleibt bei zwei Abfragen gleich), oder ob die Taste zweimal hintereinander gedrückt wurde (Phase ändert sich).

Da der Empfangschip am I^2C-Bus hängt, können die allgemeinen Schreib-und Leseoperationen verwendet werden, die in der Klasse `BT878I2CBus` gekapselt und über das Interface `I2CBus` abstrahiert werden. Die Abfrage und Dekodierung der Pakete erfolgt zweckmäßigerweise innerhalb einer Klasse, in diesem Falle in `IRReceiverImpl`. Sie stellt auch die Methoden für das Portal `IRReceiver` bereit, über welches andere Domains die Fernbedienung abfragen können. Das Portal muß nur noch von der Treiberklasse `WinTV` initialisiert werden.

Aber wie gelangt die Portal-Referenz zum Anwendungsprogramm? Es wäre möglich, das allen Treiber-Klassen gemeinsame Interface `CaptureBoard` mit einer entsprechenden Methode zu erweitern. Dies hat den Nachteil, daß das Anwendungsprogramm wissen muß, daß der jeweilige Treiber eine Fernbedienung unterstützt, um dann die Portal-Referenz des Gerätetreibers durch sog. "Casten" zu erweitern. Dies ist nicht sehr elegant und führt bei weiteren derartigen Erweiterungen sehr bald zu einem Dickicht von leicht erweiterten Framegrabber-Interfaces. Denn leider kann z.Z. ein Portal nur ein Interface anbieten, so daß schon bei einer zweiten Erweiterung, beispielsweise einem Infrarotsender zur Steuerung eines Videorecorders, schon vier Interfaces notwendig wären:

- CaptureBoard
- CaptureBoardWithIRReceiver
- CaptureBoardWithIRTransmitter
- CaptureBoardWithIRReceiverAndTransmitter

Um dies zu vermeiden bietet sich als Ausweg nur an, das Portal der Fernbedienung separat über den Namensdienst verfügbar zu machen.

Nachdem die Anwendung das Portal der Fernbedienung beim Namensdienst erfragt hat, kann sie nun mit der Methode `poll()` abfragen, ob eine Taste auf der Fernbedienung gedrückt wurde. Wird dabei anhand der Phaseninformation erkannt, daß seit der letzten Abfrage immer noch die alte Taste gedrückt ist, wird die spezielle Taste `lastKeyStillPressed` zurückgegeben. Eine vergeßliche Anwendung kann diese Taste mit der Methode `lastKey` erfragen.

Die übrigen Tasten werden im Interface `RCKeys` als Konstanten definiert, so daß die Anwendung einen leichten Zugriff auf den Code der Tasten und auch auf deren Beschriftung hat.

7.3 Andere Erweiterungen

Nachdem die Maßnahmen zur Unterstützung neuer Geräte besprochen wurde, erfolgt jetzt eine kleine Zusammenfassung, wie bei anders gearteten Erweiterungen vorzugehen und was dabei zu beachten ist.

Da der Treiber ja die Systemsicherheit gewährleisten soll, muß man bei Erweiterungen daran denken, ob und wie die gewünschte Funktionalität auf die drei Bereiche "Anwendungsprogramm", "Treiberschnittstelle" und "Treiber" zu verteilen ist.

Beispielsweise wird bisher vom Treiber ja nur der sog. "Packed Mode" des BT878 unterstützt, bei dem die drei Farbinformationen eines Bildpunktes unmittelbar hintereinander im Speicher liegen. Soll nun der "Planar Mode" unterstützt werden, bei dem die drei Farbinformationen in drei unterschiedliche Speicherbereiche geschrieben werden, muß eine Änderung an allen drei Bereichen vorgenommen werden:

- Die **Hilfsklassen**, die sich ja im Bereich des Anwenderprogrammes befinden, müssen um eine Klassenstruktur analog zu `PackedFramebuffer` erweitert werden.
- Die **Treiberschnittstelle** muß bei einigen Methoden (z.B. `setFramebufferOdd/Even`) die neuen Klassen unterstützen. Dazu ist sowohl die Schnittstellendefinition `CaptureBoard` als auch die Implementierung der Methoden in `GenericBT877` anzupassen.

- Und schließlich muß im **Treiber** selbst die Erzeugung des RISC-Codes entsprechend angepaßt werden. Dies wird z.B. bei den Clipping-Algorithmen durch die schon vorhandene Abstraktion `ClippingRasterizerOutput` stark erleichtert, so daß der Clipping-Algorithmus selbst nicht angepaßt werden muß (vgl.Kap. 5.7).

Dieses Beispiel ist insofern sehr extrem, da es alle Bereiche des Treibers berührt.

In vielen Fällen ist aber ein solch großer Aufwand gar nicht notwendig. Für die meisten Anwendungen ist es wahrscheinlich nur wichtig, die Organisation des Bildspeichers an ihre Bedürfnisse anzupassen. Im Falle von regelmäßig angeordneten Bildzeilen ist das schon mit dem vorhandenen Konzept des `PackedFramebuffer` erreichbar, wie es in Kapitel 4.9 am Beispiel von `MemoryFramebuffer` gezeigt wird, so daß eine Änderung des Treibers oder seiner Schnittstelle nicht notwendig ist.

8 Zusammenfassung und Ausblick

Nachdem in Kapitel 2 die Grundlagen für analoge Videoübertragung erklärt wurden und in Kapitel 3 die Funktionsweise des BT878 Videocapture-Chip beschrieben wurde, konnte ab Kapitel 4 der Entwurf und die Implementierung des Treibers erfolgen. Dazu wurden zuerst die Entwurfsziele festgelegt, auf die im Folgenden besonders geachtet werden sollte. Dies waren im Einzelnen:

- Universalität
  Die Anwendungsschnittstelle sollte auch für andere Videocapture-Produkte verwendbar sein.
- Flexibilität
  Alle Funktionen des BT878 sollten unterstützt werden. Dies steht natürlich in Konflikt mit dem ersten Ziel.
- Modularität
  Die einzelnen Komponenten des Treibers sollten zur leichteren Unterstützung neuer Hardware allgemein gehalten und dadurch erweiterbar und wiederverwendbar sein.
- Sicherheit
  Durch Verwendung des Treibers sollte im System kein Sicherheitsloch entstehen, das böswillige Anwendungen ausnutzen könnten.

Diese Ziele widersprechen sich teilweise. So wirkt sich z.B. die Forderung nach Sicherheit sehr stark auf die Flexibilität aus, weil es dadurch nicht möglich ist, dem Benutzer die Erzeugung des RISC-Codes zu überlassen, da damit jeglicher Speicherschutz umgangen werden könnte.

Dennoch wurde in Kapitel 4 eine Schnittstelle für die Anwendungsprogramme entworfen, die den Entwurfszielen in hohem Maße gerecht wird. Dies wurde auch dadurch erreicht, daß die Funktionalität, die verschiedene typische Anwendungsfälle erfordern, im Vorhinein abgeschätzt wurde.

Nachdem die öffentliche Schnittstelle und ihre Arbeitsweise definiert wurden, erfolgte in Kapitel 5 der Entwurf und die Implementierung der inneren Strukturen des Treibers. Hier wurde auch dargelegt, wie man im JX-System vorteilhaft Gerätetreiber implementieren kann.

Anschließend wurden in Kapitel 6 einige der in Kapitel 4.9 angedachten Anwendungsfälle prototypisch implementiert, um die Funktionalität des Treibers und seiner Schnittstelle zu prüfen.

Daran schließt sich Kapitel 7 an, in dem erklärt wird, wie das Konzept bzw. die Implementierung erweitert werden kann. Dies beinhaltet die Unterstützung neuer Videocapture-Karten und neuer Zusatzgeräte wie z.B. eine angeschlossene Fernbedienung, aber auch Gedanken zur konzeptionellen Erweiterung des Klassengerüstes.

Diese Arbeit zeigt also, daß es durchaus möglich ist, das JX-System zusammen mit geeigneter Hardware für Multimedia-Anwendungen zu verwenden. Die Implementierung der erforderlichen Treiber muß dabei nicht in den Kern, der DomainZero, integriert werden, sondern kann wie ein Anwendungsprogramm in Java implementiert werden.

Wie kann nun diese Arbeit weitergeführt oder als Grundlage weiterer Arbeiten verwendet werden? Natürlich können zuallererst die in Kapitel 7 besprochenen Erweiterungen durchgeführt werden. Ein weiterer wichtiger Punkt dabei wäre die Unterstützung der Audio-Kanäle, die zusammen mit den Videosignalen übertragen werden. Daneben wäre es sinnvoll, einen für das JX bzw. Java geeigneten Video-Codec zu implementieren, der das Volumen der Videodaten drastisch reduziert, so daß auch größere Bildauflösungen mit weniger Bildverlusten über ein Netzwerk übertragen werden kann. Damit könnte ein einfach zu bedienendes Gerät hergestellt werden, mit dem Videokonferenzen oder generell Videoübertragungen ermöglich werden. Eine weitere Möglichkeit wäre die Implementierung der Standard-AWT-Klassen von Java, damit die Videobilder nicht nur "nackt" auf dem Bildschirm erscheinen, sondern zusammen mit einer grafischen Bedienoberfläche präsentiert werden.

9 Literaturverzeichnis

[Atmel98] "*Atmel 2-Wire Serial EEPROM AT24C01A/02/0/08/16*", Atmel, 1998

[BT878] "*BT878/879 - Single-Chip Video and Broadcast Audio Capture for the PCI Bus*", Dokument-Nr. D879DSA, Rockwell Semiconductor Systems, 1998

[Fol91] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes: "*Computer Graphics - Principles and Practice - Second Edition*" Addison-Wesley, 1991

[Gam97] E. Gamma, R. Helm, R. Johnson, J. Vlissides: "*Design Patterns - Elements of reusable Object-Oriented Software*", 10. Auflage, Addison-Wesley, 1997

[Golm00] M. Golm, J. Kleinöder: "*Eine adaptierbare Java-Betriebssystemarchitektur" in "GI-Herbsttagung: Mobile Computing*", 2000

[Golm01] M. Golm, J. Kleinöder, F. Bellosa: "*Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System*", Tech-Report TR-I4-0201, Lehrstuhl für Betriebssysteme an der Universität Erlangen, 2001

[Intel97] "*82371AB PCI-To-ISA- / IDE XCelerator (PIIX4)",* Dokument-Nr. 290562-001, Intel, 1997

[Intel98] "*Intel 440BX AGPset: 82443BX Host Bridge/Controller Datasheet*", Dokument-Nr. 290633-001, Intel, 1998

[JLS00] J. Gosling, B. Joy, G. Steele, G. Bracha: "*The Java Language Specification - Second Edition*", Sun Microsystems, 2000

[JXOS] "*JX flexible Operating System Architecture*", http://www4.informatikuni-erlangen.de/Projects/JX/

[Linux] Linux Kern, ftp://ftp.kernel.org/pub/linux/kernel/v2.2/linux-2.2.18.tar.gz

[Matrox98] "*Matrox MGA-G200 Specification*", Dokument-Nr. 10582-MS-0301, Matrox Graphics Inc., 1998

[Micr99] "*MSP 3405D, MSP 3415D Multistandard Sound Processors*", Dokument-Nr. 6251-475-2PD, Micronas, 1999

[Phil97] "*FM126 - Desktop video & FM radio module - System B/G*", Philips, 1997

[Phil00] "*The I2C-Bus Specification - Version 2.1*", Philips Semiconductors , Jan. 2000

[Shan00] T. Shanley, D. Anderson: "*PCI System Architecture*", 4. Auflage, Mindshare Inc., 2000

[Wen88] Broder Wendland: "*Fernsehtechnik - Band 1: Grundlagen*", Hüthig, Heidelberg, 1988

[Wen91] Broder Wendland, Hartmut Schröder: "*Fernsehtechnik - Band 2: Systeme und Komponenten zur Farbbildübertragung*", Hüthig, Heidelberg, 1991