



Hochschule für angewandte Wissenschaften München
Fakultät für Elektrotechnik und Informationstechnik
Bachelorstudiengang Elektrotechnik und Informationstechnik

Bachelorarbeit

Modulare Ansteuerung eines FPGAs über Software

abgegeben von Maren Konrad

Bearbeitungsbeginn:	12.08.2019
Abgabetermin:	12.02.2020
lfd. Nr. gemäß Belegschein:	1883

Hochschule für angewandte Wissenschaften München
Fakultät für Elektrotechnik und Informationstechnik
Bachelorstudiengang Elektrotechnik und Informationstechnik

Bachelorarbeit

Modulare Ansteuerung eines bildverarbeitenden FPGAs über generische Kernelmodule

Modular control of an image processing FPGA via generic kernelmoduls

abgegeben von Maren Konrad

Bearbeitungsbeginn:	12.08.2019
Abgabetermin:	12.02.2020
lfd. Nr. gemäß Belegschein:	1883
Betreuer (Hochschule München):	Prof. Dr. Gerhard Schillhuber
Betreuer (Extern):	Anton Hattendorf

Erklärungen des Bearbeiters:

Name: Konrad

Vorname: Maren

1) Ich erkläre hiermit, dass ich die vorliegende Bachelorarbeit selbständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe.

Sämtliche benutzte Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate sind als solche gekennzeichnet.

München, den 3. Dezember 2019

Unterschrift

2) Der Veröffentlichung der Bachelorarbeit stimme ich hiermit **NICHT** zu.

München, den 3. Dezember 2019

Unterschrift

Kurzfassung

In dem vorliegenden Bericht geht es um die Verbesserung einer Kamerasoftware im Zusammenspiel mit der Hardware. Es wird hierzu eine Abstraktionsebene vorgestellt, welche die Zugriffe von der Software auf den FPGA übersichtlicher und einfacher gestalten soll. Der Zugriff auf die Hardware findet aktuell direkt über die Adresse statt und soll abstrahiert werden, sodass die Software lediglich den Namen benötigt um auf das richtige Modul zuzugreifen. Zum Erleichtern des Verständins wird ein kurzer Überblick über die aktuelle Implementierung und eine kurze Einführung zur Bildkette gegeben. Außerdem wird auf das Betriebssystem Linux und benötigte Komponenten eingegangen. Im Anschluss wird die Implementierung der Abstraktionsebene näher erklärt. Die Wartbarkeit und die Erweiterung der Software soll durch die Abstraktionsebene in Zukunft erleichtert werden.

Dieser Bericht ist für Leser aus dem Bereich der Elektrotechnik geschrieben.

Abstract

This report handles the improvement of an camera software in interaction with the hardware. For this, an abstraction level will be introduce, which should simplify and clearer the accesses from the software to the FPGA.

This report targets readers having an electric engineering background.

Inhaltsverzeichnis

I	Abkürzungsverzeichnis	V
II	Glossar	VI
1	Einleitung	1
1.1	Arnold & Richter Cine Technik GmbH & Co. Betriebs KG . .	1
1.2	Kontext	1
1.2.1	Arbeitsumgebung	1
1.2.2	Bildkette	2
1.2.3	Implementierung	3
1.2.4	Problematiken	3
1.3	Konzept	4
2	Grundlagen Linux	6
2.1	Linux - Kernel und Userspace	6
2.2	IO Control	7
2.3	Datenaustausch zwischen Kernel und Userspace	7
2.4	Plattformtreiber	8
2.4.1	Major- und Minornummern	9
2.5	Multifunction Device	9
3	FPGA Resource Abstraction	12
3.1	Generischer Plattformtreiber	12
3.2	Konzeptionierung der IO Controls	14
3.3	Implementierung im Kernel	16
3.3.1	Anlegen der Geräte	16

3.3.2	Öffnen und Schließen der Geräte	17
3.4	Implementierung im Userspace	18
3.4.1	FRA Bibliothek	19
3.4.2	FRA Middleware	20
3.4.3	FRA Kernelbackend	23
3.5	Einbindung ins Geometrie Framework	23
4	Frameworktest	25
4.1	Plattformunabhängige Tests	25
4.2	Tools für die Entwicklungsumgebung	28
5	Fazit und Ausblick	30
5.1	Zusammenfassung der Ergebnisse	30
5.2	Ausblick	31
Anhang		32
A.1	Literaturverzeichnis	32
A.2	Abbildungsverzeichnis	33
A.3	Tabellenverzeichnis	33
A.4	Liste der Codeausschnitte	33

Abkürzungsverzeichnis

ARRI Arnold & Richter Cine Technik GmbH & Co. Betriebs KG

FPGA Field Programmable Gate Array

FRA FPGA Resource Abstraction

Geo Framework Geometrie Framework

GUI grafische Benutzeroberfläche

IOCTL IO Control

MFD Multifunction Device

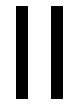
REC Aufnahme

PID Prozesskennung

SMPTE Society of Motion Picture and Television Engineers

SDI Serial Digital Interface gemäß Society of Motion Picture and Television Engineers (SMPTE) 292M [10]

Xbar Crossbar



Glossar

Dateideskriptor example

Frame example

Framework example

Handle example

Middleware example

Unittest example

1

Einleitung

1.1 Arnold & Richter Cine Technik GmbH & Co. Betriebs KG

Die Bachelorarbeit fand bei der Firma Arnold & Richter Cine Technik GmbH & Co. Betriebs KG (ARRI) statt. Gegründet wurde die Firma 1917 und auch noch über 100 Jahren Firmengeschichte liegt der Hauptsitz von ARRI noch immer in München. Weltweit sind mittlerweile um die 1500 Mitarbeiter angestellt und die Firma ist einer der führenden Hersteller und Lieferanten in der Film- und Fernsehindustrie.

Die ARRI Gruppe teilt sich in die folgenden fünf Geschäftsbereiche ein: Kamerasysteme, Licht, Postproduktion, Verleihservice und Operationskamerasysteme für die Medizin. [1]

Das Thema dieser Bachelorarbeit wurde im Bereich Kamerasysteme in der Forschungs- und Entwicklungsabteilung erarbeitet.

1.2 Kontext

Um einen Überblick über das Umfeld der Arbeit zu bekommen, werden zunächst ein paar Grundlagen und auch die Funktionsweise einer Kamera betrachtet.

1.2.1 Arbeitsumgebung

Als Erstes soll auf die Zielplattform eingegangen werden. Die Wahl der Kamera ist auf eine ARRI AMIRA gefallen, da die Entwicklungsumgebung durch vorherige Arbeit bekannt ist. Zusätzlich ist sie nicht die aktuellste

Kamera der Firma und somit ohne Probleme verfügbar. Allerdings ist die Hard- und Softwarearchitektur identisch zu den aktuellen Kameras. Die AMIRA ist eine vielseitige Kamera, die für eine Einmannbedienung ausgelegt ist und mit einem Audioboard ausgestattet. Aus diesem Grund wird sie bei Dokumentationsfilmen und der elektronischen Berichterstattung gerne verwendet. Zum Beispiel wird die ARRI AMIRA bei Sportveranstaltungen der National Football League (NFL) in Amerika eingesetzt.[2]



Abbildung 1.1: ARRI AMIRA
siehe [3]

Für Spielfilm- und Serienproduktionen wird auch manchmal die ARRI AMIRA eingesetzt, wodurch das breite Einsatzspektrum der Kamera noch deutlicher wird. Als Beispiele sind hier der bayrische Eberhofer Krimi „Sauerkrautkoma“ [7], die Netflixserie „The Ivory Game“ [12] oder auch das Fernsehmagazin „The Grand Tour“ [11] zu nennen.

Die Kamera ist ein wichtiger Bestandteil in der Entwicklungsarbeit, da das regelmäßige Testen von Änderungen unerlässlich ist, um die Funktionalität beizubehalten und grobe Fehler rechtzeitig zu erkennen. Die aktuelle Software läuft mit Linux auf der Kernelversion 5.3.7.

1.2.2 Bildkette

In jeder Kamera wird das Eingangsbild von einem Sensor aufgenommen. Durch die Verarbeitungskette werden die Bilder vom Eingang bis zum Ausgang geleitet. In dieser Arbeit wird eine schematische Bildkette (siehe Abbildung) zur Veranschaulichung weiter detailliert. Von der Quelle bis zur Senke läuft das Bild durch verschiedene Module, die für die Anpassung des Bilds sorgen. Die Ausgänge sind in diesem Fall die Aufnahme (REC) und das Serial Digital Interface (SDI).

Direkt nach dem Sensor geht das Bild durch eine Crossbar (Xbar), hier wird das identische Bild in zwei Bildpfaden weitergeführt. Für die Aufnahme wird das Bild im Crop zugeschnitten, damit wird nur ein bestimmter Bildausschnitt aufgezeichnet. In dem anderen Bildpfad wird, mithilfe des Scalers, das Bild kleiner skaliert. Nachdem eine grafische Benutzeroberflä-

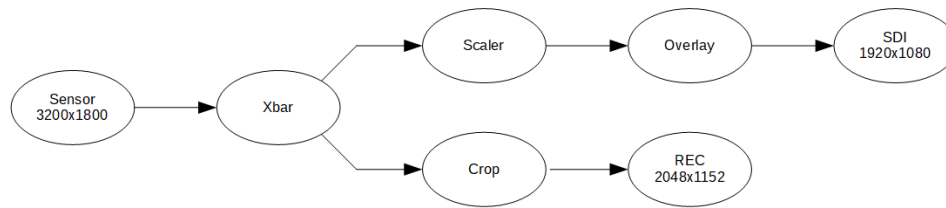


Abbildung 1.2: Schematische Bildkette

che (GUI) hinzugefügt worden ist, wird das Bild am SDI Monitor ausgegeben. Hier ist, durch die Skalierung, das komplette Sensorbild einschließlich der eingefügten GUI zu sehen.

In dieser Arbeit wird nur das Field Programmable Gate Array (FPGA)-Modul Xbar softwareseitig implementiert, da weitere Module sonst den Umfang der Arbeit überschreitet.

1.2.3 Implementierung

Bevor auf das erarbeitete Konzept eingegangen wird, soll kurz auf die Funktionsweise der Kamera und die aktuelle Implementierung eingegangen werden.

Die bildverarbeitende Hauptfunktionalität liegt im FPGA. Hier sind die Module entsprechend der Bildkette angeordnet und verbunden. Durch die Software werden bei den FPGA Modulen entsprechende Einstellungen vorgenommen.

Damit die Einstellungen auch zu dem Sensorbild passen, werden alle Module des FPGAs in dem Geometrie Framework (Geo Framework) abgebildet. Das objektorientierte Framework führt hauptsächlich Berechnungen der Bildgrößen und Offsets durch. Nach der Änderung einer Größe in der Quelle oder Senke werden alle Module in der abgebildeten Frameworkbildkette geupdatet und entsprechend der voreingestellten Parameter werden die Größen neu berechnet. Am Ende des Updates werden verschiedenen Funktionen aufgerufen, welche die Register im FPGA entsprechend der Einstellungen gesetzt.

1.2.4 Problematiken

In der aktuellen Implementierung liegen verschiedene Probleme vor, die durch ein neues Framework eliminiert werden soll.

Zum Einen muss bei den Zugriff auf ein Modul immer der FPGA gesperrt werden. Dadurch kann es passieren, dass es einen oder zwei Frames dauert, bis alle Einstellungen in der Bildkette aktuell sind. Bei einem Livebild

der Kamera ist dies besonders störend, da man die aktuelle Änderung erst später sieht. Da man beim Aufnehmen keine Änderungen am Sensorbild vornehmen kann fällt das Problem hier nicht ins Gewicht.

Des weiteren müssen die Adressen für die FPGA Module händisch eingetragen werden. Die Fehleranfälligkeit steigt dadurch weiter an, da es passieren kann, dass die Software, Einstellungen an eine Adresse schreibt, hinter der kein Modul liegt. In schlechtesten Fall werden die Register eines anderen Moduls beschrieben und es kommt zum Fehlerfall in der Bildkette.

Auch die Länge der Register wird in der aktuellen Implementierung nicht weiter berücksichtigt. So kann es passieren, dass über ein Register hinweg geschrieben wird. Auch dann kommt es zum Fehlerfall in der Bildkette, da andere Einstellungen überschrieben werden und so verloren gehen.

Im normalen Betrieb der Kamera können diese Fehlerfälle auftreten und somit in Filmproduktionen für Ausfällen sorgen. Dies ist mit der wichtigste Grund um die Software entsprechend anzupassen, sodass die Problematiken verschwinden oder zumindest minimiert werden.

1.3 Konzept

Die Probleme in der aktuellen Implementierung (siehe Kapitel 1.2.4) sollen durch ein neues Framework behoben werden, zusätzlich sollen die Zugriffe der Software auf den FPGA übersichtlicher und wartbarer gestaltet werden.

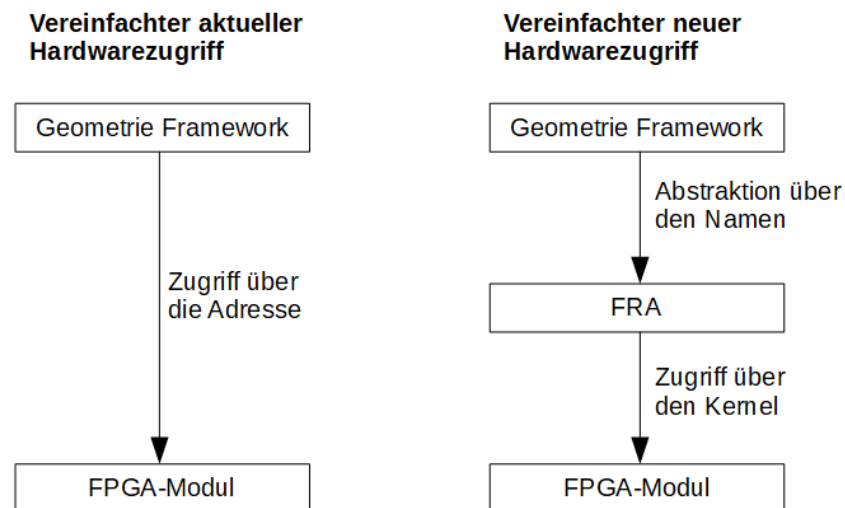


Abbildung 1.3: Vereinfachte Darstellung des Zugriffs auf den FPGA in der aktuellen und der neuen Implementierung

Aktuell benötigen die Zugriffe auf den FPGA jedes Mal die Adressen der Module. So ist es nicht nachvollziehbar welcher Prozess in der Hardware Änderungen durchführt. Dadurch wird die Fehlersuche in diesen Teilen der Software extrem kompliziert und aufwendig. Durch eine weitere Abstraktionsebene soll eine modulare Ansteuerung möglich werden und so die direkten Hardwarezugriffe über die Adressen aus dem Hauptcode eliminiert werden. In der FPGA Resource Abstraction (FRA) werden die Module im Kernel als Gerät angelegt und über Dateideskriptoren greift die Software auf den FPGA zu.

Die Geräte werden mit der Adresse, der Größe, dem Namen und Typ des dahinterliegenden Modul beim Initialisieren des Bildpfads angelegt. Dieser Teil soll generisch generiert werden, aber aufgrund von verschiedenen Abhängigkeiten überschreitet es den Umfang der Arbeit.

Über den Namen im vorhandenen Geo Framework werden die Dateideskriptoren in der Software geöffnet und in einem Handle gespeichert. Damit wird im Hauptteil des Codes ein Zugriff ohne Adresse gewährleistet. Jeder laufende Prozess muss seinen eigenen Dateideskriptor öffnen und verwaltet somit sein eigenes Handle. Pro Gerät kann lediglich eine begrenzte Anzahl von Deskriptoren geöffnet werden, geregelt wird dies durch den Kernel. Bei jedem Schreib- oder Lesezugriff auf ein Register wird überprüft, ob dieser innerhalb der angegebenen Modulgröße liegt. Damit ist es nicht mehr möglich über ein Register hinweg zuschreiben und somit falsche Module zu beschreiben.

Durch die neue Abstraktionsebene soll die Wartbarkeit sowie die Erweiterung der Kamerasoftware in Zukunft ohne tiefere Kenntnisse vom FPGA durch unterschiedliche Entwickler möglich werden.

2

Grundlagen Linux

Zu Beginn sollen einige Grundlagen näher erläutert werden, die zum Erstellen der Arbeit essenziell waren.

2.1 Linux - Kernel und Userspace

Da das Zielsystem auf Linux läuft, soll zunächst dieses Betriebssystem betrachtet werden. Im Herbst 1991 wurde die erste Version des Systems von Linus Torvalds veröffentlicht und der Gründer kümmert sich, mit Unterstützung, weiterhin um die Entwicklung des frei verfügbaren Betriebssystems.

Der Name Linux bezeichnet dabei eigentlich nur den Kern des Systems, auch Kernel genannt. Zusätzlich benötigt man noch System- und Anwendersoftware. Oft wird dieser Teil als Userspace zusammengefasst. [8, S. 46]

Der Kernel hat verschiedene Aufgaben. Unter anderem ist er für die Prozess- und die Speicherverwaltung sowie das Gerätemanagement zuständig. [9, S. 234]

Im Normalfall hat der Nutzer aus dem Userspace keinen direkten Zugriff auf die Kernelfunktionen und der Hardware. Nur über Systemaufrufe, auch Syscalls, hat ein Programm im Userspace die Möglichkeit Änderungen an der Hardware zu kommunizieren beziehungsweise bestimmte Funktionen im Kernel zu nutzen. [8, S. 124] Die Brücke zwischen der Hardware und dem Benutzer stellt somit der Kernel da.

2.2 IO Control

Um ein zuverlässig arbeitendes Betriebssystem zu haben, muss der Speicherbereich von Kernel und der vom Userspace getrennt sein. [9, S. 233]

Damit entsteht die Notwendigkeit zwischen Userspace und Kernel aktiv Daten auszutauschen. Die Anwendungen im Userspace können über das sogenannte Systemcall Interface auf die Funktionen im Kernel zugreifen. In einer Struktur vom Typ *file_operations* wird die Schnittstelle zu einem Treiber vorgegeben. In dieser Struktur werden treiberabhängige Funktionszeiger gespeichert. [9, S. 249]

Im folgenden soll lediglich der Zeiger auf das IO Control (IOCTL) betrachtet werden, da dieser im weiteren Teil der Arbeit eine wichtige Rolle spielt. Durch die IOCTL Methode wird dem Programmierer ein flexibles Werkzeug zur Verfügung gestellt.

```
1 int (*ioctl) (struct inode *node, struct file *instanz, unsigned
    int cmd, unsigned long arg);
```

Codeausschnitt 2.1: Funktionsdeklaration des IOCTL in der file_operations Struktur [9, S. 249f.]

Über die *node* wird der Dateideskriptor und über *instanz* ein Zeiger auf die Treiberinstanz an den Funktionszeiger übergeben. Das Kommando wird durch eine Nummer widerspiegelt und ist in der Funktionsdeklaration als *cmd* zu finden. Das optionale Argument *arg* wird meistens als Zeiger auf eine Dateistruktur, welche kopiert werden soll angegeben. [9, S. 249f.]

Über den ersten Übergabeparameter wird ein Zeiger auf die Treiberinstanz an den Funktionszeiger übergeben. Das auszuführende Kommando wird durch eine Nummer wieder gespiegelt und ist in der Funktionsdeklaration an der zweiten Stelle zu finden. Das letzte Argument ist optional und wird meistens als Zeiger auf eine Dateistruktur, welche kopiert werden soll angegeben. [6, S. 90f.]

Mit den Übergabeparametern und dem Dateideskriptor wird die Funktion dann in Anwendungen im Userspace aufgerufen, im Kernel werden die Daten weiterverarbeitet und wieder zurück gegeben.

2.3 Datenaustausch zwischen Kernel und Userspace

Durch die Notwendigkeit von getrennten Speicherbereichen des Kernels und des Userspaces, wie im vorausgegangen Kapitel erläutert, wird der Datenaustausch zwischen den beiden Ebenen natürlich schwieriger. Durch

copy_from_user beziehungsweise *copy_to_user* stehen im Linuxkernel zwei Funktionen als hilfreiche Werkzeuge für diesen Austausch zu Verfügung.

```
1 unsigned long copy_from_user(void *to, const void *from, unsigned
   long num);
2 unsigned long copy_to_user(void *to, const void *from, unsigned
   long num);
```

Codeausschnitt 2.2: Vereinfachte Funktionsdeklaration aus [15, linux/uaccess.h, Zeile 140ff.]

Die Hauptaufgabe beider Funktionen ist das Kopieren von Daten, aber zusätzlich werden die übergebenen Speicherbereiche auf Gültigkeit überprüft. In der *copy_from_user* werden die Daten ab *from* aus dem Userspace mit der Größe von *num* Bytes an die Stelle *to* in den Kernel kopiert. Analog arbeitet das Gegenstück *copy_to_user*. Hier gibt *from* allerdings die Speicherstelle im Kernel an und somit ist *to* die Stelle im Userspace. Im Erfolgsfall geben beide Funktionen 0 zurück, andernfalls wird die Anzahl der nicht kopierten Bytes zurückgegeben. [9, S. 250f.]

2.4 Plattformtreiber

Gerätetreiber, und damit auch Plattformtreiber, sind unter Linux im Kernel angesiedelt. Eigene Treiber werden hierzu meist modular entwickelt und nicht fest in den Kernel integriert. Allerdings muss auch der Programmierer bei den nachgeladenen Kernelmodulen auf die korrekte Nutzung des Speicherplatzes achten, da diese Module ebenfalls im Kernel laufen und somit ein Zugriffsfehler schwerwiegende Folgen hätte. [9, S. 231ff.]

Damit die Treiber richtig funktionieren gibt es einige Bestandteile, die in jedem Modul wiederzufinden sind. Standardmäßig wird die Registrierung von Treiber und Gerät an unterschiedlichen Teilen des Programms ausgeführt. [5]

Jedes Kernelmodule besitzt mindestens eine *init* und *exit* Funktion. Hierzu gibt es eigene Makros, in welchen die Funktionen übergeben werden und somit den Kernel mit dem Treiber bekannt machen. Der Aufruf ist dann entweder beim Kernelboot bzw. beim Laden oder beim Entfernen des Treibers platziert. [15, linux/module.h, Zeile 79ff.]

Beim Laden des Treibers wird eine *platform_driver* Struktur übergeben. In dieser Struktur sind Zeiger auf die Funktionen gespeichert, die beim Erzeugen oder Löschen einer Instanz benötigt werden. In dieser Arbeit werden lediglich die *probe* und *remove* Funktionszeiger betrachtet. Beim Registrieren einer Instanz wird die Funktion hinter dem *probe* Zeiger aufgerufen und analog beim Auflösen die *remove* Funktion.[5]

```
1 struct platform_driver {  
2     int (*probe)(struct platform_device *);  
3     int (*remove)(struct platform_device *);  
4     void (*shutdown)(struct platform_device *);  
5     int (*suspend)(struct platform_device *, pm_message_t state);  
6     int (*resume)(struct platform_device *);  
7     struct device_driver driver;  
8     const struct platform_device_id *id_table;  
9     bool prevent_deferred_probe;  
10 };
```

Codeausschnitt 2.3: platform_driver Struktur in [15, linux/platform_device.h, Zeile 184ff.]

Beim Anlegen der Instanz kann ein Zeiger auf eine Struktur übergeben werden, in welchem Daten gespeichert sind. In der *probe* Funktion sind somit spezielle Daten für ein entsprechendes Gerät vorhanden. [5]

2.4.1 Major- und Minornummern

In Linux bekommt jedes Gerät in dem Ordner */dev/* eine Major- und Minornummer, d.h. es wird bereits mit diesen Nummern angelegt. Die Majornummer kennzeichnet hier üblicherweise den Treiber, zu welchem das Gerät gehört, analog dazu wird die Minornummer vom Kernel genutzt um auf das exakte Gerät zu referenzieren. [6, S. 43f.] Seit der Kernelversion 2.6 arbeitet dieser intern nicht mehr mit Major- und Minornummern, sondern mit Gerätenummern. Die 32-Bit Variable setzt sich aus den beiden Nummern zusammen. [9, Seite 244]

2.5 Multifunction Device

Normalerweise wird lediglich ein Gerät angelegt, ohne weitere Unterteilungen vorzunehmen. Es ist allerdings möglich, dass ein Hardwareblock mehr als eine Funktionalität hat. Damit das gleiche Gerät in mehreren Untersystemen registriert werden kann, benötigt man die Möglichkeit es als Multifunction Device (MFD) anzulegen. [4]

Bevor die Funktion zum Anlegen von MFD näher betrachtet wird, sollen zunächst zwei benötigte Strukturen erläutert werden.

Als erste Struktur wird die *resource* näher betrachtet. Hier werden Parameter für einen Speicherbereich abgelegt, damit auf diesen zugegriffen werden kann. Die beiden Parameter sind *start* und *end*. Die beiden Werte legen die Größe und den Ort der Ressource fest. In *name* wird der Name der Datenquelle gespeichert und in der Variablen *flags* werden über Defines unter anderem der Datentyp und weitere optionale Einstellungen

```

1 struct resource {
2     resource_size_t start;
3     resource_size_t end;
4     const char *name;
5     unsigned long flags;
6     [...]
7     struct resource *parent, *sibling, *child;
8 };

```

Codeausschnitt 2.4: Struktur resource in [15, linux/ioport.h, Zeile 20ff.]

festgelegt. In dem letzten drei Parametern können abhängige Ressourcen entsprechend ihrem Grad gespeichert werden.

```

1 struct mfd_cell {
2     const char *name;
3     int id;
4     [...]
5     /* platform data passed to the sub devices drivers */
6     void *platform_data;
7     size_t pdata_size;
8     [...]
9     /*
10    * Device Tree compatible string
11    * See: Documentation/devicetree/usage-model.txt Chapter 2.2 for
12    * details
13    */
14     const char *of_compatible;
15     [...]
16     /*
17    * These resources can be specified relative to the parent device.
18    * For accessing hardware you should use resources from the
19    * platform dev
20    */
21     int num_resources;
22     const struct resource *resources;
23     [...]
24 };

```

Codeausschnitt 2.5: Struktur mfd_cell in [15, linux/mfd/core.h, Zeile 29ff.]

Die zweite Struktur wird benötigt um einem MFD wichtige Parameter zum Anlegen mitzugeben. Aus diesem Grund werden in der *mfd_cell* Struktur lediglich die benötigten Parameter betrachtet.

In der Variable *name* und *id* werden der Name des Treibers und eine spezifische Nummer gespeichert. Der *void* Zeiger *platform_data* ist ein benutzerdefinierter Datenzeiger, welcher an das untergeordnete Gerät weitergereicht wird, da der Typ variieren kann, wird in *pdata_size* die zugehörige Datengröße übergeben. In der Variable *of_compatible* wird eine

sortierte Liste von strings gespeichert. Beginnend mit dem exakten Namen des Geräts folgt dann eine optionale Liste mit weiteren kompatiblen Geräten. [13, devicetree/usage_model.txt, Zeile 116ff.] Der Zeiger *resources* speichert die zugehörige Ressource ab, bzw. ein Zeiger auf ein Array von Ressourcen. Die Anzahl der abgespeicherten Ressourcen wird in *num_resources* abgelegt.

```
1 extern int devm_mfd_add_devices(struct device *dev, int id, const
    struct mfd_cell *cells, int n_devs, struct resource *mem_base,
    int irq_base, struct irq_domain *irq_domain);
```

Codeausschnitt 2.6: Funktionsdeklaration in [15, linux/mfd/core.h, Zeile 130ff.]

Beim Anlegen eines Untergeräts über *devm_mfd_add_devices* werden mehrere Parameter benötigt. Als Erstes wird ein Zeiger auf das übergeordnete Gerät übergeben. Der zweite Übergabeparameter ist die Struktur *mfd_cell*, wie oben erwähnt wird diese benötigt um das anzulegende Gerät näher zu beschreiben. Durch den Integer *n_devs* wird die Anzahl der zu registrierenden Kindergeräten angegeben. Dies ist notwendig, da der Parameter *cells* auch ein Array beinhalten kann. Die anderen Übergabeparameter werden nicht näher betrachtet, da sie im folgenden nicht benötigt werden. [14, mfd/mfd-core.c, Zeile 359ff.]

In der Funktion ist zusätzlich implementiert, dass beim Entfernen des übergeordneten Gerät alle Untergeräte automatisch aufgelöst werden. [14, mfd/mfd-core.c, Zeile 356f.]

3

FPGA Resource Abstraction

In diesem Kapitel soll auf die Vorgehensweise und die Implementierung des, in Kapitel 1.3 vorgestellten, FRA eingegangen werden. Die Schwerpunkte liegen hierbei bei der Beschreibung des generischen Plattformtreibers, den IOCTLs, der Kernel- bzw. Userspaceimplementierung sowie der Einbindung in die bestehende Software. Damit es der Umfang der Arbeit nicht übersteigt, wird nur die Xbar softwareseitig implementiert.

3.1 Generischer Plattformtreiber

Zunächst soll auf den Plattformtreiber eingegangen werden, der eine einwandfreie Kommunikation zwischen den Firmware Modulen im FPGA und dem Kernel gewährleisten soll.

Wie in dem Kapitel 2.4 bereits erläutert, werden für die grundlegende Funktionsweise eines Treiber verschiedene Funktionen benötigt. Zunächst soll auf die Registrierungs- bzw. Aufräumfunktion des Treibers eingegangen werden.

Beim Laden des Treibers werden verschiedene allgemeingültige Parameter gesetzt und zusätzlich Speicherplatz allokiert.

Desweiteren wird beim Laden als Erstes eine Klasse erstellt, der später die einzelnen Module zugeordnet werden. Diese Klasse wird in der, dem Treiber, zugehörigen Struktur *afm_driver* in der Variablen *class* abgespeichert.

Da die Majornummer den Treiber kennzeichnet (siehe Kapitel 2.4.1) wird diese in der globalen Treiberstruktur als *major_id* gespeichert. Für die Minornummer wird in der Datenstruktur eine Liste *minor_list[AFM_MAX]* und ein zugehöriges Spinlock *minor_spinlock* initialisiert. Über die Liste


```
1 struct afm_driver
2 {
3     unsigned int major_id;
4
5     uint8_t minor_list[AFM_MAX];
6     spinlock_t minor_spinlock;
7     struct class * class;
8 };
```

Codeausschnitt 3.1: Struktur des Treibers

wird später eine freie Nummer ausgewählt, die bei jedem Module individuell ist und gleichzeitig wird dadurch die Anzahl der möglichen Module auf *AFM_MAX* begrenzt. Das Spinlock sorgt bei der Auswahl der Minornummer für einen konfliktfreien Vorgang, sodass keine Nummer doppelt vergeben werden kann.

Am Ende der initialen Funktion wird der Plattformtreiber mit der *platform_driver* Struktur (siehe Codeausschnitt 2.3) registriert. Damit werden die initiale Funktion zum Anlegen, aber auch die Funktion zum Deinitialisieren der Geräte übergeben.

Analog wird beim Freigeben des Treibers der allokierte Speicherplatz freigegeben, der Plattformtreiber abgemeldet und die erstellte Klasse zerstört.

Beim Anlegen einer Instanz vom Treiber wird die *probe* Funktion des Plattformtreibers aufgerufen und es wird der Modultyp und der Name in einer *arri_fra_mod_config* Struktur zur Verfügung gestellt. In der initialen Funktion der Instanz wird diese Struktur genutzt um spezifische Einstellungen für jede einzelnes Gerät festgelegt und in einer entsprechenden Datenstruktur (siehe Codeausschnitt 3.2) gespeichert.

Der einzige Übergabeparameter der Probefunktion ist die Struktur des *platform_device*, diese wird in dem Zeiger *pdev* gespeichert.

Aus der Liste der Minornummern (siehe Codeausschnitt 3.1) wird nach dem Sperren des zugehörigen Spinlocks eine freie Nummer ausgewählt und als *minor_id* gespeichert, sowie in der Liste als verwendet gesetzt. Danach wird das Spinlock wieder freigegeben.

Anschließend wird mit Hilfe der Majornummer und der Minornummer ein Gerät erstellt und als *cdev* gespeichert.

Damit das Öffnen der Geräte auf eine bestimmte Anzahl limitiert ist, wird beim Initialisieren der Instanz ein Array der *afm_file* Struktur angelegt. Dadurch wird gleichzeitig der Speicherplatz statisch zur Kompilierzeit reserviert. Durch den Spinlock *open_spinlock* wird später dafür gesorgt, dass Kollisionen und Raceconditions vermieden werden.

```

1 struct afm_device
2 {
3     struct arri_fra_mod_config *mod_config;
4
5     struct class *class;
6     struct cdev cdev;
7     unsigned int minor_id;
8     struct platform_device *pdev;
9     struct resource *res;
10    u8 __iomem *base;
11
12    struct afm_file fdev[AFM_FILE_MAX];
13    spinlock_t open_spinlock;
14
15    #define AFM_NOLOGGING ((uint32_t)0U)
16    #define AFM_LOGGING ((uint32_t)1U)
17    uint32_t has_logging;
18 };

```

Codeausschnitt 3.2: Auszug aus der Struktur einer Instanz

3.2 Konzeptionierung der IO Controls

Damit der Zugriff und die Steuerung der Geräte, welche man über den generischen Treiber angelegt hat möglich ist, soll dieser Abschnitt auf die Konzeptionierung der IOCTLs eingehen. Wie in Kapitel 2.2 erläutert werden diese benötigt, um zwischen Kernel und Userspace Daten auszutauschen. Da der Großteil der Software im Userspace läuft, aber der Treiber im Kernel, wird der hauptsächliche Zugriff auf die Geräte über IOCTLs geregelt. Die Überlegungen hinter den einzelnen IOCTLs werden im Folgenden näher erläutert.

Als Erstes ist eine Anmeldung der Anwendung bei dem Gerät notwendig. Um die Eindeutigkeit zu garantieren wird hier die Prozesskennung (PID) übergeben. Allerdings wird auch der Prozessname benötigt, damit eine schnelle Nachvollziehbarkeit bei der Fehlersuche auf der Kamera vorhanden ist. Ohne die Ausführung von *ARRI_FRA_MOD_HELLO* bleibt der weitere Zugriff auf das Gerät über IOCTLs verwehrt, da die Anmeldung notwendig ist, um im laufenden Betrieb eine Analyse vorzunehmen.

Die Protokollierung der Zugriffe auf ein Gerät ist ein notwendiger Bestandteil. Dadurch soll, vor allem im Fehlerfall, die Suche erleichtert werden. Aufgrund der Zugriffe durch verschiedene Prozesse besteht ohne Protokollierung kein einheitlicher Überblick über die Zugriffe. In dem Zusammenhang werden zwei IOCTLs angelegt. Dabei ist *ARRI_FRA_MOD_LOGGING* zum Aktivieren der Nachrichten zuständig und

	Schreibend	Lesend
Einzeln	*SET_REG	*GET_REG
Einzeln mit Bitmaske	*SET_REG_BITMASK	-
Blockweise	*SET_REG_BLOCK	*GET_REG_BLOCK

Tabelle 3.1: Namensgebung der IOCTLs für den Registerzugriff

analog werden diese durch *ARRI_FRA_MOD_NOLOGGING* deaktiviert.

Beim Anlegen des Geräts werden der Gerätetyp und der Name abgelegt. Im Userspace besteht die Notwendigkeit auf diese Informationen nach dem Öffnen des Geräts zuzugreifen. Dafür wird ein eigenes IOCTL (*ARRI_FRA_MOD_GET_ID*) benötigt, welches lediglich den Typ und Namen aus der Instanzstruktur (siehe Codebeispiel 3.2) zurück gibt.

Die essenziellen Funktionen des Treibers sind das Setzen bzw. Auslesen von Register. Basierend auf den, in der aktuellen Implementierung genutzten Funktionen und den entsprechenden Registerzugriffen, gibt es bis zu drei verschiedene Arten. Jede soll durch ein eigenes IOCTL abgebildet werden, da so die spätere Verwendung vereinfacht wird. In der Tabelle 3.1 sind die Namen zu den Zugriffen dargestellt. Aus Platzgründen wurde auf den identischen Prefix *ARRI_FRA_MOD_* verzichtet.

Die Größe eines Registers ist auf 4 Byte normiert. Damit werden zum einfachen Setzen eines Registers lediglich zwei Parameter benötigt. Zum Einen die Stelle des Registers im Gerät (auch Registernummer) und zum Anderen der Inhalt. Allerdings besteht auch die Notwendigkeit einzelne Bits oder mehrere hintereinander liegende Register über einen Aufruf zu setzen. Für beide Sonderfälle gibt es ein gesondertes IOCTL. Zum Setzen von einzelnen Bits wird zusätzlich zu den beiden oben genannten Parametern noch eine Bitmaske übergeben. Mithilfe der Bitmaske werden im Register erst die entsprechenden Bits gelöscht und danach die übergebenen Bits gesetzt. Dadurch ist es nicht mehr notwendig erst das entsprechende Register auszulesen, anschließend zu verändern und dann in das gleiche Register zu setzen. So werden überflüssige IOCTL und Registerzugriffe dezimiert. Der zweite Sonderfall benötigt andere Übergabeparameter als das einfache Setzen. Zusätzlich zu der Registernummer wird hier eine Registeranzahl gebraucht. Des Weiteren muss der Inhalt für die Register in einem Array mit der Größe der Anzahl übergeben werden. Damit kann dann der Reihe nach jedes Register einzeln mit dem entsprechenden Inhalt beschrieben werden.

Das Auslesen der Register erfolgt nahezu analog zu dem Setzen. Der Parameter für den Registerinhalt wird hier allerdings über das IOCTL

gefüllt und dann zurück in den Userspace übergeben. Register über eine Bitmaske auszulesen würde keine weiteren Vorteile gegenüber dem Auslesen des ganzen Registers bieten. Aus diesem Grund wird es nicht implementiert. Damit man einen ganzen Block an Registern zurück lesen kann, muss durch das Übergeben eines Arrays mit der richtigen Größe des Speicherbereichs zu Verfügung gestellt werden.

Dadurch sind die wichtigsten IOCTLs erläutert, die notwendig sind um die Grundfunktionen des Geräts abzudecken und zusätzlich eine triviale Möglichkeit zur Fehlersuche bieten.

3.3 Implementierung im Kernel

Im Kernel gibt es zwei verschiedene Stellen, an welchen der Code implementiert ist. Zum Einen muss der bestehende FPGA Treiber entsprechend erweitert werden, sodass die Geräte unter Linux angelegt werden können und zum Anderen muss ein FRA Treiber implementiert werden, um die angelegten Geräte zu öffnen und zu schließen.

Zur besseren Übersichtlichkeit wurde sich für zwei Namenskonventionen entschieden. Strukturen und Funktionen, die mit *arri_fra_mod* beginnen, werden außerhalb des FRA Treibers benötigt bzw. angelegt. Mit diesem Prefix werden die Namen recht lang, aus diesem Grund wird treiberintern auf den nicht so aussagekräftigen Prefix *afm* abgekürzt.

3.3.1 Anlegen der Geräte

Der Zugriff der Software auf die FPGA-Module soll über Geräte stattfinden. Durch die unterschiedlichen Funktionalitäten der Module, wie in Kapitel 1.2.2 erläutert, werden diese Geräte als Kindgeräte vom FPGA angelegt, d.h. kernelseitig wird dieser Bestandteil im vorhandenen FPGA Treiber implementiert.

```

1 struct arri_fra_mod_init
2 {
3     #define ARRIFPGA_FRA_MAX_NAME ((uint32_t)50)
4     char type[ARRIFPGA_FRA_MAX_NAME];
5     char name[ARRIFPGA_FRA_MAX_NAME];
6     uint32_t offset;
7     /* register size in bytes */
8     uint32_t size;
9 };

```

Codeausschnitt 3.3: Struktur zum Initialisieren des Geräts

Über ein IOCTL, mit der obigen Struktur als Übergabeparameter, wird das Gerät angelegt.

Als Erstes wird hier Speicherplatz für die drei Strukturen *resource*, *mfd_cell* und *arri_fra_mod_config* allokiert. Mithilfe dieser Strukturen wird am Ende der Funktion das Gerät angelegt.

In der *resource* Struktur (siehe Codeausschnitt 2.4) wird der Speicherbereich des FPGA-Modules abgebildet. Für die Variable *start* wird auf den bereits gemapten Registerbereich im FPGA noch der, im *arri_fra_mod_init* übergebene, *offset* aufaddiert. Die Variable *end* wird aus dem Startwert (*start*) und der Größe des Speicherbereichs (*size*) bestimmt. Durch den Parameter *flags* wird die Ressource als Speicherbereich festgelegt und als *parent* wird die Ressource des gesamten FPGAs angegeben.

Als Nächstes wird die *mfd_cell* Struktur (siehe Codeausschnitt 2.5) gefüllt. Hier wird die Variable *id* auf eine globale Variable gesetzt, die bei jedem Funktionsaufruf um eins erhöht wird. *num_resources* wird auf eins gesetzt, da es für jedes Gerät nur eine Ressource gibt. Entsprechend wird in *resources* der Zeiger der oben angegebenden Ressource übergeben. Analog wird in *platform_data* der Zeiger auf die *arri_fra_mod_config* Struktur und in *pdata_size* die Größe der Struktur abgelegt. Durch die *arri_fra_mod_config* wird der Gerätenamen und -typ an den Treiber übergeben. Bei Bedarf kann die Struktur zum Übergeben der internen Daten des Modules angepasst werden.

Über die, im Codeausschnitt 2.6 aufgezeigte Funktionsdeklaration wird das Gerät angelegt. Dadurch wird die *probe* Funktion des Plattformtreibers ausgeführt und das erfolgreich angelegte Gerät ist auf der Kamera unter */dev/fra/* zu finden.

3.3.2 Öffnen und Schließen der Geräte

Um später im Userspace auf die Geräte zuzugreifen und initiale Einstellungen vorzunehmen, muss eine *open* Methode implementiert werden. Im Umkehrschluss werden die Einstellungen durch die *release* Funktion zurückgesetzt. [6, Seite 58f.]

Die Anzahl von offenen Geräten ist durch die Speicherallokierung in der Struktur *afm_device* (siehe Codeausschnitt 3.2) auf *AFM_FILE_MAX* (hier: 4) begrenzt. Die geöffnete Instanz eines Geräts wird in der Struktur *afm_file* abgelegt.

Beim Öffnen eines Geräts wird, nachdem das entsprechende Spinlock (*open_spinlock*) gesperrt wurde, nach einem freien Gerät zum Öffnen gesucht. Hierzu wird über eine for-Schleife iteriert, bis die maximale Anzahl

```

1 struct afm_file {
2 #define AFM_FREE      (( uint8_t)0U)
3 #define AFM_USED      (( uint8_t)1U)
4     uint8_t in_use;
5     struct afm_device *dev;
6     int minor;
7     char name[ARRL_FRA_MOD_MAX_NAME];
8     uint32_t type;
9     uint32_t pid;
10    struct file *file;
11 };

```

Codeausschnitt 3.4: Struktur zum Ablegen des Geräts

erreicht ist. In jedem Durchgang wird die Variable *in_use* überprüft, entsprechend dem Define kann dann festgestellt werden, ob es noch möglich ist ein Gerät zu öffnen. Wurde eine freie Stelle gefunden, wird der Parameter auf *AFM_USED* gesetzt und das Spinlock wieder freigegeben. Wenn die maximale Anzahl erreicht ist, wird eine Fehlermeldung ausgegeben und ein Fehlercode zurückgegeben. Die Variable *dev* wird auf das geöffnete Gerät gesetzt, analog wird *minor* auf die verwendete Minornummer und *file* auf die, in der *open* Methode, übergebene Datei gesetzt. Die anderen Parameter beschreiben den Prozess, welcher das Gerät öffnet und werden durch ein IOCTL entsprechend beschrieben.

Analog wird beim Schließen eines Geräts in der *release* Funktion, die Inhalte aus *file* und *dev* auf *NULL* gesetzt. Dadurch ist keine Zuordnung mehr möglich und durch das Freigeben der Variable *in_use* kann beim nächsten Öffnen der Eintrag im Array wiederverwendet werden.

3.4 Implementierung im Userspace

Der Zugriff auf die FRA Module im Userspace ist in drei Ebenen gekapselt. In diesem Kapitel werden auf die Bibliothek, die Middleware und das Backend eingegangen und somit ein Überblick über alle drei Ebenen gegeben. Die untere Stufe besteht aus verschiedenen backendspezifischen Funktionen. Im Folgenden wird nur auf das Kernelbackend eingegangen und das Testbackend wird im Kapitel 4 erläutert. Im Kernelbackend werden mit den übergebenen Parametern die IOCTLs (siehe Kapitel 3.2) abgesetzt und somit im Kernel die Hardware gelesen oder beschrieben.

In der FRA Middleware sind erweiterte Wrapperfunktionen zu finden. Hier werden verschiedene Überprüfungen durchgeführt, damit kein fehlerhafter Zugriff stattfindet. Des Weiteren wird im Wrapper entsprechend dem Backendtyp, die Funktion aus der unteren Ebene ausgewählt.

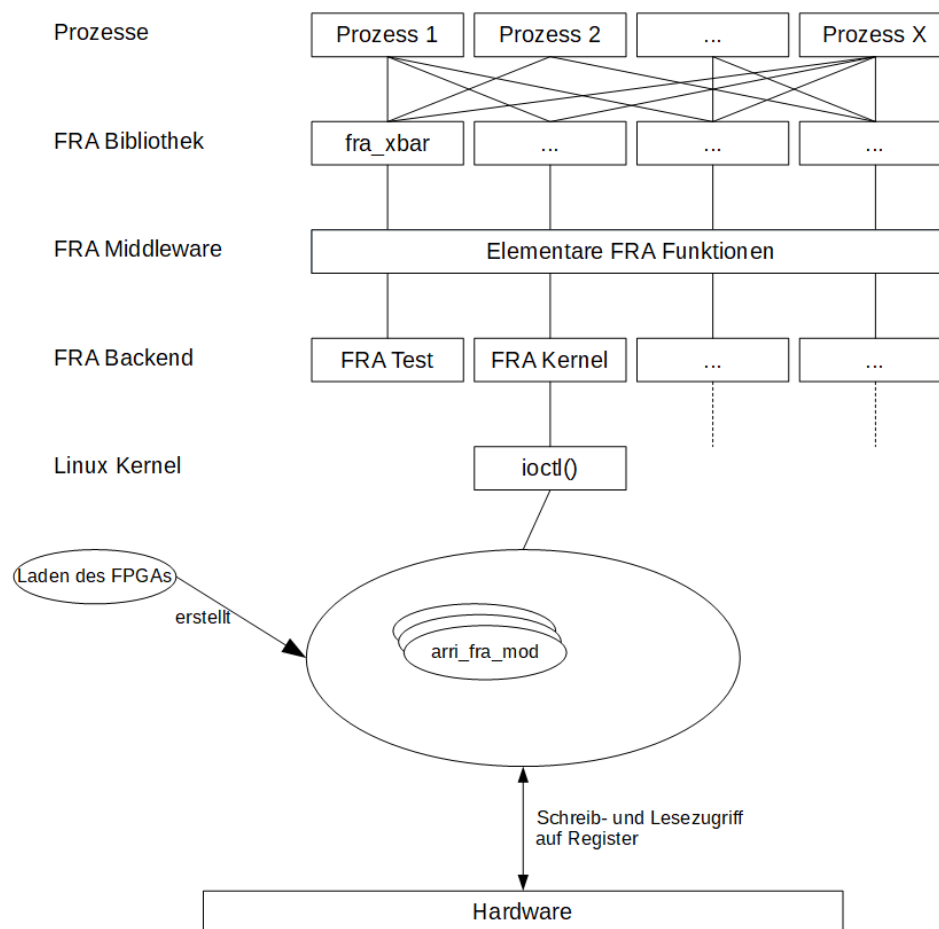


Abbildung 3.1: Darstellung des Aufbaus des FRA

Durch die gekapselten Ebenen ist eine Erweiterung um neue Backendtypen einfach gestaltet und zusätzlich die Wartbarkeit erhöht worden. Dadurch müssen bei Änderungen im Kernelzugriff, nur im Backendcode die entsprechenden Stellen geändert werden.

3.4.1 FRA Bibliothek

Um das FRA vollständig nutzen zu können, müssen die Geräte geöffnet, geupdatet und auch wieder geschlossen werden. Dies passiert in der sogenannten FRA Bibliothek. Für jeden Modultyp gibt es im FRA eigene, spezifische Funktionen. Diese Funktionen ersetzen die vorherigen Funktionen, welche direkt in den FPGA geschrieben haben. Im Rahmen dieser Arbeit

wird nur die Xbar betrachtet und somit auch nur die Implementierung dieser in der Bibliothek.

```

1  int32_t fra_xbar(struct fra_handle const *handle, const uint32_t
    transid, const uint32_t setting)
2  {
3      int32_t retval = ERRVALUE_SUCCESS;
4      uint32_t num, reg;
5
6      FRA_CHECK_HANDLE_TYPE(handle, FRA_MOD_TYPE_XBAR);
7
8      num = AVALONVIDEO_CROSSBAR_MXN_ENABLE_OUTPUT_REG;
9      reg = setting;
10
11     retval = fra_set_reg(handle, transid, num, reg);
12
13     return retval;
14 }
```

Codeausschnitt 3.5: Funktion zum Setzen der Xbar

Für die Xbar sind fünf spezifische Funktionen implementiert. In diesen Funktionen werden die Einstellungen gesetzt, die aktuellen Registerwerte protokolliert, auf aktuelle Fehlercodes überprüft und der aktuelle Status zurück gegeben. Da in allen Funktionen hauptsächlich auf die FRA Middleware zugegriffen wird und abgesehen von diesem Zugriff, die Daten aufbereitet werden, wird nur die *fra_xbar* Funktion exemplarisch betrachtet.

In der Funktion im Codeausschnitt 3.5 werden die Ein- und Ausgänge einer Xbar entsprechend dem übergebenen *setting* verbunden. Durch das Makro *FRA_CHECK_HANDLE_TYPE* wird vorher überprüft, ob das Gerät im *handle* eine Xbar ist. Wenn dies nicht der Fall ist, wird ein entsprechender Fehlercode zurückgegeben und die Funktion ist beendet. Diese Überprüfung findet auch bei den restlichen Funktionen in der FRA Bibliothek statt. Ist der Modultyp korrekt, werden die Übergabeparameter für *fra_set_reg* befüllt und die Funktion aufgerufen. Mit dem Zurückgeben des Rückgabewerts der Middleware Funktion ist die Funktion beendet.

Finden mehrere Aufrufe in die Middleware statt, wird nach jedem Aufruf der Rückgabewert überprüft und im Fehlerfall direkt die Funktion abgebrochen.

3.4.2 FRA Middleware

Damit die Funktionsaufrufe in der FRA Bibliothek unabhängig vom Backendtyp funktionieren, werden in der FRA Middleware Wrapper Funktio-

nen zur Verfügung gestellt. Daneben wird auch eine Struktur zum Verwalten der wichtigen Parameter der FRA Module im Userspace bereitgestellt.

```
1 struct fra_handle
2 {
3     int dev;
4     char dev_type[FRA_MAX_NAME];
5     char dev_name[FRA_MAX_NAME];
6     uint32_t type_id;
7     const struct fra_backend_funcs *backend_funcs;
8 };
```

Codeausschnitt 3.6: Struktur zum Abspeichern wichtiger Parameter

Der erste und wichtigste Parameter in der Struktur ist der Dateideskriptor *dev*. Hierüber kann nach dem Öffnen des Geräts weiterhin auf dieses zugegriffen werden. Die restlichen Parameter werden beim Initialisieren des Geräts gesetzt. *dev_type* und *dev_name* geben den Modultyp und den Namen an. Für spätere Überprüfungen gibt es für jeden Modultyp noch eine ID, dieses wird in der Variablen *type_id* abgelegt. In der *fra_backend_funcs* Struktur sind Prototypen aller backendspezifischen Funktionen abgelegt.

Das Öffnen eines Geräts ist nur über die *fra_init* Funktion möglich. Da neben dem Öffnen auch eine Anmeldung bei dem Gerät stattfinden muss (siehe Kapitel 3.2), wird dies über einen Funktionsaufruf abgedeckt.

In der initialen Funktion *fra_init* werden verschiedene Einstellungen vorgenommen und Funktionen aufgerufen. Zum Einen wird der Funktion ein Backendtyp übergeben und anhand von diesem die Funktionsstruktur im *fra_handle* gesetzt. Zum Anderen muss neben dem Öffnen des Geräts auch eine Anmeldung vom Prozess stattfinden (siehe Kapitel 3.2). Hierzu werden jeweils die entsprechenden Wrapper Funktionen aufgerufen um die Funktionalität bei allen Backendtypen zu garantieren. Zum Bestimmen von *dev_type* und *dev_name* werden auch über eine Wrapper Funktion die Parameter vom Gerät geholt und entsprechend im *fra_handle* gesetzt. Mithilfe einer Liste, in welcher Name, ID und Größe der einzelnen Module hinterlegt sind, wird die Variable *type_id* gesetzt. Hierfür wird *dev_type* mit den Namen der Liste verglichen und bei einem Treffer die entsprechende ID abgespeichert.

Die Grundstruktur der Wrapper Funktionen ist für alle identisch, deshalb wird im folgenden beispielhaft die *fra_set_reg* näher betrachtet. Grundsätzlich gibt es für jedes konzeptionierte IOCTL (siehe Kapitel 3.2) im Kernel eine Wrapper Funktion, lediglich für das Logging sind beide IOCTLs in einer Funktion zusammen gefasst.

Zu Beginn einer jeden Methode werden über ein Makro verschiedene Überprüfungen durchgeführt.

```

1 #define FRA_CHECK_HANDLE(p_handle)      \
2     if (p_handle == NULL)                \
3     {                                    \
4         return ERRVALUE_INVALID_PARAMETER; \
5     }                                    \
6     if (p_handle->dev == -1)              \
7     {                                    \
8         return ERRVALUE_DEVICE_NOT_OPEN;  \
9     }                                    \
10    if (p_handle->backend_funcs == NULL)   \
11    {                                    \
12        return ERRVALUE_NOT_INITIALIZED;  \
13    }

```

Codeausschnitt 3.7: Makro zum Überprüfen des Handles

Ohne ein gültiges Handle würden alle nachfolgenden Aufrufe einen Fehler zurück geben, da alles basierend auf diesem Handle erfolgt. Aus diesem Grund wird in Zeile 2 als Erstes überprüft, ob das *fra_handle* valid ist. Über den Dateideskriptor kann man herausfinden, ob das Gerät schon geöffnet wurde. Hier wird beim Initiieren der Struktur *dev* auf -1 gesetzt, bei einem geöffneten Gerät ist eine Zahl größer 0 abgespeichert. Als Letztes wird noch überprüft, ob ein Zeiger auf die *fra_backend_funcs* Struktur übergeben wurde. Die Funktion, in welcher das Makro aufgerufen wird gibt entsprechende, softwareintern definierte Fehlermeldungen zurück um die Fehlersuche zu erleichtern.

In der Funktionssignatur ist immer das *fra_handle* und die *transid* angeben. Durch das Handle werden alle notwendigen Informationen zum Zugriff auf das Gerät übergeben und die Transaktionsidentifikation *transid* ist in der Signatur implementiert, damit bei einer späteren Erweiterung nicht alle Aufrufe geändert werden müssen. Durch die Nummer soll es später möglich sein, Kernel- und damit Hardwarezugriffe zu gruppieren und direkt hintereinander abzuarbeiten. Aktuell wird sie lediglich bis zum Ende durchgereicht und nicht weiter betrachtet, da es kein Teil der Arbeit ist.

Das Makro im Codeausschnitt 3.8 Zeile 9 ist im vorhergehenden Absatz genauer erläutert. Durch die if - Bedingung in Zeile 11 wird überprüft, ob für das ausgewählte Backend die entsprechende Funktion implementiert ist. Ist die Funktion nicht implementiert, liegt an der Stelle ein *NULL* Zeiger. Durch die Überprüfung wird vermieden, dass beim Aufrufen der Funktion auf *NULL* zugegriffen wird und somit beim laufenden Programm ein Fehler auftritt. Der Rückgabewert in der Funktion in der FRA Middleware

```

1  /* fra_set_reg(handle, transid, num, reg)
2  *      sets a register at num
3  */
4  int32_t fra_set_reg(struct fra_handle const *handle,
5                     const uint32_t transid,
6                     const uint32_t num,
7                     uint32_t reg)
8  {
9      FRA_CHECK_HANDLE(handle);
10
11     if (!handle->backend_funcs->fra_set_reg) return
        ERRVALUE_FUNCTION_NOT_AVAILABLE;
12     return handle->backend_funcs->fra_set_reg(handle, transid, num,
        reg);
13 } /* fra_set_reg () */

```

Codeausschnitt 3.8: Funktion zum Setzen eines Registers

entspricht entweder einem entsprechenden Fehlerwert oder dem Rückgabewert der Backendfunktion.

3.4.3 FRA Kernelbackend

Da die Wrapper lediglich alle benötigte Übergabeparameter an die Backendfunktionen weiterreichen, haben diese eine identische Funktions-signatur. Die einzelnen Funktionen unterscheiden sich im Kernelbackend nur durch die aufgerufenen IOCTLs und entsprechenden den übergebenden Strukturen dazu. Beispielhaft soll hier wieder die Funktion zum Setzen eines Registers betrachtet werden.

Im Backend finden keine weiteren Überprüfungen statt, da dies bereits eine Ebene höher geschehen ist. Nach dem Füllen der Übergabestruktur wird das IOCTL aufgerufen und anschließend auf Fehler überprüft. Im Fehlerfall wird eine Meldung ausgegeben und der Fehlerwert zurückgegeben.

3.5 Einbindung ins Geometrie Framework

Um das FRA vollständig in der Kamerasoftware zum Implementieren müssen die Geräte angelegt, geöffnet, geupdatet und auch wieder geschlossen werden. Dies geschieht an verschiedenen Stellen in der Software und zum Großteil eng verknüpft mit dem Geo Framework.

Angelegt werden die Geräte aktuell beim Laden des FPGAs. An dieser Stelle ist bekannt, welche Firmware geladen wurde und entsprechend kann über eine Struktur bestimmt werden, ob und an welcher Stelle ein

```

1 int32_t fra_kernel_set_reg(struct fra_handle const *handle, const
    uint32_t transid, const uint32_t num, uint32_t reg)
2 {
3     (void) transid;
4     int32_t retval = ERRVALUE_SUCCESS;
5     int32_t err;
6     struct arri_fra_mod_reg frareg;
7
8     frareg.num = num;
9     frareg.reg = reg;
10    err = ioctl(handle->dev, ARRI_FRA_MOD_SET_REG, &frareg);
11    if (err < 0)
12    {
13        error_msg(EH_ERROR, "%s:_FRA_IOCTL_failed_(%u)", handle->
            dev_name, err);
14        retval = ERRVALUE_IOCTL_FAILED;
15    }
16    return retval;
17 } /* fra_kernel_set_reg () */

```

Codeausschnitt 3.9: Funktion im Kernelbackend zum Setzen eines Registers

bestimmtes Modul vorhanden ist. Mit einem vorgegebenen Namen und Modultyp werden so die Geräte im Kernel angelegt. Anschließend kann in der kompletten Kamerasoftware davon ausgegangen werden, dass, sofern kein Fehler aufgetreten ist, die Geräte verfügbar sind. Das Anlegen der Geräte soll autark von der Software stattfinden, da dieses in Zukunft ohne die Adressen der Module im FPGA auskommen soll. Da dieses Konzept den Rahmen der Arbeit übersteigen würde, wird es nicht näher betrachtet.

Im Geo Framework sind die Module der Bildkette in der entsprechenden Reihenfolge abgebildet und über einen Modulindex ist jedes Modul individuell zu unterscheiden. So kann beim initialen Hardwareupdate des Geo Framework für alle, in diesem Framework abgebildeten und vorhandenen Module, ein Gerät geöffnet und entsprechend dem Modulindex in ein *fra_handle* Array abgelegt werden. In beiden Frameworks wird somit über den identischen Modulindex auf das gleiche Modul zugegriffen. Dies spielt vor allem beim Updaten der Module eine wichtige Rolle.

In den Hardwarefunktionen des Geo Framework werden für die Xbar die Konfigurationen des Ein- und Ausgänge aus dem Framework abgeholt und anschließend die *fra_xbar* Funktion (siehe Codeausschnitt 3.5) aufgerufen. Analog sollen auch für andere Module die Hardwarefunktionen abgeändert werden.

4

Frameworktest

Überprüfen der Funktionalität von den einzelnen Bestandteilen außerhalb der eigentlichen Anwendung gehört zum Entwickeln von Software dazu. Zusätzlich werden auch Tools zum Testen und Debuggen auf der Zielentwicklungsumgebung benötigt. In diesem Kapitel sollen beide Varianten näher betrachtet werden.

4.1 Plattformunabhängige Tests

Zunächst sollen auf die Tests eingegangen werden, die unabhängig von der Entwicklungsumgebung durchgeführt werden können. Damit dies möglich ist, muss die Struktur des *fra_handle* angepasst sowie ein entsprechendes Testbackend implementiert werden. Anschließend können verschiedene Tests erstellt werden, wobei die Unittest im Vordergrund stehen. Dadurch kann die FRA Middleware und Bibliothek unabhängig von der Software getestet und bei Änderungen überprüft werden, ob die beiden Teile korrekt arbeiten.

```
1 struct fra_test_mod
2 {
3     char type[FRA_MAX_NAME];
4     char name[FRA_MAX_NAME];
5     uint32_t size;
6     uint32_t *reg;
7 };
```

Codeausschnitt 4.1: Struktur zum Abbilden der virtuellen Hardware

Durch das Anlegen eines Geräts im Kernelbackend können alle benötigten Informationen und Register über dieses Gerät abgefragt und gesetzt

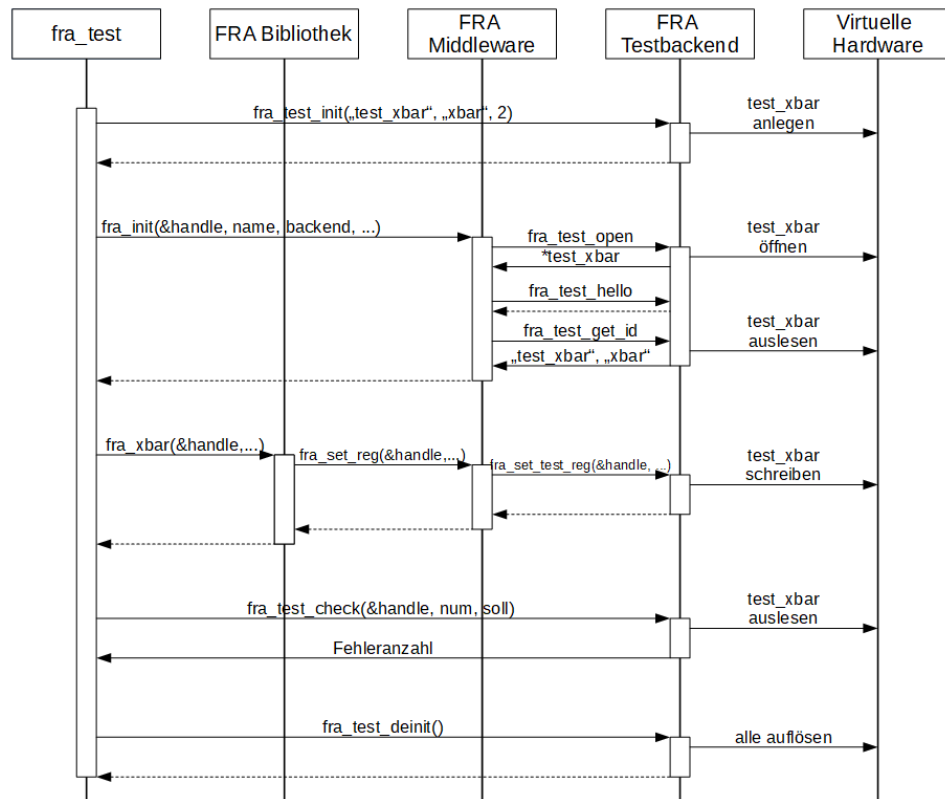


Abbildung 4.1: Beispielhaftes Sequenzdiagramm zum Unittest

werden. Da das Testbackend unabhängig laufen soll, ist das Anlegen von Geräten so nicht möglich. Um eine ähnliche Funktionalität wie im Kernel zu haben, wird das Modul über die *fra_mod_test* Struktur abgebildet. Hier werden alle notwendigen Informationen angelegt und auch entsprechende Register abgebildet, damit man diese beschreiben und auch auslesen kann. Die *fra_mod_test* Struktur wird als statisches Array im Testbackend angelegt und bildet so die virtuelle Hardware für das Testprogramm. Dadurch ist das Überprüfen der Register später möglich und auch eine maximale Anzahl der Module ist vorgegeben.

Damit die virtuelle Hardware richtig initialisiert wird, muss für die Register entsprechender Speicherplatz allokiert, aber auch die restlichen Parameter gesetzt werden. Analog dazu muss der Speicherbereich der Register beim Beenden des Programms wieder freigegeben werden. Dafür gibt es im Testbackend zusätzliche Funktionen, die sich um das initialisieren und deinitialisieren kümmern.

Beim Starten des Test wird der *fra_test_init* Funktion drei Parameter übergeben. Zum Einen wird der Name sowie der Typ des Moduls und zum Anderen wird die Größe des Registers in die Funktion gereicht. Ist die maximale Anzahl der Module noch nicht überschritten, wird mithilfe der Größe ein Registerblock allokiert. Anschließend wird die *fra_mod_test* Struktur mit allen vier Parametern gefüllt und eine statische Variable *mod_count* erhöht.

Wenn der Test am Ende angelangt ist, werden in der *fra_test_deinit* Funktion mithilfe der Variablen *mod_count* alle allokierten virtuellen Geräte wieder freigegeben.

Zum Öffnen eines Moduls im Testbackend wird analog zum Kernelbackend die Funktion *fra_init* aufgerufen. Hier wird allerdings ein anderer Backendtyp übergeben und somit in den Wrapperfunktionen entsprechend ins Testbackend weitergeleitet. Die Funktionalität der einzelnen Backendfunktionen ist, im Vergleich zum Kernelbackend, vereinfacht worden und die Funktionen kommen entsprechend ohne IOCTLs aus. Das Setzen und Lesen der Register erfolgt nun über die *fra_mod_test* Struktur im Handle. Diese wird bei der *fra_test_open* Funktion als Zeiger auf die entsprechende Stelle im statischen *fra_mod_test* Array beschrieben. Die richtige Stelle wird über einen Vergleich mit dem Modulname gefunden.

Damit überprüft werden kann, ob die Register richtig beschrieben bzw. ausgelesen worden sind, gibt es zwei Funktionen. Diese sind, wie die *fra_test_init* und *fra_test_deinit* Funktion, im Testbackend, aber werden ohne Wrapperfunktion aufgerufen, da sie spezifisch sind und lediglich beim Testen genötigt werden. Der *fra_test_check* Funktion werden neben dem Handle noch die Registernummer und der Sollwert des Registers übergeben. Über den Namen wird die Zuordnung des Handles zu der virtuellen Hardware gemacht und anschließend der Registerwert mit dem Sollwert verglichen. Bei einem fehlerhaften Wert wird eine Fehlermeldung ausgegeben und der Rückgabewert ist 1. Durch eine weitere Funktion (*fra_test_check_all*) kann das gesamte Register überprüft werden. Allerdings muss hier neben dem Handle noch ein Array der Registergröße übergeben werden. In diesem Array müssen die Sollwerte in richtiger Reihenfolge gespeichert sein. Anschließend wird für jede einzelne Registerstelle die *fra_test_check* Funktion aufgerufen. Am Ende wird die Gesamtanzahl der Fehler zurückgegeben, welche im Test ausgewertet wird.

Im Unittest wird nun die virtuelle Hardware angelegt und geöffnet, anschließend werden modulspezifische Funktionen aufgerufen und direkt danach der Check durchgeführt (siehe Abbildung 4.1). Dadurch kann die

Funktionalität der einzelnen Module überprüft und im Fehlerfall direkt gehandelt werden. Vor allem bei Änderungen an der FRA Bibliothek können so vor der Inbetriebnahme auf der Kamera Fehler gefunden und ausgebesert werden.

4.2 Tools für die Entwicklungsumgebung

Unabhängig von der Funktionstests des vorherigen Kapitels sollten dem Entwickler kleine Programme zur Verfügung gestellt werden, mit denen man für Testprogramme oder zur Fehlersuche im laufenden Betrieb ein Gerät anlegen oder das Logging aktivieren kann.

Besonders für die Kalibrierung der Kamera ist es notwendig ein Gerät über die Kommandozeile anzulegen, da es hier keinen Codeteil gibt, in welchem das Anlegen integriert werden könnte. Dadurch das die benötigten Geräte vor dem Starten der Kalibriersoftware angelegt werden, können in der Software über die Funktionen der FRA Middleware und Bibliothek die Geräte geöffnet und auf diese zugegriffen werden.

Beim Ausführen des Programms *fra_create_device* werden in der Kommandozeile über Optionen die benötigten Parameter übergeben. Hier gibt es, neben dem Namen, dem Typ, der Adresse und der Größe auch, die Möglichkeit eine Hilfe auszugeben, in welcher genauer erläutert wird, wie das Programm zu nutzen ist. Nach dem Ausführen des Programms werden die programminternen Parameter mithilfe der Optionen gefüllt und nach der Überprüfung auf Dateninkonsistenz wird das entsprechende Gerät angelegt. Danach ist dieses Gerät in Linux unter *dev/fra/* zu finden und weitere Programme oder Testtools können darauf zugreifen.

Ein weiteres nützliches Tool für die Entwicklungsarbeit ist das *fra_set_logging*. Da das Logging standardmäßig deaktiviert ist, muss dieses bei Bedarf aktiviert werden. Damit der Entwickler nicht im entsprechenden Code die Aktivierung vornehmen und zeitaufwendig neu kompilieren muss, wird durch das Testtool ein entsprechendes Werkzeug bereitgestellt. Durch das Programm kann im laufenden Betrieb der Kamera die Protokollierung eines Geräts aktiviert oder deaktiviert werden.

Hier werden beim Starten des Programms auf der Kommandozeile entsprechend den Optionen der Name des Geräts und ein Setparameter übergeben. Über den Setparameter wird entschieden, ob das Logging aktiviert (*set=1*) oder deaktiviert (*set=0*) werden soll. Anschließend wird das Gerät über den Namen geöffnet und über das entsprechende IOCTL wird das Logging gesetzt. Nach dem Schließen des Geräts wird das

Programm beendet.

Beide Tools sind für den Einsatz auf der Entwicklungsumgebung gedacht und sollen Entwicklern und Testern vor allem die Fehlersuche auf der Kamera erleichtern. Bei Bedarf können die Programme einfach erweitert werden oder weitere Tools in Anlehnung an die Codestruktur geschrieben werden.

5

Fazit und Ausblick

In diesem Kapitel soll ein Fazit über das in der Arbeit behandelte FRA gezogen werden und zusätzlich ein Ausblick auf die weitere Vorgehensweise zur Einbindung in die Software und notwendige Erweiterungen gegeben werden.

5.1 Zusammenfassung der Ergebnisse

Ein wichtiger Punkt für das FRA war die Abstraktion der Hardware. Durch die direkten Zugriffe von der Kamerasoftware auf den FPGA ist die Fehlersuche recht zeitaufwendig gewesen. Aufgrund des neuen Frameworks wird nicht mehr über direkte Adressen auf die Hardware zugegriffen. Dadurch ist es einfacher nachzuvollziehen, aus welchem Prozess die Funktionsaufrufe kommen und kann so im Fehlerfall auch einfach protokolliert werden.

Des weiteren wird durch die Einteilung in drei verschiedene Ebenen im Userspace eine einfache Möglichkeit gegeben um Tests zu implementieren. Durch das Testbackend und die FRA Middleware können die modulspezifischen Funktionen identisch wie in der Kamerasoftware aufgerufen werden und somit überprüft werden, ob die FRA Middleware und Bibliothek richtig funktioniert. So können durch die Unittests bei jeder Neuerung und Änderung in der Bibliothek oder Middleware frühzeitig Fehler gefunden und behoben werden.

Durch das FRA ist der letzte Schritt in einer Umstrukturierung der Software gemacht worden. Die letzten Jahre wurde die Software bereits funktional durch die Implementierung des Geo Framework abstrahiert. Aufgrund des Zusammenspiels der beiden Abstraktionsebenen ist es nun möglich ohne

tieferen Kenntnisse von FPGA und Bildkette die Kamerasoftware durch unterschiedliche Entwickler zu Pflegen und zu Erweitern.

5.2 Ausblick

Damit das FRA vollständig in der Software eingebunden ist, müssen noch weitere Schritte gemacht werden. Zum Einen müssen die restlichen Module der Bildkette samt ihrer spezifischen Funktionen in die FRA Bibliothek umgezogen werden und zum Anderen haben ein paar Module in FPGA Speicherbereiche um die Bilder zwischen zu speichern, auch diese müssen für die komplette Umstellung im FRA abgebildet werden. Der Arbeitsaufwand für den kompletten Umzug der Software auf das Framework wird auf ungefähr ein halbes Jahr geschätzt.

Auch die Abhängigkeit von händisch eingetragenen Adressen ist noch immer vorhanden (siehe Kapitel 1.3). Diese soll in Zukunft durch ein generisches Modell ersetzt werden und somit eine komplette Unabhängigkeit von den FPGA Adressen in der Software schaffen. Dies ist vor allem in Anbetracht der Wartbarkeit der Software ein wichtiges Thema.

Wie in Kapitel 3.4.2 erwähnt, wird in der aktuellen Implementierung des FRA die Transaktionsidentifikation durch alle Funktionsaufrufe durchgeführt. Bei bestimmten Teilen der Bildkette hängen die Einstellungen der Module stark voneinander ab, aus diesem Grund ist es sinnvoll die Zugriffe auf die Module zu gruppieren. Dadurch können die FPGA Zugriffe direkt hintereinander aufgerufen werden und so Bildfehler vermieden werden. Die Erweiterung ist relativ zeitaufwendig und auch mit Änderungen im Kernel verbunden.

Der aktuelle Stand des FRA ist bereits funktional in der Software integriert und muss noch über die obigen Teile erweitert werden, damit die Umstrukturierung komplett abgeschlossen ist.

Anhang

A.1 Literaturverzeichnis

- [1] ARRI. *About ARRI*. Webseite. Besucht 2019-09-06. URL: <https://www.arri.com/en/company/about-arri>.
- [2] ARRI. "AMIRA: mulit-purpose tool". In: *ARRI News* (Sept. 2015), 26f.
- [3] ARRI. *Bild der ARRI AMIRA*. Webseite. Besucht 2019-11-04. URL: <https://www.arri.com/resource/blob/33916/909908b1643addb99036f132d6b3582c/amira-product-image-data.jpg>.
- [4] Alexandre Belloni. *Supporting multi-function devices in the Linux kernel*. Präsentation. Besucht 2019-10-14. URL: <https://elinux.org/images/9/9a/Belloni-mfd-regmap-syscon.pdf>.
- [5] Jonathan Corbet. *The platform device API*. Webseite. Besucht 2019-09-13. URL: <https://lwn.net/Articles/448499/>.
- [6] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers: Where the Kernel Meets the Hardware*. O'Reilly Media, Inc., 2005. URL: <https://free-electrons.com/doc/books/ldd3.pdf>.
- [7] ARRI Media. *ARRI im SAUERKRAUTKOMA*. Webseite. Besucht 2019-09-25. URL: <https://www.arri-media.de/global/detail-news/arri-im-sauerkrautkoma/>.
- [8] Johannes Plötnner and Steffen Wendzel. *Linux: das umfassende Handbuch*; Galileo Press, 2012.
- [9] Joachim Schröder, Tilo Gockel, and Rüdiger Dillmann. *Embedded Linux: Das Praxisbuch*. Springer-Verlag, 2009.
- [10] SMPTE. *ST 292-1:2012 - SMPTE Standard - 1.5 Gb/s Signal/Data Serial Interface*. Webseite. Besucht 2019-12-03. URL: <https://ieeexplore.ieee.org/servlet/opac?punumber=7291768>.
- [11] Unbekannt. *The Grand Tour, Technical Specifications*. Webseite. Besucht 2019-09-25. URL: https://www.imdb.com/title/tt5712554/technical?ref_=tt_dt_spec.

- [12] Unbekannt. *The Ivory Game, Technical Specifications*. Webseite. Besucht 2019-09-25. URL: https://www.imdb.com/title/tt5952266/technical?ref_=tt_dt_spec.
- [13] Verschiedene. *Linux / Documentation*. Webseite. Besucht 2019-10-28, v5.3.7. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation?h=v5.3.7>.
- [14] Verschiedene. *Linuxquellcode / driver*. Webseite. Besucht 2019-10-24, v5.3.7. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/drivers?h=v5.3.7>.
- [15] Verschiedene. *Linuxquellcode / include*. Webseite. Besucht 2019-10-24, v5.3.7. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/include?h=v5.3.7>.

A.2 Abbildungsverzeichnis

1.1	ARRI AMIRA	2
1.2	Schematische Bildkette	3
1.3	Vereinfachte Darstellung des Zugriffs auf den FPGA in der aktuellen und der neuen Implementierung	4
3.1	Darstellung des Aufbaus des FRA	19
4.1	Beispielhaftes Sequenzdiagramm zum Unittest	26

A.3 Tabellenverzeichnis

3.1	Namensgebung der IOCTLs für den Registerzugriff	15
-----	---	----

A.4 Liste der Codeausschnitte

2.1	Funktionsdeklaration des IOCTL in der file_operations Struktur [9, S. 249f.]	7
2.2	Vereinfachte Funktionsdeklaration aus [15, linux/uaccess.h, Zeile 140ff.]	8
2.3	platform_driver Struktur in [15, linux/platform_device.h, Zeile 184ff.]	9
2.4	Struktur resource in [15, linux/ioport.h, Zeile 20ff.]	10

2.5	Struktur mfd_cell in [15, linux/mfd/core.h, Zeile 29ff.] . . .	10
2.6	Funktionsdeklaration in [15, linux/mfd/core.h, Zeile 130ff.]	11
3.1	Struktur des Treibers	13
3.2	Auszug aus der Struktur einer Instanz	14
3.3	Struktur zum Initialisieren des Geräts	16
3.4	Struktur zum Ablegen des Geräts	18
3.5	Funktion zum Setzen der Xbar	20
3.6	Struktur zum Abspeichern wichtiger Parameter	21
3.7	Makro zum Überprüfen des Handles	22
3.8	Funktion zum Setzen eines Registers	23
3.9	Funktion im Kernelbackend zum Setzen eines Registers . . .	24
4.1	Struktur zum Abbilden der virtuellen Hardware	25