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 kernelmodul

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 31. Oktober 2019 _____

Unterschrift

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

München, den 31. Oktober 2019 _____

Unterschrift

Kurzfassung

In dem vorliegenden Bericht geht es um die Verbesserung einer Kamerasoftware. Es wird hierzu eine Abstraktionsebene vorgestellt, die aus mehreren Modulen besteht. Um das Verständnis zu erleichtern wird außerdem eine kurze Einführung zur Bildkette gegeben und anschließend auf ausgewählte Module eingegangen. Durch die Abstraktionsebene wird die Übersichtlichkeit und Erweiterung der Software in Zukunft erleichtert werden.

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

Abstract

This report handle the improvement of an camera software. For this, an abstraction level will be introduce, which consists out of a few modules. At first there is a short intorduction to the image processing chain to make the understanding easier and then go into details of selected modules. Because of the abstraction level the clarity and extensions of the software will be easier in future.

This report targets readers having an electric engineering background.

Inhaltsverzeichnis

1 Abkürzungsverzeichnis **11**

2 Einleitung **12**

2.1 Arnold & Richter Cine Technik GmbH & Co. Betriebs KG . . 12

2.2 Hinführung zum Thema 12

3 Grundlagen Linux **14**

3.1 Linux - Kernel und Userspace 14

3.2 IO Control 15

3.3 Datenaustausch zwischen Kernel und Userspace 15

3.4 Plattformtreiber 16

3.4.1 Major- und Minornummern 17

3.5 Multifunction Device 17

4 Fachbezogene Grundlagen **20**

4.1 Kontext 20

4.1.1 Kamera 20

4.1.2 Bildkette 21

4.1.3 Funktionsweise 22

4.1.4 Problematik 22

4.2 Konzept 23

5 FPGA Resource Abstraction **24**
5.1 Generischer Plattformtreiber 24
5.2 Konzeptionierung der IO Controls 26
5.3 Implementierung im Kernel 27
5.3.1 Anlegen der Geräte 28
5.3.2 Öffnen und Schließen der Geräte 29
5.4 Implementierung im Userspace 30
5.4.1 FPGA Resource Abstraction (FRA) Bibliothek 30
5.4.2 Kernelbackend 32
5.5 Einbindung ins Geometrie Framework 33

6 Testbackend **34**

7 Ausblick **35**

1

Abkürzungsverzeichnis

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

FPGA Field Programmable Gate Array

FRA FPGA Resource Abstraction

Geometrie 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 [1]

Xbar Crossbar

[1] *https://ieeexplore.ieee.org/document/7291770*

2

Einleitung

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

Die Bachelorarbeit fand bei der Firma Arnold & Richter Cine Technik GmbH & Co. Betriebs KG (ARRI) statt. Nach der Gründung 1917 liegt der Hauptsitz von ARRI immer noch in München. Mittlerweile sind weltweit um die 1500 Mitarbeiter angestellt und die Firma ist einer der führenden Hersteller und Lieferanten in der Film- und Fernsehindustrie.

Die ARRI Gruppe ist in fünf Geschäftsbereiche eingeteilt: Kamerasysteme, Licht, Postproduktion, Verleihservice und Operationskamerasystem für die Medizin. [1]

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

2.2 Hinführung zum Thema

Damit die Kamera einwandfrei funktioniert müssen Firmware und Software zusammenspielen. Im Geometrie Framework (Geometrie Framework) werden die verschiedenen Module aus dem Field Programmable Gate Array (FPGA) abgebildet und entsprechend der Kameraeinstellungen die Bildgrößen berechnet.

In der Software werden dann in verschiedenen Funktionen die einzelnen Register im FPGA entsprechend gesetzt. Durch die aktuelle Implementierung können keine Module gleichzeitig im FPGA geupdatet werden.

Damit eine modulare Ansteuerung möglich wird, soll eine weitere Abstraktionsebene erstellt werden. Dieses sogenannte FRA soll die einzelnen Module im Kernel darstellen und entsprechend aus dem Userspace über IO Control (IOCTL) angesprochen werden können. Zusätzlich werden dann die direkten Hardwarezugriffe aus dem Hauptcode eliminiert.

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

3

Grundlagen Linux

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

3.1 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 Betriebssystem.

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. [7, S. 46]

Der Kernel hat verschiedene Aufgaben. Unter anderem ist er für die Prozess- und die Speicherverwaltung sowie das Gerätemanagement zuständig. [8, 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. [7, S. 124] Die Brücke zwischen der Hardware und dem Benutzer stellt somit der Kernel da.

3.2 IO Control

Um ein zuverlässig arbeitendes Betriebssystem zu haben, muss der Speicherbereich von Kernel und Userspace getrennt sein. [8, S. 232] Damit entsteht die Notwenigkeit zwischen Userspace und Kernel 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. [8, 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 3.1: *Funktionsdeklaration des IOCTL in der file_operations Struktur [13, linux/fs.h]*

Über die *node* wird der Dateideskriptor und über *instanz* ein Zeiger auf die Treiberinstanz an den Funktionszeiger übergeben. Das Kommando wird durch eine Nummer widergespiegelt 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. [5, 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.

3.3 Datenaustausch zwischen Kernel und Userspace

Im vorausgegangen Kapitel wurde schon die Notwendigkeit von getrennten Speicherbereichen des Kernels und des Userspaces erläutert. Dadurch 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 3.2: *Funktionsdeklaration in asm/uaccess.h*

Die Hauptaufgabe beider Funktionen ist das Kopieren von Daten, 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. [8, S. 250f]

3.4 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. [8, S. 231ff]
Damit die Treiber richtig funktionieren werden müssen gibt es einige Bestandteile, die in jedem Modul wiederzufinden sind. Standardmäßig wird die Registrierung von Treiber und Device an unterschiedlichen Teilen des Programms ausgeführt. [4]
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. [13, linux/module.h]

```
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 3.3: *platform_driver Struktur in [13, linux/platform_driver.h]*

Beim Laden des Treibers wird die eine *platform_driver* Struktur übergeben. In dieser Struktur sind Zeiger auf die Funktionen gespeichert, die beim Erzeugen oder Löschen einer Instanz gebraucht 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.[4]

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 Device vorhanden. [4]

3.4.1 Major- und Minornummern

Jedes Device hat eine Major- und Minornummer mit der sie angelegt werden. Die Majornummer kennzeichnet hier üblicherweise dem Device zugehörigen Treiber analog dazu wird die Minornummer vom Kernel genutzt um auf das exakte Device zu referenzieren. [5, S. 43f]

3.5 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. [3]

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

```
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 3.4: *Struktur resource in [13, linux/ioport.h, Zeile 20ff.]*

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 andere optionale Einstellungen festgelegt. In dem letzten drei Parametern können die abhängige Ressource 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
       details
12 */
13  const char    *of_compatible;
14 [...]
15 /*
16 * These resources can be specified relative to the parent device.
17 * For accessing hardware you should use resources from the
       platform dev
18 */
19  int     num_resources;
20  const struct resource *resources;
21 [...]
22 };
```

Codeausschnitt 3.5: *Struktur mfd_cell in [13, 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. [11, 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 3.6: *Funktionsdeklaration in [13, 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 Parame-

ter *cells* auch ein Array beinhalten kann. Die anderen Übergabeparameter werden nicht näher betrachtet, da sie im folgenden nicht benötigt werden. [12, mfd/mfd-core.h]
In der Funktion ist zusätzlich implementiert, dass beim Entfernen des übergeordneten Gerät alle Untergeräte automatisch aufgelöst werden. [12, mfd/mfd-core.h, Zeile 356f.]

4

Fachbezogene Grundlagen

Zunächst sollen die Entwicklungsumgebung und die aktuelle Implementation in der Software näher betrachtet werden. Im Anschluss wird noch das erarbeitete Konzept des FRA vorgestellt.

4.1 Kontext

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

4.1.1 Kamera

Um die Funktionalität des FRA festzustellen und grobe Fehler rechtzeitig zu erkennen, ist das regelmäßige Testen auf der Zielplattform unerlässlich. 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 ARRI und somit ohne Probleme verfügbar.

Abbildung 4.1: *ARRI AMIRA* [2]

Die AMIRA ist eine vielseitige Kamera, die für eine Einmannbedienung ausgelegt ist. Zusätzlich ist sie mit einem Audioboard ausgestattet und aus diesem Grund bei Dokumentationsfilmen und der elektronischen Berichtserstattung gerne verwendet. Zum Beispiel wird die ARRI AMIRA bei Sportveranstaltungen der NFL in Amerika eingesetzt. [2]
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“ [6], die Netflixserie „The Ivory Game“ [10] oder auch das Fernsehmagazin „The Grand Tour“ [9] zu nennen.

4.1.2 Bildkette

Unter einer Bildkette versteht man die Verarbeitungskette der Bilder vom Eingang - dem Sensorbild bis zu zum Ausgang, in unserem Fall die Aufnahme (REC) und das Serial Digital Interface (SDI).

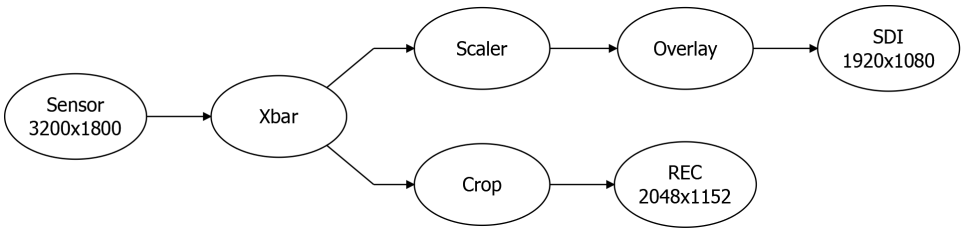

Abbildung 4.2: *Schematische Bildkette*

In der Abbildung 4.2 sieht man eine schematische Bildkette, die in dieser Arbeit zur Veranschaulichung weiter detailliert wird. Von der Quelle bis zur jeweiligen Senke läuft das Bild durch verschiedene Module, die für die Anpassung des Bilds sorgen.
Direkt nach dem Sensor geht das Bild durch eine Crossbar (Xbar), hier wird das identische Bild in zwei Bildpfade weitergeführt. Für die Aufnahme wird das Bild im Crop zugeschnitten, somit wird nur ein bestimmter Bildausschnitt aufgezeichnet. In dem anderen Bildpfad wird, mithilfe des Scalers, das Bild kleiner skaliert. Nachdem eine grafische Benutzeroberfläche hinzugefügt worden ist, wird das Bild am SDI ausgegeben. Hier ist durch die Skalierung das komplette Sensorbild einschließlich der eingefügten Oberfläche zu sehen.
In dieser Arbeit wird nur das FPGA-Modul Xbar softwareseitig implementiert, da weitere Module sonst den Umfang der Arbeit überschreitet.

[2] *https://www.arri.com/resource/blob/33916/909908b1643addb99036f132d6b3582c/amira-product-image-data.jpg*

4.1.3 Funktionsweise

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 werden 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 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.
Nach der Änderung der Größen wird in dem entsprechenden Modul ein Flag gesetzt, welches später dafür sorgt, dass auch das Modul im FPGA aktualisiert wird.
Beim Updaten des Modules wird an den entsprechenden Offset im FPGA die übergebenen Einstellungen geschrieben und gleichzeitig das gesetzte Flag wieder gelöscht.
Damit der Zugriff auf den FPGA möglich ist, wird dieser beim Starten der Software über ein IOCTL initialisiert und anschließend kann über eine Variable im Shared Memory in allen Prozesseen darauf zugegriffen werden.

4.1.4 Problematik

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

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 natürlich weiter, 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 Modules 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 dann passieren, dass über ein Register hinweg geschrieben wird. Auch dann kommt es zum Fehlerfall in der Bildkette, da andere Einstellungen überschrieben werden.

4.2 Konzept

Die Probleme in der aktuellen Implementierung (siehe Kapitel 4.1.4) sollen durch ein neues Framework behoben werden, aber die Zugriffe der Software auf den FPGA sollen auch übersichtlicher und wartbarer gestaltet werden.

In der momentanen Implementierung benötigen die Zugriffe auf den FPGA immer die Adressen der Module dort. Dadurch wird die Fehlersuche in diesen Teilen der Software extrem kompliziert und aufwendig. Durch das FRA sollen die Firmware Module im Kernel als Gerät angelegt werden und in der Software wird auf die FPGA Module über die entsprechenden Dateideskriptoren zugegriffen.

Die Geräte werden mit der Adresse, dem Name, dem Type und der Größe des dahinterstehenden Modul angelegt. Dieser Teil soll generisch generiert werden, aber aufgrund von verschiedenen Abhängigkeiten überschreitet es den Umfang der Arbeit. Deswegen werden die Geräte beim Initialisieren des Bildpfads mit den entsprechenden Parametern angelegt.

Das Öffnen der Dateideskriptoren erfolgt in der Software über den Namen und wird in einem Handle gespeichert. Damit ist der Zugriff ohne Adresse gewährleistet. Jeder Prozess in der Software muss seinen eigenen Dateideskriptor öffnen und verwaltet somit ein eigenes Handle. Im Kernel wird gewährleistet, dass nur eine maximale Anzahl von Deskriptoren geöffnet werden kann. Zusätzlich soll implementiert werden, dass es verschiedene Applikationstypen gibt. Damit kann immer nur eine echte Applikation Zugriff bekommen, allerdings sollen Debugtools immer die Möglichkeit haben zu lesen und teilweise auch zusätzlich Schreibrechte.

Der Zugriff auf die Module erfolgt über ein IOCTL des Treibers. Da beim Anlegen der Geräte eine Größe festgelegt wird, soll bei allen Zugriffen auf die Register überprüft werden, ob diese innerhalb der angegebenen Modulgröße liegen. Damit ist es nicht mehr möglich über ein Register hinweg zu schreiben.

5

FPGA Resource Abstraction

In diesem Kapitel soll auf die Vorgehensweise und die Implementierung des in 2.2 vorgestellte FRA eingegangen werden. schwerpunkte.... In dieser Arbeit wird nur die Crossbar softwareseitig implementiert, da es sonst den Umfang der Arbeit übersteigt.

5.1 Generischer Plattformtreiber

Um eine einwandfreie Kommunikation zwischen den Firmware Modulen im FPGA und dem Kernel zu gewährleisten, muss ein generischer Treiber erstellt werden.
Wie in dem Kapitel 3.4 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.

```
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 5.1: *Struktur des Treibers*

Im Treiber 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 3.4.1) 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 wird später eine freie Nummer ausgewählt, die bei jedem Gerät 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 unterbrechungsfreien Vorgang, sodass keine Nummer doppelt vergeben werden kann.
Am Ende der initialen Funktion wird der Plattformtreiber mit der *platfrom_driver* Struktur (siehe Codebeispiel 3.3) 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 erstelle Klasse zerstört.

Beim Anlegen einer Instanz vom Treiber wird die *.probe* Funktion aufgerufen und zusätzlich wird der Modultyp und Name in einer *arri_fra_mod_config* Struktur zur Verfügung gestellt.

```
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   %atomic_t use_count;
15
16   #define AFM_NOLOGGING    ((uint32_t)0U)
17   #define AFM_LOGGING      ((uint32_t)1U)
18   uint32_t has_logging;
19 };
```

Codeausschnitt 5.2: *Auszug aus der Struktur einer Instanz*

In der initialen Funktion der Instanz wird diese Struktur genutzt um spezifische Einstellungen für jede einzelne Instanz festgelegt und in einer entsprechenden Datenstruktur (siehe Codebeispiel 5.2) gespeichert.

Der einzige Übergabeparameter der Probefunktion ist die Struktur des *platform_device*, diese wird in dem Zeiger **pdev* gespeichert. Auch die, beim Laden des Treibers, angelegte Klasse wird ohne weitere Änderungen unter **class* abgelegt.
Aus der Liste der Minornummern (siehe Codebeispiel 5.1) wird nach dem Sperren des zugehörigen Spinlocks eine freie Nummer ausgewählt und als *minor_id*, sowie auf verwendet gesetzt.
Anschließend wird mit Hilfe der Majornummer und der Minornummer ein Device erstellt und als *cdev* gespeichert.
Damit das Öffnen der Devices auf eine bestimmte Anzahl limitiert ist, wird beim Initialisieren der Instanz ein Array der *afm_file* Struktur angelegt. Gleichzeitig wird der Speicherplatz statisch zur Kompilierzeit reserviert. Durch den Spinlock *open_spinlock* wird später dafür gesorgt, dass die Auswahl nicht unterbrochen und somit verfälscht wird.

5.2 Konzeptionierung der IO Controls

In dem Abschnitt soll auf die Konzeptionierung der IOCTLs eingegangen werden. Wie in Kapitel 3.2 erläutert werden IOCTLs 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.

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 vorhanden ist. Wie in Kapitel 4.2 erläutert, soll die Unterscheidung zwischen Standardapplikation und Debugtool möglich sein. Aus diesem Grund wird bei der Anmeldung zusätzlich zur PID und dem Prozessnamen auch der Typ der Applikation übergeben. Ohne die Ausführung von dem *ARRI_FRA_MOD_HELLO* IOCTL bleibt der weitere Zugriff auf das Gerät über IOCTLs verwehrt.

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 diese. In dem Zusammenhang werden zwei IOCTLs angelegt. Dabei ist *ARRI_FRA_MOD_LOGGING* zum Aktivieren der Protokollierung zuständig und analog wird diese durch *ARRI_FRA_MOD_NOLOGGING* deaktiviert.

Beim Anlegen des Geräts werden der Gerätetyp und der Name abgelegt. Im Userspace soll nach dem Öffnen des Geräts auf diese Informationen

zugegriffen werden. Dafür wird ein eigenes IOCTL benötigt, welches den Typ und Namen aus der Instanzstruktur (siehe Codebeispiel 5.2) 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.
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 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 gesetzt. Dadurch ist es nicht notwendig erst über ein IOCTL das entsprechende Register auszulesen, anschließend im Userspace zu verändern und dann, wieder über ein IOCTL, in das gleiche Register zu setzen. Der zweite Sonderfall benötigt andere Übergabeparameter als das einfache Setzen. Zusätzlich zu der Registernummer wird hier eine Registeranzahl gebraucht. Des Weiteren muss der Inhalt der 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 den 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 leeren Arrays mit der richtigen Größe der Speicherbereich zu Verfügung gestellt werden.

Dadurch sind die wichtigsten IOCTLs erläutert, die notwendig sind um die Grundfunktionen des Geräte abzudecken und zusätzlich eine triviale Debug Möglichkeit bieten.

5.3 Implementierung im Kernel

Im Kernel gibt es zwei verschiedene Stellen, an welchen der Code implementiert ist. Zum einen muss der FPGA Treiber entsprechend erweitert

werden, damit die Geräte unter Linux angelegt werden und zum anderen der FRA Treiber, um die angelegten Geräte zu Öffnen bzw. zu Schließen.

Zur besseren Übersichtlichkeit wurde sich für zwei Nameskonventionen 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.

5.3.1 Anlegen der Geräte

Der Zugriff der Software auf die FPGA-Module soll über Geräte stattfinden. Durch die unterschiedlichen Funktionalitäten der Module, wie in Kapitel 4.1.2 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 5.3: *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 am Ende der Funktion, die durch das IOCTL aufgerufen wird, das MFD angelegt.
In der *resource* Struktur (siehe Codebeispiel 3.4) wird der Speicherbereich des FPGA-Modules abgebildet. Für die Variable *start* wird auf den bereits allokierten FPGA Bereich noch der im *arri_fra_mod_init* übergebene *offset* aufaddiert. Entsprechend ist wird für den *end* Wert noch die *size* addiert und zusätzlich eins abgezogen, da alles null basiert ist. 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 Codebeispiel 3.5) 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 aktuelle Zeiger der 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 Gerätename und -typ an den Treiber übergeben.
Über die im Codebeispiel 3.6 aufgezeigt 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.

5.3.2 Öffnen und Schließen der Geräte

Um später im Userspace auf die Geräte zuzugreifen und um initiale Einstellungen vorzunehmen, muss eine *open* Methode implementiert werden. Im Umkehrschluss werden die Einstellungen durch die *release* Funktion rückgängig gemacht. [5, Seite 58f.]

Die Anzahl von offenen Geräten ist durch die Speicherallokierung in der Struktur *afm_device* (siehe Codebeispiel 5.2) auf *AFM_FILE_MAX* (hier: 4) begrenzt. Die geöffnete Instanz eines Geräts wird in der Struktur *afm_file* abgelegt.

```
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[ARRI_FRA_MOD_MAX_NAME];
8    uint32_t type;
9    uint32_t pid;
10 #define AFM_UNLOCK   ((uint32_t)0U)
11 #define AFM_LOCK     ((uint32_t)1U)
12   uint32_t has_lock;
13   struct file *file;
14 };
```

Codeausschnitt 5.4: *Struktur zum Ablegen des Geräts*

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 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 zurückgehen. 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. *has_lock* wird beim Öffnen des Geräts initial freigegeben. Die anderen Parameter

beschreiben den Prozess, welcher das Gerät öffnet und werden durch ein IOCTL entsprechend beschrieben.

Analog werden 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 überschrieben werden.

5.4 Implementierung im Userspace

Der Zugriff auf die FRA Module im Userspace ist in zwei Ebenen gekapselt. Die untere Stufe besteht aus verschiedenen backendspezifischen Funktionen. Entsprechend sind in der anderen Ebene erweiterte Wrapper Funktionen zu finden. Hier werden verschiedene Überprüfungen durchgeführt, damit kein fehlerhafter Zugriff stattfindet. Des Weiteren wird im Wrapper entsprechend dem Backendtyp, die unterliegende Funktion ausgewählt. Damit es den Umfang der Arbeit nicht übersteigt, wird nur das Kernelbackend implementiert. Dies wird benötigt um über die, im Kernel implementierten, IOCTLs auf die FPGA Module zuzugreifen.
Durch die gekapselten Ebenen ist eine Erweiterung um neue Backendtypen einfach gestaltet. Insbesondere über ein Testbackend sollen Unittest zu den verschiedenen Modulen möglich sein, damit können Fehler frühzeitig erkannt, eingegrenzt und behoben werden. Durch die Kapselung ist die Wartbarkeit erhöht worden. Dadurch müssen bei Änderungen im Kernelzugriff, nur im Backendcode die entsprechenden Stellen geändert werden.

5.4.1 FRA Bibliothek

Neben den Wrapper Funktionen wird in der FRA Bibliothek auch eine Struktur zum Abspeichern 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 5.5: *Struktur zum Abspeichern wichtiger Parameter*

Der erste und wichtigste Parameter in der Struktur ist der File Descriptor. *dev* Hierüber kann nach dem Öffnen des Geräts weiterhin auf dieses

zugegriffen werden. Die restlichen Parameter werden beim initiieren des Geräts gesetzt. *dev_type* und *dev_name* geben den Modultyp und den Namen an. Für spätere Überprüfungen wird gibt es für jeden Modultyp noch ein Define, 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 5.2), wird dies über einen Funktionsaufruf abgedeckt.
Der initiialen Funktion wird auch ein Backendtyp übergeben und anhand von diesem die Funktionsstrukur im *fra_handle* gesetzt. Anschließend wird das Gerät geöffnet und danach meldet sich der Prozess direkt an. Hierzu werden jeweils die entsprechenden Wrapper Funktionen aufgerufen um die Funktionalität bei allen Backendtypen zu garantieren. Zum Bestimmen von *dev_type* und *dev_name* wird auch über eine Wrapper Funktion, diese Parameter vom Gerät geholt und entsprechend im *fra_handle* gesetzt. Durch das Iterieren über eine *fra_type_list* mit Name, ID und Größe und dem Vergleichen der Strings in Name und dem *dev_name* wird die Variable *type_id* gesetzt.

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 konzeptioniertes IOCTL im Kernel eine Wrapperfunktion, lediglich das Logging ist in einer Funktion zusammengefasst.
Zu Beginn einer jeden Methode wird 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 5.6: *Makro zum Überprüfen des Handles*

Ohne ein gültiges Handle machen alle nachfolgenden Aufrufe keinen Sinn, da alles basierend auf diesem Handle erfolgt. Aus diesem Grund wird dies in Zeile 2 als erstes überprüft. Über den File Descriptor kann

man herausfinden, ob das Gerät schon geöffnet wird. Hier wird beim initiieren der Struktur *dev* auf -1 gesetzt. 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, intern 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 ist in der Signatur implementiert, damit bei einer späteren Erweiterung nicht alle Aufrufe geändert werden müssen. Aktuell wird sie lediglich bis zum Ende durchgereicht und nicht weiter beachtet.

```
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 5.7: *Funktion zum Setzen eines Registers*

Das Makro in Zeile 9 ist im Codeausschnitt 5.6 näher ausgeführt. Durch die if - Bedingung in Zeile 11 wird überprüft, ob für das ausgewählte Backend die entsprechende Funktion implementiert ist. Damit 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 Bibliothek entspricht entweder einem entsprechenden Fehlerwert bzw. dem Rückgabewert der Backendfunktion.

5.4.2 Kernelbackend

Da die Wrapper lediglich alle benötigte Übergabeparameter an die Backendfunktionen weiterreichen, haben diese eine identische Funktionssignatur. 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.
Hier 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.

```
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 5.8: *Funktion im Kernelbackend zum Setzen eines Registers*

5.5 Einbindung ins Geometrie Framework

Um das FRA vollständig in der Kamerasoftware zum Implementieren müssen die Geräte angelegt, geöffnet, geupdatet und auch wieder geschlossen werden.

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 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.

Beim initialen Hardwareupdate durch das Geometrie Framework werden für alle vorhandenen, in diesem Framework abgebildeten Module ein Gerät geöffnet und entsprechend dem Modulindex in ein *fra_handle* abgelegt.

Damit die Module geupdatet werden können, müssen die vorhandenen Funktionen, welche direkt an die Offsets im FPGA schreiben geändert werden. Im Rahmen dieser Arbeit wird lediglich die Xbar näher betrachtet, aber andere Module müssen analog abgeändert werden.

6

Testbackend

7

Ausblick

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] 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*.

[4] Jonathan Corbet. *The platform device API*. Webseite. Besucht 2019-09-13. URL: *https://lwn.net/Articles/448499/*.

[5] 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*.

[6] ARRI Media. *ARRI im SAUERKRAUTKOMA*. Webseite. Besucht 2019-09-25. URL: *https://www.arrimedia.de/global/detail-news/arri-im-sauerkrautkoma/*.

[7] Johannes Plötner and Steffen Wendzel. *Linux: das umfassende Handbuch;* Galileo Press, 2012.

[8] Joachim Schröder, Tilo Gockel, and Rüdiger Dillmann. *Embedded Linux: Das Praxisbuch*. Springer-Verlag, 2009.

[9] Unbekannt. *The Grand Tour, Technical Specifications*. Webseite. Besucht 2019-09-25. URL: *https://www.imdb.com/title/tt5712554/technical?ref_=tt_dt_spec*.

[10] Unbekannt. *The Ivory Game, Technical Specifications*. Webseite. Besucht 2019-09-25. URL: *https://www.imdb.com/title/tt5952266/technical?ref_=tt_dt_spec*.

[11] 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*.

[12] 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*.

[13] 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*.