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:	1234

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 camera software

abgegeben von Maren Konrad

Bearbeitungsbeginn:	12.08.2019
Abgabetermin:	12.02.2020
lfd. Nr. gemäß Belegschein:	1234
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. Oktober 2019 _____

Unterschrift

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

München, den 3. 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

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

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 FPGA Resource Abstraction (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

Theoretische Grundlagen

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. [6, S. 46]

Der Kernel hat verschiedene Aufgaben. Unter anderem ist er für die Prozess- und die Speicherverwaltung sowie das Gerätemanagement zuständig. [7, 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. [6, 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. [7, 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. [7, 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);
```

Codebeispiel 3.1: *Funktionsdeklaration des IOCTL in der file_operations Struktur*

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. [4, 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);
```

Codebeispiel 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. [7, 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. [7, 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. [3]
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. (https://elixir.bootlin.com/linux/v4.15.9/source/include/linux/module.h ab Zeile 77)

```
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 };
```

Codebeispiel 3.3: *Struktur in linux/platform_device.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.[3]

Beim Anlegen der Instanz kann ein Zeiger auf ein struct übergeben werden, in welchem Daten übergeben werden. In der probe Funktion sind somit spezielle Daten für ein entsprechendes Device vorhanden. [3]

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. [4, S. 43f]

3.5 Multifunction Device

Normalerweise wird lediglich ein Device angelegt, ohne weitere Unterteilungen zu machen. Es ist allerdings möglich unter einem Parentdevice noch weitere Childdevices zu registrieren. Diese Art wird dann Multifunction Device (MFD) genannt.

```
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);
```

Codebeispiel 3.4: *Funktionsdeklaration in mfd/core.h*

Die obere Funktion legt Childdevices an und beim Entfernen des Parentdevice werden alle Childdevices automatisch aufgelöst.
Als ersten Parameter wird ein Zeiger auf das Parentdevice übergeben. *mfd_cell* ist eine Struktur, welche das Childdevice näher beschreibt und über die *n_devs* wird die Anzahl der zu registrierenden Childdevices übergeben. Die anderen Übergabeparameter werden nicht näher betrachtet, da sie im folgenden nicht benötigt werden. (https://elixir.bootlin.com/linux/v5.2.14/source/drivers/mfd/mfd-core.c)

4

Fachbezogene Grundlagen

Zunächst soll 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, sollen ein paar Grundlagen und auch die Funktionsweise einer Kamera betrachtet werden.

4.1.1 Kamera

Um die Funktionalität des FRA zu festzustellen und grobe Fehler rechtzeitig zu erkennen, ist das regelmäßige Testen auf einer Kamera unerlässlich. Da die Wahl der Kamera keine weiteren Einschränkungen unterliegt, wurde eine ARRI sAMIRA gewählt.

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 elektronische 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“ [5], die Netflixserie „The Ivory Game“ [9] oder auch das Fernsehmagazin „The Grand Tour“ [8] zu nennen.

4.1.2 Bildkette

Unter einer Bildkette versteht man den Durchlauf des Sensorbilds bis zu zum Ausgang, in unserem Fall die Aufnahme (REC) und das Serial Digital Interface (SDI).

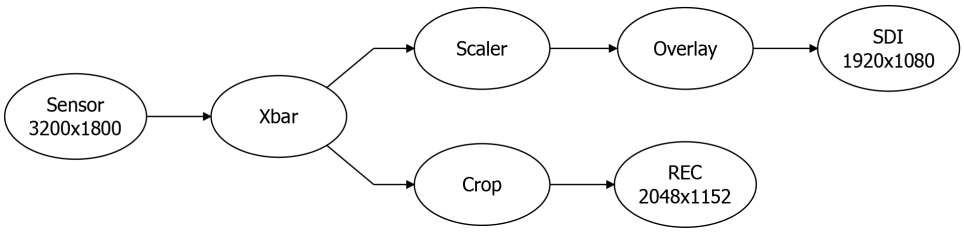

Abbildung 4.2: *Schematische Bildkette*

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, hier wird das identische Bild in zwei Bildpfade weiterführt. 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-Module Crossbar (Xbar) softwareseitig implementiert, da es 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 Module 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 Parametern 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 Problematiken 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.

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. Natürlich kann es dann passieren, dass über ein Re-

gister 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 auch die Zugriffe der Software auf den FPGA sollen übersichtlicher und wartbarer gestaltet werden.

Die Zugriffe auf den FPGA benötigen in der momentanen Implementierung immer die Adressen der Module im FPGA. 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 reine 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 zuschreiben.

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 };
```

Codebeispiel 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 };
```

Codebeispiel 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 IOCTL ist eine Anmeldung der Anwendung bei dem Gerät notwendig. Um die Eindeutigkeit zu garantieren wird hier die 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.

5.3 Implementierung im Kernel

5.4 Implementierung im Userspace

5.5 Einbindung ins Geometrie Framework

6

Testbackend

7

Ausblick

Literaturverzeichnis

[1] ARRI. *About ARRI*. Webseite. Accessed 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] Jonathan Corbet. *The platform device API*. Webseite. Accessed 2019-09-13. URL: *https://lwn.net/Articles/448499/*.

[4] 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*.

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

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

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

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

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