Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Jan Henke

Modellbasierte Entwicklung von Linux-Treibern

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Jan Henke

Modellbasierte Entwicklung von Linux-Treibern

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Thomas Lehmann
Zweitgutachter : Prof. Dr. rer. nat. Henning Dierks

Abgegeben am 21. Juli 2011

Jan Henke

Thema der Bachelorarbeit
Modellbasierte Entwicklung von Linux-Treibern
Stichworte
UML, Linuxkernel, Treiber, modellbasiert, Codegeneration
Kurzzusammenfassung
Diese Arbeit untersucht die Anwendbarkeit modellbasierter Entwicklungsverfahren auf die Entwicklung von Linuxkernelmodulen (Linuxtreibern). Es werden zwei unterschiedliche Konzepte hinsichtlich ihrer Machbarkeit und ihres Nutzens bewertet.
Im ersten Konzept wird versucht ein Kernelmodul komplett in der UML zu beschreiben, sodass aus dem UML-Modell eine Generierung des Programmcodes des Moduls möglich ist. Hier zeigte sich, dass eine Beschreibung nur teilweise in der UML möglich und sinnvoll ist.
Im zweiten Konzept wird die Fragestellung untersucht, ob eine beliebige Schnittstelle zwischen Kernel- und Userspace automatisch generiert werden kann. Ziel hierbei ist es, eine bessere Schnittstelle für Treiber und Geräte zu erhalten, welche nur schlecht durch eines der drei bestehendes Treibersubsysteme abgebildet werden kann. Diese konnte erreicht werden, ist jedoch mit Einschränkungen verbunden.

Jan Henke

Title of the paper
Model Based Development of Linux Drivers
Keywords
UML, Linux kernel, driver, model based, code generation
Abstract
This thesis investigates the application of model based development processes on the development of Linux kernel modules (Linux device drivers). There are two different concepts under evaluation in respect of their feasibility and possible gain.
The first concepts tries to completely describe a kernel module by using a UML model with the intent to make it possible to generate the complete code again using just the model. The result of this investigation showed that a description by using the UML is only partly possible and reasonable.
The second concept studies the possibility of generating a user-defined interface between the kernel and user space with the intention to achieve an interface better suiting the driver and the underlying hardware than any of the three existing driver sub systems in the kernel. This has been accomplish successfully, but there remain some restrictions.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Diese Arbeit beschäftigt sich mit der Treiberentwicklung im Linuxkernel unter Anwendung modellbasierter Softwareentwicklung auf Basis der Unified Modeling Language (UML).

Die Unified Modeling Language (UML) dient dazu, objektorientierte Systeme standardisiert in verschiedenen Aspekten zu beschreiben. Dazu definiert die UML verschiedene Diagramme, welche jeweils einen bestimmten Aspekt des Systems beschreiben. Das so erzeugte Modell kann als Basis zur automatischen Codesynthese dienen, wenn das Modell alle nötigen Informationen enthält. Im Vergleich zur direkten Programmierung, erhält man hierdurch eine größere Abstraktion vom konkreten System, welches die Wartbarkeit des erzeugten Programms erhöhen kann. [1]

Der Linuxkernel zeichnet sich durch seine gute Skalierbarkeit und Unterstützung vieler verschiedener Hardwareplattformen aus. Dies hat dazu geführt, dass dieser neben dem klassischen Einsatzgebiet auf Computern (Server, Laptop und Desktop) verstärkt auch im Bereich der eingebetteten Systeme (*engl.* embedded systems) als Betriebssystem verwendet wird. Eine Besonderheit der eingebetteten Systeme besteht darin, dass vielfach Hard- und Software vom selben Hersteller zusammen entwickelt wird und insbesondere Hardwarekomponenten verwendet werden, die nur in einer sehr begrenzten Zahl von unterschiedlichen Geräten zum Einsatz kommen. Dies führt dazu, dass im Entwicklungsprozess eines neuen Systems häufig auch die Treiber für ein oder mehrere Hardwarekomponenten entwickelt werden müssen.

Da der Linuxkernel unter Verwendung bestimmter objektorientierter Konzepte in den nicht objektorientierten Programmiersprachen C und Assembler geschrieben ist, wird z.B. der Code in klassenartigen Konstrukten organisiert, jedoch ohne Polymorphie und Vererbung zu unterstützten. Diese Arbeit untersucht welche Möglichkeiten der Modellierung durch die UML in der Treiberentwicklung für den Linuxkernel angewandt werden können und in wie

[1] Siehe hierzu auch Abschnitt 2.1

weit diese Modellierung zur automatischen Codegenerierung genutzt werden kann. Dabei werden zwei unterschiedliche Konzepte untersucht, welche jeweils einen unterschiedlichen Ansatz verfolgen.

Im ersten Konzept, das im Kapitel 3 beschrieben wird, wird versucht beispielhaft ein Kernelmodul komplett mit den Methoden, welche die UML zur Verfügung stellt, zu beschreiben. Ziel dabei ist das Aufzeigen genauer Korrespondenzen des Programmcodes mit der entsprechenden UML-Darstellung, um hieraus einen Codegenerator entwickeln zu können, der die UML-Darstellung wieder in ein komplettes Kernelmodul überführt. Im Erfolgsfalle würde dies die Entwicklung von Kernelmodulen komplett auf der abstrakteren Ebene eines UML-Modells ermöglichen, mit den damit verbundenen Vorteilen, wie z.B. höhere Abstraktion und leichtere Wartbarkeit.

Einen völlig anderen Ansatz verfolgt das in Kapitel 4 vorgestellte zweite Konzept. Für die Kommunikation zwischen dem Kernel und normalen „Userspace"-Anwendungen kennt der Linuxkernel nur drei verschiedene Arten von Geräten, nämlich zeichenorientierte, blockorientierte und sockelorientierte, mit jeweils einer festen Schnittstelle[2]. Reale Geräte, insbesondere im Bereich der eingebetteten Systeme, lassen sich aber nicht immer zu 100% einer dieser Kategorien zuordnen, vor allem ist eine an das Gerät angepasste Schnittstelle effektiver. So ließe sich z.B. ein Beschleunigungssensor zwar als zeichenorientiertes Gerät auffassen, welches in einem Strom die aktuellen Beschleunigungswerte aller Achsen nacheinander ausgibt. Effektiver wäre es jedoch, eine eigene Schnittstelle zu definieren, anhand derer entweder die Werte aller Achsen auf einmal abgerufen werden oder selektiv nur bestimmte Achsen direkt abgefragt werden können.

Hierzu wird untersucht, ob es möglich ist, aus der Schnittstellendefinition in einem UML-Modell automatisch Code zu generieren, der im Sinne des Stellvertreter-Entwurfsmusters (*engl.* proxy design pattern) diese Schnittstelle zwischen Kernel und Userspace darstellt, sodass man aus dem Userspace direkt die spezifizierten Funktionen im Kernel aufrufen kann, ohne auf die Standardsystemaufrufe angewiesen zu sein. Dazu bedarf es einer Bibliothek im Userspace, die diese Schnittstelle für verschiedene Programme zur Verfügung stellt und eines Kernelmoduls, welches die eigentlichen Funktionsaufrufe im Kernel vornimmt. Diese müssen dabei über die Standardsystemaufrufe kommunizieren, sodass für die Kommunikation ein allgemeingültiges System spezifiziert werden muss.

[2] siehe hierzu auch Abschnitt 2.2

Kapitel 2

Grundlagen

Dieses Kapitel soll Hintergrundwissen über die verwendeten Techniken vermitteln, um die nachfolgenden Kapitel besser verstehen zu können.

Der *Abschnitt 2.1* geht dabei auf die Technik der modellbasierten Entwicklung ein.

Während der *Abschnitt 2.2* die Unterschiede und Besonderheiten der Programmierung im Linuxkernel erläutert.

Schließlich wirft der *Abschnitt 2.3* einen Blick auf die verfügbaren Werkzeuge zur modellbasierten Entwicklung und zeigt die Gründe der Wahl von Artisan Studio für diese Arbeit auf.

2.1 Grundlagen der modellbasierten Entwicklung

Ziel einer Entwicklung ist in der Regel die Lösung eines gegebenen Problems. Die klassische Vorgehensweise besteht darin, direkt in der Sprache der Zielumgebung eine Lösung für dieses Problem zu formulieren. In der modellbasierten Entwicklung wird stattdessen zunächst ein abstraktes Modell der Problemlösung entworfen, welches zunächst unabhängig von der späteren Zielumgebung ist, daher keine spezifischen Elemente der Zielumgebung enthält. In diesem abstrakten Modell ist es möglich die Problemlösung einfacher und präziser zu beschreiben, als es bei einer konkreten Implementierung möglich wäre, da nicht auf die Eigenschaften eines konkreten Zielsystems eingegangen werden muss. Dies reduziert die Redundanz in der Beschreibung und verringert damit den Aufwand, der für eine spätere Wartung nötig ist.

Um aus dieser abstrakten Darstellung ein konkretes Zielsystem zu erhalten, bedarf es einer Transformation zwischen diesen Elementen. Im Bereich der Softwareentwicklung bedeutet

dies in der Regel, dass basierend auf dem Modell eine Codesynthese durchgeführt wird. Die zur Beschreibung des abstrakten Modells verwendete Unified Modelling Language (UML) lässt sich jedoch meistens nicht direkt synthetisieren, da meistens für das Zielsystem spezifische Zusatzinformationen nötig sind, um die Transformationen durchführen zu können, die die reine UML nicht enthält. Es ist daher im Laufe des Entwicklungsprozesses eine Abwägung zu treffen zwischen der gewünschten Abstraktion vom Zielsystem und zusätzlicher Information über das konkrete Zielsystem, ohne die eine Synthese des Modells nicht möglich ist.

Ein klarer Nachteil dieses Problemlösungsansatzes ist der zusätzliche Aufwand, der durch die zunächst abstrakte Modellierung entsteht. Dafür erhält man eine bessere Wartbarkeit des Zielsystem, da Änderungen auf der Ebene des abstrakten Modells einfacher durchzuführen sind, außerdem lassen sich Teile des Modells einfach wiederverwenden.

2.2 Grundlagen über die Entwicklung von Linuxtreibern

Linux ist wie die meisten modernen Betriebssysteme in zwei Bereiche mit unterschiedlichen Privilegierungen aufgeteilt. In den Bereich des Betriebssystemkerns (Kernel), der im sogenannten Kernelspace ausgeführt wird, und in den Bereich der Benutzerprogramme, das heißt aller Programme, die selbst nicht Teil des Kernels sind, Userspace genannt. Diese Unterteilung dient dazu, die Sicherheit und die Stabilität des Systems zu verbessern, indem allen Programmen des Userspaces eine Reihe von bekannten Beschränkungen auferlegt werden. Dazu gehört unter anderem, dass sie nur Zugriff auf den Teil des Speichers haben, welcher ihnen vom Betriebssystem zugewiesen wurde, und ihnen nicht alle Maschinenbefehle des Prozessors zur Verfügung stehen, so ist zum Beispiel ein direkter Zugriff auf die Hardware nicht erlaubt.

Linuxtreiber jedoch werden als sogenannte Kernelmodule realisiert. Wie der Name bereits aussagt, werden diese als Teil des Kernels ausgeführt und unterliegen damit nicht den eben genannten Beschränkungen, können allerdings dafür nicht auf die Standard C-Bibliothek zurückgreifen. Kernelmodule können sowohl als feste Bestandteile des Kernels kompiliert werden, als auch als zur Laufzeit nachladbare kompilierte Objekte vorliegen.

Die Kommunikation zwischen dem Userspace und dem Kernelspace findet, entsprechend der UNIX-Philosophie „alles ist eine Datei”, über spezielle Dateien statt, die zur Laufzeit im Dateisystem unter /dev/ (kurz für devices) erzeugt werden. Wenn ein Benutzerprogramm eine der verfügbaren Funktionen auf eine dieser Dateien aufruft, wird ein Systemaufruf (*engl.* system call) ausgelöst. Das ist ein Softwareinterrupt, der einen Kontextwechsel in den Kernel auslöst und im mit der Datei verbunden Modul eine entsprechende Funktion aufruft. Über diesen Weg ist es auch Benutzerprogrammen möglich mit der Hardware zu interagieren, indem über entsprechende Systemaufrufe jeweils ein für die Hardware passendes

Kernelmodul mit der Hardware direkt interagiert und das Ergebnis dann entsprechend zur Verfügung stellen kann. Es ist dabei zu beachten, dass die verschiedenen system calls fest definiert und jeweils fest mit einer bestimmten Funktion der Standard C-Bibliothek verbunden sind.

Für häufig benötigte Arbeiten, zum Beispiel das Erstellen der eben beschriebenen Gerätedateien oder die Kommunikation mit USB-Geräten, stellt der Linuxkernel sogenannte Subsysteme zur Verfügung. Wenn ein Modul die Ressourcen eines dieser Subsysteme in Anspruch nehmen möchte, kann es sich bei diesem registrieren und bekommt dann von diesem entsprechende Ressourcen zugeteilt. Für viele Bereiche wird so ein großer Teil des Verwaltungsaufwandes für Ressourcen bereits von den verschiedenen Subsystemen geleistet, allerdings bedeutet dies auch, dass man sich bei Benutzung dieser Erleichterung für eins der existierenden Subsysteme entscheiden muss. Um zum Beispiel eine solche Gerätedatei zu erstellen, muss man sich bei einem von nur drei existierenden I/O-Subsystemen anmelden, sein Modul also als blockorientiert, zeichenorientiert oder sockelorientiert kategorisieren, mit allen damit verbunden Einschränkungen.

Möchte man einen Treiber unter Benutzung eines oder mehrerer Subsysteme schreiben, ergibt sich daher eine immer gleiche Grundstruktur. Der Treiber muss somit über die folgenden Komponenten verfügen:

Eine Initialisierungsfunktion wird vom Kernel immer dann aufgerufen, wenn das Modul geladen wird. Hier müssen alle benötigten Ressourcen angefordert werden und das Modul muss sich bei den benötigten Subsystemen anmelden. Es ist dabei zu beachten, dass ein Modul zur Laufzeit des Systems durchaus mehrmals geladen und wieder entladen werden kann.

Eine Aufräumfunktion ist das Gegenstück zur Initialisierungsfunktion, entsprechend muss sich das Modul hier von allen Subsystemen abmelden und alle Ressourcen wieder freigeben. Da ein Modul, wie bereits erwähnt, zur Laufzeit des Kernels auch wieder entladen werden kann, ist es wichtig darauf zu achten, immer sauber aufzuräumen, um über eine längere Laufzeit keine Problem zu verursachen.

Modulweit genutzte Daten zum Beispiel Verweise auf von Subsystemen erhaltene Ressourcen, die wieder zum Abmelden benötigt werden.

Eine oder mehrere Funktionen, die als Einsprungspunkt für an dieses Modul gerichtete Systemaufrufe dienen. Die Adressen dieser Funktionen werden während der Initialisierung dem Kernel mitgeteilt, sodass der Handler des Softwareinterrupts direkt die entsprechende Funktion anspringen kann.

2.3 Auswahl der Entwicklungswerkzeuge

Bei der modellbasierten Entwicklung kommt der Wahl der benutzten Entwicklungswerkzeuge eine besondere Bedeutung zu. Denn der Schritt von dem abstrakten Modell zum mehr oder weniger fertigen System, muss von dem verwendeten Werkzeug bewältigt werden. Daher haben die Fähigkeiten dieses Werkzeugs großen Einfluss darauf, inwieweit die Vorteile dieser Entwicklungstechnik überhaupt genutzt werden können.

Um die Eignung für die Realisierung dieser Arbeit zu bewerten, werden die folgenden Kriterien herangezogen:

- Das Werkzeug muss in der Lage sein, aus einem UML-Modell entsprechenden C-Code automatisch zu generieren.
- Es muss möglich sein, die Generierung nach eigenen Bedürfnissen anzupassen, sodass das Ergebnis sich als Kernelmodul kompilieren und ausführen lässt.
- Es muss möglich sein, sich in kurzer Zeit so weit einzuarbeiten, dass eine produktive Verwendung im Rahmen dieser Arbeit möglich ist.
- Vorzugweise soll das verwendete Werkzeug Open-Source sein, sodass die Ergebnisse dieser Arbeit ohne zusätzliche Kosten frei nutzbar sind.

Die Auswahl an zur Verfügung stehenden Werkzeugen ließ sich sehr schnell auf drei konkret in Frage kommende Werkzeuge einschränken. Zur Auswahl standen daher Topcased, Papyrus und Artisan Studio, auf die im folgenden jeweils kurz eingegangen wird.

2.3.1 Topcased

Topcased ist ein Open-Source Programm, welches auf der Eclipse IDE aufsetzt und das auch in der Industrie für verschiedene Projekte eingesetzt wird. Obwohl auch damit geworben wird, dass dieses Werkzeug in der Lage sein soll die geforderte Codesynthese durchzuführen, zeigte sich sehr schnell, dass diese Funktionalität kaum in öffentlichen Quellen dokumentiert wird. Hinzu kommt, dass die allgemeine Bedienung für Einsteiger sehr umständlich ist. Topcased stand somit nicht mehr zur Wahl, da eine produktive Verwendung im Rahmen dieser Arbeit nicht abzusehen war.

2.3.2 Papyrus

Papyrus ist ebenso wie Topcased ein Open-Source Programm, welches auf der Eclipse IDE aufsetzt, jedoch wird Papyrus von der Eclipse Foundation selbst betreut. Leider treten die

Kritikpunkte, die gegenüber Topcased vorhanden sind, auch hier wieder auf. Das Resultat ist daher, dass Papyrus auch nicht für diese Arbeit geeignet war. Es scheint daher, dass die Probleme bei der Bedienung der beiden Werkzeuge von dem gemeinsamen Merkmal ausgehen, dass beide als Erweiterung der Eclipse IDE realisiert sind.

2.3.3 Artisan Studio

Artisan Studio ist ein proprietäres Programm, welches von der Firma Artego entwickelt und vertrieben wird. Es zeichnete sich im Rahmen dieses Vergleiches dadurch aus, dass es über einen fertigen Codegenerator für C verfügt, der sich auch gut anpassen lässt. Insbesondere sind hier die Funktionen gut dokumentiert und es wird im Internet vom Hersteller Trainingsmaterial zur Verfügung gestellt, das den Einstieg in diese Software deutlich erleichterte. Aufgrund dieser deutlichen Vorteile im Vergleich mit den beiden Open-Source Alternativen, verwende ich dieses Programm zur Realisierung des praktischen Teils dieser Arbeit. Dafür habe ich von der Firma Artego eine kostenlose Universitätslizenz für die Bearbeitungszeit dieser Arbeit erhalten.

Kapitel 3

Konzept 1: UML-Modellierung eines Kernelmoduls

Das Ziel dieses Konzeptes besteht darin, herauszufinden, wie sich allgemein ein Kernelmodul komplett in der UML beschreiben lässt. Ist dies erreicht, so lassen sich aus dem Vergleich der zwei Beschreibungen (UML und Quellcode) die direkten Korrespondenzen zwischen den einzelnen Elementen der UML und den entsprechenden Codefragmenten herleiten. Auf Basis dieser Korrespondenzen lässt sich dann ein passender Generator entwickeln, welcher automatisch die umgekehrte Umformung aus einem UML-Modell in den Zielcode durchführt.

Ich begann daher zunächst damit, den Code eines existierenden und lauffähigen Kernelmoduls zu analysieren. Im Rahmen dieser Arbeit habe ich dafür aus dem Begleitmaterial (Quade und Kunst, 2011) zu dem Buch von Quade und Kunst (2006), welches unter der GNU Public Licence version 2 steht und somit frei verbreitet werden kann, die Treiber für eine einfache sogenannte „RAM-Disk" und ein allgemeines Muster für zeichenorientierte Treiber ausgesucht, anhand derer ich die Besonderheiten der Treiberentwicklung herausarbeite.

Eine RAM-Disk verhält sich wie jeder andere Massenspeicher, wie zum Beispiel eine Festplatte oder ein Flashspeicher. Sie ist also ein blockorientiertes Gerät, auf dem ein Dateisystem existieren und beliebig Daten gelesen und geschrieben werden können. Der Unterschied besteht darin, dass sämtliche Daten nur im flüchtigen Hauptspeicher vorliegen, somit nach dem Ausschalten entweder verloren gehen oder auf einem nichtflüchtigen Massenspeicher gesichert werden müssen. Anwendung finden solche RAM-Disks vor allem bei den Live-CD genannten Datenträgern, welche häufig von Linux-Distributionen zur Installation genutzt werden. Diese enthalten auf dem Datenträger meistens eine große RAM-Disk, die zunächst in den Hauptspeicher geladen und anschließend das Betriebssystem von diesem

gestartet wird. So kann das Betriebssystem ohne Installation gestartet werden, kann aber mit der RAM-Disk wie mit einer normalen Festplatte interagieren.

Der komplette Programmcode der verwendeten Treiber ist im Anhang auf der beiliegenden CD enthalten[3]. Im Folgenden wird jeweils nur der Ausschnitt wiedergegeben, auf den sich eine Textstelle bezieht.

3.1 Konzeptentwurf

Bei der Betrachtung des Programmcodes der beiden Treiber lassen sich sehr schnell mehrere Programmstrukturen erkennen, die charakteristisch für Linuxkernel sind.

Obwohl die Programmiersprache C keine Unterstützung für Klassen bietet, so ist der Kernel doch von der Idee her in Klassen strukturiert. Durch die Kapselung sämtlicher Daten jeweils in eine Struktur, wird eine Wiederverwendung von Code möglich. Eine C-Struktur (Schlüsselwort *struct*) kann als Objekt und ihre Elemente als Attribute dieses Objektes gesehen werden. Alle Funktionen, die als erstes Argument den Zeiger auf eine Struktur erhalten, können somit als Objektmethode verstanden werden. Eine Technik, die in dieser Art auch bei anderen Programmiersprachen zur Realisierung der Objektorientierung verwendet wird, zum Beispiel Python. Alle Objekte des Linuxkernels verfügen über eine Initialisierungsfunktion, die daher einem Konstruktor entspricht, und eine Funktion um am Ende die Ressourcen des Objektes korrekt freizugeben, diese Funktion entspricht somit einem Destruktor. Alle Funktionen, die nicht in eine dieser drei Kategorien fallen, können somit als statische Methoden begriffen werden. Über diese Zuordnung erhält man eine objektorientierte Sicht auf sämtliche Elemente des Linuxkernels.

Listing 3.1: Die Definition der Objekte

```
1 static struct gendisk *disk;
2 static struct request_queue *bdqueue;
```

Listing 3.2: Eine Konstruktoraufruf

```
1    disk=alloc_disk(1)
```

Listing 3.3: Ein Destruktoraufruf

```
1    del_gendisk(disk);
```

Listing 3.4: Ein Objektmethodenaufruf

```
1    set_capacity( disk, (SIZE_IN_KBYTES * 1024)>>9 );
```

[3] der gesamte Inhalt der CD mit den genauen Pfaden ist in Anhang A auf Seite 24 aufgelistet

Die Listings 3.1 bis 3.4 belegen das vorher Gesagte anhand der Verwendung im Code der RAM-Disk. Die grundlegenden Elemente eines UML-Klassendiagramms sind somit im Quellcode identifiziert. Damit kann somit die grobe Struktur des Treibers und des Linuxkernels in einem Klassendiagramm beschrieben werden.

Bei der Betrachtung verschiedener Linuxtreiber fällt auf, dass diese häufig Gebrauch von Makros machen, sowohl um Metadaten im Quellcode einzubinden, zum Beispiel die verwendete Lizenz, aber auch zur Initialisierung von Objekten oder auch zum Aufrufen der Initialisierungs- und Deinitialisierungsfunktionen des Moduls. Der UML-Standard sieht für diese Makros keine Elemente vor, sodass ich eine eigene Lösung entwickelt habe. Diese besteht darin, die Makros als Schlüssel-Werte-Paare (der Name des Makros ist der Schlüssel, Parameter sind der Wert) im Kommentarfeld des Objektes abzulegen, welches den Treiber repräsentiert.

Für diese Lösung spricht, dass das Kommentarfeld allgemein vorhanden ist, seinem Inhalt jedoch im Allgemeinen keine feste Bedeutung zugewiesen ist, somit kann die Struktur des Inhaltes in diesem Kontext frei definiert werden. Zur besseren Unterscheidung von „normalen" Kommentaren sollte der Bereich der Makros durch ein Paar geschweifte Klammern eingeschlossen werden. Damit muss der UML-Standard an dieser Stelle erweitert werden, um diese Informationen im Model unterzubringen.

Im Abschnitt 2.2 auf Seite 6 wurde bereits erläutert, dass zur Kommunikation zwischen Userspace und Kernelspace Systemaufrufe benutzt werden, die einen Softwareinterrupt auslösen, bei dessen Behandlung der Kernel jeweils eine zum Systemaufruf passende Funktion im jeweiligen Modul aufruft. Damit ein Treiber mit dem Userspace kommunizieren kann, ist es nötig, dem Kernel mitzuteilen, an welcher Adresse jeweils die aufzurufende Funktion für die verschiedenen Systemaufrufe beginnt. Der Linuxkernel macht zu diesem Zweck intensiven Gebrauch von Zeigern auf Funktionen.

Listing 3.5: Zuordnung der Einsprungpunkte für Systemaufrufe

```
1 static struct file_operations fops = {
2    .open = driver_open,
3    .release = driver_close,
4    .read = driver_read,
5    .write = driver_write,
6    .poll = driver_poll,
7 };
```

Listing 3.5 zeigt die Verwendung im Kernelkontext. „driver_..." ist dabei jeweils der Name einer in diesem Treiber definierten Funktion. Man sieht, dass dort in der Struktur „file_operations" jeweils dem Namen eines Systemaufrufes die Adresse der dabei aufzurufenden Funktion zugeordnet wird.

Um diese Zuordnung in der objektorientierten Sichtweise der UML zu beschreiben, kann das Objekt, welches die Funktionszeiger erhält (hier struct file_operations), im Rahmen dieser Sichtweise als Schnittstellenbeschreibung, die vom Treiber implementiert wird, gesehen und entsprechend im Klassendiagramm kenntlich gemacht werden. Wenn sich die Funktionen im Treiber an ein bestimmtes Namensschema halten (hier „driver_<Systemaufrufbezeichnung>"), ist es dem Codegenerator dadurch möglich, die gewünschte Beziehung herzustellen.

Bis hierhin hat sich gezeigt, dass sich bereits große Teile des Treiberquellcodes in einem UML-Klassendiagramm beschreiben lassen. Lediglich der Inhalt der Funktionen lässt sich nicht auf ein Klassendiagramm abbilden. Es fällt in diesem Zusammenhang auf, dass ein großer Teil dieses Codes aus Funktionsaufrufen besteht, die sich sehr gut in Sequenzdiagrammen beschreiben lassen. Zusätzlich werden jedoch auch prozedurale Passagen benötigt, sodass sich in der Summe ein in der UML Version 2 eingeführter Diagrammtyp gut dafür eignet, das Interaktionsübersichtsdiagramm. Dieser Diagrammtyp entspricht einem Aktivitätsdiagramm, bei dem die einzelnen Aktivitäten jeweils ein Sequenzdiagramm sind, womit sich der gesamte Inhalt einer Funktion in einem Diagramm abbilden lassen sollte.

Auf Basis der in diesem Abschnitt gemachten Überlegungen und Erkenntnisse, soll das so entwickelte Konzept im Abschnitt 3.2 angewandt werden, um den verwendeten RAM-Disk-Treiber komplett durch ein UML-Modell in Artisan Studio zu beschreiben.

3.2 Konzeptumsetzung

Bei der Anwendung des im letzten Abschnitt entwickelten Konzeptes, hat sich das Problem ergeben, dass das Interaktionsübersichtsdiagramm zum Zeitpunkt der Erstellung dieser Arbeit, weder in der favorisierten Software „Artisan Studio", noch in den alternativen Programmen zur Verfügung gestanden hat. Es bestand daher die Notwendigkeit für diese Umsetzung das Konzept dahingehend zu ändern, dass stattdessen nur Aktivitätsdiagramme verwendet werden. Reine Sequenzdiagramme können in diesem Zusammenhang nicht genutzt werden, da in diesen eine Modellierung prozeduraler Passagen nicht möglich ist.

Ich begann daher damit ein Klassendiagramm zu erstellen, welches neben der Schnittstellenbeschreibung der eigentlichen RAM-Disk auch die im Quellcode verwendeten Teile der vom Kernel bereitgestellten Objekte beinhaltet, um diese dadurch dem Generator bekannt zu machen und sich im Aktivitätsdiagramm darauf beziehen zu können. Abbildung 3.1 auf der nächsten Seite zeigt das erstellte Klassendiagramm. Parallel dazu habe ich die Arbeit am Aktivitätsdiagramm aufgenommen, das die Funktion bd_request beschreiben soll und in Abbildung 3.2 auf Seite 15 dargestellt ist.

Beide Diagramme sind nicht vollständig. Denn bereits nach kurzer Bearbeitungszeit zeigten

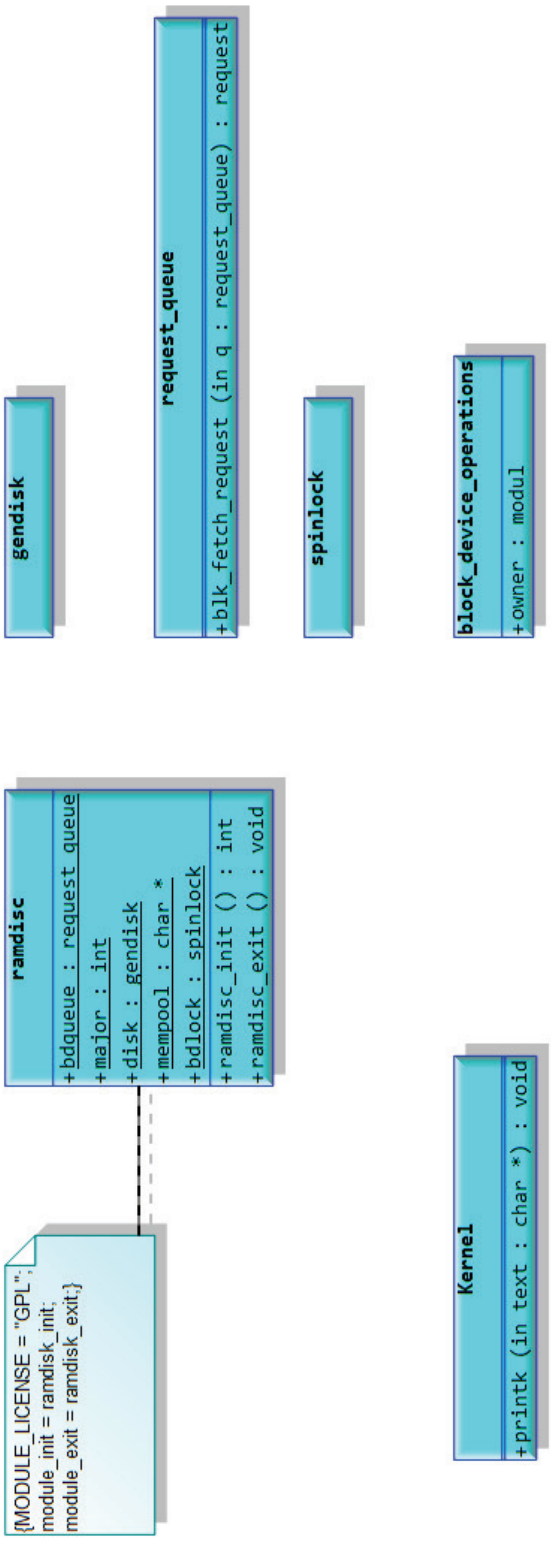

Abbildung 3.1: Das erstellte Klassendiagramm

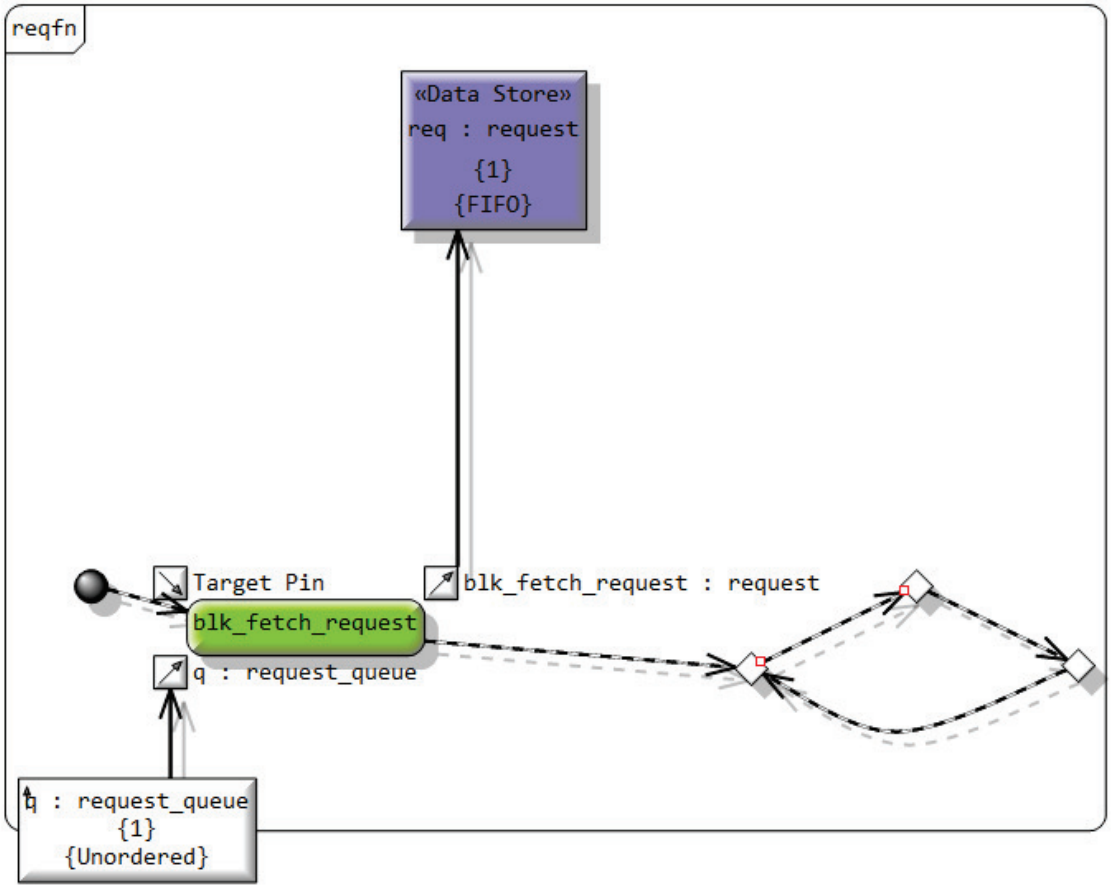

Abbildung 3.2: Das erstellte Aktivitätsdiagramm für die Funktion bd_request()

sich Probleme bei der Erstellung des Aktivitätsdiagramms, die zeigten, dass dieser Ansatz nicht in der Lage ist, die gewünschten Ziele zu erreichen.

Um den Programmablauf innerhalb der Funktion so zu beschreiben, dass der Generator später daraus wieder Programmcode generieren kann, ist die Verwendung von speziellen Elementen erforderlich, die eine werkzeugabhängige Erweiterung des UML-Standards darstellen. Zum Beispiel ist der in Abbildung 3.2 auf der vorherigen Seite verwendete „Data Store" kein standardisiertes UML-Element in diesem Kontext, aber für ein synthesefähiges Modell notwendig. Dies hat zwei Probleme zur Folge:

Zum einen wäre eine derartige Lösung nur mit diesem speziellen Werkzeug zu benutzten und damit keine allgemeine Lösung des Problems, insbesondere da Artisan Studio kein Open-Source Programm ist. Zum anderen, und das ist das Entscheidende, wäre eine derartige Lösung damit verbunden, dass die Anweisungen des Quellcodes eins zu eins in einzelne Elemente des Diagramms umgesetzt werden müssten, was zur Folge hat, dass praktisch keine Abstraktion mehr zwischen dem Aktivitätsdiagramm und dem eigentlichen Quellcode existiert. Dabei ist, wie eingangs erwähnt, aber gerade diese Abstraktion das wesentliche Ziel der modellbasierten Entwicklungsmethodik. Daher kann daraus geschlossen werden, dass dieses Konzept keinen wesentlichen Mehrwert gegenüber der direkten Entwicklung auf Basis des Quellcodes bietet, im Vergleich mit diesem jedoch mehr Aufwand der Modellierung erfordert.

Davon unbeeinflusst lässt sich auch feststellen, dass die UML-Klassendiagramme zwar nur einen Teil eines Treibers, nämlich in erster Linie seine Schnittstelle nach außen, beschreiben, für diesen Zweck aber gut geeignet sind.

Kapitel 4

Konzept 2: Generierung eines Stellvertreter-Kernelmoduls

Das Konzept, das in diesem Kapitel vorgestellt wird, verfolgt das Ziel, die modellbasierten Entwicklungsmethodiken zu nutzen, um Kernelmodulen eine auf das spezielle Modul angepasste Schnittstelle zum Userspace zu ermöglichen. Wie bereits im Abschnitt 2.2 auf Seite 6 vorgestellt, werden zur normalen Kommunikation zwischen Kernel- und Userspace spezielle Gerätedateien verwendet, welche im Verzeichnis /dev/ dynamisch erstellt und wieder entfernt werden. Diese ermöglichen jedoch nur die Kommunikation über eine standardisierte Schnittstelle.

Die Idee dieses Konzeptes besteht darin, basierend auf der Schnittstellendefinition in einem UML-Klassendiagramm, automatisch einen Satz von Dateien zu erzeugen, der diese Einschränkungen umgeht, indem er anstelle der Standardschnittstelle, die in dem Klassendiagramm definierte Schnittstelle in Form einer Userspace Bibliothek anbietet. Dazu ist neben der Generierung der Userspace Bibliothek zusätzlich ein zu generierendes Kernelmodul[4] nötig, welches mit der Bibliothek über die Standardschnittstelle kommuniziert, die eigentliche Funktion im Zielkernelmodul aufruft und einen eventuellen Rückgabewert wieder an den Aufrufer im Userspace zurückgibt.

Ein ähnliches Konzept wird bereits von der FUSE[5] genannten Bibliothek verfolgt, die Dateisystemzugriffe im Userspace zur Verfügung stellt. Im Unterschied dazu soll jedoch bei dem hier verfolgten Konzept ein allgemeinerer Ansatz gewählt werden, der nicht auf Dateisysteme beschränkt ist und automatisch generiert wird.

Für die zu generierenden Dateien wird ein festes Namensschema verwendet. Sei *nnn* der

[4] Wenn im Folgenden von einem Proxymodul die Rede ist, so ist dieses Modul gemeint, entsprechend dem englischen Namen für das Stellvertreter-Entwurfsmuster: proxy design pattern

[5] steht für Filesystem in Userspace

Name des Moduls für das ein Stellvertreter generiert werden soll, so werden die folgenden Dateien erstellt.

libnnn.h Diese C-Headerdatei enthält lediglich die Signatur der von der Bibliothek bereitgestellten Funktionen. Das sind zum einen die Funktionen, die von der Schnittstelle vorgegeben werden, zum anderen die Hilfsfunktionen, um die Bibliothek korrekt zu initialisieren und nach Benutzung die Ressourcen korrekt freizugeben.

libnnn.c Diese C-Quellcodedatei enthält die Programmlogik für Kommunikation mit dem Proxymodul. Entsprechend kann diese Datei, kompiliert in Binärform, als statische oder dynamische Bibliothek verteilt werden.

nnnProxy.c Diese C-Quellcodedatei enthält das Proxymodul und damit die Gegenstelle, mit der die libnnn.c genannte Bibliothek im Kernel kommuniziert.

Wesentliche Aufgabe dieser Dateien ist die Signalisierung der Funktionsaufrufe. Das heißt, das Kernelmodul muss die Information erhalten, welche Funktion mit welchen Parametern aufzurufen ist und einen eventuellen Rückgabewert wieder zurückmelden. Dabei treten zwei prinzipielle Probleme auf.

Die für die Kommunikation zwischen Kernel und Userspaceprogrammen zur Verfügung stehenden Funktionen erlauben die Datenübermittlung nur in eine Richtung, es können jeweils nur Daten gelesen oder nur geschrieben werden. Dies macht es nötig den Rückgabewert im Kernel zwischen zu speichern, bevor dieser in einem zweiten Schritt vom Userspaceprogramm ausgelesen werden kann. Für das Userspaceprogramm ist es daher nötig zusätzliche Maßnahmen zu treffen, um entsprechende Codeabschnitte reentrantfähig zu machen, zum Beispiel durch die Verwendung von Mutexen.

Das Kernelmodul kann nicht direkt mit den Daten im Speicherbereich des Userspaces arbeiten, sondern muss diese erst in den Speicherbereich des Kernels kopieren. Umgekehrt ist ebenso ein Kopiervorgang nötig, um Daten vom Kernelmodul in den Userspace zu bekommen. Dies macht die Verwendung von Zeigern als Parameter oder Rückgabewert äußerst schwierig, da zum Zeitpunkt der Generierung des Proxies keinerlei Informationen vorliegen, wie diese Zeiger im übrigen Programm verwendet werden. Der Zeiger könnte zum Beispiel einen weiteren Zeiger in einer Struktur beinhalten oder auf den Beginn eines Feldes verweisen, beide Fälle sind möglich, erfordern aber unterschiedliche Lösungen. Für eine allgemeine Lösung dieses Problems wäre es daher nötig, bereits bei der Erstellung der Daten zusätzliche Information über die Verwendung des Zeigers zu haben. Da die Schnittstelle aber als Bibliothek nach außen hin offen sein soll, was ja genau das Ziel ist, kann bei der Generierung der Dateien gar nicht fest stehen, welcher Art eventuelle Zeiger als Parameter sind. Für die Verwendung von Zeigern kann es daher keine allgemeingültige Implementierung nur auf Basis der Information über die Schnittstellendefinition geben.

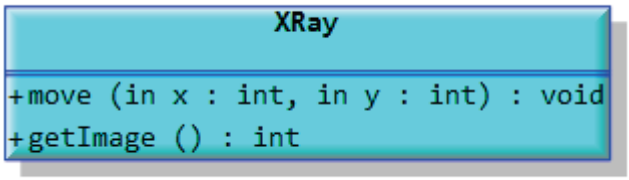

Abbildung 4.1: Klassendiagramm des zum Testen verwendeten Kernelmoduls

4.1 Implementierung des Konzeptes

Um die Machbarkeit dieses Konzeptes zu zeigen, wurde im Rahmen dieser Arbeit in Artisan Studio ein Generator geschrieben, der das beschriebene Konzept umsetzt.[6] Die korrekte Funktionsweise wurde mit einem für diesen Test erstellen Modul mit dem Namen XRay getestet, dessen Schnittstelle Abbildung 4.1 zeigt.

Zweck dieses Moduls ist es, die Schnittstelle im Kernel bereitzustellen, damit das Proxymodul überhaupt vom Kernel geladen werden kann. Dazu exportiert es die beiden Funktionen, damit diese anderen Modulen zur Verfügung stehen (siehe Listing 4.1). Die Funktionen selbst sind so implementiert, dass diese nur eine Logmeldung mit den Parametern generieren und einen festen Wert zurückgeben, womit die korrekte Funktion des Proxymoduls verifiziert werden kann.

Listing 4.1: Bereitstellung der Schnittstelle im Zielmodul "XRay.c"

```
1 EXPORT_SYMBOL(move);
2 EXPORT_SYMBOL(getImage);
```

Listing 4.2: Implementierung der Schnittstelle im Zielmodul "XRay.c"

```
1  void move(int x, int y)
2  {
3        printk("move(int, int) called with arguments: %d, %d\n", x,
           y);
4     return;
5  }
6
7  int getImage(void)
8  {
9        printk("getImage() called with no arguments\n");
10    return -1;
11 }
```

[6] Das Artisan Studio Model mit dem Generator ist Teil des elektronischen Anhangs auf CD

Entsprechend wurde ebenso ein Userspaceprogramm geschrieben, das die Bibliothek nutzt um beide Funktionen der Schnittstelle jeweils einmal aufzurufen. Zusammen mit dem Kernelmodul lässt sich so die korrekte Funktion verifizieren.

Listing 4.3: Testprogramm "textXRay.c"für die Userspacebibliothek

```
1  #include <stdio.h>
2
3  #include "libXRay.h"
4
5  int main(char *argv[], int argc)
6  {
7     int i = 0;
8     printf("Starting test.\n");
9     _init_();
10    move(19, 5);
11    printf("move(int, int) called.\n");
12    i = getImage();
13    printf("getImage() returned: %d\n", i);
14    _close_();
15    printf("End of test.\n");
16
17    return 0;
18 }
```

Im Folgenden werde ich auf einige Merkmale der Implementierung eingehen.

Zur Kommunikation zwischen Userspace und Kernelspace verwende ich hier einen union Datentyp, wie er in Listing 4.4 definiert ist. Entsprechend findet sich diese Typendeklaration sowohl im Proxymodul, als auch in der Bibliothek. Hier sind nur primitive Datentypen enthalten, es lassen sich jedoch nach einem festen Namensschema auch beliebige andere Datentypen hinzufügen. Die Verwendung eines unions bedeutet natürlich, dass relativ viel Speicher benötigt wird, allerdings vereinfacht dies auch den weiteren Programmcode, da das union so jeden anderen Datentyp aufnehmen kann.

Listing 4.4: Zur Kommunikation genutztes union

```
1  typedef union{
2     float __float;
3     unsigned __unsigned;
4     double __double;
5     char __char;
6     short __short;
7     long __long;
8     int __int;
```

```
}parameter;
```

Die Kommunikation mit dem Proxymodul kann daher so realisiert werden, dass einfach ein Feld des definierten Datentyps über die write-Funktion an das Proxymodul übergeben wird. Dieses ist so strukturiert, dass das erste Element immer die aufzurufende Funktion enthält, anschließend kommen dann eventuelle Parameter in der richtigen Reihenfolge.

Listing 4.5: Zur Kommunikation mit Proxymodul ohne Rückgabewert (libXRay.c)

```
    parameter data[3];
    data[0].__int = _move;
    data[1].__int = x;
    data[2].__int = y;

    write(fd, data, sizeof(data));
    return;
```

Ein anschließendes Lesen des Rückgabewertes findet nur statt, wenn dieser von „void" verschieden ist.

Listing 4.6: Zur Kommunikation mit Proxymodul mit Rückgabewert (libXRay.c)

```
    parameter data[1];
    data[0].__int = _getImage;

    write(fd, data, sizeof(data));
    parameter returnValue;
    read(fd, &returnValue, sizeof(parameter));
    return returnValue.__int;
```

Im Proxymodul findet anschließend der entsprechende Funktionsaufruf statt.

Listing 4.7: Funktionsaufruf im Kernel (XRayProxy.c)

```
switch(data[0].__int)
{
   case _move:
      move(data[1].__int, data[2].__int);
      break;
   case _getImage:
      ((parameter *)(instance->private_data))->__int = getImage
         ();
      break;
}
```

Diese Implementierung funktioniert mit den eingangs vorgestellten Testmöglichkeiten problemlos, sodass die Machbarkeit diese Konzeptes damit erwiesen ist.

Kapitel 5

Fazit

Ich habe in dieser Arbeit die Anwendungsmöglichkeiten der modellbasierten Entwurfsmethodiken auf die Entwicklung von Treibern für den Linuxkernel untersucht. Dabei bin ich zwei unterschiedlichen Konzepten gefolgt.

Zunächst ging es darum, einen Treiber soweit in der UML zu beschreiben, dass ein passendes Entwicklungswerkzeug aus diesem Modell den gesamten benötigten Quellcode generieren kann. Hier kam es zu einem gespaltenen Ergebnis. Es zeigte sich, dass es sehr gut möglich ist, die allgemeinen Strukturen des Linuxkernels und eines Treibers in einem UML-Klassendiagramm darzustellen. Die Modellierung des Funktionskörpers in einer Art und Weise, die einen Vorteil gegenüber der direkten Programmierung aufweisen würde, ist jedoch nicht möglich.

In dem zweiten Ansatz geht es darum, die modellbasierte Entwicklungsmethodik zu nutzten, um eine auf den speziellen Treiber angepasste Schnittstelle für Benutzerprogramme zu ermöglichen und diese automatisch zu generieren. Die dafür von mir erstellte Implementierung zeigt die gute Machbarkeit dieses Vorhabens. Dieser Ansatz ließe sich vor allem in eingebetteten Systemen gut nutzen, um die Ansteuerung der gerätespezifischen Hardwarekomponenten zu vereinfachen.

Literaturverzeichnis

[Booch u. a. 1999] BOOCH, Grady ; RUMBAUGH, Jim ; JACOBSON, Ivar: *Das UML-Benutzerhandbuch*. 1. Auflage. Addison-Wesley-Longman, 1999

[Corbet u. a. 2005] CORBET, Jonathan ; RUBINI, Alessandro ; KROAH-HARTMAN, Greg ; ORAM, Andy (Hrsg.): *Linux Device Drivers*. 3rd Editon. O'Reilly, 2005

[Fowler 2004] FOWLER, Martin: *UML konzentriert*. 3. Auflage. Addison-Wesley Verlag, 2004

[Gleditsch 2011] GLEDITSCH, Arne G.: *The Linux Cross Reference*. 2011. – URL `http://lxr.linux.no/#linux+v2.6.38/`. – Abgerufen am 06.07.2011

[Quade und Kunst 2006] QUADE, Juergen ; KUNST, Eva-Katherina: *Linux-Treiber entwickeln*. 2. aktualisierte Auflage. dpunkt.verlag, 2006

[Quade und Kunst 2011] QUADE, Juergen ; KUNST, Eva-Katherina: *Quellcode zum Buch "Linux-Treiber entwickeln", 3. Auflage*. 2011. – URL `https://ezs.kr.hsnr.de/TreiberBuch/Download/TreiberEntwickeln2011068.tgz`. – Abgerufen am 02.07.2011

Anhang A

Inhalt der CD

Teil dieser Arbeit ist ein Anhang in elektronischer Form, der auf der beliegenden CD enthalten ist. Die Tabelle A.1 listet die enthaltenen Dateien mit meiner kurzen Beschreibung auf.

Datei	Beschreibung
./Konzept 1/5-28-drvrtemplate.c	Das in Kapitel 3 verwendete Treiber Muster für allgemeine Treiber, Quelle: Quade und Kunst (2011)
./Konzept 1/8-1-blockdevice.c	Das in Kapitel 3 verwendete Treiber Muster für eine RAM-Disk, Quelle: Quade und Kunst (2011)
./Konzept 2/BacProxyGenCustom v0.zip	Das Artisan Studio Modell, das den im Kapitel 4 beschriebenen Generator implementiert.
./Konzept 2/libXRay.c	Die generierte Userspacebibliothek.
./Konzept 2/libXRay.h	Die generierte Userspacebibliothek (Headerdatei).
./Konzept 2/Makefile	Makefile zur Kompilierung der beiden Kernelmodule, Quelle: Quade und Kunst (2011)
./Konzept 2/XRay.c	Das zum Testen erstellte Kernelmodul, welches die Schnittstelle bereitstellt.
./Konzept 2/XRayProxy.c	Das generierte Proxymodul.

Tabelle A.1: Der Inhalt der beiliegenden CD

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 21. Juli 2011
Ort, Datum

Unterschrift