FOM Hochschule für Oekonomie & Management

Studienzentrum München

Bachelor-Thesis

zur Erlangung des Grades eines

Bachelor of Science (B.Sc.)

über das Thema

Implementation eines FAT-Treibers für das Lehrbetriebssystem ULIX

von

Simon Brugger

Erstgutachter	Dr.-Ing. Hans-Georg Eßer
Matrikelnummer	240612
Abgabedatum	2016-01-18

Inhaltsverzeichnis

Abkürzungsverzeichnis

BPB BIOS Parameter Block

CLN Clusternummer

DR-DOS Digital Research Disk Operating System

EOF End Of File

FAT File Allocation Table

LFD Lokaler Dateidesktiptor

MS-DOS Microsoft Disk Operating System

NTFS New Technology File System

UC Use Case

VFS Virtuelles Dateisystem

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Betriebssysteme spielen eine zentrale Rolle in der Informatik. UNIX-ähnliche Betriebssysteme wie Linux oder Mac OS X, aber auch die Windows-Betriebssysteme, übernehmen im Grunde zwei Hauptaufgaben: die Interaktion mit dem Benutzer auf der Anwendungsebene und die Bereitstellung und Verwaltung von Hardwareressourcen (vgl. Tanenbaum, 2009, S. 33). Ein Betriebssystem ist eine sehr komplexe Software. Mit Hilfe von Lehrbetriebssystemen erhalten Studenten einen Einblick in die Funktionsweise von Betriebssystemen. Zentraler Gegenstand dieser Bachelor-Thesis ist die Implementierung eines FAT-Dateisystemtreibers für das Lehrbetriebssystem ULIX. Dieser FAT-Dateisystemtreiber bildet eine zusätzliche Betriebssystemkomponente zur Vermittlung der Funktionsweise von FAT-Dateisystemen. Dieses Kapitel geht zuerst auf die Problemstellung ein, führt an den Untersuchungsgegenstand heran und grenzt diesen ab. Weiter werden das Vorgehen und die angewandten Forschungsmethoden beleuchtet. Zum Schluss wird der Aufbau der Arbeit beschrieben.

1.1. Problemstellung

Das Betriebssystem ULIX wird zur Zeit von Hans-Georg Eßer und Felix Freiling an der Universität Erlangen-Nürnberg entwickelt. Das UNIX-ähnliche ULIX ist ein Lehrbetriebssystem mit dem Ziel die Vermittlung von Betriebssystemprinzipien zu fördern. Auch andere Lehrbetriebssysteme, wie zum Beispiel Minix, verfolgen diesen Ansatz.

ULIX ist das erste Betriebssystem, das komplett in *Literate Programming* geschrieben wurde, mit dem Ziel herauszufinden, ob und wie durch ein in *Literate Programming* implementiertes Betriebssystem die Vermittlung von Betriebssystemprinzipien gefördert werden kann. ULIX unterstützt in der Version 0.10 als einziges Dateisystem Minix. Eine Anforderung der finalen ULIX-Version ist die zusätzliche Unterstützung des FAT-Dateisystems. Der FAT-Dateisystemtreiber für ULIX bildet den *Untersuchungsgegenstand* dieser Arbeit. Vor der Implementierung müssen folgende Grundlagen erarbeitet werden:

1. Welche Eigenschaften hat FAT und welche Varianten werden unterschieden?
2. Wie verhält sich FAT zur UNIX Dateisemantik?
3. Welche Anforderungen hat der FAT-Dateisystemtreiber?
4. Wie ist die Integration eines neuen Dateisystemtreibers in den ULIX-Kernel möglich?

Die *Untersuchungsperspektive* ist die Implementierung des FAT-Dateisystemtreibers in der Programmiersprache C und der Methode *Literate Programming*. Dabei liegt der Fokus auf der grundlegenden Funktionsweise von FAT und der Integration in ULIX 0.10. Daraus ergibt sich folgendes *Ziel der Arbeit*: Implementierung eines FAT-Dateisystemtreibers für das Lehrbetriebssystem ULIX 0.10. Somit trägt diese Komponente zum *übergeordneten Ziel* bei: Der Vermittlung der Funktionsweise von FAT-Dateisystemen an Studenten.

1.2. Abgrenzung und Einordnung des Untersuchungsgegenstandes

Das Lehrbetriebssystem ULIX soll um das FAT-Dateisystem erweitert werden. Das FAT-Dateisystem kennt gegenwärtig die Varianten FAT12 (1977), FAT16 (1988) und FAT32 (1996) (vgl. Baumgarten & Siegert, 2007, S. 198). Die Implementierung im Rahmen dieser Arbeit konzentriert sich auf FAT12.

Dabei geht es um die grundlegende Realisierung des FAT-Konzeptes: Die Initialisierung des Dateisystems, Dateien lesen und schreiben und den Umgang mit FAT-Tabelleneinträgen. Weiter muss der Treiber in ULIX integriert werden. Ebenfalls die Interoperabilität zwischen Dateisystemen muss der Treiber unterstützen. Darunter das Kopieren von Dateien eines anderen Dateisystems auf das FAT-Dateisystem, sowie unter ULIX veränderte Image-Dateien müssen unter Linux gemountet und verarbeitet werden können.

1.3. Forschungsmethodik und Vorgehen

Da durch die Implementierung eines FAT-Dateisystemtreibers in *Literate Programming* für ULIX eine neue, für sich alleinstehende IT-Komponente entwickelt wird, bietet es sich an, diese Arbeit entsprechend der Forschungsmethodik Design-Science anzufertigen. Die sieben Richtlinien der Design-Science-Methodik bilden die Grundlage zur Erlangung wissenschaftlicher Erkenntnisse bei Informationssystemen (vgl. Hevner et al., 2004, S. 77). Die Struktur dieser Arbeit wird den sieben Richtlinien nach der Design-Science-Methodik (Abbildung 1.1) zugeordnet.

1. Richtlinie: Design als Komponente (vgl. Hevner et al., 2004, S. 82)

   Der FAT-Dateisystemtreiber wird als eigene Komponente bzw. *Untersuchungsgegenstand* implementiert.

2. Richtlinie: Problemrelevanz (vgl. Hevner et al., 2004, S. 84)

   Aktuelle Arbeiten auf dem Gebiet der Betriebssystemlehre zeigen, dass ein Lehrbetriebssystem, welches komplett in *Literate Programming* implementiert wurde, eine Forschungslücke darstellt. ULIX besitzt die Anforderung, FAT als weiteres Dateisys-

Guideline	Description
Guideline 1: Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation.
Guideline 2: Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems.
Guideline 3: Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods.
Guideline 4: Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies.
Guideline 5: Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact.
Guideline 6: Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment.
Guideline 7: Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences.

Abbildung 1.1.: Sieben Design-Science Richtlinien zur Erlangung wissenschaftlicher Erkenntnisse (vgl. Hevner et al., 2004, S. 83)

tem zu unterstützen. Daraus ergibt sich die Relevanz eines in *Literate Programming* geschriebenen FAT-Dateisystemtreibers.

3. Richtlinie: Artefakt-Evaluierung (vgl. Hevner et al., 2004, S. 85)

   Nach der Implementierung folgt eine Testphase, in welcher die Funktionalität der Komponente ausführlich getestet wird. Dabei werden die Analytical Design Evaluations Methods (Architecture Analysis und Static Analysis) nach Design Science angewendet. Die Implementierung bezieht sich auf die ULIX-Version 0.10. Die Evaluation des übergeordneten Zieles und der finalen Version von ULIX erfolgte im Rahmen der Dissertation von Herrn Eßer an der Technischen Hochschule Nürnberg (vgl. Eßer, 2015, S. 144) und an den FOM-Standorten München und Nürnberg (vgl. Eßer, 2015, S. 161).

4. Richtlinie: Beitrag zur Forschung (vgl. Hevner et al., 2004, S. 87)

   Der FAT-Dateisystemtreiber wird zum ersten Mal in *Literate Programming* implementiert und bildet somit einen Beitrag zur Forschung sowie zur übergeordneten Zielsetzung.

5. Richtlinie: Richtigkeit der Forschung (vgl. Hevner et al., 2004, S. 87)

   Die Evaluierung und Implementierung der Komponente FAT-Dateisystemtreiber werden nach anerkannten Methoden durchgeführt. Darüber hinaus liegt der Fokus auf dem Kontext der übergeordneten Arbeit und der Integration in das Lehrbetriebssystem ULIX mit Hilfe von *Literate Programming*.

6. Richtlinie: Design als Suchprozess (vgl. Hevner et al., 2004, S. 88)

   Durch einen iterativen Softwareentwicklungsprozess wird der FAT-Dateisystemtreiber in mehreren Entwicklungszyklen implementiert. Dadurch werden Probleme frühzeitig erkannt und die Zielerreichung kontinuierlich überprüft.

7. Richtlinie: Kommunikation der Forschung (vgl. Hevner et al., 2004, S. 90)

   Der FAT-Dateisystemtreiber wird im Rahmen einer Bachelor-Thesis beschrieben und implementiert. Die Arbeit wird nach Fertigstellung im Rahmen eines Kolloquiums präsentiert.

1.3.1. Methodik der Literaturrecherche

Das Vorgehen der Literaturrecherche basiert auf der Arbeit *Reconstructing the Giant* (vgl. vom Brocke et al., 2009). Zur Erarbeitung der oben genannten Grundlagen existiert eine Vielzahl an Fachbüchern und Spezifikationen. Die Anzahl der Google-Scholar-Zitate (>1000) diente als Hilfestellung zur Eingrenzung relevanter Literatur.

Die Dissertation von Herrn Eßer dient als übergeordnetes Werk zur Einarbeitung in das Lehrbetriebssystem ULIX. Der bereits in *Literate Programming* geschriebene Sourcecode dient zur Einarbeitung in die Implementierung. Die mit Hilfe einer MindMap definierten Schlagwörter dienten als Grundlage zur Recherche in Meta-Suchmaschinen. Grundsätzlich wurde bei der Literaturrecherche das Vorwärts- und Rückwärts-Suchverfahren angewandt (vgl. Webster & Watson, 2002).

Tabelle 1.1 gibt einen Überblick über die Suchergebnisse aus den Suchmaschinen. Verwendete Suchbegriffe: „fat filesystem“, „unix“, „literate programming“, „filesystems“, „filesystem driver“, „filesystem protocol“, „virtual filesystem“.

1.3.2. Softwareentwicklungsmethode

Die Implementierung erfolgt nach einem iterativen Softwareentwicklungsprozess. Ausgehend von einem Prototyp und dem Grobentwurf folgen Feinentwurf, Implementierung und Tests. Diese Phasen werden iterativ durchlaufen und bei jedem Iterationsdurchlauf verfeinert. Somit können die Zielerreichung und die Qualität kontinuierlich überprüft werden (vgl. Kleuker, 2013, S. 31).

Suchmaschine	Suche	Hits	Gesichtet
Springerlink	All fields	90	<10
IEEE Xplore	All fields	1.003	<10
ACM Digital Library	All fields	1.145	>20
SienceDirect	All fields	639	>50
Google Scholar	All fields	1.522	>100

Tabelle 1.1.: Ergebnis der Literaturrecherche

1.3.3. Literate Programming

ULIX wurde vollständig in *Literate Programming* implementiert. Der FAT-Dateisystemtreiber dieser Arbeit wird ebenfalls in *Literate Programming* implementiert, um nicht mit der Methodik zu brechen. Während sich traditionelle Programmiermethoden darauf konzentrieren, dem Computer Instruktionen zu geben, liegt der Fokus von *Literate Programming* darauf, dem Menschen zu vermitteln was, mit dem Computer gemacht wird (vgl. Knuth, 1984, S. 97).

Mit `WEB` führt Knuth eine Programmsammlung ein, die es dem Programmierer ermöglicht, ausführbaren Programmcode und die dazugehörige Dokumentation in einem Dokument zu schreiben. Darüber hinaus bietet `WEB` dem Programmierer die Möglichkeit, eine komplexe Software in kleine, verständliche Probleme zu unterteilen und die einzelnen Programmteile nach menschlichem Verständnis so zu strukturieren, beschreiben und zu ordnen (vgl. Knuth, 1984, S. 107).

`WEB` sorgt mit den Routinen „Weaving“ und „Tangling“ dafür, dass zum einen eine Dokumentation erzeugt wird und zum anderen der Programmcode so zusammengesetzt wird, dass eine kompilierbare Datei entsteht (vgl. Knuth, 1984, S. 97-98). `WEB` kombiniert dabei Textauszeichnungssprachen mit Programmiersprachen.

`noweb` ist die Literate-Programming-Toolsammlung, mit der diese Arbeit und auch ULIX implementiert wurden. Das von Norman Ramsey entwickelte `noweb` ist eine Variante von Donald E. Knuth's `WEB`. Programmcode und Dokumentation werden in sog. Chunks geschrieben. Ein `noweb`-Dokument besteht aus einer Reihe von Chunks. Code-Chunks werden mit ⟨*chunk name* (never defined)⟩= eingeleitet und können beliebige Reihenfolgen haben. Das Programm `notangle` sorgt u.a. für eine korrekte Code Reihenfolge und fasst Chunks mit gleichem Namen zusammen. Dokumentations-Chunks werden mit einem `@` eingeleitet. Im Gegensatz zu `WEB` lassen sich die Dokumentationen mit LaTeX formatieren. Das `noweave`-Programm erzeugt die Dokumentation, Indizierung und Referenzierung der Chunks (vgl. Ramsey, 1994, S. 97-100).

`notangle` extrahiert die Dateien `module.c` und `module.h` aus diesem Dokument. Diese Dateien werden im Build-Prozess zusammen mit den ULIX Sourcode-Dateien verlinkt.

1.4. Aufbau der Arbeit

In Kapitel 2 geht die Arbeit zuerst auf die Grundlagen von Dateisystemen und im speziellen auf die FAT-Dateisysteme ein. Dabei werden die unterschiedlichen FAT-Varianten betrachtet. Darauf folgen in Kapitel 3 die Aufnahme und Auswertung der Anforderungen sowie der Entwurf und die Konzeption des FAT-Dateisystemtreibers. Anschließend folgt in Kapitel 4 die Implementierung des Treibers in *Literate Programming*. Kapitel 5 dokumentiert die funktionalen Tests der Implementierung. Es folgen zum Schluss in Kapitel 6 eine Zusammenfassung, ein Fazit und eine kritische Würdigung der Ergebnisse.

2. Grundlagen

Computer speichern Informationen als Dateien auf verschiedenen Speichermedien wie Festplatten, USB Sticks, CDs, etc. Dateien repräsentieren persistente Informationseinheiten, welche durch Dateisysteme verwaltet werden (vgl. Tanenbaum, 2009, S. 319).

Das vorliegende Kapitel behandelt die Grundlagen der FAT-Dateisysteme. Dabei wird zuerst auf die unterschiedlichen Varianten eingegangen. Im Anschluss wird die FAT-Architektur beschrieben und die einzelnen Regionen eines FAT-Dateisystems anhand von Beispielen detailliert erklärt. Zum Schluss wird erklärt wie FAT-Dateneinheiten (Cluster) gebildet werden.

2.1. Das FAT-Dateisystem

Das FAT-Dateisystem hat seinen Ursprung in den späten 70er und frühen 80er Jahren. Es wurde von Microsoft für das MS-DOS Betriebssystem entwickelt (vgl. Microsoft, 2000, S. 1).

Über die Jahre entstand eine ganze FAT-Dateisystem-Familie, mit unterschiedlichen Derivaten und Erweiterungen. Das NTFS-Dateisystem, welches durch Microsoft mit Windows NT 3.1 eingeführt wurde, gilt als Nachfolger der FAT-Dateisysteme. Obwohl FAT kein Standard mehr ist, wird FAT von einer Vielzahl von Betriebssystemen unterstützt und ist weit verbreitet in embedded Systemen wie zum Beispiel MP3-Playern, Digital Kameras und USB-Sticks (vgl. Tanenbaum, 2009, S. 383). Die Gründe dafür sind u.a. dass das FAT-Dateisystem robust, relativ einfach und leicht zu implementieren ist (vgl. Microsoft, 2010b).

Das grundlegende Konzept eines FAT-Dateisystems beruht darauf, dass Dateien und Verzeichnisse durch eine Datenstruktur den sog. Verzeichniseinträgen repräsentiert werden. Verzeichniseinträge enthalten Informationen wie Name, Dateigröße, das erste Start-Cluster der Datei und weitere Metainformationen. Der Inhalt der Dateien ist in gleichen Einheiten sog. Clustern gespeichert. Ein Verzeichnis oder eine Datei kann ein oder mehrere Cluster belegen. Die Beziehung zwischen Verzeichniseinträgen und den Inhalten wird über eine dritte Datenstruktur, der FAT definiert. Über die FAT können alle zu einer Datei gehörenden Cluster identifiziert werden (vgl. Carrier, 2005, S. 156 - 157).

Auch wenn das FAT-Dateisystem robust und einfach ist, bringt dieses Konzept auch Nachteile mit sich. Das Schreiben und Löschen von Dateien führt zur Belegung neuer Cluster, welche nicht zusammenhängend geschrieben werden. Das verursacht eine Frag-

mentierung des Dateisystems. Die Fragmentierung hat starken Einfluss auf die Performance. Beim vollständigen sequentiellen Lesen einer Datei wird die Cluster-Kette von Anfang bis Ende durchlaufen. Sind die einzelnen Cluster auf dem Datenträger verteilt, erfordert das viele Plattenkopf Bewegungen (vgl. Friedman & Pentakalos, 2002, S. 423).

Ein weiteres Problem ist die interne Fragmentierung. Die kleinste adressierbare Einheit eines FAT-Dateisystems ist ein Cluster. Wenn Dateien keine vollständigen Cluster belegen, führt das zu ungenutzten Speicherverbrauch. So würde eine 65 KB Datei, welche sich auf einer 3 GB, FAT16-Partition befindet, bei einer Cluster Größe von 64 KB, volle zwei Cluster belegen und somit blieben 63 KB ungenutzt (vgl. Friedman & Pentakalos, 2002, S. 424).

2.2. Die FAT-Varianten

Grundsätzlich existieren drei verschiedene Varianten des FAT-Dateisystems FAT12, FAT16 und FAT32. Alle Versionen haben gemeinsam das sie eine FAT verwenden, um Daten zu organisieren und zu verwalten. Die FAT ist das Herzstück des Dateisystems und kann auch mehrfach auf dem Dateisystem abgelegt sein. Das FAT-Dateisystem funktioniert nach dem Prinzip einer verketteten Allokation. Im Gegensatz zur zusammenhängenden Allokation bei der die Daten am Stück auf den Datenträger abgelegt werden, sind bei der verketteten Allokation Daten in Clustern verteilt abgelegt.

FAT12 Das Dateisystem des IBM-PC 1981 war FAT12 und ist heute noch auf allen FAT-formatierten 3,5 Zoll-Disketten im Einsatz. FAT wurde ursprünglich für kleine Datenträger und einfache Ordnerstrukturen entworfen (vgl. Microsoft, 2010c). FAT12 verwendet 12 Bit für die Cluster Adressierung. Damit lassen sich maximal $2^{12} = 4.096$ Cluster adressieren. Die Cluster können eine Größe von 512 Byte bis 4.069 Byte besitzen. In der ursprünglichen Variante sind Datei- und Verzeichnisnamen auf das 8.3-Format beschränkt. Das *Root-Verzeichnis* ist limitiert auf eine Größe von 14 Cluster. Dadurch ist auch die Anzahl der Einträge im *Root-Verzeichnis* auf 224 begrenzt, wodurch hingegen im Unterverzeichnis beliebig viele Einträge erstellt werden können (vgl. Wikipedia, 2015).

FAT16 Mit der Zeit wuchsen die Festplattenkapazitäten, so dass ein größerer Adressraum notwendig wurde. Die nächste Variante, FAT16 unterscheidet sich zu FAT12 im Wesentlichen in der Anzahl der zur Datenadressierung verwendeten Bits. Mit 16 Bits lassen sich unter FAT16 maximal $2^{16} = 65.536$ Cluster adressieren. Davon sind 12 Cluster reserviert. Damit sind Partitionsgrößen von 2 GB (auf DOS basierende Windows 9x) und 4 GB (Windows NT, Enhanced DR-DOS mit einer Clustergröße von 64 KB) möglich (vgl. Wikipedia, 2015).

FAT32 Unter Windows 95B wurde 1996 FAT32 eingeführt. FAT32 verwendet 32 Bit zur Addressierung wobei 4 Bit reserviert sind. Somit können $2^{28} = 268.435.456$ Cluster

adressiert werden. Neben der Adressierung größerer Datenträger (maximal 8 TB) unterscheidet sich FAT32 auch im Layout des Dateisystems zu den anderen Varianten. So wurde der BPB erweitert, damit zusätzliche Dateisysteminformationen abgelegt werden können, siehe Kapitel 2.4.1. Eine neue Datenstruktur, *File-System-Information-Sector* wurde eingeführt und das *Root-Verzeichnis* befindet sich nicht wie bei FAT12/16 an einer definierten Position hinter den FATs, sondern in der *Datenregion*. Dadurch kann es beliebig groß werden (vgl. Wikipedia, 2015). Da FAT32 nicht Bestandteil dieser Arbeit ist, wird an dieser Stelle nicht weiter auf die Unterschiede zwischen FAT12/16 und FAT32 eingegangen.

2.3. Die FAT-Derivate

Neben den oben beschriebenen, grundsätzlichen FAT-Varianten existieren noch einige Derivate und Erweiterungen. Nachfolgend zwei Beispiele von FAT-Derivaten.

TFAT Mit TFAT wurde ein transaktionssicheres FAT-Dateisystem implementiert. Dabei arbeitet TFAT immer mit zwei FATs. Eine Arbeitskopie und eine mit dem letzten stabilen Stand. Änderungen in der Arbeitskopie werden erst freigegeben, wenn alle Transaktionen getätigt wurden. Das macht das Dateisystem deutlich robuster gegenüber unvorhergesehenen Unterbrechungen bei kritischen Aktionen. Die theoretische Dateisystemgrenze liegt bei 2 TB mit 512 Byte Clustern. Dateisysteme mit bis zu 500 GB wurden getestet (vgl. Microsoft, 2010d).

exFAT Das Extended FAT-Dateisystem richtet sich an die zukünftigen Anforderungen Mobiler Datenspeicherung. Dabei löst es nicht nur alte Beschränkungen wie z. B. das 4 GB Dateigrößen Limit, sondern bietet neben besserer Performance, die Verwaltung von Datenträgern mit einer Größe von bis zu 64 ZB (theoretisch). Grundsätzlich steht die Interoperabilität zwischen Mobilen- und Desktop-Geräten im Mittelpunkt von exFAT (vgl. Microsoft, 2010a). Zu exFAT existiert auch eine transaktionssichere Version TexFAT. TexFAT arbeitet wie TFAT mit zwei Tabellen, um die Transaktionssicherheit zu realisieren, basiert aber auf exFAT und nicht wie TFAT auf FAT (vgl. Microsoft, 2015b).

2.4. Die Architektur des FAT12 Dateisystems

Untersuchungsgegenstand dieser Arbeit ist das FAT12/16 Dateisystem. Nachfolgend wird im Detail ausschließlich auf die Datenstrukturen der FAT12/16 Dateisysteme eingegangen.

Das FAT-Dateisystem ist grundlegend in vier Regionen eingeteilt, siehe Abbildung 2.1.

1. **Reservierte Region:** Jedes FAT-Dateisystem beginnt mit einem reservierten Bereich. Der reservierte Bereich enthält den BPB. Bei bootfähigen Datenträgern kann dieser Bereich auch den Boot-Loader-Code enthalten.

Abbildung 2.1.: Aufbau eines FAT-Dateisystems. Die Bereiche FAT 1 und FAT 2 werden unter dem Begriff FAT-Region zusammengefasst

2. **FAT-Region:** Die nächsten Datenstrukturen sind die FATs. Normalerweise führen FAT-Dateisysteme zwei identische FATs, welche bei FAT12/16 direkt nach der *Reservierten Region* beginnen. Die Anzahl der FATs kann variieren.

3. **Root-Verzeichnis:** Das *Root-Verzeichnis* enthält die Verzeichniseinträge auf oberster Dateisystemebene.

4. **Datenregion:** Bereich der Daten-Cluster. In diesem Bereich werden die Dateiinhalte gespeichert.

Alle FAT-Dateisysteme wurden ursprünglich für die IBM-PC Architektur entworfen. Aus diesem Grund werden in FAT alle Datenstrukturen als *little endian* gespeichert (vgl. Microsoft, 2000, S. 7). Bei *little endian* wird das kleinstwertige Byte an der Anfangsadresse gespeichert und alle folgenden Bytes der Reihe nach aufsteigend abgelegt. So wird zum Beispiel die Hexzahl `ABCD` im Speicher als `CD AB` abgelegt. Zeichenketten werden dabei nicht in umgekehrter Reihenfolge gespeichert (vgl. Saltzer & Kaashoek, 2009, S. 158).

In den nachfolgenden Kapiteln, wird anhand eines Beispiels einer FAT12 formatierten Diskette, näher auf die einzelnen Regionen eingegangen.

2.4.1. Reservierte Region und BPB

Im ersten Sektor der *Reservierten Region* befindet sich der BPB. Manchmal wird der BPB auch Boot-Sektor oder Sektor 0 genannt. Er befindet sich immer an erster Stelle eines jedem FAT-Dateisystems (vgl. Microsoft, 2000, S. 7). Der BPB enthält grundlegende Informationen zum Dateisystem. Mit Hilfe der Informationen aus dem BPB lassen sich zum Beispiel die genauen Positionen der Regionen auf dem Datenträger ermitteln oder die entsprechende FAT-Variante bestimmen. Der Aufbau des BPB unterscheidet sich zwischen FAT12/FAT16 und FAT32 ab dem Offset 36, abhängig vom verwendeten Medium. Die Standardisierung der ersten BPB-Felder vereinfacht die Entwicklung kompatibler FAT-Dateisystemtreiber (vgl. Microsoft, 2000, S. 8).

Folgender *Hexdump* zeigt einen BPB des in dieser Arbeit verwendeten FAT12-Dateisystems:

```
$ hexdump -C -n512 floppy.img
00000000  eb 3c 90 6d 6b 66 73 2e  66 61 74 00 02 01 01 00  |.<.mkfs.fat.....|
00000010  02 e0 00 40 0b f0 09 00  12 00 02 00 00 00 00 00  |...@............|
00000020  00 00 00 00 00 00 29 55  0e 3c a5 55 4c 49 58 46  |......)U.<.ULIXF|
00000030  41 54 20 20 20 20 46 41  54 31 32 20 20 20 0e 1f  |AT    FAT12   ..|
```

```
00000040  be 5b 7c ac 22 c0 74 0b  56 b4 0e bb 07 00 cd 10  |.[|."t.V.......|
00000050  5e eb f0 32 e4 cd 16 cd  19 eb fe 54 68 69 73 20  |^..2.......This |
00000060  69 73 20 6e 6f 74 20 61  20 62 6f 6f 74 61 62 6c  |is not a bootabl|
00000070  65 20 64 69 73 6b 2e 20  20 50 6c 65 61 73 65 20  |e disk.  Please |
00000080  69 6e 73 65 72 74 20 61  20 62 6f 6f 74 61 62 6c  |insert a bootabl|
00000090  65 20 66 6c 6f 70 70 79  20 61 6e 64 0d 0a 70 72  |e floppy and..pr|
000000a0  65 73 73 20 61 6e 79 20  6b 65 79 20 74 6f 20 74  |ess any key to t|
000000b0  72 79 20 61 67 61 69 6e  20 2e 2e 2e 20 0d 0a 00  |ry again ... ...|
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa  |..............U.|
00000200
```

Tabelle 2.1 schlüsselt die einzelnen Daten der BPB-Felder nach Microsoft Spezifikation (vgl. Microsoft, 2000, S. 9-10) auf.

BPB Feld	**Offset**	**Größe**	**Beschreibung**
`BS_jmpBoot`	0	3	Sprungmarke zum Bootcode welcher nach dem BPB den Rest von Sektor 0 belegen kann. In diesem Fall `0xeb 0x3c 0x90`, was übersetzt bedeutet JMP 003C NOP. `0xeb` gibt an, dass das nächste Byte die Position des Bootcodes bestimmt. `0x90` steht für no operation. Beginnt dieses Feld mit `0xe9`, geben die nächsten 2 Bytes die Position an (vgl. Microsoft, 2000, S. 9).
`BS_OEMName`	3	8	Beschreibt das System mit dem das Dateisystem formatiert wurde. In diesem Beispiel stehen `0x6d 0x6b 0x66 0x73 0x2e 0x66 0x61 0x74` für mkfs.fat (vgl. Microsoft, 2000, S. 9).
`BPB_BytsPerSec`	11	2	Je nach FAT-Version kann dieses Feld folgende Werte enthalten: 512, 1024, 2048 oder 4096. In diesem Fall handelt es sich um `0x00 0x02` in little endian Darstellung sprich `0x0200` = 512 Bytes pro Sektor (vgl. Microsoft, 2000, S. 9).
`BPB_SecPerClus`	13	1	Dieser Wert gibt an, wieviel Sektoren in einem Cluster zusammengefasst werden. Dieser Wert muss eine Potenz von 2 sein und > 0. Hier ist er `0x01`, was bedeutet dass ein Cluster nur aus einem Sektor besteht (vgl. Microsoft, 2000, S. 9).
`BPB_RsvdSecCnt`	14	2	Gibt an wie viele reservierten Sektoren vor der FAT existieren. Das Dateisystem hat einen reservierten Sektor `0x01`. Das ist bei FAT12/FAT16 der Standardwert (vgl. Microsoft, 2000, S. 9).

BPB Feld	Offset	Größe	Beschreibung
BPB_NumFATs	16	1	Das nächste Byte gibt an, wie viele FAT-Datenstrukturen vorhanden sind. Dieser Wert sollte für alle FAT-Versionen immer auf 2 stehen. Wie auch im vorliegenden Fall 0x02 = 2 FATs (vgl. Microsoft, 2000, S. 10).
BPB_RootEntCnt	17	2	Maximale Anzahl der 32-Byte-Einträge im *Root-Verzeichnis*. 0xe0 0x00 = 0x00e0 = 224 Einträge (vgl. Microsoft, 2000, S. 10).
BPB_TotSec16	19	2	Maximale Anzahl der vorhandenen Sektoren in allen Regionen: 0x40 0x0b = 0x0b40 = 2880 Sektoren (80 Tracks * 18 Sektoren * 2 Seiten) (vgl. Microsoft, 2000, S. 10).
BPB_Media	21	1	0xf0 steht für einen bestimmten Medium Typ. In Tabelle 2.2 werden die möglichen Medien Typen beschrieben. Dieser Eintrag muss identisch mit dem ersten Eintrag der FAT sein (vgl. Microsoft, 2000, S. 10).
BPB_FATSz16	22	2	Für FAT12/FAT16 Dateisysteme gibt dieser Wert an, wie viele Sektoren eine FAT belegt. 0x09 0x00 = 0x0009 = 9 Sektoren (vgl. Microsoft, 2000, S. 10).
BPB_SecPerTrk	24	2	Definiert die Sektoren pro Track im vorliegenden Fall 0x12 0x00 = 0x0012 = 18 Sektoren pro Track (vgl. Microsoft, 2000, S. 10).
BPB_NumHeads	26	2	Anzahl der Lese- und Schreibköpfe im Laufwerk. 0x02 0x00 = 0x0002 = 2 Köpfe (vgl. Microsoft, 2000, S. 10).
BPB_HiddSec	28	4	Anzahl der versteckten Sektoren. Da das Beispiel-Image zu einer Diskette gehört und somit keine Partitionen existieren, gibt es keine versteckten Sektoren: 0x00 0x00 0x00 0x00 (vgl. Microsoft, 2000, S. 10).
BPB_TotSec32	32	4	Die maximale Anzahl der vorhandenen Sektoren für FAT32. Diese Werte sind 0x00 0x00 0x00 0x00 da ein FAT12 Dateisystem vorliegt und somit BPB_TotSec16 gesetzt ist (vgl. Microsoft, 2000, S. 10).

Tabelle 2.1.: BPB Felder bis Offset 32

BPB_Media enthält Identifikationswerte für das genutzte Medium. Tabelle 2.2 listet mögliche Media Typen auf (vgl. Microsoft, 2015a).

Byte	Kapazität	Media-Größe und Typ
0xF0	2,88 MB	3,5 Zoll, zweiseitig, 36 Sektoren
0xF0	1,44 MB	3,5 Zoll, zweiseitig, 18 Sektoren
0xF9	720 KB	3,5 Zoll, zweiseitig, 9 Sektoren
0xF9	1,2 MB	5,25 Zoll, zweiseitig, 15 Sektoren
0xFD	360 KB	5,25 Zoll, zweiseitig, 9 Sektoren
0xFF	320 KB	5,25 Zoll, zweiseitig, 8 Sektoren
0xFC	180 KB	5,25 Zoll, einseitig, 9 Sektoren
0xFE	160 KB	5,25 Zoll, einseitig, 8 Sektoren
0xF8	—	Festplattenpartition

Tabelle 2.2.: Media Typen und ihre Eigenschaften

Ab Offset 36 unterscheiden sich FAT12/16-Dateisysteme von FAT32. Tabelle 2.3 beschreibt nach der Microsoft Spezifikation (vgl. Microsoft, 2000, S. 11), die Datenstruktur ab Offset 36 für FAT12-Dateisysteme.

BPB Feld	Offset	Größe	Beschreibung
`BS_DrvNum`	36	1	Bezieht sich auf die physikalische BIOS Laufwerksnummer. Floppy-Laufwerke werden ab `0x00` nummeriert, während Festplatten bei `0x80` beginnen. Da der Wert nur bei bootfähigen Datenträgern relevant ist, wird für Floppy-Laufwerke immer der Wert `0x00` gesetzt. Unabhängig wieviele Laufwerke existieren (vgl. Microsoft, 2000, S. 11).
`BS_Reserved1`	37	1	Reserviert für Windows NT `0x00`. Wird grundsätzlich auf 0 gesetzt (vgl. Microsoft, 2000, S. 11).
`BS_BootSig`	38	1	Erweiterte Boot Signatur. Enthält den Wert `0x29`. Gibt an, das die folgenden 3 Felder vorhanden sind (vgl. Microsoft, 2000, S. 11).
`BS_VolID`	39	4	Die 32 Bit Volume Number ist eine Seriennummer welche bei der Formatierung vergeben wird. Wird das Laufwerk erneut formatiert so wird eine neue Seriennummer gesetzt. Anhand der Seriennummer und des Laufwerks Labels (`BS_VolLab`), können FAT-Dateisystemtreiber erkennen ob die richtige Diskette eingelegt wurde. Die Seriennummer ist eine 32 Bit Kombination aus aktuellem Datum und Zeit `0x55 0x0e 0x3c 0xa5` (vgl. Microsoft, 2000, S. 11).
`BS_VolLab`	43	11	Laufwerksnamen welcher auch im Root-Verzeichins gespeichert wurde. `0x55 0x4c 0x49 0x58 0x46 0x41 0x54 0x20 0x20 0x20 0x20` steht in diesem Fall für `ULIXFAT` (vgl. Microsoft, 2000, S. 11).

BPB Feld	Offset	Größe	Beschreibung
BS_FilSysType	54	8	Kann folgende Werte enthalten: FAT12, FAT16 oder FAT. 0x46 0x41 0x54 0x31 0x32 0x20 0x20 0x20 = FAT12 (vgl. Microsoft, 2000, S. 11).

Tabelle 2.3.: BPB Felder ab Offset 36

Das FAT32-Dateisystem wird in dieser Arbeit nicht behandelt.

2.4.2. Die File Allocation Table

Nach der *Reservierten Region* folgen die FATs. FAT-Dateisysteme führen in der Regel zwei Tabellen. Die Anzahl der FATs kann allerdings variieren. Die genaue Anzahl wird im BPB definiert, siehe Tabelle 2.1. Redundante FATs dienen der Sicherheit (vgl. Friedman & Pentakalos, 2002, S. 422). So können Datenwiederherstellungsprogramme bei einer korrupten FAT1 auf FAT2 zurückgreifen. Ein FAT-Dateisystem speichert Dateiinhalte in Clustern. Die FAT definiert verkettete Listen von Clustern einer Datei. Der Startpunkt einer verketteten Liste wird im Verzeichniseintrag gespeichert. Von dort aus kann die gesamte Liste durchlaufen werden. Das Verfahren mit verketteten Listen verhindert, dass Speicherplatz durch Fragmentierung verloren geht. Allerdings ist der wahlfreie Zugriff extrem langsam da immer n-1 Blöcke durchlaufen werden müssen (vgl. Tanenbaum, 2009, S. 337). Die FAT zur Speicherung aller Cluster und ihrer verketteten Listen wächst proportional mit der Größe des Datenträgers. Besitzt ein Datenträger n Cluster Einträge, benötigt die FAT n Einträge (vgl. Tanenbaum, 2009, S. 340). Die FAT-Einträge selbst werden von 0 durchnummeriert, wobei die ersten beiden Einträge reserviert sind (vgl. Microsoft, 2000, S. 13).

Abbildung 2.2 zeigt die Funktionsweise einer FAT. Ein Verzeichniseintrag, zum Beispiel aus dem *Root-Verzeichnis*, enthält den ersten FAT-Eintrag. Im Beispiel oben verweist der Verzeichniseintrag, der Datei TWO.TXT auf Eintrag 76 der FAT. Dieser enthält den Eintrag

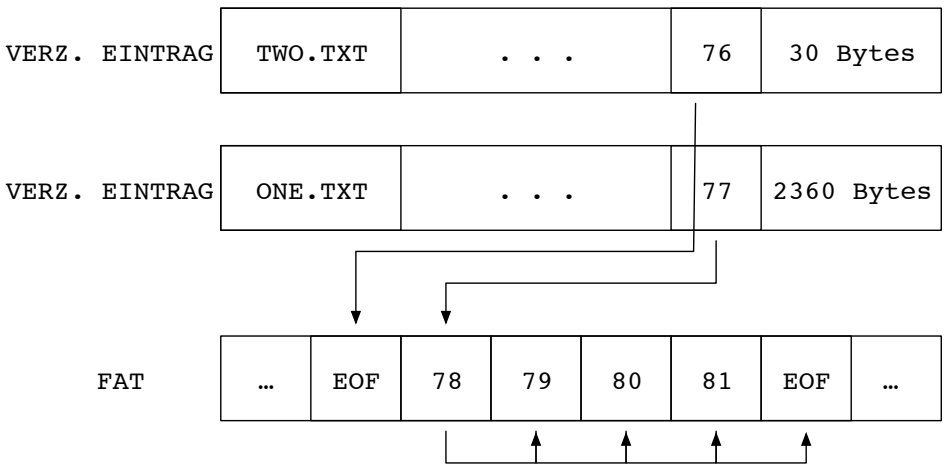


Abbildung 2.2.: Zwei Verzeichniseinträge mit unterschiedlich langen Cluster-Ketten.

EOF und somit belegt die Datei keine weiteren Cluster. Die Datei `TWO.TXT` beginnt und endet im Daten-Cluster 76.

Anders beim Beispiel `ONE.TXT`. Hier zeigt der Verzeichniseintrag auf den Eintrag 77 der FAT. Dieser wiederrum auf 78, dieser auf 79, usw. bis EOF erreicht ist. Die Datei `ONE.TXT` belegt somit die Daten-Cluster 77 bis 82.

Das erste adressierbare Cluster der FAT ist das Cluster 2. FAT-Eintrag 0 enthält eine Kopie des Media Deskriptors siehe Kapitel 2.4.1. FAT-Eintrag 1 enthält den *dirty* Status. Wurde das Dateisystem nicht korrekt ausgehängt (umount), zum Beispiel durch plötzliches Herunterfahren des Systems, kann das mit Hilfe des *dirty* Status festgestellt werden und ein Filesystem-Check gestartet werden. FAT12 verwendet diesen Status nicht und setzt pauschal immer ein EOF (vgl. Microsoft, 2000, S. 18).

Ein FAT12 Eintrag kann folgende Werte enthalten:

- `0x000` Freies Cluster, der Eintrag ist frei und kann allokiert werden.
- `0xFF7` Bad Cluster, fehlerhaftes Cluster. Fehlerhafte Cluster sollten nicht zu freien Clustern gezählt werden.
- `0xFF8 - 0xFFF` EOF, letzter Wert einer verketten Liste. Letztes Daten-Cluster einer Datei.
- `0x002 - 0xFF6` Allokiertes Cluster, Cluster ist belegt und einem Daten-Cluster zugeordnet.

2.4.3. Das Root-Verzeichnis

Im Grunde ist das *Root-Verzeichnis* nichts anderes als eine Auflistung 32-Byte Strukturen. Wie in Kapitel 2.4 beschrieben, hat das *Root-Verzeichnis* für FAT12/16 Dateisysteme, seinen festen Platz direkt hinter der letzten FAT. Weiter hat das *Root-Verzeichnis* eine feste Anzahl an Sektoren, welche im Boot Sektor definiert ist (vgl. Microsoft, 2000, S. 22).

Folgender *Hexdump* stellt einen Ausschnitt aus einem, in dieser Arbeit verwendeten FAT12 Dateisysteme dar.

```
$ hexdump -C -s 9728 -n 512  floppy.img
00002600  55 4c 49 58 46 41 54 20  20 20 20 08 00 00 91 bd  |ULIXFAT    .....|
00002610  71 45 71 45 00 00 91 bd  71 45 00 00 00 00 00 00  |qEqE....qE......|
*
00002620  41 66 00 6f 00 6c 00 64  00 65 00 0f 00 b1 72 00  |Af.o.l.d.e....r.|
00002630  00 00 ff ff ff ff ff ff  ff ff 00 00 ff ff ff ff  |................|
*
00002640  46 4f 4c 44 45 52 20 20  20 20 20 10 00 64 f1 b5  |FOLDER     ..d..|
00002650  71 45 71 45 00 00 f1 b5  71 45 1c 00 00 00 00 00  |qEqE....qE......|
00002660  41 6f 00 6e 00 65 00 2e  00 74 00 0f 00 53 78 00  |Ao.n.e...t...Sx.|
00002670  74 00 00 00 ff ff ff ff  ff ff 00 00 ff ff ff ff  |t...............|
00002680  4f 4e 45 20 20 20 20 20  54 58 54 20 00 64 d4 b5  |ONE     TXT .d..|
00002690  71 45 71 45 00 00 d4 b5  71 45 1b 00 0f 00 00 00  |qEqE....qE......|
```

```
000026a0  e5 6d 00 65 00 2e 00 74  00 78 00 0f 00 55 74 00  |.m.e...t.x...Ut.|
000026b0  2e 00 73 00 77 00 70 00  00 00 00 00 ff ff ff ff  |..s.w.p.........|
000026c0  e5 2e 00 6c 00 6f 00 6f  00 6f 00 0f 00 55 6e 00  |...l.o.o.o...Un.|
000026d0  67 00 66 00 69 00 6c 00  65 00 00 00 6e 00 61 00  |g.f.i.l.e...n.a.|
000026e0  e5 4f 4f 4f 4e 47 7e 31  53 57 50 20 00 00 0a b6  |.OOONG~1SWP ....|
000026f0  71 45 71 45 00 00 0a b6  71 45 00 00 00 00 00 00  |qEqE....qE......|
00002700  42 65 00 2e 00 74 00 78  00 74 00 0f 00 9a 00 00  |Be...t.x.t......|
00002710  ff ff ff ff ff ff ff ff  ff ff 00 00 ff ff ff ff  |................|
00002720  01 6c 00 6f 00 6f 00 6f  00 6e 00 0f 00 9a 67 00  |.l.o.o.o.n....g.|
00002730  66 00 69 00 6c 00 65 00  6e 00 00 00 61 00 6d 00  |f.i.l.e.n...a.m.|
00002740  4c 4f 4f 4f 4e 47 7e 31  54 58 54 20 00 00 13 b6  |LOOONG~1TXT ....|
00002750  71 45 71 45 00 00 13 b6  71 45 3e 00 1a 00 00 00  |qEqE....qE>.....|
00002760  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00002800
```

Die Tabelle 2.4 beschreibt nach der Microsoft Spezifikation (vgl. Microsoft, 2000, S. 23), einen 32-Byte Verzeichniseintrag, am Beispiel des oben erzeugten *Hexdump* eines FAT-Dateisystems.

Name	**Offset**	**Größe**	**Beschreibung**
`DIR_shortName`	0	11	Dateiname im 8.3-Format. Die Dateinamen `0x4f 0x4e 0x45 0x20 0x20 0x20 0x20` = ONE werden mit Leerzeichen `0x20` aufgefüllt, sollten sie kürzer als 8 Zeichen sein. Die drei Extension-Bytes `0x54 0x58 0x54` = TXT werden ebenfalls mit Lesezeichen aufgefüllt, sollten sie weniger Zeichen beinhalten (vgl. Microsoft, 2000, S. 23).
`DIR_attr`	11	1	Die Datei `one.txt` hat das Attribut `ATTR_ARCHIVE` (`0x20`). In Kapitel 2.4.3 werden die Dateiattribute detaillierter beschrieben (vgl. Microsoft, 2000, S. 23).
`DIR_reserv`	12	1	Reserviertes Byte für Windows NT. Beim Erstellen einer Datei wird dieses Byte auf `0x00` gesetzt (vgl. Microsoft, 2000, S. 23).
`DIR_cnto`	13	1	Da die Erstellungszeit, siehe `DIR_TimeCreate`, nur eine Genauigkeit von 2 Sekunden hat, repräsentiert dieser Wert die Millisekunden der Erstellungszeit. Zulässiger Bereich 0 <= `DIR_cnto` <= 199. Im Fall von `one.txt`: `0x64` (vgl. Microsoft, 2000, S. 23).
`DIR_TimeCreate`	14	2	Erstellungszeit `0xd4 0xb5` = 23:48:38 Uhr mit einer Genauigkeit von 2 Sekunden (vgl. Microsoft, 2000, S. 23).
`DIR_DateCreate`	16	2	Erstellungsdatum `0x71 0x45` = 17.11.2014 (vgl. Microsoft, 2000, S. 23).

Name	Offset	Größe	Beschreibung
DIR_DateLastAccess	18	2	Datum des letzten Zugriffs 0x71 0x45 = 17.11.2014. Der Wert wird bei jedem Zugriff gesetzt, sowohl beim Lesen als auch beim Schreiben. Unmittelbar nach einem Schreibzugriff identisch mit DIR_TimeLastMod (vgl. Microsoft, 2000, S. 23).
DIR_FirstClusterHi	20	2	High Word der ersten CLN. Für FAT12/16 ist der Wert immer 0x00 0x00 (vgl. Microsoft, 2000, S. 23).
DIR_TimeLastMod	22	2	Uhrzeit der letzten Dateimodifikation 0xd4 0xb5 = 23:48:38 Uhr. Der Wert wird bei jedem Zugriff gesetzt, sowohl beim Lesen als auch beim Schreiben. Unmittelbar nach einem Schreibzugriff identisch mit DIR_TimeCreate (vgl. Microsoft, 2000, S. 23).
DIR_DateMod	24	2	Datum der letzten Dateimodifikation 0x71 0x45 = 17.11.2014. Muss identisch mit DIR_DateCreate sein (vgl. Microsoft, 2000, S. 23).
DIR_FirstClusterLow	26	2	Low Word der ersten CLN 0x1b 0x00 (vgl. Microsoft, 2000, S. 23).
DIR_FileSize	28	4	Die letzten vier Bytes beschreiben die Dateigröße in Bytes 0x0f 0x00 0x00 0x00 = 15 Bytes (vgl. Microsoft, 2000, S. 23).

Tabelle 2.4.: FAT Verzeichniseintrag

Verzeichnisnamen

Eine besondere Bedeutung hat das erste Byte des Dateinamens DIR_shortName[0]. Folgende Werte sind Spezialfälle für das erste Byte in DIR_shortName:

- DIR_shortName[0] == 0xE5 der Verzeichniseintrag ist frei (vgl. Microsoft, 2000, S. 23).
- DIR_shortName[0] == 0x00 der Verzeichniseintrag ist frei und es gibt nach diesem Eintrag keine Weiteren (vgl. Microsoft, 2000, S. 23).
- DIR_shortName[0] == 0x05 Das erste Byte Zeichen ist eigentlich 0xE5. Da 0xE5 ein valides Zeichen im japanischen KANJI Zeichensatz ist, wurde 0x05 eingeführt, um den Eintrag nicht fälschlicherweise als frei zu kennzeichnen (vgl. Microsoft, 2000, S. 23).

Abgespeichert werden die Dateinamen in 8.3-Format. Dabei besteht der Dateinamen aus zwei Komponenten. Einem 8 Zeichen langen Namen und einer 3 Zeichen langen Dateien-

dung. Bei weniger Zeichen werden die beiden Komponenten mit Leerzeichen `0x20` aufgefüllt. Das . Zeichen zwischen Name und Dateierweiterung wird nicht in `DIR_shortName` abgelegt. Lower Case Zeichen und folgende Zeichen sind nicht zulässig in einem Verzeichnisnamen: `0x22`, `0x2A`, `0x2B`, `0x2C`, `0x2E`, `0x2F`, `0x3A`, `0x3B`, `0x3C`, `0x3D`, `0x3E`, `0x3F`, `0x5B`, `0x5C`, `0x5D`, `0x7C` und alle Zeichen niedriger als `0x20` (vgl. Microsoft, 2000, S. 24). Beispiele für Dateinamen im 8.3-Format in Tabelle 2.5.

Dateiname	**Inhalt** `DIR_shortName`
`one.txt`	`"ONE␣␣␣␣␣TXT"`
`ONE.TXT`	`"ONE␣␣␣␣␣TXT"`
`One.txt`	`"ONE␣␣␣␣␣TXT"`
`one`	`"ONE␣␣␣␣␣␣␣␣"`
`one.`	`"ONE␣␣␣␣␣␣␣␣"`
`one.tx`	`"ONE␣␣␣␣␣TX␣"`
`oneoneoe.txt`	`"ONEONEOETXT"`

Tabelle 2.5.: Dateinamen im 8.3-Format

Dateiattribute

Dateiattribute werden direkt am Verzeichniseintrag gespeichert. Die Feldlänge für Dateiattribute beträgt 1 Byte. Jedes Bit repräsentiert eine Dateieigenschaft. Somit ist auch eine Kombination aus Dateiattributen möglich. Die oberen 2 Bits sind reserviert.

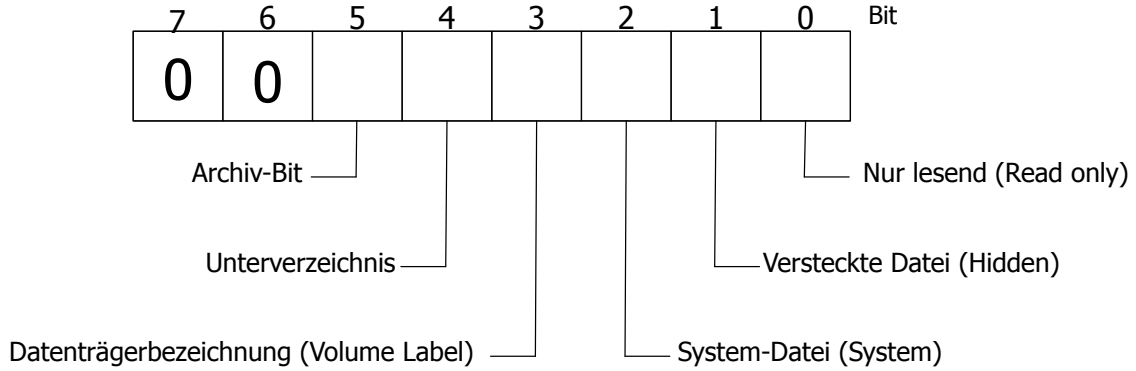


Abbildung 2.3.: Darstellung der FAT-Dateiattribute in Bits und ihre Bedeutung

Tabelle 2.6 schlüsselt alle möglichen Dateiattribute, anhand der Microsoft Spezifikation (vgl. Microsoft, 2000, S. 24) auf.

Bit	Maske	Beschreibung
0	0x01	ATTR_READ_ONLY, die Datei kann nicht modifiziert werden. Schreibversuche werden mit einem Fehler abgebrochen (vgl. Microsoft, 2000, S. 24).
1	0x02	ATTR_HIDDEN, versteckte Datei oder Unterverzeichnis welches nicht bei Dateiauflistung angezeigt werden darf (vgl. Microsoft, 2000, S. 24).
2	0x04	ATTR_SYSTEM, die Datei ist eine Komponente des Betriebssystems. Sie muss nicht angezeigt werden, es sei denn die Anwendung erzwingt es (vgl. Microsoft, 2000, S. 24).
3	0x08	ATTR_VOLUME_ID, Der Verzeichniseintrag enthält den Datenträgernamen. Nur das *Root-Verzeichnis* darf einen Eintrag mit diesem Attribut enthalten (vgl. Microsoft, 2000, S. 24).
4	0x10	ATTR_DIRECTORY, der Verzeichniseintrag ist ein Ordner und keine Datei (vgl. Microsoft, 2000, S. 24).
5	0x20	ATTR_ARCHIVE, dieses Attribut muss gesetzt werden sobald die Datei geschrieben, umbenannt oder modifiziert wurde. Backup-Programme können mit Hilfe dieses Attributes geänderte Dateien identifizieren (vgl. Microsoft, 2000, S. 24).

Tabelle 2.6.: FAT-Dateiattribute

Verzeichnisse

Werden Verzeichnisse erstellt, haben ihre Verzeichniseinträge das Attribut ATTR_DIRECTORY und immer eine feste Dateigröße von 0. Trotzdem wird immer ein Cluster allokiert, was bedeutet, dass ein freies Cluster ermittelt und in DIR_FstClusLO gesetzt werden muss. Da in dieser Arbeit mit einem FAT12-Dateisystem gearbeitet wird, hat DIR_FstClusHI den Wert 0. Der ermittelte FAT-Eintrag enthält den Wert 0xFFF (EOF). Das allokierte Cluster in der *Datenregion* wird mit 0 initialisiert (vgl. Microsoft, 2000, S. 24).

2.4.4. Unterverzeichnisse

Bei Unterverzeichnissen werden die ersten beiden 32-Byte Strukturen im Cluster mit . (dot) und .. (dotdot) belegt.

Diese beiden Verzeichniseinträge besitzen als DIR_Name ".␣␣␣␣␣␣␣␣␣␣" beziehungsweise "..␣␣␣␣␣␣␣␣␣" und eine Dateigröße von 0. Die Datum- und Zeit-Felder beider Einträge, sind identisch mit den Feldern des Verzeichniseintrages in dem sie erstellt wurden. DIR_FstClusLO und DIR_FstClusHI des ersten Eintrages (dot) enthalten die gleichen Werte wie der Verzeichniseintrag in dem sie erstellt wurden. Der dot-Eintrag zeigt somit auf sich selbst. DIR_FstClusLO und DIR_FstClusHI des dotdot-Eintrages zeigen auf die CLN des Eltern-Verzeichnisses in dem das aktuelle Verzeichnis erstellt wurde. Handelt es sich dabei um das *Root-Verzeichnis* ist der Wert 0 (vgl. Microsoft, 2000, S. 25).

Unterverzeichnisse besitzen in FAT-Dateisystemen die gleiche Struktur wie das *Root-Verzeichnis*. Im Unterschied zum *Root-Verzeichnis* befinden sich die Unterverzeichnisse nicht an einer definierten Position auf dem Datenträger. Unterverzeichnisse werden in Daten-Clustern der *Datenregion* abgebildet.

Angenommen die Datei `LEA` liegt im Ordner `/SIMON/NICOLE/FYNN`. Der Ordner `SIMON` (ein Eintrag mit dem Namen `SIMON` vom Typ `ATTR_DIRECTORY`) befindet sich im *Root-Verzeichnis* und verweist auf ein Cluster in der *Datenregion*. Innerhalb dieses Clusters, befinden sich weitere Verzeichniseinträge u. a. der Eintrag `NICOLE`, ebenfalls vom Typ `ATTR_DIRECTORY`. Dieser zeigt auf ein weiteres Daten-Cluster mit einem Verzeichniseintrag `FYNN`. Ein Unterverzeichnis kann n Daten-Cluster belegen und mehr Einträge als das *Root-Verzeichnis* führen.

Datum und Uhrzeit

Ein Datumsfeld, z. B. `DIR_DateCreate` ist ein 16-Bit-Feld. Tag, Monat und Jahr sind relativ zum Jahr 01/01/1980 kodiert. Die folgende Aufzählung zeigt welche Bits, welchen Wert repräsentiert:

- Bits 0-4: Tag des Monats hier sind die Werte 1 bis 31 zulässig.
- Bits 5-8: Monate 1 bis 12.
- Bits 9-15: Anzahl der Jahre ab dem Jahr 1980. Werte von 0 bis 127 sind möglich.

Das bedeutet das maximal das Jahr 2107 abgebildet werden kann (vgl. Microsoft, 2000, S. 25).

Das Zeitformat ist ähnlich wie das Datumsformat aufgebaut. 16 Bit repräsentieren mit einer Genauigkeit von zwei Sekunden die Uhrzeit.

- Bits 0-4: 2 Sekunden Zähler mit zulässigen Werten von 0-29 (0-58 Sekunden).
- Bits 5-10: 0 bis einschließlich 59 Minuten.
- Bits 9-15: Stunden mit Werten von 0 bis einschließlich 23.

Nach diesem System können Uhrzeiten von 00:00:00 bis 23:59:58 dargestellt werden (vgl. Microsoft, 2000, S. 25).

2.4.5. Die Datenregion

Cluster welche Dateiinhalte enthalten, befinden sich in der *Datenregion*. Sie enthält alle Dateien und Verzeichnisse eines FAT-Dateisystems. Jedes Cluster im Datenbereich wird über einen Eintrag in der FAT adressiert. Mit Hilfe der BPB-Werte, wird die Anzahl der Daten-Cluster in der *Datenregion* berechnet, siehe Kapitel 4.4. Der erste allokierbare Cluster in der *Datenregion* hat die Nummer 2, da in der FAT die ersten beiden Einträge reserviert sind (vgl. Microsoft, 2000, S. 14).

2.4.6. Cluster

Eine wichtige Einheit eines FAT-Dateisystems ist ein Cluster. FAT-Dateisysteme verwalten ihre Daten in Cluster und nicht in Sektoren. Ein Cluster kann dabei mehrere Sektoren beinhalten. Die Abbildung 2.4 veranschaulicht den Unterschied zwischen Cluster und Sektor.

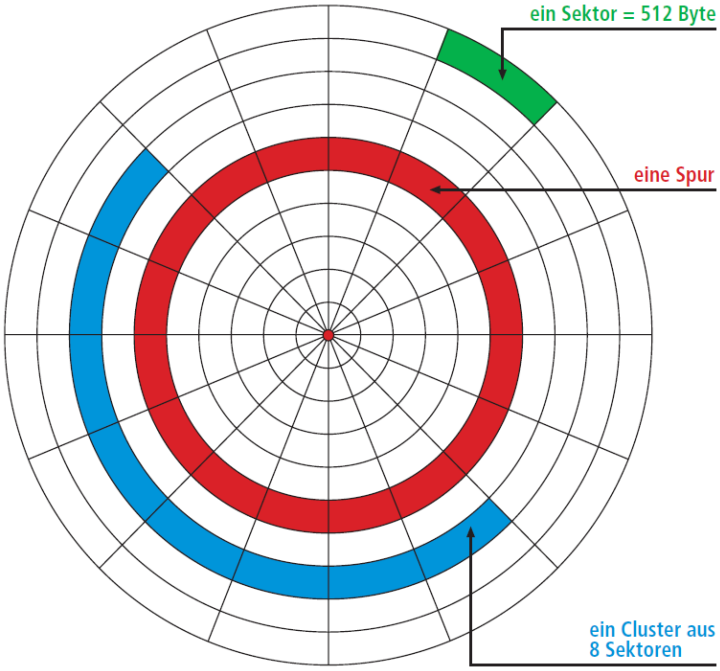


Abbildung 2.4.: Geometrie eines Datenträgers (vgl. com! professional, 2013).

Im Rahmen dieser Arbeit wird das FAT12-Dateisystem, auf einem Floppy-Image behandelt. Bei diesem Setup hat in der Regel jedes Cluster nur einen Sektor. Das FAT12-Dateisystem kann maximal 4.096 Cluster adressieren, siehe Kapitel 2.2. Mehr Cluster kann FAT12 nicht verwalten. Eine 3,5"1.44 MB Floppy-Diskette besitzt 2.880 Sektoren.

$$18 \text{ Spuren} \times 80 \text{ Sektoren pro Spur} \times 2 \text{ Seiten} = 2.880 \text{ Sektoren}$$

FAT12 reicht somit mehr als genug aus, um die gesamte Kapazität einer Diskette zu verwalten. Müsste FAT12 einen größeren Datenträger mit mehr Sektoren verwalten, könnte man Sektoren in Cluster zusammenfassen. Würde man in diesem Fall vier Sektoren in einem Cluster zusammenfassen, könnte FAT12 16.384 Sektoren verwenden.

Ein Cluster ist die kleinste adressierbare Einheit die ein FAT-Dateisystem verwalten kann. Eine Datei belegt immer mindestens einen kompletten Cluster. Das in dieser Arbeit verwendete FAT12 verwaltet 1 Sektor = 1 Cluster = 512 Byte. Das bedeutet das jede Datei und jeder Verzeichniseintrag mindestens 512 Byte belegt. Diese Verschwendung von Speicherplatz (Slack) steigt mit zunehmender Clustergröße (vgl. Phillips, 2008, S. 294). Die Tabelle 2.7 aus dem Microsoft Technet, veranschaulicht für ein FAT16-Dateisystem,

dass bei größerer Laufwerkskapazität, größere Cluster verwendet werden müssen.

Laufwerkskapazität	**Sektoren pro Cluster**	**Cluster-Größe**
0 MB - 32 MB	1	512 Bytes
33 MB - 64 MB	2	1 KB
65 MB - 128 MB	4	2 KB
129 MB - 255 MB	8	4 KB
256 MB - 511 MB	16	8 KB
512 MB - 1.023 MB	32	16 KB
1.024 MB - 2.047 MB	64	32 KB
2.048 MB - 4.095 MB	128	64 KB

Tabelle 2.7.: FAT16-Datenträgerkapazität im Verhältnis zu Cluster-Größen

3. Konzeption

Im Rahmen der Konzeption wird auf Basis der Grundlagenrecherche die Funktion des FAT-Treibers spezifiziert. Zuerst wird auf die Integration von Dateisystemen in UNIX-artigen Betriebssystemen eingegangen. Behandelt werden Kernel-Funktionen und das virtuelle Dateisystem. Es folgt die Spezifikation grundlegender Verfahren zur Realisierung des Dateisystemtreibers.

3.1. Anforderungen an einen FAT-Dateisystemtreiber

Zunächst werden grob die grundlegenden Anforderungen an den FAT-Dateisystemtreiber definiert. UCs eignen sich hierbei sehr gut um diese Anforderungen zu definieren. Ein UC fasst das Verhalten eines Systems, unter definierten Szenarien auf Anfragen eines Akteur zusammen. Dabei beschreibt der UC, was bei der Interaktion zwischen Akteur und System, zur Erreichung eines Zieles geschieht. Unabhängig von der konkreten technischen Umsetzung (vgl. Cockburn, 2001, S. 1).

3.1.1. UC: Öffnen und Schließen von Dateien

Beschreibung Um mit einem Dateisystem zu arbeiten, muss der Treiber in der Lage sein, Dateien zu öffnen und zu schließen.

Akteur User.

Vorbedingungen Keine.

Ablauf

1. User ruft im User Mode die Funktion `open()` auf.
2. Überprüfung ob die Datei *Read-only* ist.
3. Erstellen eines Dateideskriptors und Inodes.
4. Rückgabe des Dateideskriptors.
5. User ruft im User Mode die Funktion `close()` auf.
6. Löschen eines Dateideskriptors und Inodes.

Bedingung

1. Bei bereits geöffneten Dateien, bestehende Inodes mit neuem Deskriptor verknüpfen.

2. Sind mehrere Dateien geöffnet, Inode erst löschen, wenn die letzte Datei geschlossen wurde.

Alternative Abläufe Keine.

Das Öffnen einer Datei ist keine triviale Aufgabe. Das Flowchart in Abbildung 3.1 verdeutlicht diesen Ablauf auf einem höheren Detailgrad.

- Wie in Kapitel 3.4 beschrieben, kennt FAT das Inode-Konzept nicht. Aus diesem Grund werden Inodes zur Laufzeit erzeugt.
- Weiter müssen die Dateiattribute eines Verzeichniseintrages berücksichtigt werden. Es darf nicht möglich sein, *Read-only*-Dateien schreibend zu öffnen.
- Bei der Erzeugung von neuen Deskriptoren muss überprüft werden, ob diese Datei bereits geöffnet ist. Ist das der Fall, muss der Inode der geöffneten Datei, dem neuen Deskriptor zugeordnet werden.
- Diese Implementierung orientiert sich an den Limits von Minix. Somit sind maximal 256 Deskriptoren und Inodes gleichzeitig möglich. Es können nicht mehr Deskriptoren und Inodes erstellt werden.

3.1.2. UC: Zeitgleicher Dateizugriff

Beschreibung Eine Datei kann mehrfach geöffnet werden. Der Treiber muss in der Lage sein, mit gleichzeitigen Lese-/Schreibzugriffen umgehen zu können.

Akteur User.

Vorbedingungen Datei wurde zweimal geöffnet.

Ablauf
1. User ruft die generische Funktion `lseek()` auf und definiert für den ersten Dateideskriptor die Lese-/Schreibposition A.
2. User ruft die generische Funktion `lseek()` auf und definiert für den zweiten Dateideskriptor die Lese-/Schreibposition B.
3. Anschließend schreibt der User mit beiden Dateideskriptoren in die Datei.
4. Die Daten des ersten Dateideskriptors sind an Position A platziert und die des zweiten Dateideskriptors an Position B.

Bedingung Keine.

Alternative Abläufe Keine.

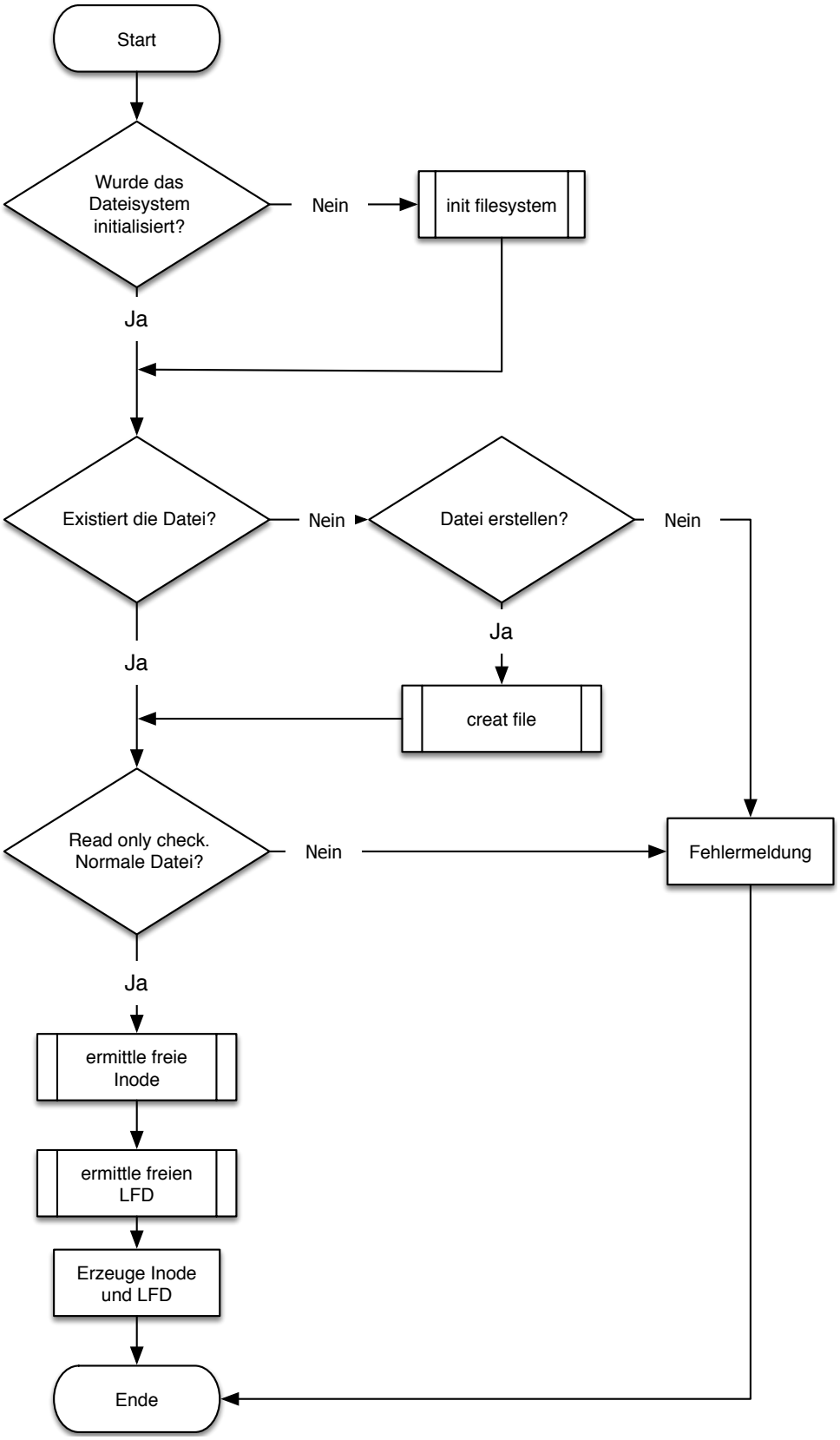


Abbildung 3.1.: Detaillierter Ablauf zum Öffnen von Dateien.

3.1.3. UC: Lesen von Dateien

Beschreibung Der FAT-Dateisystemtreiber muss in der Lage sein, Dateien auf dem FAT-Dateisystem sequenziell und wahlfrei auszulesen.

Akteur User.

Vorbedingungen Datei wurde geöffnet.

Ablauf
1. User ruft die generische Funktion `read()` auf.
2. Ermitteln der aktuellen Lese-/Schreibposition.
3. n-Bytes ab ermittelter Position auslesen.
4. Lese solange Daten aus der Datei, bis die angeforderte Menge oder das Dateiende erreicht ist.

Bedingung Keine.

Alternative Abläufe Keine.

3.1.4. UC: Schreiben von Dateien

Beschreibung Dieser UC beschreibt, wie der Treiber Dateien sequenziell und wahlfrei schreibt.

Akteur User.

Vorbedingungen Datei wurde geöffnet.

Ablauf
1. Der User ruft die generische Funktion `write()` auf.
2. Ermitteln der aktuellen Lese-/Schreibposition.
3. n-Bytes ab ermittelter Position schreiben.
4. Schreibe solange, bis alle Daten aus dem Buffer geschrieben wurden.

Bedingung
1. Ist das Dateiende erreicht, allokiere neue Daten-Cluster.
2. Liegt die Lese-/Schreibposition über der Dateigröße, fülle den Bereich zwischen Dateiende und Lese-/Schreibposition mit `0`-Werten und schreibe dann die Daten.
3. Schreibe über Daten-Cluster-Grenzen hinweg.

Alternative Abläufe Keine.

3.1.5. UC: Kopieren von Dateien

Beschreibung Es muss möglich sein Dateien von anderen Dateisystemen auf das FAT-Dateisystem zu kopieren.

Akteur User.

Vorbedingungen Keine.

Ablauf
1. Der User kopiert Dateien mit dem Programm `cp` von Minix nach FAT.
2. Datei wird angelegt.
3. Daten werden geschrieben.
4. Die identische Datei befindet sich am Ende auf dem FAT-Dateisystem.

Bedingung Keine.

Alternative Abläufe Keine.

Das Flowchart 3.2 beschreibt den Vorgang zum Erstellen neuer Dateien detaillierter.

3.1.6. UC: Löschen von Dateien

Beschreibung Der User muss in der Lage sein Dateien zu löschen. Dabei werden die belegten Daten-Cluster in der FAT wieder freigegeben und der Verzeichniseintrag entfernt.

Akteur User.

Vorbedingungen Keine.

Ablauf
1. User erstellt oder kopiert eine Datei unter ULIX auf dem FAT-Dateisystem.
2. User führt die genersiche Funktion `unlink()` aus um die Datei zu löschen.
3. Datei wird gelöscht und der Speicherplatz wird freigegeben.

Bedingung Keine.

Alternative Abläufe
1. Eine Datei, welche nicht unter ULIX erstellt wurde, befindet sich auf dem FAT-Dateisystem.
2. User führt die genersiche Funktion `unlink()` aus um die Datei zu löschen.
3. Datei wird gelöscht und der Speicherplatz wird freigegeben.

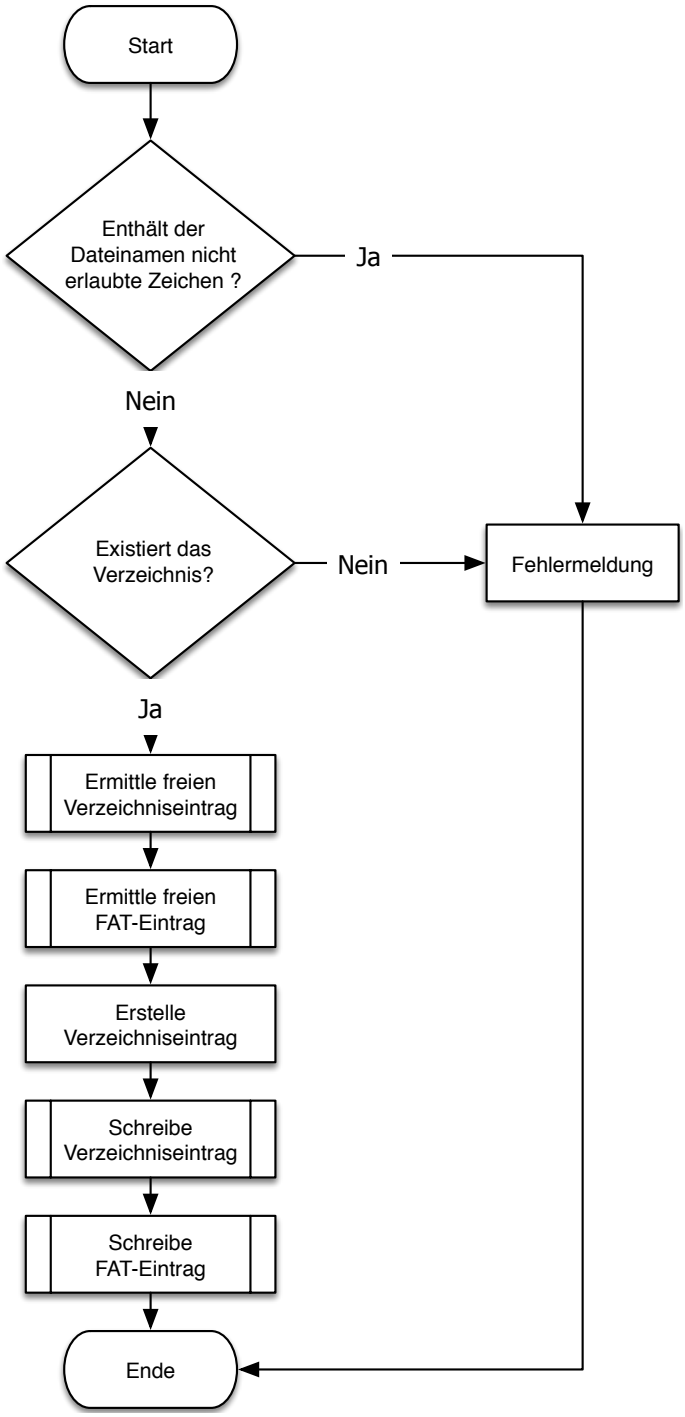


Abbildung 3.2.: Detaillierter Ablauf zum Erstellen einer leeren Datei.

3.1.7. UC: Verzeichnisse auflisten

Beschreibung Um unter UNIX-artigen Systemen, Inhalte von Verzeichnissen aufzulisten, verwendet der User das Programm `ls`. Die Ausgabe von `ls` ist eine Liste aller Verzeichniseinträge aus dem jeweiligen Verzeichnis.

Akteur User.

Vorbedingungen Keine.

Ablauf
1. User wechselt in ein Verzeichnis.
2. User führt das Programm `ls` aus um den Verzeichnisinhalt darzustellen.
3. Der Verzeichnisinhalt wird ausgegeben.

Bedingung Keine.

Alternative Abläufe Keine.

3.1.8. UC: Anwendung unter Linux

Beschreibung Dateien welche unter ULIX erstellt worden sind, müssen von anderen UNIX-artigen Systemen wie Linux gelesen werden können und umgekehrt.

Akteur User.

Vorbedingungen User erstellt eine Textdatei auf einem gemounteten FAT-Dateisystem.

Ablauf
1. User erstellt eine Textdatei in ULIX.
2. Das FAT-Image wird unter Linux gemountet.
3. Auslesen der unter ULIX erstellten Datei.

Bedingung Keine.

Alternative Abläufe
1. User erstellt eine Textdatei in ULIX.
2. Das FAT-Image wird unter Linux gemountet.
3. Auslesen der unter ULIX erstellten Datei.

Die folgenden Kapitel beschreiben u.a. konkrete technische Konzepte zu den oben definierten UCs.

3.2. Dateisystem-Funktionen

Dateioperationen lassen sich in sechs Grundfunktionen zusammenfassen (vgl. Pate, 2003, S. 37-38). Tabelle 3.1 beschreibt die Funktionen.

Funktion	Beschreibung
open()	Öffnet eine Datei oder erstellt eine neue
creat()	Erstellt eine neue Datei
close()	Schließt eine bereits geöffnete Datei
lseek()	Springt zu einer bestimmten Stelle in einer Datei
read()	Liest Daten abhängig von der aktuellen Lese-/Schreibposition
write()	Schreibt Daten abhängig von der aktuellen Lese-/Schreibposition

Tabelle 3.1.: Dateisystem-Funktionen

Aber woher weiß das Betriebssystem, welches Dateisystem bzw. welcher Datenträger, mit einem Aufruf von `open()` angesprochen werden soll? Das nächste Kapitel gibt eine Antwort auf diese Frage.

3.2.1. Verzeichnis auflisten

In UNIX-artigen Betriebssystemen gibt das Programm `ls` den Inhalt von Verzeichnissen aus. Dabei werden die Verzeichniseinträge und ihre Eigenschaften dargestellt. In der Regel handelt es sich dabei um folgende Eigenschaften:

- Dateityp und Zugriffsrechte
- Anzahl Links auf die Datei
- Dateibesitzer und Gruppe
- Dateigröße
- Datum der letzten Modifizierung
- Dateiname

Um diese Informationen darzustellen, muss das Programm `ls` zwei Funktionen aufrufen. Die eine Funktion holt alle Verzeichniseinträge des aktuellen Verzeichnisses. Die andere Funktion ermittelt zu jedem Verzeichniseintrag die Eigenschaften (vgl. Pate, 2003, S. 20).

Nach einer Analyse des ULIX 0.10 Sourcecodes wurde festgestellt, dass mit der in dieser Arbeit verwendeten ULIX-Version, eine Implementierung der Verzeichnisauflistung nicht möglich ist, da die generischen Funktionen `u_getdent` und `u_stat` noch nicht implementiert wurden.

Dennoch soll an dieser Stelle mit Hilfe von Pseudocode-Chunks beleuchtet werden, wie eine Implementierung dieser Funktionalität für eine aktuellere ULIX-Version aussehen könnte. Dabei wird die Datenstruktur `stat` verwendet, welche in ULIX definiert ist (vgl. Eßer & Freiling, 2014, S. 453).

In einer Funktion `fat_getdent()` werden Root- oder Unterverzeichnisse, Eintrag für Eintrag durchlaufen. Für jeden Eintrag wird `fat_stat` aufgerufen und das Verzeichnis gelesen. Anschließend wird der Name des Verzeichniseintrages und die Referenz auf den Inode im Array `DirList` abgelegt.

31a ⟨*fat getdent* 31a⟩≡

```
int fat_getdent(int device, const char * path, struct dir_entry *DirList[]){
   FATVerzeichniseintrag *dir
   struct stat s

   Wenn path == Rootverzeichnis dann
      wiederhole Schleife solange bis zum letzten Verzeichniseintag
         ino = fat_stat(device, AktuellerVerzeichniseintrag, &s)
         fat_read_dir_entry(device, AktuellerVerzeichniseintrag, &dir)
         DirList[i]->inode = ino
         DirList[i]->name = dir.name
         i++
         Wenn dir.name[0] = 0x00 dann
            VerlasseSchleife
      wiederhole_ende
   sonst
      ErmittleDasVerzeichnis()
      wiederhole Schleife solange bis zum letzten Daten"=Cluster
         /* Das gleiche Verfahren wie im Root-Verzeichnis */
      wiederhole_ende
   Rückgabe 0
}
```

Uses `device` 66a, `dir` 77, `fat_getdent`, `fat_stat`, `i`, and `inode`.

Die Funktion `fat_stat` holt die Informationen aus der Inode, siehe Kapitel 4.7.1 und weist sie den Komponenten der globalen Struktur `stat` zu. Dabei muss für jeden Verzeichniseintrag ein neuer Inode erstellt werden. Die Informationen des Inodes und der `stat`-Datenstruktur werden mit den Informationen aus dem Verzeichniseintrag `dir` bzw. FAT-Default Werten belegt. Der Rückgabewert ist die Inodenummer.

31b ⟨*fat stat* 31b⟩≡

```
int fat_stat(int device, const char * path, struct stat *buf){
   FATVerzeichniseintrag *dir

   wiederhole Schleife solange bis zum letzten fat_inodes[] Eintrag
      wenn path == fat_inodes[i].path dann
         struct int_fat_inode *inode = &(fat_inodes[i])
      sonst erstelle neuen Inode
         int_ino = fat_get_free_inode_entry()
         struct int_fat_inode *inode = &(fat_inodes[int_ino])
      i++
   wiederhole_ende

   fat_read_dir_entry(device, path, &dir)

   buf->st_size = inode -> i_size = dir.size
   buf->st_ctime = inode -> i_ctime = dir.ctime
```

```
    ...

    Rückgabe int_ino
}
```

Uses `device` 66a, `dir` 77, `fat_get_free_inode_entry` 59b, `fat_inodes`, `fat_stat`, `i`, `inode`, `int_fat_inode` 56 64a 64d 66a 87c, and `int_ino` 61b.

3.3. Das virtuelle Dateisystem in ULIX

Um ULIX in die Lage zu versetzen, mit einem neuen Dateisystem umzugehen, ist eine grundlegende Erklärung eines VFS notwendig.

Moderne Betriebssysteme müssen gleichzeitig mehrere Dateisysteme unterstützen. UNIX-artige Betriebssysteme lösen das mit Methoden der Modularisierung und Abstraktion. Dabei werden die dateisystemspezifischen Funktionen durch generische Funktionen getrennt. Dadurch bietet sich zum Beispiel dem User die Möglichkeit, mit Hilfe der Funktion `open()` auf eine Datei zuzugreifen, ohne sich darum zu kümmern, auf welchem Dateisystem die Datei liegt (vorausgesetzt das Dateisystem wird vom Betriebssystem unterstützt). Folgende Abbildung veranschaulicht das Konzept eines VFS (vgl. Silberschatz et al., 2008, S. 468).

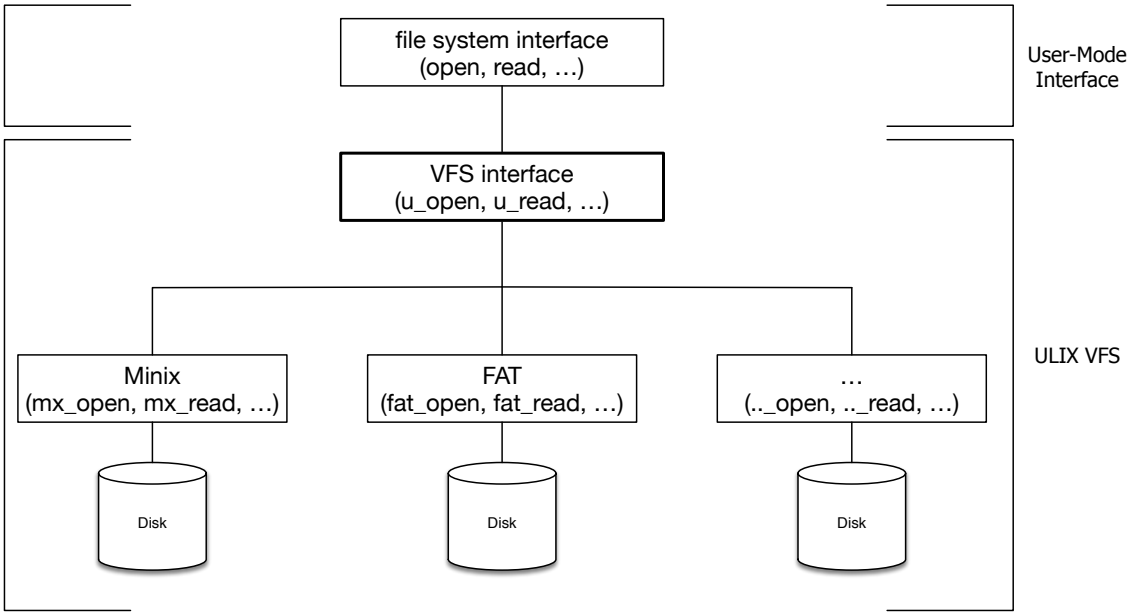


Abbildung 3.3.: Abgrenzung von ULIX User-Mode und ULIX-VFS

UNIX-Betriebssysteme verwalten alle Datenträger in einem Verzeichnisbaum mit einem Wurzelverzeichnis ("/"). Weitere Dateisysteme werden mit Hilfe des `mount`-Programms in den Verzeichnisbaum integriert. Dabei können Datenträger den Verzeichnisbaum erweitern, in dem sie als neuer Unterverzeichnisbaum an eine beliebige Stelle (mountpoint) integriert werden (vgl. Ritchie & Thompson, 1974, S. 367). Auch ULIX organisiert unterschiedliche Datenträger in einem großen Verzeichnisbaum. Anders als viele UNIX- und Linux-Distributionen, stellt ULIX nicht die Programme `mount` und `umount` zur Verfügung.

Stattdessen verwendet ULIX eine fest definierte Mount-Tabelle mit maximal 16 Einträgen, um Datenträger zu verwalten (vgl. Eßer & Freiling, 2014, S. 369).

Damit das VFS Kenntnis über ein FAT-Dateisystem erhält, wird die Mount-Tabelle um den Eintrag `/mnt2/, FS_FAT, DEV_FD1, 0` ergänzt. Der erste Parameter ist der mountpoint, in diesem Fall das Verzeichnis `/mnt2/`. Die Konstante `FS_FAT` definiert das FAT-Dateisystem. ULIX definiert eine Reihe von Konstanten, unterstützter Dateisysteme oder von Dateisystemen, welche in Zukunft unterstützt werden könnten (vgl. Eßer & Freiling, 2014, S. 374). In ULIX stehen zwei Floppy-Laufwerke zur Verfügung. Der dritte Parameter steht für das zweite Floppy-Laufwerk `DEV_FD1` (vgl. Eßer & Freiling, 2014, S. 458). Der letzte Parameter definiert die mount-Option, in diesem Fall, Lese-/Schreibzugriff. Der folgende Code-Chunk zeigt die Änderung im Sourcecode von ULIX.

33 ⟨*ulix mount table* 33⟩≡

```
mount_table_entry mount_table[16] = {
  { "/",     FS_MINIX, DEV_HDA, 0 },
  { "/mnt/", FS_MINIX, DEV_FD0, 0 },
  { "/mnt2/", FS_FAT, DEV_FD1, 0 },
  0
};
```

Defines:
`mount_table`, never used.

Weiter wurde das Verzeichnis `/mnt2/`, manuell auf dem Disketten-Image `ulixboot.img` angelegt. Somit steht nach dem Kompilieren und Ausführen von ULIX der Mountpoint zur Verfügung.

Über die Mount-Tabelle kann ULIX herausfinden unter welchem Verzeichnis sich welches Dateisystem befindet und welche spezifischen Dateisystemfunktionen aufzurufen sind. Das reicht aber noch nicht um mit Dateien zu arbeiten, dazu sind weitere Datenstrukturen notwendig.

Globaler Dateideskriptor

Unter ULIX verwalten die einzelnen Dateisysteme ihre lokalen Dateideskriptoren selbst. So kann, sowohl Minix als auch FAT die lokalen Deskriptoren 0,1,2,... erzeugen. Damit das VFS aus der Vielzahl von geöffneten Dateien unterschiedlicher Dateisysteme, die richtige Zuordnung herstellen kann, identifiziert das VFS Dateioperationen über globale Dateideskriptoren.

Dazu verwaltet ULIX fest definierte Nummernkreise für globale Dateideskriptoren. Für das FAT-Dateisystem ist der Nummernkreis 512-767 definiert (vgl. Eßer & Freiling, 2014, S. 374).

Lokaler Dateideskriptor

Im VFS findet die Umrechnung von globalen Dateideskriptoren zu lokalen Dateideskriptoren statt. Die generischen Funktionen wie zum Beispiel `u_open()` kommen nicht mit den lokalen Dateideskriptoren in Kontakt. Das VFS ruft die dateisystemspezifischen Funktionen, wie zum Beispiel `fat_open()` mit dem lokalen Dateideskriptor `ffd` auf (vgl. Eßer &

Freiling, 2014, S. 374).

Der folgende Code-Chunk erweitert die `u_open`-Funktion um einen Aufruf der `fat_open`-Funktion (vgl. Eßer & Freiling, 2014, S. 376).

34 ⟨*u_ open* 34⟩≡

```
[...]
switch (fs) {
   case FS_MINIX:
     fd = mx_open (device, localpath, oflag);
     if (fd == -1) return -1;   // error
     else          return (fs << 8) + fd;
   case FS_FAT:
     fd = fat_open (device, localpath, oflag);
     if (fd == -1) return -1;   // error
     else          return (fs << 8) + fd;
   case FS_ERROR: return -1;  // error
   }
 }
[...]
```

Uses `device` 66a and `fat_open` 60b.

Damit die in Kapitel 3.2 genannten generischen Funktionen auf die FAT-spezifischen Funktionen zugreifen können, werden die entsprechenden `u_*`-Funktionen des ULIX-VFS um FAT-Funktionen ergänzt.

3.4. Inodes

ULIX ist ein UNIX-artiges Betriebssystem und verwendet somit das Inode-Schema. Inodes sind Datenstrukturen, welche die Adressen der Datenblöcke und die Dateiattribute verwalten (vgl. Tanenbaum, 2009, S. 339). Andere UNIX-artige Betriebssysteme wie z.B. FreeBSD stellen über Header-Dateien Datenstrukturen zur Verfügung, welche von Dateisystemen verwendet werden können (vgl. FreeBSD, 2006). In UNIX-Dateisystemen werden Inode-Verweise zusammen mit dem Verzeichniseintrag gespeichert. FAT-Dateisysteme kennen das Schema der Inodes nicht und berücksichtigen in ihrem Design auch keinen Platz zur Speicherung dieser Informationen. Aus diesem Grund müssen FAT-Dateisystemtreiber, welche für UNIX-Systeme implementiert werden, diese Datenstrukturen zur Laufzeit erzeugen und zur Verfügung stellen.

Die UNIX-Dateioperationen wie `stat()` können dann auf diese Informationen zugreifen (vgl. Pate, 2003, S. 69).

3.5. Lesen und Schreiben in ULIX

ULIX stellt die Funktionen `readblock` und `writeblock` zur Verfügung (vgl. Eßer & Freiling, 2014, S. 392). Beide Funktionen lesen bzw. schreiben mit einer fest kodierten Blockgröße von 1024 Byte (ULIX Version 0.10) (vgl. Eßer & Freiling, 2014, S. 359). FAT12 ist

auf einem Floppy-Image mit einer Sektoren bzw. Cluster Größe von 512 Byte organisiert. Es wird ein Verfahren benötigt, um die korrekten FAT-Cluster zu lesen und zu schreiben, da ULIX 1024er Blöcke liest und schreibt.

35a ⟨*fat function prototypes* 35a⟩≡ (89a) 42a▷
```
extern void readblock (int device, int blockno, char* buffer);
extern void writeblock (int device, int blockno, char* buffer);
```
Uses `device` 66a.

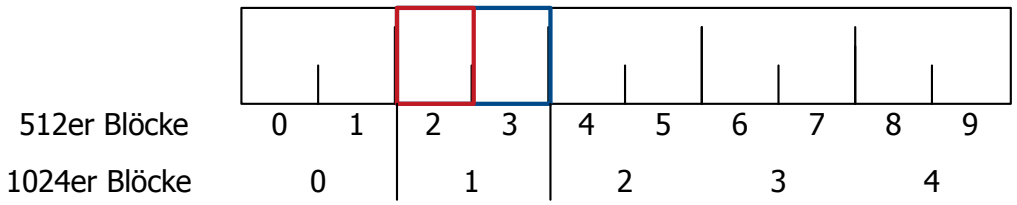


Abbildung 3.4.: FAT-Layout mit 512er und 1024er Blöcken.

Damit an der richtigen Stelle gelesen bzw. das richtige Cluster ausgelesen wird, kommen zwei unterschiedliche Verfahren zum Einsatz. Entweder wird über die absolute Byte Position die korrekte Stelle ermittelt:

35b ⟨*absolute Byte Position* 35b⟩≡
```
...
int clusteroff = (512 * fat_FirstSectorOfRootDir);
int diroff = clusteroff + (32 * dirno);
int reblock = diroff / 1024;

readblock (device,reblock, (char *)fbuf);
int blockoff = (int)(diroff%1024);

wdentry = (dirEntry*) &fbuf[blockoff];
...
```
Uses `blockoff` 73, `clusteroff`, `device` 66a, `dirEntry` 49a, `diroff`, `fbuf` 43b, `reblock` 73, and `wdentry` 73.

oder über die CLN. Bei geraden CLNs wird der gelesene 1.024er Block ab einem Offset von 512 Byte ausgelesen und bei ungeraden Clustern bei 0 Byte.

35c ⟨*Clusternummer* 35c⟩≡
```
...
if (clusternummer % 2 > 0){
   blockoff = 0;
} else {
   blockoff = 512;
}

readblock (device, block, (char *)fbuf);
wdentry = (dirEntry*) &fbuf[blockoff];
...
```
Uses `block`, `blockoff` 73, `device` 66a, `dirEntry` 49a, `fbuf` 43b, and `wdentry` 73.

3.6. FAT Einträge ermitteln

Im Rahmen dieser Arbeit wird die FAT-Variante FAT12 betrachtet. Wie bereits in Kapitel 2 beschrieben, verwendet FAT12 zwölf Bits um die CLN in der FAT zu speichern. Abbildung 3.5 beschreibt das Verfahren, mit dem CLNs ermittelt werden.

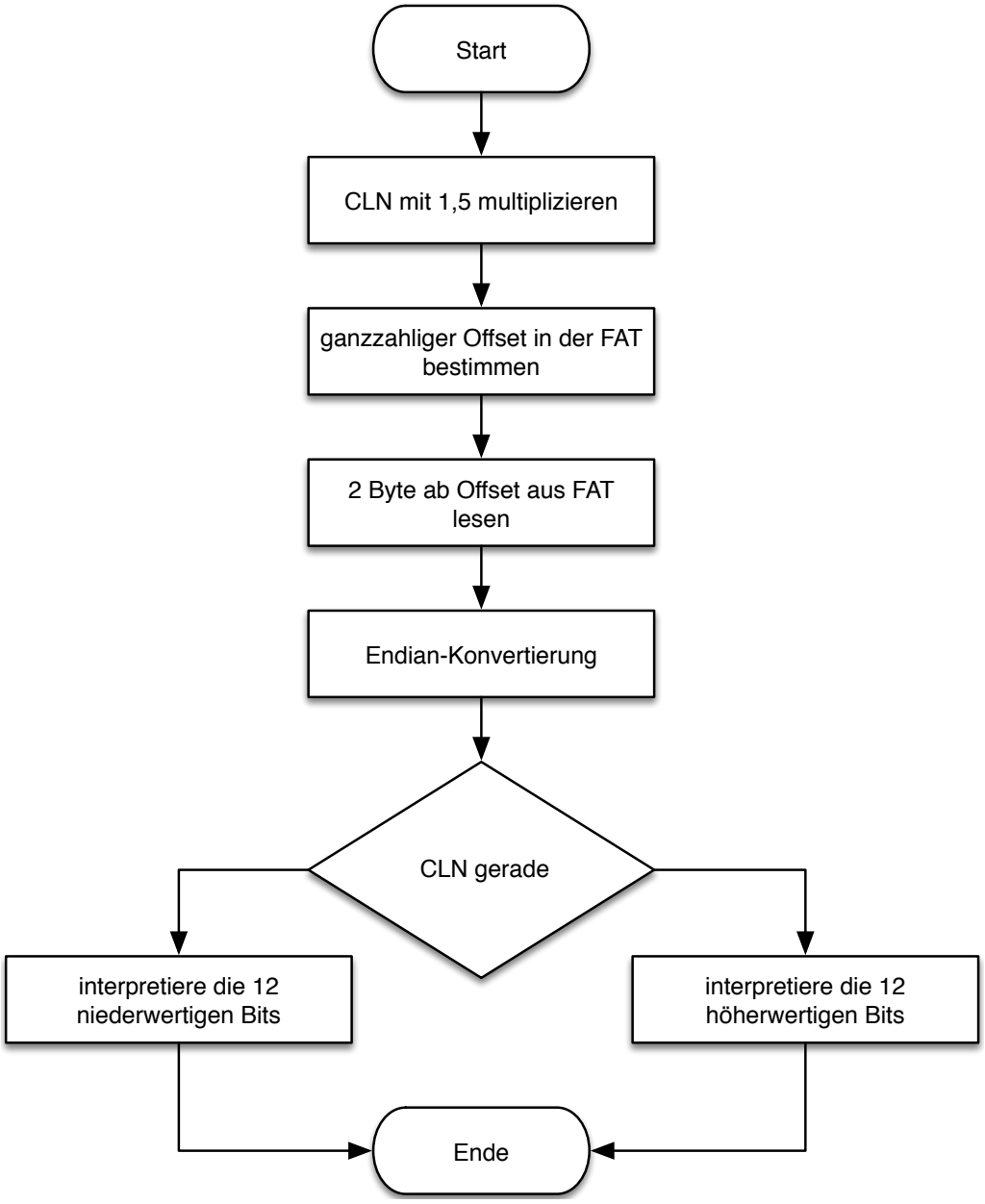


Abbildung 3.5.: Detaillierter Ablauf zur Ermittlung des korrekten FAT-Eintrages

Anhand eines Beispiels wird der oben definierte Prozess nachvollzogen. Die Cluster-Kette beginnt mit einem CLN von 17 `0x11`. Der folgende *Hexdump* hat ein Offset von (17 * 1,5 + 512 = 537):

```
ulix@ulixdevel:~/ulix/bin-build$ hexdump -C -n 512 -s 537 floppy.img
00000219  2f 01 13 40 01 ff 0f 00  00 00 00 00 00 00 00 00  |/..@............|
00000229  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Eintrag CLN 17: `2f 01` → `01 2f` ungerade CLN → `0x012` = `18`
Eintrag CLN 18: `13 40` → `04 13` gerade CLN → `0x013` = `19`
Eintrag CLN 19: `40 01` → `01 40` ungerade CLN → `0x013` = `20`
Eintrag CLN 20: `ff 0f` → `0f ff` gerade CLN → `0xfff` = `EOF`

Der Dump beginnt mit den Werten `2f 01` als *Big Endian* konvertiert `01 2f`. Da 17 eine ungerade Zahl ist, werden nur die zwölf höherwertigen Bits ausgewertet `0x012`. Diese entsprechen 18 in Dezimal. Die CLN 18 enthält die Werte `13 40` bzw. `04 13` und ist gerade. Die niederwertigen Bits enthalten die Werte `0x013` bzw. 19. Nach diesem Verfahren wird die Cluster-Kette durchlaufen bis `EOF` erreicht wird (vgl. Microsoft, 2000, S. 16).

3.7. Zugriff auf die Datenregion

Die in der FAT gespeicherten CLNs referenzieren die Daten-Cluster. Mit folgender Formel werden die Sektoren bzw. Daten-Cluster bestimmt:

$$\text{Daten-Cluster} = ((\text{ CLN} - 2)) \times \text{ Sektoren pro Cluster} + \text{ Offset-Sektor der Datenregion}$$

Die ersten beiden Einträge in der FAT sind reserviert und referenzieren keine Cluster. Zur Bestimmung des entsprechenden Daten-Clusters wird die CLN um 2 reduziert und mit der Anzahl der Sektoren pro Cluster multipliziert. Anschließend wird der Offset der *Datenregion* addiert (vgl. Microsoft, 2000, S. 14).

Ein Beispiel: CLN = 10, Anzahl der Sektoren pro Cluster = 1, Start-Sektor der *Datenregion* = 33. Mit Hilfe der oben genannten Formel wird der Cluster 41 berechnet. Die genaue Position in Bytes wird wie folgt berechnet:

$$\text{Byte Position} = \text{ Berechnete CLN} \times \text{ Bytes pro Sektor}$$

Die Berechnung der ULIX-Blockgröße, wurde in Kapitel 3.5 beschrieben.

Die *Datenregion*, der in dieser Arbeit verwendeten FAT12-Dateisysteme beginnen bei Sektor bzw. Cluster 33. Aus diesem Grund arbeitet die Implementierung mit einem Offset von 31 für den Zugriff auf ein Daten-Cluster.

37 ⟨*Datenregion Zugriff* 37⟩≡

```
...
readblock (device,(int)((31 + (int)CLN) / 2), (char *)fbuf);
...
```

Uses `device` 66a and `fbuf` 43b.

3.8. Zugriffsrechte

In ULIX ist ein einfaches Gruppenkonzept realisiert. Unter ULIX und anderen UNIX-artigen Systemen gehören die Benutzer ein oder mehreren Gruppen an. Dateien sind immer eindeutig einer Gruppe zugeordnet. Wer was mit einer Datei machen darf wird über Zugriffsrechte gesteuert. Zugriffsrechte werden getrennt von einander für Besitzer, Gruppe und alle anderen Benutzer verwaltet. Dabei gibt es drei verschiedene Rechte:

1. Lesen (r): Dateien können zum Lesen geöffnet werden. Veränderungen sind nicht erlaubt. Verzeichnisse können gelesen werden.
2. Schreiben (w): Dateien in Verzeichnissen können gelöscht und erstellt werden. Dateien können verändert werden.
3. Ausführen (x): Berechtigung um Dateien auszuführen oder in Verzeichnisse zu wechseln.

Diese Rechte können unabhängig von einander und separat für Besitzer, Gruppe und Andere definiert werden (vgl. Kappes, 2013, S. 75).

In seiner Implementierung definiert ULIX folgende Konstanten bzw. Masken um die Zugriffsrechte zu verarbeiten (vgl. Eßer & Freiling, 2014, S. 421-422).

38 ⟨*fat type definitions* 38⟩≡ (89a) 42b▷

```
#define S_IRWXU  0000700    /* RWX mask for owner */
#define S_IRUSR  0000400    /* R for owner */
#define S_IWUSR  0000200    /* W for owner */
#define S_IXUSR  0000100    /* X for owner */

#define S_IRWXG  0000070    /* RWX mask for group */
#define S_IRGRP  0000040    /* R for group */
#define S_IWGRP  0000020    /* W for group */
#define S_IXGRP  0000010    /* X for group */

#define S_IRWXO  0000007    /* RWX mask for other */
#define S_IROTH  0000004    /* R for other */
#define S_IWOTH  0000002    /* W for other */
#define S_IXOTH  0000001    /* X for other */

#define S_ISUID  0004000    /* set user id on execution */
#define S_ISGID  0002000    /* set group id on execution */
#define S_ISVTX  0001000    /* save swapped text even after use */

#define S_IFMT   0170000    /* mask the fie type part */
#define S_IFDIR  0040000    /* directory */
#define S_IFREG  0100000    /* regular */
```

Uses S_IFDIR, S_IFMT, S_IFREG, S_IRGRP, S_IROTH, S_IRUSR, S_IRWXG, S_IRWXO, S_IRWXU, S_ISGID, S_ISUID, S_ISVTX, S_IWGRP, S_IWOTH, S_IWUSR, S_IXGRP, S_IXOTH, and S_IXUSR.

3.8.1. Zugriffsrechte und FAT

Das FAT-Dateisystem besitzt kein Konzept zur Unterstützung von Zugriffsrechten. Mehr als einfache Dateimerkmale wie *Read-only*, *Hidden* und *System* unterstützt FAT nicht (vgl. Smith, 2015, S. 201).

Die nachfolgende Analyse soll das Verhalten von FAT-Dateisystemen unter anderen UNIX-artigen Systemen, im Bezug auf das Gruppenkonzept und die Zugriffsrechte beleuchten. Analysiert wurde unter Debian 6.

mount

Unter UNIX-artigen Systemen werden Dateien immer eindeutig Gruppen und Besitzern zugeordnet. FAT kennt weder das Gruppenkonzept noch besitzt FAT Inode-Datenstrukturen um diese Informationen persistent zu den Dateien zu speichern. Wenn ein FAT-Dateisystem unter Linux eingebunden wird, werden die UNIX-Dateirechte simuliert. So erhalten alle Dateien und Verzeichnisse des gesamten FAT-Datenträgers den gleichen Benutzer und die gleiche Gruppe im Regelfall, `root/root`.

```
$ sudo mount -t vfat -o loop /home/ulix/ulix/bin-build/floppy.img floppy/

drwxr-xr-x 3 root root  7168  1. Jan 1970  .
drwxr-xr-x 5 root root  4096  8. Nov 23:24 ..
-rwxr-xr-x 1 root root 12753  7. Okt 03:07 fat.txt
drwxr-xr-x 2 root root   512 17. Nov 2014  folder
-rwxr-xr-x 1 root root    15 17. Nov 2014  one.txt
-rwxr-xr-x 1 root root   200  7. Okt 03:06 xaa
-rwxr-xr-x 1 root root   200  7. Okt 03:06 xab
```

Sollen die Rechte angepasst werden kann, das nur beim Einbinden des Datenträgers geschehen. Das Programm `mount` bietet hier folgende Parameter: `umask`, `dmask` und `fmask` zur Definition der Zugriffsrechte für Dateien und/oder Verzeichnisse. `uid` und `gid` zur Definition von Besitzern und Gruppen.

```
$ sudo mount -t vfat -o loop,uid=1001,gid=1002 -rw \
  /home/ulix/ulix/bin-build/floppy.img floppy/

drwxr-xr-x 3 ulix ulix  7168  1. Jan 1970  .
drwxr-xr-x 5 root root  4096  8. Nov 23:24 ..
-rwxr-xr-x 1 ulix ulix 12753  7. Okt 03:07 fat.txt
drwxr-xr-x 2 ulix ulix   512 17. Nov 2014  folder
-rwxr-xr-x 1 ulix ulix    15 17. Nov 2014  one.txt
-rwxr-xr-x 1 ulix ulix   200  7. Okt 03:06 xaa
-rwxr-xr-x 1 ulix ulix   200  7. Okt 03:06 xab
```

Das nachträgliche ändern der Zugriffs-und Gruppenrechte mit den Programmen `chmod`, `chown` und `chgrp` ist nicht mehr möglich.

4. Implementierung eines FAT-Dateisystemtreibers

Dieses Kapitel behandelt die Implementierung des FAT-Dateisystemtreibers nach den in Kapitel 3 definierten Anforderungen. Zuerst geht das Kapitel auf die Integration in ULIX ein, während es im Anschluss die Umsetzung der einzelnen Funktionen in *Literate Programming* beschreibt.

4.1. Das Floppy-Image

Der im Rahmen dieser Arbeit zu entwickelnde FAT12-Dateisystemtreiber greift auf ein Floppy-Image zu. Dieser Abschnitt verdeutlicht alle Schritte zur Erstellung eines Floppy-Images. Anschließend wird das Floppy-Image formatiert und es werden einige Testdateien und Verzeichnisstrukturen initial angelegt. Anhand des erstellten Images sollen die Funktionen des FAT12-Dateisystemtreibers erklärt und verdeutlicht werden. Alle vorbereitenden Maßnahmen wurden unter einem Debian Linux ausgeführt.

```
$ cat /proc/version
Linux version 2.6.32-5-686 (Debian 2.6.32-31) (ben@decadent.org.uk)
(gcc version 4.3.5 (Debian 4.3.5-4) ) #1 SMP Tue Mar 8 21:36:00 UTC 2011
```

Der folgende Befehl erstellt ein 1.44 MB großes Floppy-Image mit dem Dateinamen `floppy.img`, welches Null-Bytes enthält.

```
$ dd if=/dev/zero of=floppy.img bs=1024 count=1440
```

Anschließend wird das Image mit `mkfs` ein FAT-Dateisystem erstellt. Das Programm `mkfs` formatiert unter UNIX-artigen Systemen Block-Dateisysteme wie FAT. Mit folgendem Befehl wird der Laufwerksname `ULIXFAT` (maximal 11 Zeichen) gesetzt. Mit welcher FAT-Version formatiert werden soll, ermittelt das Programm automatisch (vgl. Hudson et al., 2015).

```
mkfs.vfat -n ULIXFAT floppy.img
```

Nach der Formatierung wurde das Laufwerk gemountet und folgende Dateistruktur angelegt:

```
$ sudo mount -t vfat -o loop floppy.img /media/floppy
```

```
insgesamt 13
    1 drwxr-xr-x 3 root root 7168  1. Jan 1970  .
40001 drwxr-xr-x 5 root root 4096  5. Apr 01:10 ..
    5 drwxr-xr-x 2 root root  512 17. Nov 23:47 folder
    7 -rwxr-xr-x 1 root root   26 17. Nov 23:48 looongfilename.txt
    6 -rwxr-xr-x 1 root root   15 17. Nov 23:46 one.txt
```

Damit sind sämtliche Schritte zur Erstellung des Testobjektes (Floppy-Image) dokumentiert.

Die Konstante `DEV_FD1` definiert die Gerätenummer des zweiten Floppy-Laufwerkes. Der folgende Befehl startet ULIX in einer virtuellen Umgebung mit dem erstellten FAT12-Floppy-Image.

```
$ qemu -m 64 -rtc base=localtime -boot a -fda ulixboot.img -fdb floppy.img -hda \
  $(HD_IMG) -d cpu_reset -s -serial mon:stdio | tee ulix.output
```

Nachdem ULIX gestartet wurde, steht das zweite Floppy-Laufwerk mit dem FAT12-Dateisystem am Mountpoint `/mnt2` zur Verfügung.

4.2. Modul-Integration in ULIX

Diese Bachelorarbeit wurde in einer separaten `.nw`-Datei geschrieben. Während des Build-Prozesses wird zuerst mit Hilfe des Programmes `notangle` der C-Quelltext extrahiert. Die `noweb`-Umgebung stellt dieses Programm zur Verfügung.

```
$ notangle -L -Rmodule.c $(TEXSRC_MODULE_FILE) > module.c
```

Der Modul-Quelltext in der Datei `module.c` wird im Anschluss mit folgendem Befehl kompiliert und der Linker integriert den Modul-Programmcode mit dem ULIX-Programmcode.

```
$ gcc-4.4 -O0 -m32 -fno-stack-protector -std=c99 -g -nostdlib -nostdinc \
  -fno-builtin -I./include -c -o module.o module.c
```

Alle globalen Typdefinitionen, Makros, Prototypen und Variablen werden in der Modul-Header-Datei definiert.

41a ⟨*module header* 41a⟩≡ (89b)

```
#include "module.h"
```

Die Funktion `initialize_module` initialisiert das Modul beim Start von ULIX.

41b ⟨*init module* 41b⟩≡ (89b)

```
void initialize_module () {
}
```

Uses `initialize_module`.

Mit der Erstellung des Floppy-Images, der Integration des Dateisystems und der Integration des Moduls in ULIX sind alle Voraussetzungen erfüllt, um mit der Implementierung des FAT-Dateisystemtreibers zu beginnen.

4.3. ULIX-Funktionen

Die Implementierung bindet einige ULIX-Funktionen ein. Im folgenden Code-Chunk werden die Funktionen deklariert. Eine genaue Beschreibung zu welchem Zweck, welche Funktion verwendet wird, erfolgt an der entsprechenen Stelle in der Implementierung.

42a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁35a 44a▷

```
extern int strlen (const char* str);
extern void splitpath (const char *path, char *dirname, char *basename);
extern int strcmp (const char *str1, const char *str2);
extern int strlen (const char* str);
extern int printf(const char *format, ...);

typedef int size_t;
extern void *memcpy(void *dest, const void *src, size_t count);
extern void *strncpy(void *dest, const void *src, size_t count);
extern void *memset(void *dest, char val, size_t count);
```

Defines:
`memcpy`, used in chunks 52–54, 68, 75a, 80, and 85c.
`memset`, used in chunk 64a.
`printf`, used in chunks 60, 69b, 76–78, 80, 88a, and 102–105.
`splitpath`, used in chunk 76b.
`strcmp`, used in chunks 52, 53, and 61b.
`strlen`, used in chunks 50c, 54, 55, 63a, and 77.
`strncpy`, used in chunks 50c and 63a.
Uses `dirname` 76b and `size_t`.

4.4. FAT-Dateisystem-Initialisierung

Bei der Initialisierung des FAT-Dateisystems werden der Bootsektor und der BPB ausgelesen und interpretiert. In diesem Kapitel wird beschrieben, wie der Bootsektor des in Kapitel 4.1 erstellten FAT12-Floppy-Images ausgelesen wird. Der BPB enthält globale Informationen über das Layout und die Anordnung der Datenstrukturen des FAT-Dateisystems.

Zunächst werden zwei Strukturen definiert. Die Struktur `bootblock` definiert den in Kapitel 2.4.1 beschriebenen Bootsektor. Dabei werden Bitfields verwendet um den Rückgabe-Buffer der `readblock`-Funktion bitgenau zu mappen.

42b ⟨*fat type definitions* 38⟩+≡ (89a) ◁38 43a▷

```
struct bootblock {
   unsigned int BS_jmpB           : 24;
   unsigned char BS_OEMName[8];
   unsigned int BPB_BytsPerSector : 16;
   unsigned int BPB_SecPerCluster : 8;
   unsigned int BPB_RsvdSecCnt    : 16;
```

```
    unsigned int BPB_NumFATs       : 8;
    unsigned int BPB_RootEntCnt    : 16;
    unsigned int BPB_TotSec16      : 16;
    unsigned int BPB_Media         : 8;
    unsigned int BPB_FATSz16       : 16;
    unsigned int BPB_SecPerTrk     : 16;
    unsigned int BPB_NumHeads      : 16;
    unsigned int BPB_HiddSec       : 32;
    unsigned int BPB_TotSec32      : 32;
    unsigned int BS_DrvNum         : 8;
    unsigned int BS_Reserved1      : 8;
    unsigned int BS_BootSig        : 8;
    unsigned int BS_VolID          : 32;
    unsigned char BS_VolLab[11];
    unsigned char BS_FilSysType[8];
} __attribute__((packed));
```

Defines:
bootblock, used in chunk 44.

Alle ausgelesenen und berechneten Werte werden in den Komponenten der globalen Struktur `fatMount` gespeichert. Die FAT-Treiberfunktionen greifen auf die berechneten Werte in `fatMount` zu. Somit muss nicht jedesmal der Bootsektor ausgelesen werden, wenn z.B. die Position der FAT ermittelt werden soll.

43a ⟨*fat type definitions* 38⟩+≡ (89a) ◁42b 44b▷

```
struct fatBPB{
    unsigned int fat_TotalCountOfFATs;
    unsigned int fat_SecPerCluster;
    unsigned int fat_TotalCountOfSectors;
    unsigned int fat_TotalCountOfDataClusters;
    unsigned int fat_FirstSectorOfFAT;
    unsigned int fat_SizeOfFATSectors;
    unsigned int fat_FirstSectorOfRootDir;
    unsigned int fat_SizeOfRootDirSectors;
    unsigned int fat_FirstSectorOfData;
    unsigned int fat_TotalCountOfDataSectors;
    unsigned int fat_BytesPerSector;
    unsigned int fat_FATType;
    unsigned int fat_ReservedSectors;
    unsigned int fat_MaxRootDirEntries;
} __attribute__((packed));
```

Defines:
fatBPB, used in chunk 43b.

Mit Hilfe des Markers `fatinit` wird verhindert, das bei jedem Aufruf der Funktion `fat_open`, erneut der Bootsektor gelesen wird. Die Variablen `fbuf` und `fatMount` werden initialisiert.

43b ⟨*fat global variables* 43b⟩≡ (89a) 57b▷

```
unsigned char fbuf[1024];
struct fatBPB fatMount = { 0 };
int fatinit = 0;
```

Defines:

`fatinit`, used in chunks 44c and 60b.
`fatMount`, used in chunks 44–47, 82d, 84a, 86c, 102b, and 104a.
`fbuf`, used in chunks 35, 37, 44d, 48, 53, 68, 72a, 73, 75a, 83c, and 85–88.
Uses `fatBPB` 43a.

Die Funktion `fat_filesystem_init` liest den Bootsektor aus, analysiert diesen und schreibt das Ergebnis in die Komponenten der globalen Struktur `fatMount`.

44a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁42a 48a▷
```
int fat_filesystem_init (int device);
```
Uses `device` 66a and `fat_filesystem_init` 44c.

Der Bootsektor startet immer bei Sektor `0`. Es wird die Konstante `FAT_bootsector` mit dem Wert `0` definiert.

44b ⟨*fat type definitions* 38⟩+≡ (89a) ◁43a 49a▷
```
#define FAT_bootsector 0
```
Defines:
`FAT_bootsector`, used in chunk 44d.

Nachdem die Initialisierung erfolgreich durchlaufen wurde, wird die globale Varibale `fatinit` auf `1` gesetzt. Somit können die Treiberfunktionen feststellen, ob das FAT-Dateisystem bereits initialisiert wurde.

44c ⟨*fat function implementations* 44c⟩≡ (89b) 48b▷
```
int fat_filesystem_init (int device) {
   struct bootblock* FAT_bootblock;
   struct directory* FAT_DirEntry;
   ⟨init 44d⟩
   fatinit = 1;
}
```
Defines:
`fat_filesystem_init`, used in chunks 44a, 60b, and 102b.
Uses `bootblock` 42b, `device` 66a, and `fatinit` 43b.

Mit Hilfe der Funktion `readblock` wird ein 1024 Byte großer Datenblock eingelesen und in den Buffer `fbuf` geschrieben. Der Buffer wird der Struktur `FAT_bootblock` zugewiesen. Somit werden die Daten aus dem Buffer, in die Komponenten (Bitfields) der Struktur geschrieben.

44d ⟨*init* 44d⟩≡ (44c) 44e▷
```
readblock(device, FAT_bootsector, (char *)fbuf);
FAT_bootblock = (struct bootblock*) &fbuf;
```
Uses `bootblock` 42b, `device` 66a, `FAT_bootsector` 44b, and `fbuf` 43b.

Anschließend werden die ersten Werte aus dem BPB den Komponenten in `fatMount` zugewiesen.

44e ⟨*init* 44d⟩+≡ (44c) ◁44d 45a▷
```
fatMount.fat_BytesPerSector = FAT_bootblock->BPB_BytsPerSector;
fatMount.fat_TotalCountOfFATs = FAT_bootblock->BPB_NumFATs;
fatMount.fat_MaxRootDirEntries = FAT_bootblock->BPB_RootEntCnt;
```

```
fatMount.fat_ReservedSectors = FAT_bootblock->BPB_RsvdSecCnt;
fatMount.fat_SecPerCluster = FAT_bootblock->BPB_SecPerCluster;
```
Uses fatMount 43b.

Zunächst wird überprüft, um welchen FAT-Typ (FAT12, FAT16 oder FAT32) es sich handelt. Damit die Anzahl der Cluster ermittelt werden kann, wird zuerst berechnet, wie viele Sektoren vom *Root-Verzeichnis* und der *Datenregion* belegt werden.

Wie in Kapitel 2.4.3 beschrieben, hat ein Verzeichniseintrag eine Länge von 32 Byte. Die Komponente `BPB_RootEntCnt` enthält die maximale Anzahl von Verzeichniseinträgen im *Root-Verzeichnis*. Die Anzahl der Sektoren im *Root-Verzeichnis* werden mit folgender Formel berechnet:

(BPB_RootEntCnt × 32/BPB_BytsPerSec)

Da bei FAT32-Dateisystemen `BPB_RoootEntCnt` 0 ist, wird `BPB_BytsPerSec -1` zum Wert `BPB_RoootEntCnt` addiert.

45a ⟨*init* 44d⟩+≡ (44c) ◁44e 45b▷
```
fatMount.fat_SizeOfRootDirSectors =
((fatMount.fat_MaxRootDirEntries * 32) +
(fatMount.fat_BytesPerSector - 1)) / fatMount.fat_BytesPerSector;
```
Uses fatMount 43b.

Als Nächstes wird die Anzahl der Cluster in der *Datenregion* bestimmt. Hierzu wird zuerst der Wert `BPB_TotSec16` ausgelesen. Für FAT12- und FAT16-Dateisysteme enthält `BPB_TotSec16` die Anzahl der Sektoren auf dem Laufwerk. Bei einem FAT32-Dateisystem beträgt der Wert von `BPB_TotSec16` `0`. Da FAT32 nicht unterstützt wird, bricht die Initialisierung an dieser Stelle ab.

45b ⟨*init* 44d⟩+≡ (44c) ◁45a 45c▷
```
if (FAT_bootblock->BPB_TotSec16 != 0)
  fatMount.fat_TotalCountOfSectors = FAT_bootblock->BPB_TotSec16;
else
  return -1; //FAT32 wird nicht unterstützt
```
Uses fatMount 43b.

`BPB_FATSz16` enthält die Anzahl der Sektoren pro FAT. Bei FAT32-Dateisystemen ist dieser Wert `0`, deshalb bricht die Funktion hier ab.

45c ⟨*init* 44d⟩+≡ (44c) ◁45b 46a▷
```
if (FAT_bootblock->BPB_FATSz16 != 0)
  fatMount.fat_SizeOfFATSectors = FAT_bootblock->BPB_FATSz16;
else
  return -1; //FAT32 wird nicht untersützt
```
Uses fatMount 43b.

An diesem Punkt wurden alle Voraussetzungen ermittelt, um die Anzahl der Sektoren in der *Datenregion* zu ermitteln. Dazu wird von der Gesamtanzahl aller Sektoren, die Sektoren-Zahl von Bootsektor, FATs und *Root-Verzeichnis* abgezogen, siehe Kapitel 2.4 Aufbau von FAT12.

46a ⟨*init* 44d⟩+≡ (44c) ◁45c 46b▷
```
fatMount.fat_TotalCountOfDataSectors =
  fatMount.fat_TotalCountOfSectors - (fatMount.fat_ReservedSectors +
  (fatMount.fat_TotalCountOfFATs * fatMount.fat_SizeOfFATSectors) +
  fatMount.fat_SizeOfRootDirSectors);
```
Uses `fatMount` 43b.

Nachdem die Sektoren bestimmt wurden, können auch die Cluster berechnet werden. Hierzu wird die Anzahl der Sektoren durch die Anzahl der Sektoren pro Cluster ganzzahlig geteilt. Dabei ist es wichtig zu verstehen, dass ein Cluster mehrere Sektoren enthalten kann, siehe Kapitel 2.4.6.

46b ⟨*init* 44d⟩+≡ (44c) ◁46a 46c▷
```
fatMount.fat_TotalCountOfDataClusters = fatMount.fat_TotalCountOfDataSectors /
                                        fatMount.fat_SecPerCluster;
```
Uses `fatMount` 43b.

Mit Hilfe der Cluster-Anzahl kann der FAT-Typ bestimmt werden.

- Ein FAT12-Dateisystem hat höchstens 4.084 Cluster.
- Ein FAT16-Dateisystem hat mindestens 4.085 Cluster und höchstens 65.524 Cluster.
- Ein FAT32-Dateisystem hat mindestens 65.525 Cluster.

Der FAT-Typ wird ausschließlich über die Anzahl der Cluster definiert (vgl. Microsoft, 2000, S. 15). Je nach FAT-Typ wird der entsprechende Wert der Komponente `fat_FATType` der globalen Struktur zugewiesen. Werden die Dateisysteme FAT16 und FAT32 ermittelt, so bricht die Funktion ab und gibt den Fehlercode `-1` zurück.

46c ⟨*init* 44d⟩+≡ (44c) ◁46b 46d▷
```
if(fatMount.fat_TotalCountOfSectors < 4085) {
  fatMount.fat_FATType = 12;
} else if(fatMount.fat_TotalCountOfSectors < 65525) {
  return -1; //FAT16 wird nicht unterstützt
} else {
  return -1; //FAT32 wird nicht unterstützt
}
```
Uses `fatMount` 43b.

Als letzter Schritt der Initialisierung werden die Positionen (Offsets) von *Root-Verzeichnis*, FAT und *Datenregion* berechnet. Bei FAT12- und FAT16-Dateisystemen befindet sich das *Root-Verzeichnis* direkt hinter den FATs. Der Offset des *Root-Verzeichnisses* lässt sich bestimmen, indem die Anzahl der FATs mit den Sektoren pro FAT multipliziert und anschließend die Bootsektoren addiert werden (vgl. Microsoft, 2000, S. 22).

46d ⟨*init* 44d⟩+≡ (44c) ◁46c 47a▷
```
fatMount.fat_FirstSectorOfRootDir = ((fatMount.fat_SizeOfFATSectors *
                                      fatMount.fat_TotalCountOfFATs) +
                                      fatMount.fat_ReservedSectors);
```
Uses `fatMount` 43b.

Die FATs beginnen direkt nach dem Bootsektor. Der folgende Code-Chunk definiert die Startposition der ersten FAT.

47a ⟨*init* 44d⟩+≡ (44c) ◁46d 47b▷
```
fatMount.fat_FirstSectorOfFAT = fatMount.fat_ReservedSectors;
```
Uses `fatMount` 43b.

Der Offset des ersten Datensektors ermittelt sich durch die Addition von Bootsektor, FATs und *Root-Verzeichnis-Sektoren* (vgl. Microsoft, 2000, S. 14).

47b ⟨*init* 44d⟩+≡ (44c) ◁47a
```
fatMount.fat_FirstSectorOfData = fatMount.fat_ReservedSectors +
                                 (fatMount.fat_SizeOfFATSectors *
                                  fatMount.fat_TotalCountOfFATs) +
                                 ((fatMount.fat_MaxRootDirEntries * 32) /
                                  fatMount.fat_BytesPerSector);
```
Uses `fatMount` 43b.

Die Abbildung 4.1 zeigt den Aufbau des FAT12-Dateisystems, des verwendeten Floppy-Images. Nach der Initialisierung wurden die Regionen BPB (Blau), FAT (Grün), *Root-Verzeichnis* (Rot) und *Datenregion* (ab Cluster 33) ermittelt. Neben den 512-Byte-Clustern sind auch die 1024er-Blöcke eingezeichnet, wie sie von `readblock` gelesen werden.

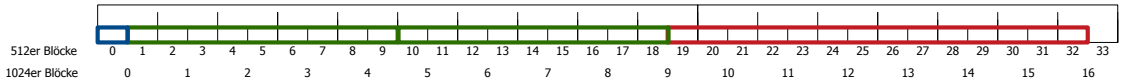


Abbildung 4.1.: FAT Layout nach Initialisierung

4.5. Verkettete Listen

Ein zentrales Element eines FAT-Dateisystems sind die FATs. Ohne die Einträge und die verketteten Listen sind die Daten in den Daten-Clustern nutzlos. Dieses Kapitel beschreibt die Implementierung der Funktion `read_fat_entry`. Das Grundprinzip findet sich in späteren Implementierungen, wie z. B. `fat_get_free_fat_entry` wieder. Bereits in Kapitel 2.4.2 wurde beschrieben, dass ein FAT12-Eintrag zwölf Bit lang ist. Das bedeutet, dass drei Bytes zwei Einträge enthalten. Die Abbildung 4.2 verdeutlicht noch einmal den Aufbau einer FAT12-Tabelle und die Verteilung von drei Einträgen auf vier Bytes.

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4
0000 0000	0000 1111	1111 1111	2222 2222	2222 3333

Abbildung 4.2.: CLNs belegen unter FAT12 1,5 Bytes, zwölf Bits.

Die Funktion `read_fat_entry` liest den Wert eines FAT-Eintrages aus und gibt ihn anschließend zurück.

48a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁44a 50b▷
```
int read_fat_entry (int device, int fatpos);
```
Uses `device` 66a and `read_fat_entry` 48b.

Aus Abbildung 4.2 erkennt man, dass die CLN eins die Bytes eins und zwei belegt. Zuerst wird die echte Position in Bytes `fateoff` des Eintrages in der FAT ermittelt. Dazu wird die Nummer des FAT-Eintrages mit 1,5 multipliziert, da ein Eintrag eineinhalb Bytes belegt (12 Bit). Da die FAT direkt nach dem Bootsektor beginnt und ein Sektor = ein Cluster 512 Byte belegt, wird ein Offset von 512 Byte addiert. Somit enthält `fateoff` die Byte Position des Eintrages in der FAT. Da die Funktion `readblock` 1024-Byte-Blöcke liest, wird `fateoff` durch 1024 geteilt und `reblock` gespeichert.

48b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁44c 50c▷
```
int read_fat_entry (int device, int fatpos){
  int fateoff = (int)(fatpos * 1.5) + 512;
  int reblock = (int)(fateoff / 1024);

  readblock (device,reblock, (char *)fbuf);

  ⟨little to big endian 48c⟩
  ⟨get fat value 48d⟩
}
```
Defines:
`read_fat_entry`, used in chunks 48a, 53, 67a, 68, 71a, 74a, 75b, 78b, and 88b.
Uses `device` 66a, `fbuf` 43b, and `reblock` 73.

Nachdem der Block ermittelt und gelesen wurde, wird im nächsten Schritt der Offset innerhalb des gelesenen Blocks ermittelt. Die Variable `blockoff` enthält das Ergebnis der Modulo-Operation und somit den Offset. Mit Hilfe des Offsets lässt sich die exakte Position der beiden Bytes bestimmen. Da FAT-Einträge in *Little Endian* gespeichert sind, werden die beiden Bytes, nach *Big Endian* konvertiert und in der Variable `fatentry` gespeichert.

48c ⟨*little to big endian* 48c⟩≡ (48b)
```
int blockoff = (int)(fateoff%1024);
unsigned short fatentry;

fatentry = (fbuf[blockoff+1]<<8) | fbuf[blockoff];
```
Uses `blockoff` 73, `fatentry` 82d, and `fbuf` 43b.

Nach der Konvertierung in *Big Endian* lässt sich der FAT-Eintrag auswerten. Mit Hilfe einer Modulo-Operation auf die FAT-Nummer wird ermittelt, ob die zwölf höheren oder die zwölf niedrigeren Bits der beiden Bytes betrachtet werden.

48d ⟨*get fat value* 48d⟩≡ (48b)
```
if (fatpos % 2 > 0)
  return fatentry >> 4;
else
  return (fatentry & 0x0FFF);
```
Uses `fatentry` 82d.

4.6. Verzeichniseinträge

Dieses Kapitel beschreibt, wie mit Verzeichniseinträgen gearbeitet wird. Dazu wird auf die Funktionen `fat_file_exists` und `converto_short` eingegangen. Die Funktion `fat_file_exists` durchsucht das *Root-Verzeichnis* und ggf. Unterverzeichnisse nach einem Verzeichniseintrag. Mit Hilfe der Funktion `converto_short` werden Dateinamen in das 8.3-Format konvertiert. Ein Verzeichniseintrag ist eine 32-Byte-Datenstuktur, welche in Kapitel 2.4.3 genauer beschrieben wurde. Verzeichniseinträge befinden sich im *Root-Verzeichnis* oder als Unterverzeichniseinträge in einem Daten-Cluster. Das *Root-Verzeichnis*, beispielsweise, ist eine lineare Liste aus 32-Byte-Strukturen. Um auf die einzelnen Elemente eines Verzeichniseintrages zugreifen zu können, wird folgende Datenstruktur eingeführt:

49a ⟨*fat type definitions* 38⟩+≡ (89a) ◁44b 56▷

```
typedef struct {
   unsigned char DIR_shortName[11];
   unsigned int DIR_attr             : 8;
   unsigned int DIR_reserv           : 8;
   unsigned int DIR_cnto             : 8;
   unsigned int DIR_TimeCreate       : 16;
   unsigned int DIR_DateCreate       : 16;
   unsigned int DIR_DateLastAccess   : 16;
   unsigned int DIR_FirstClusterHi   : 16;
   unsigned int DIR_TimeLastMod      : 16;
   unsigned int DIR_DateMod          : 16;
   unsigned int DIR_FirstClusterLow  : 16;
   unsigned int DIR_FileSize         : 32;
}__attribute__((packed)) dirEntry;
```

Defines:
`dirEntry`, used in chunks 35, 50, 52, 53, 60c, 73, 77, 80, and 86–88.

Zur einfacheren Verarbeitung der Verzeichnisattribute werden folgende Konstanten definiert. Die Dateiattribute wurden in Kapitel 2.4.3 genau beschrieben.

49b ⟨*fat constants* 49b⟩≡ (89a) 50a▷

```
#define ATTR_LONG_NAME      0x0F
#define ATTR_READ_ONLY      0x01
#define ATTR_HIDDEN         0x02
#define ATTR_SYSTEM         0x04
#define ATTR_VOLUME_ID      0x08
#define ATTR_DIRECTORY      0x10
#define ATTR_ARCHIVE        0x20
```

Defines:
`ATTR_ARCHIVE`, used in chunk 80.
`ATTR_DIRECTORY`, used in chunk 63a.
`ATTR_HIDDEN`, never used.
`ATTR_LONG_NAME`, never used.
`ATTR_READ_ONLY`, used in chunk 60c.
`ATTR_SYSTEM`, never used.
`ATTR_VOLUME_ID`, never used.

Weiter wird in der Implementierung u.a. überprüft, ob Verzeichniseinträge frei sind.

Dazu werden ebenfalls Konstanten definiert.

50a ⟨*fat constants* 49b⟩+≡ (89a) ◁49b 57a▷

```
#define SPACE                0x20
#define DIR_ENTRY_IS_FREE    0xE5
#define FIRST_LONG_ENTRY     0x01
#define SECOND_LONG_ENTRY    0x42
```

Defines:
`DIR_ENTRY_IS_FREE`, used in chunk 85.
`FIRST_LONG_ENTRY`, never used.
`SECOND_LONG_ENTRY`, never used.
`SPACE`, never used.

4.6.1. fat_file_exists

Die Funktion `fat_file_exists` überprüft, ob eine Datei bzw. ein Verzeichniseintrag im Dateisystem existiert. Wurde die Datei gefunden, gibt sie die absolute Byte-Position im Dateisystem zurück und schreibt den Inhalt des Verzeichniseintrages in `dentry`. War die Suche erfolglos, gibt sie `-1` zurück.

50b ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁48a 54a▷

```
long fat_file_exists(int device, const char *path, dirEntry *dentry);
```

Uses `device` 66a, `dirEntry` 49a, and `fat_file_exists` 50c.

Zu Beginn der Funktion wird ein Buffer deklariert, in welchen der übergebene Pfad geschrieben und anschließend Null-terminiert wird. Die Funktion wird sofort beendet, wenn der Buffer nur die Terminierung enthält. In einer Schleife wird der Buffer sequenziell durchlaufen.

50c ⟨*fat function implementations* 44c⟩+≡ (89b) ◁48b 54b▷

```
long fat_file_exists (int device, const char *path, dirEntry *dentry) {
   char searchbuf[256] = { 0 };
   char *fsearch = (char*)searchbuf;

   char comparestr[12] = { 0 };

   int lookinroot = 1;
   int hitcluster = -1;

   dirEntry *rdentry;
   int gotit = 0;
   unsigned long totalposinbytes;

   strncpy(fsearch, path, strlen(path));
   fsearch[strlen(path)+1] = '\0';

   fsearch++;
   if(*fsearch == '\0')
      return 0;

   while(*fsearch != '\0'){
```

```
        ⟨search loop 51⟩
    }
    if (hitcluster == -1 || gotit == 0){ return -1;}

    *dentry = *rdentry;
    return totalposinbytes;
}
```

Defines:
  `fat_file_exists`, used in chunks 50b, 60c, 73, 77, and 88a.
Uses `device` 66a, `dirEntry` 49a, `strlen` 42a 42a, and `strncpy` 42a.

Innerhalb der Schleife wird nach den einzelnen Pfadkomponenten `subpath` gesucht. Verzeichnisse eines Pfades werden durch einen Slash getrennt. Der Pfad selbst beginnt im *Root-Verzeichnis* und kann über n-Unterverzeichnisse zu einer Datei führen. Die Suche im *Root-Verzeichnis* wird im Code-Chunck ⟨*search loop root* 52⟩ beschrieben. Alle weiteren Suchen in den Unterverzeichnissen bzw. den Datenregionen werden im Code-Chunk ⟨*search loop data* 53⟩ beschrieben. Mit Hilfe des Markers `lookinroot` wird die erste Pfadkomponente `subpath` im *Root-Verzeichnis* gesucht. Nachdem das *Root-Verzeichnis* durchsucht wurde, wird der Marker auf `0` gesetzt. Damit `subpath` vergleichbar mit den ausgelesenen Verzeichniseinträgen ist, wird `subpath` in das 8.3-Format konvertiert.

51 ⟨*search loop* 51⟩≡ (50c)

```
char subpathshort[12] = { 0 };
char subpath[31] = { 0 };
int i = 0;

while(*fsearch != '\0' && *fsearch != '/'){
   subpath[i] = *fsearch;
   fsearch++; i++;
}
subpath[i] = '\0';

convertto_shortname(subpath, subpathshort);
subpathshort[11] = '\0';

if (lookinroot == 1) {
   ⟨search loop root 52⟩
} else {
   ⟨search loop data 53⟩
}

if (*fsearch != '\0')
  fsearch++;
else
  break;
```

Uses `convertto_shortname` 54b, `i`, `subpath`, and `subpathshort`.

Im Unterschied zur Suche in der *Datenregion* wird bei der Suche im *Root-Verzeichnis* zuerst das gesamte *Root-Verzeichnis* eingelesen (erste Schleife) und im Anschluss in 32-Byte-Schritten durchlaufen. Die Nummern der Blöcke, die das *Root-Verzeichnis* enthalten, werden im `root_blocks` fest definiert. Dabei handelt es sich um 1024er-Blöcke, wie

sie `readblock` liest. `rootoffset` wird benötigt, da das *Root-Verzeichnis* bei Block 9,5 beginnt, siehe Abbildung 4.1. In einer zweiten Schleife werden die Verzeichniseinträge ab `rootoffset` in 32-Byte-Schritten durchlaufen und der Datenstruktur `rdentry` zugewiesen. Dadurch kann auf die Verzeichnisnamen `DIR_shortName` zugegriffen werden. Damit `DIR_shortName` und `subpathshort` verglichen werden können, wird `DIR_shortName` in `comparestr` terminiert. Sobald ein Eintrag gefunden wurde, wird die Variable `nextcluster` mit der Nummer des Daten-Cluster belegt, in welchem beim nächsten Schleifendurchlauf gesucht werden soll. Zuletzt wird der Marker `lookinroot` gelöscht, da nur im ersten Durchlauf im *Root-Verzeichnis* gesucht wird. Wie in Kapitel 2.4.4 beschrieben, befinden sich *Root-Verzeichnis* und Unterverzeichnisse an unterschiedlichen Positionen auf dem Datenträger. Der Pfad zur Datei, nach der gesucht wird, beginnt immer mit dem *Root-Verzeichnis*, ggf. gefolgt von Unterverzeichnissen. Damit die absolute Byte-Position am Ende der Funktion zurückgegeben werden kann, wird `totalposinbytes` hochgezählt.

52 ⟨*search loop root* 52⟩≡ (51)

```
int root_blocks[8] = {9,10,11,12,13,14,15,16};
char tmp_root[8*1024] = { 0 };
totalposinbytes = 9 * 1024;

for (i = 0; i < 8; i++)
   readblock (device, root_blocks[i], tmp_root + i * 1024);

int rootoffset = 512;
totalposinbytes += 512;

for (i = rootoffset ;i < (8*1024)-512; i += 32){
   rdentry = (dirEntry*) &tmp_root[i];

   memcpy(comparestr,&rdentry->DIR_shortName,12 * sizeof(char));
   comparestr[11] = '\0';
   if (strcmp(comparestr,subpathshort)){
      hitcluster = rdentry->DIR_FirstClusterLow;
      lookinroot = 0;
      gotit = 1;
      break;
   }
   totalposinbytes += 32;
}
```

Uses `device` 66a, `dirEntry` 49a, `i`, `memcpy` 42a, `root_blocks`, `rootoffset`, `strcmp` 42a, `subpathshort`, and `tmp_root`.

Nachdem das *Root-Verzeichnis* durchsucht wurde, führt die Schleife die Suche in den Unterverzeichnissen fort. Dazu werden die Cluster in der *Datenregion* durchsucht. Grundsätzlich ist die Suche identisch wie im *Root-Verzeichnis*. Auch hier werden Cluster in 32-Byte-Schritten durchlaufen und `totalposinbytes` mitgezählt. Im Unterschied zur Suche in *Root-Verzeichnissen* werden bei diesen Schleifendurchläufen einzelne Daten-Cluster durchsucht. Unterverzeichnisse können im Gegensatz zum *Root-Verzeichnis* beliebig groß werden. Aus diesem Grund muss bei der Suche in Unterverzeichnissen die Cluster-Kette durchlau-

fen werden. Gelöst wird das durch eine zusätzliche Schleife, die bis ans Ende der Kette `0xFFF` läuft. Wird ein Eintrag vorher gefunden, bricht die Schleife ab.

53 ⟨*search loop data* 53⟩≡ (51)

```
int curcluster = hitcluster;
int nextclstno;

while(curcluster != 0xFFF){
   gotit = 0;
   int blockoffset;
   totalposinbytes = (31 + curcluster) * 512;
   char dircluster[512] = { 0 };

   if (curcluster % 2 > 0){
      blockoffset = 0;
   } else {
      blockoffset = 512;
   }
   readblock (device,(int)((31+(int)curcluster)/2), (char *)fbuf);
   memcpy (dircluster, &fbuf[blockoffset], 512 * sizeof(char));


   for (i = 0; i < 512; i += 32){
      rdentry = (dirEntry*) &dircluster[i];
      memcpy(comparestr,&rdentry->DIR_shortName,12 * sizeof(char));
      comparestr[11] = '\0';
      if (strcmp(comparestr,subpathshort)){
         hitcluster = rdentry->DIR_FirstClusterLow;
         gotit = 1;
         break;
      }
      totalposinbytes += 32;
   }
   if(gotit == 1){
      break;
   } else {
      nextclstno = read_fat_entry (device, curcluster);
      curcluster = nextclstno;
   }
}
```

Uses `blockoffset` 67b 71b, `curcluster`, `device` 66a, `dircluster`, `dirEntry` 49a, `fbuf` 43b, `i`, `memcpy` 42a, `nextclstno`, `read_fat_entry` 48b, `strcmp` 42a, and `subpathshort`.

4.6.2. convertto_shortname

`convertto_shortname` ist eine Hilfsfunktion, welche ein beliebiges `char`-Array in ein 8.3-Format konvertiert. Diese Funktionalität wird zum einen benötigt, um Dateipfade, welche aus dem User Mode übergeben wurden, zu konvertieren. Konvertierte Dateinamen lassen sich mit den ausgelesenen Dateinamen der FAT-Verzeichniseinträgen vergleichen. Zum anderen müssen Dateinamen, welche in Verzeichniseinträge geschrieben werden, in das 8.3-Format konvertiert werden. Tabelle 4.1 zeigt einige Beispiele der Konvertierung.

Pfad	8.3 Format
ONE.TXT	"ONE␣␣␣␣␣TXT"
ONEONEONEONEONE.TXT	"ONEONE~1TXT"
ONE	"ONE␣␣␣␣␣␣␣␣"
ONEONEONEONEONE	"ONEONE~1␣␣␣"

Tabelle 4.1.: Beispiele für Konvertierung in 8.3-Format

54a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁50b 59a▷
```
void convertto_shortname(char *name, char *shortname);
```
Uses convertto_shortname 54b and shortname 80.

Die Funktion verarbeitet den übergebenen Dateinamen in einem von zwei Blöcken. Enthält der Dateiname eine Dateiendung, wird der Block ⟨*conv ext* 55a⟩ verarbeitet. Ist der Dateiname ohne Dateiendung steigt die Funktion in den Block ⟨*conv noext* 55b⟩ ein. Der Dateiname im 8.3-Format wird in der Variable `filename` aufgebaut. Wurde der Dateiname konvertiert, wird das Ergebnis in `shortname` kopiert.

Zuerst wird überprüft, ob der Dateiname `name` eine Dateiendung hat. Nach dem Schleifendurchlauf enthält `dotpos` die Position des `.` im Dateinamen.

54b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁50c 59b▷
```
void convertto_shortname (char *name, char *shortname) {
  char filename[11] = {' '};
  int lenpath;
  int dotpos = 0;
  int i;

  for (i = strlen(name); i >= strlen(name)-4; i-){
    if (name[i] == '.'){
      dotpos = i;
    }
  }

  if (dotpos > 0){
    ⟨conv ext 55a⟩
  } else {
    ⟨conv noext 55b⟩
  }
  memcpy (shortname, filename, 11);
}
```
Defines:
convertto_shortname, used in chunks 51, 54a, 80, and 103b.
Uses filename 76b, i, memcpy 42a, shortname 80, and strlen 42a 42a.

Ist `dotpos` größer `0` so werden Dateiendung und Dateiname separat betrachtet. Zunächst wird die Dateiendung ermittelt. In einer Switch/Case-Operation werden abhängig von der

Anzahl der Positionen in der Dateiendung die Zeichen in die entsprechende Position des Arrays `filename` kopiert.

Anschließend wird der Dateiname verarbeitet. Dabei kommt es darauf an, ob der Dateiname mehr als sieben Zeichen hat. In diesem Fall muss ab dem sechsten Zeichen gekürzt werden, ist der Dateiname kürzer, müssen die verbleibenen Zeichen mit Leerzeichen aufgefüllt werden. Zusätzlich muss überprüft werden das der . nicht Teil des `shortname` wird, da er sich an einer Position kleiner sieben befindet, siehe Tabelle 4.1.

55a ⟨*conv ext* 55a⟩≡ (54b)

```
switch(strlen(name) - dotpos-1){
  case 1:   filename[8] = name[dotpos+1];
            break;
  case 2:   filename[8] = name[dotpos+1];
            filename[9] = name[dotpos+2];
            break;
  case 3:   filename[8] = name[dotpos+1];
            filename[9] = name[dotpos+2];
            filename[10] = name[dotpos+3];
            break;
}

int strborder=0;

if (strlen(name) > 7){
  lenpath = 0;
  for (int i = 0; i < 6; i++){
    filename[i] = name[lenpath];
    lenpath++;
  }
  filename[6] = 0x7E; //~
  filename[7] = 0x31; //1
} else {
  lenpath = 0;
  for (int i = 0; i <= 7; i++){
    if (name[i] == '.')
       strborder = 1;
    if (strborder != 1){
      filename[i] = name[lenpath];
      lenpath++;
    } else {
      filename[i] = ' ';
      lenpath++;
    }
  }
}
```

Uses `filename` 76b, `i`, `strborder`, and `strlen` 42a 42a.

Ist keine Dateiendung vorhanden, so muss lediglich überprüft werden, ob der Dateiname gekürzt oder aufgefüllt werden muss.

55b ⟨*conv noext* 55b⟩≡ (54b)

```
if (strlen(name) > 7){
```

```
    for (int i = 0; i < 6; i++){
      filename[i] = name[i];
    }
    filename[6] = 0x7E; //~
    filename[7] = 0x31; //1
    filename[8] = 0x20;
    filename[9] = 0x20;
    filename[10] = 0x20;
  } else {
    for (int i = 0; i < 12; i++){
      if (i > strlen(name)-1){
        filename[i] = ' ';
      } else {
        filename[i] = name[i];
      }
    }
  }
```

Uses `filename` 76b, `i`, and `strlen` 42a 42a.

4.7. Öffnen und Schließen von Dateien

Wenn der User die Standard-Dateisystemfunktionen `open`, `close`, `read` und `write` aufruft, koordiniert das VFS die Operationen auf den unterstützten Dateisystemen. Dazu müssen die bereits vorhandenen Funktionen `u_open`, `u_write`, usw. erweitert werden, damit sie das FAT-Dateisystem unterstützen. In diesem Kapitel wird die Implementierung von `fat_open`, `fat_close`, `fat_read` und `fat_write` behandelt. Damit der Kernel die geöffneten Dateien verwalten kann, werden einige Datenstrukturen benötigt.

4.7.1. Interne FAT Inode

Da das FAT-Dateisystem das Konzept von Inodes nicht kennt, siehe Kapitel 3.4, werden Inodes zur Laufzeit erstellt. Die erstellten Inodes werden im Speicher vorgehalten. Sämtliche Inode-Operationen werden ausschließlich im Speicher getätigt. Das Zurückschreiben auf den Datenträger ist mit FAT-Dateisystemen nicht möglich. Die folgende Struktur repräsentiert einen internen Inode. Jedes mal wenn eine Datei geöffnet wird, wird ein interner Inode reserviert.

56 ⟨*fat type definitions* 38⟩+≡ (89a) ◁49a 57d▷

```
struct int_fat_inode {
    char path[256];          //Lokaler Dateipfad
    unsigned int ino;        //Array Index aus fat_inodes
    unsigned int i_uid;      //User ID
    unsigned int i_gid;      //Group ID
    unsigned int i_atime;    //Time last access
    unsigned int i_mtime;    //Time last mod
    unsigned int i_ctime;    //Time creation
    unsigned int i_size;     //File size
    unsigned int i_attr;     //File Attribute
```

```
    unsigned long i_totalpos;  //Absolute Position im Dateisystem
    short    i_mode;           //Dateizugriffsmodus
    short    device;           //Datenträger
    unsigned int refcount;     //Anzahl geöffneter Dateien
};
```

Defines:
`int_fat_inode`, used in chunks 31b, 57, 62b, and 69b.
Uses `device` 66a and `fat_inodes`.

Die interne Inode-Tabelle `fat_inodes` kann 256 verschiedene Inodes für geöffnete Dateien verwalten.

57a ⟨*fat constants* 49b⟩+≡ (89a) ◁50a 57c▷

```
#define MAX_FAT_INT_INODES 256
```

Defines:
`MAX_FAT_INT_INODES`, used in chunks 57b, 59b, and 61b.

57b ⟨*fat global variables* 43b⟩+≡ (89a) ◁43b 58c▷

```
struct int_fat_inode fat_inodes[MAX_FAT_INT_INODES] = { 0 };
```

Uses `fat_inodes`, `int_fat_inode` 56 64a 64d 66a 87c, and `MAX_FAT_INT_INODES` 57a.

Für das zurücksetzen von Werten wird ein NULL-Pointer definiert.

57c ⟨*fat constants* 49b⟩+≡ (89a) ◁57a 58a▷

```
#define NULL ((void*) 0)
```

Defines:
`NULL`, used in chunks 59c, 64a, and 69b.

Lokaler Dateideskriptor

Der LFD ist ein positiver Wert, welcher von der Funktion `fat_open` zurückgegeben wird. Da der Dateideskriptor eine Datei eindeutig identifiziert und über das gesamte FAT-Dateisystem seine Gültigkeit besitzt, können weitere Funktionen auf geöffnete Dateien zugreifen.

Datei Status

Die Struktur `fat_filestat` ist mit `int_fat_inode` verknüpft. Des Weiteren enthält die Struktur Informationen zur aktuellen read/write Position auf Cluster und Byte Ebene, Dateinamen und Pfad, das erste Cluster der Datei und den Zugriffsmodus.

57d ⟨*fat type definitions* 38⟩+≡ (89a) ◁56

```
struct fat_filestat {
    struct int_fat_inode *int_inode;
    unsigned int curcluster;         //aktuelles Cluster
    char name[256];                  //Dateiname
    char path[256];                  //Lokaler Dateipfad
    unsigned int FirstClusterL;      //Erstes Daten Cluster
    int pos;
    short mode;
};
```

Defines:
`fat_filestat`, used in chunks 58c and 69b.
Uses `curcluster` and `int_fat_inode` 56 64a 64d 66a 87c.

ULIX unterstützt folgende Modi zum Öffnen von Dateien (vgl. Eßer & Freiling, 2014, S. 424):

58a ⟨*fat constants* 49b⟩+≡ (89a) ◁57c 58b▷
```
#define O_RDONLY        0x0000     /* read only */
#define O_WRONLY        0x0001     /* write only */
#define O_RDWR          0x0002     /* read and write */
#define O_APPEND        0x0008     /* append mode */
#define O_CREAT         0x0200     /* create file */
```
Defines:
`O_APPEND`, used in chunks 60c, 63b, 64d, 104, and 105.
`O_CREAT`, used in chunks 60c, 104, and 105.
`O_RDONLY`, used in chunks 69b, 104, and 105.
`O_RDWR`, used in chunks 60c, 104, and 105.
`O_WRONLY`, used in chunks 60c, 104, and 105.

- **O_RDONLY und O_WRONLY** Datei nur zum Lesen bzw. Schreiben öffen.
- **O_RDWR** Sowohl Lese- als auch Schreibzugriff.
- **O_APPEND** Kann in Verbindung mit `O_WRONLY` verwendet werden. In diesem Fall werden alle Schreiboperationen am Dateiende ausgeführt.
- **O_CREAT** Mit diesem Modus werden Dateien neu erstellt. Dateien welche nicht existieren, werden mit `O_CREAT` neu angelegt.

Das Array `fat_status` wird mit `0` initialisiert und enthält 256 Einträge mit der Struktur `fat_filestat`. Somit ist es möglich, maximal 256 Dateien gleichzeitig zu öffnen. Der Index des Arrays ist der LFD, welcher von `fat_open` zurückgegeben wird.

58b ⟨*fat constants* 49b⟩+≡ (89a) ◁58a 64b▷
```
#define FAT_MAX_FILES 256
```
Defines:
`FAT_MAX_FILES`, used in chunks 58c, 59c, 64, 66a, and 69b.

58c ⟨*fat global variables* 43b⟩+≡ (89a) ◁57b 81a▷
```
struct fat_filestat fat_status[FAT_MAX_FILES] = { 0 };
```
Uses `fat_filestat` 57d 64a 64d 66a 87c, `FAT_MAX_FILES` 58b, and `fat_status`.

Abbildung 4.3 veranschaulicht die Beziehung zwischen `fat_status` und `fat_inodes`. Wenn eine Datei zweimal geöffnet wird, so existiert intern nur ein Inode-Eintrag. Die Inodes werden über den LFD (Index `fat_status`) referenziert.

Zum Ermitteln freier Einträge aus den Listen von Inodes und LFDs werden zwei Hilfsfunktionen implementiert, welche über das jeweilige Array iterieren und den nächst möglichen freien Eintrag zurückgeben. Freie Inode-Einträge werden über `refcount == 0` ermittelt und Status-Einträge über eine leere Inode-Referenz `int_inode == NULL`. Wurde kein

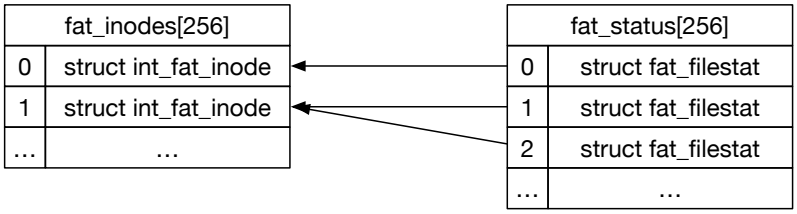


Abbildung 4.3.: n:1-Beziehung zwischen LFD und Inode.

freies Element gefunden, was bedeutet, dass 256 Dateien geöffnet bzw. alle Inodes belegt sind, so wird `-1` zurückgegeben.

59a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁54a 60a▷

```
int fat_get_free_inode_entry();
int fat_get_free_status_entry();
```

Uses `fat_get_free_inode_entry` 59b and `fat_get_free_status_entry` 59c.

59b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁54b 59c▷

```
int fat_get_free_inode_entry() {
   for (int i = 0; i < MAX_FAT_INT_INODES; i++) {
      if (fat_inodes[i].refcount == 0) return i;
   }
return -1;
}
```

Defines:
`fat_get_free_inode_entry`, used in chunks 31b, 59a, and 62a.
Uses `fat_inodes`, `i`, and `MAX_FAT_INT_INODES` 57a.

59c ⟨*fat function implementations* 44c⟩+≡ (89b) ◁59b 60b▷

```
int fat_get_free_status_entry() {
   for (int i = 0; i < FAT_MAX_FILES; i++) {
       if (fat_status[i].int_inode == NULL) return i;
   }
   return -1;
}
```

Defines:
`fat_get_free_status_entry`, used in chunks 59a and 61a.
Uses `FAT_MAX_FILES` 58b, `fat_status`, `i`, and `NULL` 57c.

4.7.2. fat_open

Das VFS von ULIX ist in der Lage, aus einem FAT-LFD einen globalen FD zu erzeugen. Damit das VFS die globalen FDs der unterschiedlichen Dateisysteme auseinanderhalten kann, wurden Nummernkreise für einzelne Dateisysteme definiert. Der Nummernkreis für das FAT-Dateisystem beträgt `512 - 767`. Das VFS von ULIX definiert die globalen FDs über einen Wert des Typs `short`. Dabei repräsentieren die ersten 8 Bit das Dateisystem (MINIX, FAT, etc.) und die zweiten 8 Bit den lokalen FD (`0 - 255`) des jeweiligen Dateisystems (vgl. Eßer & Freiling, 2014, S. 295).

Die Aufgabe von `fat_open` ist es, einen LFD zu erzeugen und zurückzugeben. Beim Erzeugen eines neuen LFD muss geprüft werden, ob nicht bereits eine Inode-Datenstruktur existiert. Ist das der Fall, so referenziert der neue LFD den existierenden Inode, wenn nicht, wird ein neuer Inode erstellt und mit dem LFD verknüpft.

60a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁59a 63c▷

```
int fat_open (int device, const char *path, int oflag);
```

Uses `device` 66a and `fat_open` 60b.

Wird zum ersten Mal durch `fat_open` auf das FAT-Dateisystem zugegriffen, wird das Dateisystem initialisiert. Dabei wird, wie im Kapitel 4.4 beschrieben, der Bootsektor ausgelesen, analysiert und seine Informationen dateisystemweit zur Verfügung gestellt. Durch die globale Variable `fatinit` wird sichergestellt, dass nicht bei jedem `fat_open` Aufruf das ganze Dateisystem initialisiert wird.

60b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁59c 64a▷

```
int fat_open (int device,const char *path, int oflag) {
   if (fatinit == 0)
      if (fat_filesystem_init (device) == -1)
         printf("FAT-Dateisystem kann nicht initalisiert werden.");

   ⟨fat open 60c⟩

   return ffd;
}
```

Defines:
`fat_open`, used in chunks 34 and 60a.
Uses `device` 66a, `fat_filesystem_init` 44c, `fatinit` 43b, `ffd` 61b, and `printf` 42a.

Die Funktion `fat_file_exists` sucht auf dem Dateisystem nach der zu öffnenden Datei. Wurde die Datei gefunden, enthält `file_found` die absolute Position in Byte auf dem Dateisystem und `fatdir` die Werte aus dem Verzeichniseintrag. Wurde kein Verzeichniseintrag gefunden, gibt `fat_file_exists` den Wert `-1` zurück. Existiert keine Datei (`file_found == - 1`) und wurde die Funktion mit `O_CREAT` aufgerufen, wird die Datei mit `fat_creat_empty_file` neu erstellt. Nachdem erfolgreich eine neue Datei erstellt wurde, werden mit einem erneuten Aufruf von `fat_file_exists` die Informationen des gerade erstellten Verzeichniseintrages gelesen.

Wurde eine Datei gefunden, so wird gleich der Zugriff auf die Datei überprüft. Die einzigen Zugriffsrechte welche FAT kennt, werden in den Dateiattributen gespeichert, siehe Kapitel 4.6. Möchte der User auf eine *Ready-Only*-Datei schreibend zugreiffen, bricht die Funktion mit `-1` ab.

60c ⟨*fat open* 60c⟩≡ (60b) 61a▷

```
dirEntry fatdir;
unsigned long file_found;
int newfile;

file_found = fat_file_exists(device, path, &fatdir);
```

```
if (file_found == -1){
   if ((oflag & O_CREAT) != 0){
      newfile = fat_creat_empty_file(device, path);
      file_found = fat_file_exists(device, path, &fatdir);
   } else {
      return -1;
      printf("ERROR: Datei existiert nicht.\n");
   }
} else {
   if (fatdir.DIR_attr == ATTR_READ_ONLY &&
      (((oflag & O_RDWR) != 0)             ||
      ((oflag & O_APPEND) != 0)            ||
      ((oflag & O_WRONLY) != 0)            ||
      ((oflag & O_WRONLY | O_CREAT) != 0)  ||
      ((oflag & O_WRONLY | O_APPEND) != 0))){
         printf ("ERROR: Datei ist Read Only\n");
         return -1;
   }
}
```

Defines:
fatdir, used in chunks 62b, 63a, and 88.
file_found, used in chunk 63a.
newfile, never used.
Uses ATTR_READ_ONLY 49b, device 66a, dirEntry 49a, fat_creat_empty_file 76b, fat_file_exists 50c, O_APPEND 58a, O_CREAT 58a, O_RDWR 58a, O_WRONLY 58a, and printf 42a.

Unabhängig davon, ob im vorherigen Schritt eine Datei neu erstellt oder eine existierende Datei gefunden wurde, wird mit Hilfe der Funktion `fat_get_free_status_entry` ein neuer LFD `ffd` ermittelt. Konnte kein freier LFD ermittelt werden, so bricht die Funktion mit `-1` ab.

61a ⟨*fat open* 60c⟩+≡ (60b) ◁60c 61b▷

```
int ffd = fat_get_free_status_entry();

if(ffd == -1)
   return -1; //Keine freien LFDs
```

Uses fat_get_free_status_entry 59c and ffd 61b.

Nachdem ein freier LFD ermittelt wurde, wird die Liste der vorhandenen Inodes nach bereits geöffneten Inodes durchsucht. Der Pfad zur Datei dient hier zur eindeutigen Identifikation. Existiert ein Inode mit dem gleichen Dateipfad, so wird der neue LFD mit dem bereits existierenden Inode verküpft.

61b ⟨*fat open* 60c⟩+≡ (60b) ◁61a 62a▷

```
short file_already_open = 0;
int int_ino = -1;

for (int i = 0; i < MAX_FAT_INT_INODES; i++) {
   if (strcmp (fat_inodes[i].path,path) && fat_inodes[i].device == device){
      file_already_open = 1;
      int_ino = i;
   }
```

```
}
```

Defines:
ffd, used in chunks 60–66 and 69.
file_already_open, used in chunk 63a.
int_ino, used in chunks 31b, 62, and 63a.
Uses device 66a, fat_inodes, i, MAX_FAT_INT_INODES 57a, and strcmp 42a.

Wurde kein Inode gefunden, wird ein Neuer erstellt. Dazu wird die Funktion `fat_get_free_inode_entry` aufgerufen. Die Funktion liefert einen freien Eintrag aus `fat_inodes` und weist ihn `int_ino` zu. Konnte kein freier Inode-Eintrag gefunden werden, bricht die Funktion mit `-1` ab.

62a ⟨*fat open* 60c⟩+≡ (60b) ◁61b 62b▷

```
if (int_ino == -1)
   int_ino = fat_get_free_inode_entry();

if(int_ino == -1)
   return -1; //Keine freie Inode
```

Uses fat_get_free_inode_entry 59b and int_ino 61b.

Wurden ein freier LFD und ein freier Inode ermittelt und verknüpft, werden im letzten Schritt die Datenstrukturen gefüllt.

fat_status Die LFD-Struktur enthält die Referenz zur Inode. Weiter wird die Lese-/Schreibposition auf `0` gesetzt, `fat_status[ffd].mode` speichert den `oflag`-Parameter welcher beim Funktionsaufruf übergeben wurde und das erste Dateicluster wird in `fat_status[ffd].curcluster` gespeichert.

62b ⟨*fat open* 60c⟩+≡ (60b) ◁62a 63a▷

```
struct int_fat_inode *inode = &(fat_inodes[int_ino]);

fat_status[ffd].int_inode = inode;
fat_status[ffd].pos = 0;
fat_status[ffd].mode = oflag;
fat_status[ffd].curcluster = fatdir.DIR_FirstClusterLow;
```

Uses curcluster, fat_inodes, fat_status, fatdir 60c, ffd 61b, inode, int_fat_inode 56 64a 64d 66a 87c, and int_ino 61b.

inode Wurde ein existierender Inode erneut verknüpft, so wird `refcount` um `1` erhöht. Bei neuen Inodes wird der Wert von `refcount` auf `1` gesetzt. Weiter werden die Benutzer- und Gruppen-IDs fest mit `0` belegt. Unter ULIX gibt es die Benutzer- und Gruppen-ID `0`. Sie steht für den `root`-User und die `root`-Gruppe. `root` ist der System Administrator und kann sämtliche Zugriffe kontrollieren (vgl. Eßer & Freiling, 2014, S. 537). Wie in Kapitel 3.8 beschrieben, werden grundsätzlich bei FAT-Mounts, alle Verzeichnisse und Dateien eines gemounteten FAT-Datenträgers mit den Benutzer und Gruppen der Mountpoints belegt, welcher den Datenträger gemountet hat. Aus diesem Grund wurde sich entschieden, `i_uid` und `i_gid` fest mit dem Wert `0` (root) zu belegen. Informationen wie Uhrzeiten und Daten werden aus dem Verzeichniseintrag übernommen. Abhängig davon, ob es sich bei der geöffneten Datei um ein Verzeichnis oder um eine reguläre Datei handelt, wird `i_mode`

mit den Standard-ULIX-Zugriffsrechten 0644 belegt.

63a ⟨*fat open* 60c⟩+≡ (60b) ◁62b 63b▷

```
if(file_already_open){
   inode->refcount++;
}else{
   strncpy(inode->path,path,strlen(path));
   inode->ino = int_ino;
   inode->i_uid = 0;
   inode->i_gid = 0;
   inode->i_atime = fatdir.DIR_DateLastAccess;
   inode->i_ctime = fatdir.DIR_DateCreate;
   inode->i_mtime = fatdir.DIR_TimeLastMod;
   inode->i_size = fatdir.DIR_FileSize;
   inode->i_totalpos = file_found;
   inode->i_attr = fatdir.DIR_attr;
   if (fatdir.DIR_attr == ATTR_DIRECTORY)
      inode->i_mode = S_IFDIR | 0644;
   else
      inode->i_mode = S_IFREG | 0644;
   inode->device = device;
   inode->refcount = 1;
}
```

Uses `ATTR_DIRECTORY` 49b, `device` 66a, `fatdir` 60c, `file_already_open` 61b, `file_found` 60c, `inode`, `int_ino` 61b, `S_IFDIR`, `S_IFREG`, `strlen` 42a 42a, and `strncpy` 42a.

Wurde die Datei im `O_APPEND`-Modus geöffnet, wird die Lese-/Schreibposition ans Ende der Datei gesetzt.

63b ⟨*fat open* 60c⟩+≡ (60b) ◁63a

```
if ((oflag & O_APPEND) != 0){
   fat_status[ffd].pos = inode->i_size;
}
```

Uses `fat_status`, `ffd` 61b, `inode`, and `O_APPEND` 58a.

4.7.3. fat_close

Geöffnete Dateien werden im Array `fat_inodes` geführt, die entsprechenden LFD in `fat_status`. Die Funktion `fat_close` schließt bzw. entfernt einen LFD zu einer Datei (Inode).

63c ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁60a 64c▷

```
int fat_close (int fd);
```

Uses `fat_close` 64a.

Nach einer Validierung des übergebenen LFD und des dazu referenzierten Inodes wird der Zähler `refcount` um `1` reduziert, da ein LFD weniger den Inode referenziert. LFD und Inodes stehen in einer *n:1*-Beziehung zu einander. Anschließend wird die Bindung zwischen LFD und Inode aufgehoben.

Entscheidend ist, dass, wenn der letzte LFD vom Inode entfernt wird (`refcount == 0`), der Pfad zur Datei im Inode selbst gelöscht wird.

64a ⟨*fat function implementations* 44c⟩+≡ (89b) ◁60b 64d▷

```
int fat_close (int ffd){
   if (ffd < 0 || ffd >= FAT_MAX_FILES) return -1;

   struct fat_filestat *st = &fat_status[ffd];
   struct int_fat_inode *inode = st->int_inode;

   if (inode == 0)
      return -1;

   inode->refcount-;
   st->int_inode = NULL;

   if(inode->refcount == 0)
      memset(&inode->path[0], 0, sizeof(inode->path));
return 0;
}
```

Defines:
`fat_close`, used in chunk 63c.
`fat_filestat`, used in chunks 58c and 69b.
`int_fat_inode`, used in chunks 31b, 57, 62b, and 69b.
Uses `FAT_MAX_FILES` 58b, `fat_status`, `ffd` 61b, `inode`, `memset` 42a, and `NULL` 57c.

4.7.4. fat_lseek

Die Aufgabe der `fat_lseek`-Funktion ist die Neupositionierung der Lese-/Schreibposition. Dadurch wird ULIX es ermöglicht, wahlfrei auf das FAT-Dateisystem zuzugreifen. Die folgenden Konstanten definieren im Rahmen der Implementierung, wie die neue Position berechnet wird. Am Ende der Funktion, wird die neue Lese-/Schreibposition zurückgegeben.

64b ⟨*fat constants* 49b⟩+≡ (89a) ◁58b 65a▷

```
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2
```

Uses `SEEK_CUR`, `SEEK_END`, and `SEEK_SET`.

64c ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁63c 65b▷

```
int fat_lseek (int ffd, int offset, int whence);
```

Uses `fat_lseek` 64d, `ffd` 61b, and `offset` 68 75a.

Nachdem eine Datei mit `fat_open` geöffnet wurde, führt der LFD die aktuelle Lese-/Schreibposition. `fat_lseek` greift über `ffd` auf diesen Wert zu und berechnet die neue Position. Dabei kann die Position absolut bestimmt (`SEEK_SET`), relativ zur aktuellen Postion (`SEEK_CUR`) oder an das Dateiende angefügt werden (`SEEK_END`).

64d ⟨*fat function implementations* 44c⟩+≡ (89b) ◁64a 66a▷

```
int fat_lseek (int ffd, int offset, int whence){
   if (ffd < 0 || ffd >= FAT_MAX_FILES) return -1;
   struct fat_filestat *st = &fat_status[ffd];
```

```
    struct int_fat_inode *inode = st->int_inode;

    if (inode == 0) return -1;

    if (whence < 0 || whence > 2) return -1;
    if ((st->mode & O_APPEND) != 0) return st->pos;

    switch (whence) {
       case SEEK_SET: st->pos = offset; break;
       case SEEK_CUR: st->pos += offset; break;
       case SEEK_END: st->pos = inode->i_size + offset;
    };

    if (st->pos < 0) st->pos = 0;

    return st->pos;
  }
```

Defines:
  `fat_filestat`, used in chunks 58c and 69b.
  `fat_lseek`, used in chunk 64c.
  `int_fat_inode`, used in chunks 31b, 57, 62b, and 69b.
Uses `FAT_MAX_FILES` 58b, `fat_status`, `ffd` 61b, `inode`, `O_APPEND` 58a, `offset` 68 75a, `SEEK_CUR`, `SEEK_END`, and `SEEK_SET`.

4.8. Lesen und Schreiben von Dateien

Das Öffnen von Dateien und das Verwalten der LFDs sind Voraussetzungen für Lese- und Schreiboperationen. In diesem Kapitel werden die Implementierungen von `fat_read` und `fat_write` sowie ihre Hilfsfunktionen behandelt.

Die Implementierung verwendet die Makros `MIN` und `MAX` aus dem ULIX-Quellcode (vgl. Eßer & Freiling, 2014, S. 435).

65a ⟨*fat constants* 49b⟩+≡ (89a) ◁64b 82b▷

```
  #define MIN(a,b) ((a) <= (b) ? (a) : (b))
  #define MAX(a,b) ((a) >= (b) ? (a) : (b))
```

Uses `MAX` and `MIN`.

4.8.1. fat_read

Die Funktion `fat_read` liest `nbyte` Bytes aus einer Datei und schreibt diese in `buf`. Am Ende der Funktion, gibt sie die Anzahl der gelesenen Bytes `read_bytes` zurück.

65b ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁64c 69a▷

```
  int fat_read (int ffd, void *buf, int nbyte);
```

Uses `fat_read` 66a and `ffd` 61b.

Eine geöffnete Datei wird über den LFD `ffd` identifiziert. Zu Beginn der Funktion wird der übergebene LFD auf einen gültigen Wertebereich validiert. Ist der LFD ungültig, bricht die Funktion mit `-1` ab.

66a ⟨*fat function implementations* 44c⟩+≡ (89b) ◁64d 69b▷

```
int fat_read (int ffd, void *buf, int nbyte){
   if (ffd < 0 || ffd >= FAT_MAX_FILES) return -1;

   struct fat_filestat *st = &fat_status[ffd];
   struct int_fat_inode *inode = st->int_inode;

   int fileSize = inode->i_size;
   short device = inode->device;

   ⟨fat read 66b⟩

   return read_bytes;
}
```

Defines:
`device`, used in chunks 31, 34, 35, 37, 44, 48, 50, 52, 53, 56, 60, 61b, 63a, 67–69, 71–78, 80, 82–88, and 102–104.
`fat_filestat`, used in chunks 58c and 69b.
`fat_read`, used in chunk 65b.
`fileSize`, used in chunks 66b and 69–72.
`int_fat_inode`, used in chunks 31b, 57, 62b, and 69b.
Uses `FAT_MAX_FILES` 58b, `fat_status`, `ffd` 61b, `inode`, and `read_bytes` 67b.

Die Lese-/Schreibposition `pos` definiert, wo in der Datei mit dem Lesen begonnen wird. `pos` wird im LFD geführt und jedesmal, nachdem Bytes gelesen wurden, aktualisiert. Lesepositionen größer als die Dateigröße sind nicht zulässig. Dieser Fall wird geprüft und die Funktion wird mit `0` beendet. Liegt `pos + nbyte - 1` über der Dateigröße, dann wird `nbyte` mit dem Dateiende neu definiert, da nicht mehr Bytes gelesen werden können als die Datei groß ist. `startfileblock` definiert den Datenblock in welchem die Datei beginnt. Dieser wird ermittelt indem `startfilebyte` durch 512 (FAT Cluster Größe) dividiert wird, siehe Abbildung 4.1.

66b ⟨*fat read* 66b⟩≡ (66a) 67a▷

```
int startfilebyte = st->pos;

if (startfilebyte >= fileSize) {return 0;}

int endfilebyte = st->pos + nbyte - 1;
if (endfilebyte >= fileSize) {
   nbyte = nbyte - (endfilebyte - fileSize + 1);
   endfilebyte = fileSize - 1;
}

int startfileblock = startfilebyte / 512;
int endfileblock = endfilebyte / 512;
```

Defines:
`endfileblock`, used in chunk 67a.
`endfilebyte`, never used.
`startfileblock`, used in chunk 67a.
`startfilebyte`, used in chunks 68 and 75a.
Uses `fileSize` 66a.

Zusätzlich zur aktuellen Lese-/Schreibposition befindet sich in der LFD-Struktur das

aktuelle Cluster `curcluster` der geöffneten Datei. Bei neu geöffneten Dateien ist das immer das erste Datei-Cluster aus dem Verzeichniseintrag.

In zwei Schleifen werden `startfilecluster` und `endfilecluster` ermittelt. Dabei handelt es sich um die Datei-Cluster mit der aktuellen Lese-/Schreibposition bzw. Dateiende. Angenommen, es wurde zuvor ermittelt das sich die aktuelle Lese-/Schreibposition im Datei-Cluster `3` befindet. Die Schleife wird drei Mal durchlaufen und mit Hilfe der Funktion `read_fat_entry`, das dritte Datei-Cluster ermittelt.

67a ⟨*fat read* 66b⟩+≡ (66a) ◁66b 67b▷

```
unsigned short curCluster = st->curcluster;
unsigned short nextclst;
unsigned short startfilecluster;

for (int i=0; i <= startfileblock; i++){
   nextclst = read_fat_entry (device, curCluster);
   startfilecluster = curCluster;
   curCluster = nextclst;
}

curCluster = st->curcluster;

unsigned short endfilecluster;

for (int i=0; i <= endfileblock; i++){
   nextclst = read_fat_entry (device, curCluster);
   endfilecluster = curCluster;
   curCluster = nextclst;
}
```

Defines:
`curCluster`, used in chunks 67b, 68, and 75.
`endfilecluster`, never used.
`nextclst`, never used.
`startfilecluster`, used in chunks 67b, 68, and 75a.
Uses `curcluster`, `device` 66a, `endfileblock` 66b 74a, `i`, `read_fat_entry` 48b, and `startfileblock` 66b 74a.

In einer Schleife werden solange Bytes aus der Datei gelesen und nach `buf` kopiert, bis keine Bytes mehr gelesen werden müssen (`nbyte < 0`). Der Code-Chunk ⟨*read loop* 68⟩ beschreibt im Detail wie die Daten gelesen und die Werte berechnet werden.

67b ⟨*fat read* 66b⟩+≡ (66a) ◁67a

```
curCluster = startfilecluster;

int read_bytes = 0;
char temp_clst[512];
int blockoffset;

while(nbyte > 0){
  ⟨read loop 68⟩

}
```

Defines:

`blockoffset`, used in chunks 53, 68, 72a, and 85c.
`read_bytes`, used in chunks 66a and 68.
`temp_clst`, used in chunk 68.
Uses `curCluster` 67a 74a and `startfilecluster` 67a 74a.

Begonnen wird im ersten Schleifendurchlauf mit dem ermittelten `startfilecluster`. Nachdem der richtige `readblock`-Offset berechnet wurde, wird ein kompletter Block gelesen und das richtige Cluster nach `temp_clst` kopiert.

Abhängig davon, ob im aktuellen Schleifendurchlauf das `startfilecluster` gelesen wird, müssen der Cluster-Offset und die zu lesende Anzahl an Bytes auf unterschiedliche Art und Weise berechnet werden. Handelt es sich um das Start-Cluster, so wird lediglich die Byte-Differenz zwischen Offset und Blockgröße gelesen. Bei allen anderen Clustern wird entweder der komplette Cluster oder die restlichen `nbyte` gelesen.

Nachdem die Daten gelesen und nach `buf` kopiert wurden, werden die Zähler aktualisiert. Dabei werden `nbyte` um die gelesene Anzahl an Bytes reduziert und `read_bytes`, `buf` und `pos` um die gelesene Anzahl erhöht. Die letzte Schleifenanweisung setzt `curCluster` auf das nächste Cluster, das im nächsten Schleifendurchlauf gelesen werden soll.

68 ⟨*read loop* 68⟩≡ (67b)

```
if (curCluster % 2 > 0){
   blockoffset = 0;
} else {
   blockoffset = 512;
}

readblock (device,(int)((31+(int)curCluster)/2), (char *)fbuf);
memcpy (temp_clst, &fbuf[blockoffset], 512 * sizeof(char));

int offset, length;
if(curCluster == startfilecluster){
   offset = startfilebyte % 512;
   length = MIN(nbyte, 512 - offset);
} else{
   offset = 0;
   length = MIN(nbyte, 512);
}
memcpy (buf, temp_clst+offset, length);

nbyte -= length;
buf += length;
read_bytes += length;
st->pos += length;
curCluster = read_fat_entry (device, curCluster);
```

Defines:
`length`, used in chunks 72, 73, and 75b.
`offset`, used in chunks 64 and 82.
Uses `blockoffset` 67b 71b, `curCluster` 67a 74a, `device` 66a, `fbuf` 43b, `memcpy` 42a, `MIN`, `read_bytes` 67b, `read_fat_entry` 48b, `startfilebyte` 66b 74a, `startfilecluster` 67a 74a, and `temp_clst` 67b.

4.8.2. fat_write

Die Aufgabe von `fat_write` ist es, Daten aus einem Buffer korrekt in das Dateisystem zu schreiben und die Anzahl der geschriebenen Bytes `written_bytes` zurückzugeben. Dabei können Fälle auftreten, bei denen zuerst „Fülldaten“ geschrieben werden müssen, bevor die eigentlichen Daten geschrieben werden. Die Funktion ist in zwei groben logischen Blöcken implementiert.

Block 1: Schreibt, wenn notwendig, „Fülldaten“

Block 2: Schreibt die Daten aus dem Buffer

69a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁65b 76a▷
```
int fat_write(int ffd, void *buf, int nbyte);
```
Uses `fat_write` 69b and `ffd` 61b.

Zu Beginn der Funktion finden Validierungen rund um den übergebenen Dateideskriptor `ffd` statt. Dabei wird geprüft, ob es sich um einen gültigen Wert handelt und eine Inode-Struktur referenziert ist. Des Weiteren wird geprüft, ob die Datei im Read-only-Modus geöffnet wurde. Schlagen alle Validierungen fehl bricht die Funktion mit `-1` ab. In beiden Blöcken wird der Verzeichniseintrag aktualisiert. Aus diesem Grund werden die entsprechenden Variablen oben in der Funktion deklariert.

69b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁66a 76b▷
```
int fat_write(int ffd, void *buf, int nbyte){
   if (ffd < 0 || ffd >= FAT_MAX_FILES){
      printf ("ERROR: Ungültiger File Deskriptor\n");
      return -1;
   }

   struct fat_filestat *st = &fat_status[ffd];
   struct int_fat_inode *inode = st->int_inode;
   int fileSize = inode->i_size;
   short device = inode->device;

   if (st->int_inode == NULL){
      printf ("ERROR: Datei nicht geöffnet\n");
      return -1;
   }
   if (st->mode == O_RDONLY){
      printf ("ERROR: Datei ist Read only geöffnet\n");
      return -1;
   }

   ⟨fat write 70⟩

   return written_bytes;
}
```
Defines:
`dirpos`, used in chunk 88a.
`fat_write`, used in chunk 69a.
Uses `device` 66a, `fat_filestat` 57d 64a 64d 66a 87c, `FAT_MAX_FILES` 58b, `fat_status`, `ffd` 61b, `fileSize` 66a, `inode`, `int_fat_inode` 56 64a 64d 66a 87c, `NULL` 57c, `O_RDONLY` 58a, `printf` 42a,

and written_bytes 74b.

Block 1:

Grundsätzlich funktioniert das wahlfreie Schreiben von Blöcken wie das Lesen. Jedoch exisitiert beim Schreiben der Sonderfall, dass die Schreibposition mittels fat_lseek weit über das Dateiende hinaus gesetzt werden kann, während beim Lesen von Dateien, nicht über das Dateiende hinaus gelesen werden kann. Das bedeutet, der Bereich zwischen Dateiende und Schreibposition muss mit 0-Werten gefüllt werden. Abbildung 4.4 veranschaulicht diesen Fall:

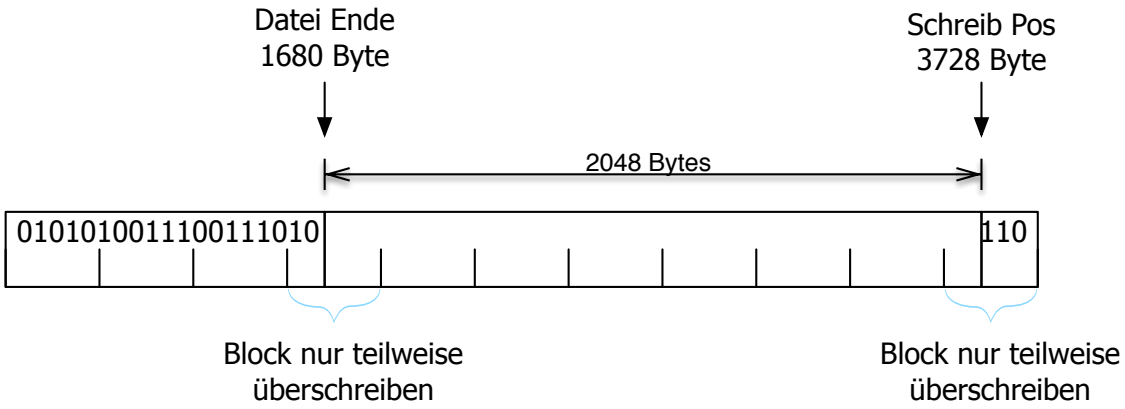


Abbildung 4.4.: Die Schreibposition liegt nach dem Dateiende. Der Bereich von 2.048 Bytes muss mit 0-Werten gefüllt werden.

Die Datei TEST besitzt eine Dateigröße von 1.680 Bytes. Die Schreibposition wurde 2.048 Bytes über das Dateiende hinaus gesetzt und hat damit die Position 3.728. An dieser Stelle sollen nun nbyte Bytes geschrieben werden. Der Bereich zwischen Schreibposition und Dateiende wird mit 0-Werten gefüllt. Dabei ist zu beachten, dass sowohl das Dateiende als auch die neue Schreibposition mitten in einem FAT-Cluster enden bzw. beginnen kann. Hier müssen die jeweiligen Offsets bestimmt werden.

Folgender Code-Chunck ermittelt zuerst, ob der oben beschriebene Fall vorliegt. Dabei muss berücksichtigt werden, dass auch in leere bzw. neu erstellte Dateien geschrieben werden kann. Bei diesen Dateien liegen die Dateigröße und die Lese-/Schreibposition bei 0. Die Variable gapsize bestimmt die Größe des zu füllenden Bereiches in Byte.

70 ⟨*fat write* 70⟩≡ (69b) 74a ▷

```
int gapsize = 0;

if (fileSize <= st->pos && fileSize > 0){
  ⟨fat write block 1 71a⟩
}
```

Defines:
gapsize, used in chunks 71 and 72.
Uses fileSize 66a.

Anschließend werden die Start- und Endpositionen ermittelt. Hierbei werden zuerst die

Byte-Positionen in der Datei berechnet und darauf aufbauend die FAT-Blöcke der Datei bestimmt. Um das letzte Cluster einer Datei zu ermitteln, wird die gesamte Cluster-Kette bis Ende (`0xFFF`) durchlaufen. Am Ende enthält `gapcurCluster` die CLN des letzten Daten-Clusters.

71a ⟨*fat write block 1* 71a⟩≡ (70) 71b▷

```
int gapstartfilebyte = fileSize;
int gapendfilebyte = st->pos + nbyte - 1;

int gapstartfileblock = gapstartfilebyte / 512;
int gapendfileblock = gapendfilebyte / 512;

unsigned short gapcurCluster = st->curcluster;
unsigned short gapnextclst;
unsigned short gapstartfilecluster;

while(gapnextclst != 0xFFF){
   gapnextclst = read_fat_entry (device, gapcurCluster);
   gapstartfilecluster = gapcurCluster;
   gapcurCluster = gapnextclst;
}
gapcurCluster = gapstartfilecluster;
```

Defines:
`gapcurCluster`, used in chunk 72.
`gapendfileblock`, never used.
`gapendfilebyte`, never used.
`gapnextclst`, used in chunk 72b.
`gapstartfileblock`, never used.
`gapstartfilebyte`, never used.
`gapstartfilecluster`, used in chunk 72a.
Uses `curcluster`, `device` 66a, `fileSize` 66a, and `read_fat_entry` 48b.

Die folgende Schleife schreibt solange `0`-Werte, bis die Lücke zwischen Dateiende und Schreibposition gefüllt ist. Bei jedem Schleifendurchlauf wird der Verzeichniseintrag aktualisiert.

71b ⟨*fat write block 1* 71a⟩+≡ (70) ◁71a

```
gapsize = st->pos - fileSize;

int gapoffset, length;
int blockoffset;

while(gapsize > 0){
   ⟨fill gap 72a⟩
   ⟨update gapvalues 72b⟩
   ⟨update direntry 73⟩
}
```

Defines:
`blockoffset`, used in chunks 53, 68, 72a, and 85c.
`gapoffset`, used in chunk 72a.
`length`, used in chunks 72, 73, and 75b.
Uses `fileSize` 66a and `gapsize` 70.

Beim Schreiben der `0`-Werte werden bis auf den Start/End-Cluster immer volle FAT-

Cluster (512 Byte) geschrieben. Der Offset `blockoffset` definiert den Offset innerhalb des 1024-Byte-Buffers `fbuf`.

72a ⟨*fill gap* 72a⟩≡ (71b)

```
if (gapcurCluster == gapstartfilecluster) {
   gapoffset = (fileSize % 512);
   length = 512 - gapoffset;
}else{
   length = MIN (512 , gapsize);
   gapoffset = 0;
}

if (gapcurCluster % 2 > 0){
   blockoffset = 0;
} else {
   blockoffset = 512;
}

int gapblock = (31+(int)gapcurCluster)/2;
readblock (device,gapblock,(char *)fbuf);

gapoffset += blockoffset;

for(int w = 0; w < length; w++){
   fbuf[gapoffset+w] = '0';
}

writeblock (device, gapblock, (char*)fbuf);
```

Defines:
`gapblock`, never used.
Uses `blockoffset` 67b 71b, `device` 66a, `fbuf` 43b, `fileSize` 66a, `gapcurCluster` 71a, `gapoffset` 71b, `gapsize` 70, `gapstartfilecluster` 71a, `length` 68 71b 75a, and `MIN`.

Nachdem das erste Cluster aufgefüllt bzw. ein kompletter Cluster geschrieben wurde, wird mit Hilfe der Funktionen `fat_get_free_fat_entry` und `fat_write_fat_entry` die Cluster-Kette der neu allokierten CLN zusammengesetzt. Die letzte CLN wird mit dem Wert `0xFFF` terminiert.

Am Schluss eines Schleifendurchlaufes werden `gapcurCluster` mit der nächsten CLN belegt, die geschriebene Anzahl von Bytes von `gapsize` abgezogen und die Dateigröße im Verzeichniseintrag und der Inode aktualisiert.

72b ⟨*update gapvalues* 72b⟩≡ (71b)

```
gapnextclst = fat_get_free_fat_entry(device,gapcurCluster);
fat_write_fat_entry(device, gapcurCluster, gapnextclst);
fat_write_fat_entry(device, gapnextclst, 0xFFF);

gapcurCluster = gapnextclst;
gapsize -= length;
```

Uses `device` 66a, `fat_get_free_fat_entry` 82c, `fat_write_fat_entry`, `gapcurCluster` 71a, `gapnextclst` 71a, `gapsize` 70, and `length` 68 71b 75a.

Am Schluss eines Schleifendurchlaufes werden `gapcurCluster` mit der nächste CLN

belegt, die geschriebene Anzahl von Bytes von `gapsize` abgezogen und die Dateigröße im Verzeichniseintrag und der Inode aktualisiert.

73 ⟨*update direntry* 73⟩≡ (71b 74b)

```
unsigned long dirpos;
dirEntry dir, *wdentry;

dirpos = fat_file_exists(device, inode->path, &dir);

dir.DIR_FileSize += length;
inode->i_size += length;

int reblock = dirpos / 1024;
int blockoff = (dirpos)%1024;

readblock (device,reblock, (char *)fbuf);
wdentry = (dirEntry*) &fbuf[blockoff];
*wdentry = dir;
writeblock (device, reblock, (char*)&fbuf);
```

Defines:
`blockoff`, used in chunks 35, 48c, 83c, and 86–88.
`dirpos`, used in chunk 88a.
`reblock`, used in chunks 35b, 48b, 83, 84a, 86c, and 88a.
`wdentry`, used in chunks 35 and 86–88.
Uses `device` 66a, `dir` 77, `dirEntry` 49a, `fat_file_exists` 50c, `fbuf` 43b, `inode`, and `length` 68 71b 75a.

Block 2:

Unabhängig davon, ob eine Lücke gefüllt werden musste oder nicht, schreibt `fat_write` jetzt `nbyte` Bytes aus `buf` an die Lese-/Schreibposition. Abbildung 4.5 veranschaulicht eine typische Situation des wahlfreien Schreibens.

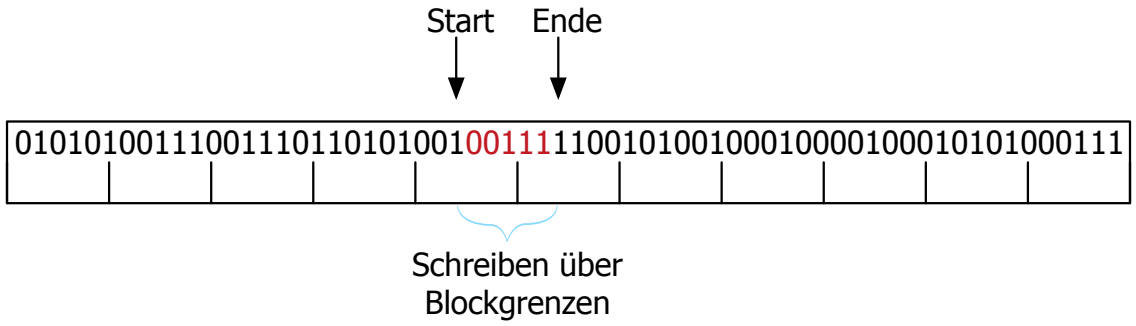


Abbildung 4.5.: Die Bytes zwischen Start und Ende in rot, werden über Block- bzw. Clustergrenzen hinweg geschrieben.

An einer beliebigen Stelle in einer Datei sollen `nbyte` Bytes geschrieben werden. Das Vorgehen ist fast identisch wie beim Füllen der Lücke zwischen Dateiende und Schreibposition. Zuerst werden die Start- und Endpositionen sowie ihre Cluster ermittelt. Im Unterschied zu Block 1 wird an dieser Stelle nicht das letzte Cluster ermittelt sondern das Cluster mit der aktuellen Lese-/Schreibposition `startfileblock`.

74a ⟨*fat write* 70⟩+≡ (69b) ◁70 74b▷

```
int startfilebyte = st->pos;
int endfilebyte = st->pos + nbyte - 1;

int startfileblock = startfilebyte / 512;
int endfileblock = endfilebyte / 512;

unsigned short curCluster = st->curcluster;
unsigned short nextclst;
unsigned short startfilecluster;

for (int i=0; i <= startfileblock; i++){
   nextclst = read_fat_entry (device, curCluster);
   startfilecluster = curCluster;
   curCluster = nextclst;
}

curCluster = startfilecluster;
```

Defines:
`curCluster`, used in chunks 67b, 68, and 75.
`endfileblock`, used in chunk 67a.
`endfilebyte`, never used.
`nextclst`, never used.
`startfileblock`, used in chunk 67a.
`startfilebyte`, used in chunks 68 and 75a.
`startfilecluster`, used in chunks 67b, 68, and 75a.
Uses `curcluster`, `device` 66a, `i`, and `read_fat_entry` 48b.

Wie auch beim Füllen der Lücke werden die Schreiboperationen innerhalb einer Schleife ausgeführt. Dabei wird die Schleife solange durchlaufen, bis alle `nbyte` Bytes geschrieben wurden.

Berücksichtigt werden muss ebenfalls, ob die geschriebenen Bytes die Dateigröße verändern oder lediglich Veränderungen innerhalb einer Datei stattfinden. Im Code-Chunk ⟨*write nbyte* 75a⟩ wird ermittelt ob der Verzeichniseintrag aktualisiert werden muss oder nicht. Der Marker `updatefilesize` definiert das Verzeichnis-Update.

74b ⟨*fat write* 70⟩+≡ (69b) ◁74a

```
int written_bytes = 0;
int updatefilesize = 0;

while(nbyte > 0){
   ⟨write nbyte 75a⟩
   ⟨update values 75b⟩
   if(updatefilesize == 1){
      ⟨update direntry 73⟩
   }
}
```

Defines:
`updatefilesize`, used in chunk 75a.
`written_bytes`, used in chunks 69b and 75a.

Besondere Fälle sind auch hier das Start- und das End-Cluster, welche nur teilweise beschrieben werden dürfen. Alle anderen Blöcke werden komplett geschrieben. Nachdem die

Blöcke beschrieben wurden, wird `nbyte` und die Anzahl der geschriebenen Byte reduziert und `buf` sowie `written_bytes` um die Anzahl geschriebener Bytes erhöht. Die Variable `written_bytes` enthält die Anzahl tatsächlich geschriebener Bytes und wird am Ende der Funktion zurückgegeben.

75a ⟨*write nbyte* 75a⟩≡ (74b)

```
int offset, length;

if(st->pos + nbyte > inode->i_size)
   updatefilesize = 1;

if (curCluster == startfilecluster) {
   offset = startfilebyte % 512;
   length = MIN (nbyte, 512 - offset);
}else{
   offset = 0;
   length = MIN (nbyte, 512);
}

int block = (31+(int)curCluster)/2;
readblock (device,block,(char *)fbuf);

if (curCluster % 2 > 0){
   memcpy (fbuf+offset, buf, length);
} else {
   memcpy (fbuf+offset+512, buf, length);
}
writeblock (device, block, (char*)fbuf);

nbyte -= length;
buf += length;
written_bytes += length;
```

Defines:
`length`, used in chunks 72, 73, and 75b.
`offset`, used in chunks 64 and 82.
Uses `block`, `curCluster` 67a 74a, `device` 66a, `fbuf` 43b, `inode`, `memcpy` 42a, `MIN`, `startfilebyte` 66b 74a, `startfilecluster` 67a 74a, `updatefilesize` 74b, and `written_bytes` 74b.

Ist der aktuelle Cluster der Letzte, so müssen ein neuer Cluster allokiert und die Cluster-Kette weitergeführt werden. Handelt es sich nicht um das letzte Cluster wird der nächste Cluster ermittelt, in welchen beim nächsten Schleifendurchlauf geschrieben wird. Des Weiteren wird die Lese-/Schreibposition aktualisiert und das aktuelle Cluster gesetzt.

75b ⟨*update values* 75b⟩≡ (74b)

```
unsigned short nclst;

if (nbyte > 0){
   if(read_fat_entry(device,(int)curCluster) == 0xFFF){
      nclst = fat_get_free_fat_entry(device,curCluster);
      fat_write_fat_entry(device, curCluster, nclst);
      fat_write_fat_entry(device, nclst, 0xFFF);
   } else {
      nclst = read_fat_entry(device,(int)curCluster);
```

```
    }
  }

  st->pos = st->pos + length;
  if (st->pos > inode->i_size) inode->i_size = st->pos;
  st->curcluster = nclst;

  curCluster = nclst;
```

Defines:
nclst, never used.
Uses curCluster 67a 74a, curcluster, device 66a, fat_get_free_fat_entry 82c, fat_write_fat_entry, inode, length 68 71b 75a, and read_fat_entry 48b.

4.9. Eine leere Datei erstellen

Dieses Kapitel beschreibt das Erstellen einer neuen Datei. `fat_creat_empty_file` wird unter anderem durch die Funktion `fat_open` angestoßen, wenn die zu öffnende Datei nicht vorhanden ist und das `O_CREAT`-Flag gesetzt wurde. Beim Erstellen einer neuen Datei sind sowohl Operationen in der FAT-Tabelle als auch bei den Verzeichniseinträgen notwendig. Auf alle relevanten Funktionen wird in diesem Kapitel eingegangen.

76a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁69a 81b▷

```
int fat_creat_empty_file(int device, const char *path);
```

Uses device 66a and fat_creat_empty_file 76b.

Wurde die Datei erfolgreich angelegt, so gibt die Funktion `0` zurück, ansonsten `-1`.

Zuerst wird der Verzeichnisname aus dem übergebenen Pfad analysiert. Enthält dieser verbotene Zeichen, bricht die Funktion mit `-1` ab. Laut FAT-Spezifikation sind alle Zeichen unter `0x20` und folgende Zeichen nicht in Dateinamen zulässig (vgl. Microsoft, 2000, S. 24).

```
", *, +, ,, ., /, :, ;, <, >, =, ?, [, ], \, |
```

Die Zeichen werden Hex-kodiert im Array `forbiddenchar` geführt. Das `.` Symbol wird bei der Überprüfung ausgeschlossen, da es ein zulässiges Zeichen im Dateinamen ist, wie zum Beispiel `TEST.TXT`. Bei der späteren Konvertierung in das 8.3-Format wird der Punkt berücksichtigt. Wurde eines der Sonderzeichen gefunden, gibt die Funktion `-1` zurück.

Zur Trennung von Dateinamen (`filename`) und dem Verzeichnis-Pfad (`dirname`) wird die ULIX-Funktion `splitpath` verwendet (vgl. Eßer & Freiling, 2014, S. 419).

76b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁69b 81e▷

```
int fat_creat_empty_file(int device, const char *path){
   int i = 0,j;
   char forbiddenchar[36] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09,
                             0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13,
                             0x14,0x22,0x2A,0x2B,0x2C,0x2F,0x3A,0x3B,0x3C,0x3D,
                             0x3E,0x3F,0x5B,0x5C,0x5D,0x7C};

   char dirname[256] = { 0 }; char filename[30] = { 0 };
```

```
    splitpath (path, dirname, filename);

    while (filename[i]){
       for (j = 0; j <= 16; j++){
          if (filename[i] == forbiddenchar[j]){
             printf ("ERROR: Nicht erlaubte Zeichen!\n");
             return -1;
          }
       }
       i++;
    }
    ⟨define parent dir 77⟩
    ⟨write file 80⟩
    return 0;
 }
```

Defines:
`dirname`, used in chunks 42a and 77.
`fat_creat_empty_file`, used in chunks 60c and 76a.
`filename`, used in chunks 54, 55, and 80.
`forbiddenchar`, never used.
Uses `device` 66a, `i`, `printf` 42a, and `splitpath` 42a.

Im nächsten Schritt wird das Verzeichnis ermittelt, in welches die Datei geschrieben werden soll. Enthält `dirname` nur das /-Symbol, so wird die Datei im *Root-Verzeichnis* erstellt. Ansonsten wird das Unterverzeichnis ermittelt. Für Unterverzeichnisse ist die Variable `clusterno` relevant, sie enthält das Daten-Cluster in welches der Verzeichniseintrag geschrieben werden soll.

77 ⟨*define parent dir* 77⟩≡ (76b)

```
dirEntry dir;
int writetoroot = 0;

if (strlen(dirname) == 1 && dirname[0] == '/'){
   writetoroot = 1;
} else {
   if(fat_file_exists(device, dirname, &dir) == -1){
      printf ("ERROR: Verzeichnis existiert nicht!\n");
      return -1;
   }
}

int freedirentry;
unsigned short freefatentry;
int clusterno;

if(writetoroot){
   ⟨writeinroot 78a⟩
}else{
   ⟨writeinsubdir 78b⟩
}
```

Defines:
`dir`, used in chunks 31, 73, 78, 86, and 87a.
`writetoroot`, used in chunk 80.

Uses clusterno, device 66a, dirEntry 49a, dirname 76b, fat_file_exists 50c, freedirentry 78b, freefatentry 78b, printf 42a, and strlen 42a 42a.

Code-Chunk ⟨*writeinroot* 78a⟩ beschreibt das Vorgehen, wenn ein Verzeichniseintrag in das *Root-Verzeichnis* geschrieben wird. Mit der `fat_get_free_dir_entry`-Funktion wird die Nummer eines freien Verzeichniseintrages bestimmt. Wurde kein freier Eintrag gefunden, bricht die Funktion ab. Das *Root-Verzeichnis* ist dann voll.

78a ⟨*writeinroot* 78a⟩≡ (77)

```
freedirentry = fat_get_free_dir_entry(device, 0, ROOTSEARCH);
if(freedirentry == -1){
   printf ("ERROR: Das Root-Verzeichnis ist voll.\n");
   return -1;
} else {
   clusterno = dir.DIR_FirstClusterLow;
}
```

Uses clusterno, device 66a, dir 77, fat_get_free_dir_entry, freedirentry 78b, printf 42a, and ROOTSEARCH 84c.

Anders sieht es bei Unterverzeichnissen aus. Wie in Kapitel 2.4.4 beschrieben, können Unterverzeichnisse beliebig viele Daten-Cluster belegen. Das FAT12-Dateisystem in dieser Arbeit verwendet eine Clustergröße von 512 Byte, das entspricht 16 Verzeichniseinträgen. Soll das Verzeichnis weitere Einträge führen, muss ein neuer Daten-Cluster allokiert und die Cluster-Kette erweitert werden. Der Code-Chunk ⟨*writeinsubdir* 78b⟩ beschreibt dieses Vorgehen.

In der ersten `while`-Schleife wird die Cluster-Kette von Anfang bis Ende durchlaufen und nach einem freien Verzeichniseintrag `freedirentry` gesucht.

Wurde kein freier Eintrag gefunden, muss ein neues Daten-Cluster allokiert werden und das Unterverzeichnis erweitert werden. Mit Hilfe der Funktion `fat_get_free_dir_entry` wird ein freier FAT-Eintrag ermittelt.

In der zweiten Schleife wird die Cluster-Kette noch einmal bis zum Ende durchlaufen, um den letzten Eintrag, `lastclst` zu erhalten. Dieser wird mit `fat_write_fat_entry` aktualisiert und enhählt dann die Nummer des gerade allokierten Daten-Cluster `freefatentry`. Dieser Eintrag wiederum, wird mit `0xFFF` terminiert.

78b ⟨*writeinsubdir* 78b⟩≡ (77)

```
unsigned short nextclstno;
unsigned short curclstno = dir.DIR_FirstClusterLow;

while(nextclstno != 0xFFF){
   freedirentry = fat_get_free_dir_entry(device, curclstno,
                                         CLUSTERSEARCH);
   if (freedirentry == -1){
      nextclstno = read_fat_entry (device, curclstno);
      curclstno = nextclstno;
   } else {
     clusterno = curclstno;
     break;
   }
```

```
 }

if (freedirentry == -1){
   freefatentry = fat_get_free_fat_entry(device, BEGINFAT);

   if (freefatentry == -1){
      printf ("ERROR: Der Datenträger ist voll\n");
      return -1;
   }

   nextclstno = 0;
   unsigned short lastclst = 0;
   curclstno = dir.DIR_FirstClusterLow;

   while(nextclstno != 0xFFF){
      nextclstno = read_fat_entry (device, curclstno);
      lastclst = curclstno;
      curclstno = nextclstno;
   }

   fat_write_fat_entry(device, lastclst, freefatentry);
   fat_write_fat_entry(device, freefatentry, 0xFFF);

   clusterno = freefatentry;
   freedirentry = fat_get_free_dir_entry(device, nextclstno,
                                         CLUSTERSEARCH);
}
```

Defines:
freedirentry, used in chunks 77, 78a, and 80.
freefatentry, used in chunks 77 and 80.
Uses BEGINFAT 82b, clusterno, CLUSTERSEARCH 84c, curclstno, device 66a, dir 77, fat_get_free_dir_entry, fat_get_free_fat_entry 82c, fat_write_fat_entry, nextclstno, printf 42a, and read_fat_entry 48b.

Jetzt können die Daten auf den Datenträger geschrieben werden. Bevor jedoch der Verzeichniseintrag geschrieben wird, muss noch einmal ein freies Daten-Cluster allokiert werden, da der Verzeichniseintrag selbst (Datei oder Verzeichnis) ebenfalls ein Cluster belegt. Die Funktion `fat_write_fat_entry` schreibt den Eintrag in beide FAT-Tabellen. Da eine leere Datei angelegt wird, lautet der Wert des Eintrages `0xFFF`.

Die zweite Datenstruktur, welche geschrieben werden muss, ist der Verzeichniseintrag selbst. Dazu wird eine Datenstruktur `wdir` eingeführt. Diese Struktur definiert den zu schreibenden Verzeichniseintrag. Bevor der Verzeichniseintrag geschrieben werden kann, müssen die Informationen in das richtige Format gebracht werden. Für die 8.3-Formatierung des Verzeichnisnamens wird die Funktion `convertto_shortname` aufgerufen. Datum und Uhrzeiten werden in der aktuellen Implementierung nicht berücksichtigt. Als Default-Werte werden hier `0`-Werte eingetragen. Der erste Daten-Cluster der verketteten Liste ist der zuvor ermittelte freie FAT-Eintrag. Die Dateigröße ist `0` Byte, und als Attribut wird gemäß FAT-Spezifikation `ATTR_ARCHIVE` gewählt (vgl. Microsoft, 2000, S. 24), siehe Kapitel 2.4.3 zur Definition der Dateiattribute.

80 ⟨*write file* 80⟩≡ (76b)

```
char shortname[12];
dirEntry wdir;
unsigned short fattime = 0;
unsigned short fatdate = 0;

fat_get_time(&fattime);
fat_get_date(&fatdate);

freefatentry = fat_get_free_fat_entry(device, BEGINFAT);

if (freefatentry == -1){
   printf ("ERROR: Der Datenträger ist voll\n");
   return -1;
}

fat_write_fat_entry(device,freefatentry,0xFFF);

convertto_shortname(filename, shortname);
memcpy(wdir.DIR_shortName,shortname, 11);

wdir.DIR_attr = ATTR_ARCHIVE;
wdir.DIR_reserv = 0;
wdir.DIR_cnto = 0;
wdir.DIR_TimeCreate = fattime;
wdir.DIR_DateCreate = fatdate;
wdir.DIR_DateLastAccess = fatdate;
wdir.DIR_FirstClusterHi = 0;
wdir.DIR_TimeLastMod = fattime;
wdir.DIR_DateMod = fatdate;
wdir.DIR_FirstClusterLow = freefatentry;
wdir.DIR_FileSize = 0;

if (fat_write_dir_entry(device, wdir, clusterno,
    freedirentry, writetoroot) == -1){
   printf ("ERROR: Die Datei konnte nicht geschrieben werden!\n");
   return -1;
}
```

Defines:
`fatdate`, never used.
`fattime`, never used.
`shortname`, used in chunk 54.
`wdir`, never used.

Uses `ATTR_ARCHIVE` 49b, `BEGINFAT` 82b, `clusterno`, `convertto_shortname` 54b, `device` 66a, `dirEntry` 49a, `fat_get_date`, `fat_get_free_fat_entry` 82c, `fat_get_time`, `fat_write_dir_entry`, `fat_write_fat_entry`, `filename` 76b, `freedirentry` 78b, `freefatentry` 78b, `memcpy` 42a, `printf` 42a, and `writetoroot` 77.

4.9.1. fat_get_date und fat_get_time

Die beiden Funktionen `fat_get_date` und `fat_get_time` konvertieren einen UNIX-Timestamp in das FAT-Zeit-Format. Den UNIX-Timestamp führt ULIX in der globalen Variable `system_time`, welche über eine Deklaration in Modul-Header (`module.h`) zugänglich ge-

macht wird.

81a ⟨*fat global variables* 43b⟩+≡ (89a) ◁58c
```
extern unsigned int system_time;
```

81b ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁76a 81c▷
```
void fat_get_date(unsigned short *fat_date);
void fat_get_time(unsigned short *fat_time);
```
Uses `fat_get_date` and `fat_get_time`.

Sowohl `fat_get_date` als auch `fat_get_time` verarbeiten den UNIX-Timestamp mit der von ULIX zur Verfügung gestellten Funktion `rev_unixtime()` (vgl. Eßer & Freiling, 2014, S. 325).

81c ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁81b 82a▷
```
extern void rev_unixtime (unsigned int unixtime, short *year, char *month,
                          char *day, char *hour, char *minute, char *second);
```

81d ⟨*get dates* 81d⟩≡ (81)
```
unsigned char second, minute, hour, day, month;
unsigned short year;

rev_unixtime (system_time, &year, &month, &day, &hour, &minute, &second);
```

Wie Daten und Uhrzeiten in FAT-Dateisystemen aufgebaut sind, wurde in Kapitel 2.4.4 beschrieben. Bei der Jahreszahl ist darauf zu achten das es sich um einen Offset von 1980 handelt.

81e ⟨*fat function implementations* 44c⟩+≡ (89b) ◁76b 81f▷
```
void fat_get_date(unsigned short *fat_date){
    ⟨get dates 81d⟩
    *fat_date = (year - 1980) << 9 | month << 5 | day;
}
```
Uses `fat_get_date`.

Die Genauigkeit der Sekunden in FAT-Dateisystemen beträgt 2 Sekunden.

81f ⟨*fat function implementations* 44c⟩+≡ (89b) ◁81e 82c▷
```
void fat_get_time(unsigned short *fat_time){
    ⟨get dates 81d⟩
    *fat_time = hour << 11 | minute << 6 | second;
}
```
Uses `fat_get_time`.

4.9.2. fat_get_free_fat_entry

Diese Funktion ermittelt in der FAT einen freien Eintrag und gibt diesen zurück. Wurde kein freier Eintrag gefunden, so gibt die Funktion `-1` zurück.

82a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁81c 83a▷
```
unsigned short fat_get_free_fat_entry(int device,int offset);
```
Uses `device` 66a, `fat_get_free_fat_entry` 82c, and `offset` 68 75a.

Der Parameter `offset` legt fest, ab welcher CLN mit der Suche in der FAT begonnen werden soll. Bei einem Offset von `0` beginnt die Suche am Anfang der FAT. Hierzu wird eine Konstante eingeführt.

82b ⟨*fat constants* 49b⟩+≡ (89a) ◁65a 84c▷
```
#define BEGINFAT 0
```
Defines:
`BEGINFAT`, used in chunks 78b and 80.

Auch wenn mehrere FAT-Tabellen im Dateisystem existieren wird, nur die erste FAT Tabelle durchsucht. Die Implementierung vertraut darauf, dass beide Tabellen identisch sind. Freie FAT-Einträge enthalten den Wert `0x000`.

Im ersten Schritt wird die komplette FAT eingelesen und in das Array `temp_fat` geschrieben.

82c ⟨*fat function implementations* 44c⟩+≡ (89b) ◁81f 83b▷
```
unsigned short fat_get_free_fat_entry(int device, int offset){
   int fat_blocks[5] = {0,1,2,3,4};
   char tmp_fat[5*1024] = { 0 };

   for (int i = 0; i < 5; i++)
      readblock (device, fat_blocks[i], tmp_fat + i*1024);
   ⟨search free fat entry 82d⟩
return -1;
}
```
Defines:
`fat_blocks`, used in chunk 104a.
`fat_get_free_fat_entry`, used in chunks 72b, 75b, 78b, 80, and 82a.
`tmp_fat`, used in chunks 82d and 104a.
Uses `device` 66a, `i`, and `offset` 68 75a.

Die FAT beginnt direkt nach dem BPB mit einem Offset von 512 Bytes. Eine `for`-Schleife durchläuft Cluster-Eintrag für Cluster-Eintrag, bis das Maximum an Einträgen erreicht ist und prüft auf freie FAT-Einträge. Hier kommt das gleiche Verfahren wie bei der Funktion `fat_read` zum Einsatz. Die Variable `clustercount` zählt die Schleifendurchläufe. Wurde ein freier Eintrag ermittelt, wird der Inhalt von `clustercount` zurückgegeben.

82d ⟨*search free fat entry* 82d⟩≡ (82c)
```
unsigned short fatentry = 0;
unsigned short clustercount = 1+offset;
int fatoff;

for (int j = 1+offset; j <= fatMount.fat_TotalCountOfDataSectors; j++){
   fatoff = (int)(j * 1.5) + 512;
   fatentry = (tmp_fat[fatoff+1]<<8) | tmp_fat[fatoff];

   if (j % 2 > 0){
```

```
        if (fatentry>>4 == 0x000)
           return clustercount;
     }else{
         if ((fatentry & 0x0FFF) == 0x000)
            return clustercount;
     }
     clustercount++;
  }
```

Defines:
`clustercount`, never used.
`fatentry`, used in chunks 48, 83, 84a, and 104a.
`fatoff`, used in chunks 83 and 84a.
Uses `fatMount` 43b, `offset` 68 75a, and `tmp_fat` 82c.

4.9.3. fat_write_fat_entry

Mit

83a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁82a 84b▷

```
int fat_write_fat_entry(int device, unsigned short fatentry,
                        unsigned short value);
```

Uses `device` 66a, `fat_write_fat_entry`, and `fatentry` 82d.

wird ein FAT-Tabellen-Eintrag geschrieben. Beim Schreiben und Verändern von FATs ist es wichtig, dass beide FATs beschrieben werden, sofern zwei Tabellen vorhanden sind. Wie bereits in Kapitel 3.6 beschrieben, wird zuerst die Byte-Position eines Eintrages in der FAT und anschließend der zu lesende Block berechnet.

83b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁82c 85a▷

```
int fat_write_fat_entry(int device, unsigned short fatentry,
                        unsigned short value){
   int fat1offset = 512;

   int fatoff = (int)(fatentry * 1.5) + fat1offset;
   int reblock = (int)(fatoff / 1024);

   ⟨write fat entry 83c⟩
   ⟨write second FAT 84a⟩
   return 0;
}
```

Uses `device` 66a, `fat_write_fat_entry`, `fatentry` 82d, `fatoff` 82d, and `reblock` 73.

Nachdem der Block gelesen wurde, werden abhängig vom Verzeichniseintrag die oberen oder die unteren zwölf Bit aus zwei Bytes von `fbuf[blockoff]` analysiert.

83c ⟨*write fat entry* 83c⟩≡ (83b 84a)

```
readblock (device,reblock, (char *)fbuf);
int blockoff = (int)(fatoff%1024);

if (fatentry % 2 > 0){
   fbuf[blockoff+1] = value >> 4;
   fbuf[blockoff] = ((value % 16) << 4) | (fbuf[blockoff] % 16);
```

```
}else{
   fbuf[blockoff+1] = (fbuf[blockoff+1] & 0b11110000) | (value >> 8);
   fbuf[blockoff] = value % 256;
}

writeblock (device, reblock, (char*)&fbuf);
```

Uses `blockoff` 73, `device` 66a, `fatentry` 82d, `fatoff` 82d, `fbuf` 43b, and `reblock` 73.

Die gleiche Schreiboperation wird in der zweiten FAT ausgeführt. Wenn eine zweite FAT vorhanden ist, folgt diese gleich auf die erste. Die Position der Tabelle wird über folgende Formel bestimmt:

(Offset FAT1 + (BytesPerSector ∗ Anzahl Sektoren pro FAT)

84a ⟨*write second FAT* 84a⟩≡ (83b)

```
if (fatMount.fat_TotalCountOfFATs == 2){
    int fat2off = fat1offset + (fatMount.fat_BytesPerSector *
                                fatMount.fat_SizeOfFATSectors);
   fatoff = (int)(fatentry * 1.5) + fat2off;
   reblock = (int)(fatoff / 1024);
   ⟨write fat entry 83c⟩
}
```

Uses `fatentry` 82d, `fatMount` 43b, `fatoff` 82d, and `reblock` 73.

4.9.4. fat_get_free_dir_entry

Zum Ermitteln freier Verzeichniseinträge wird die Funktion

84b ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁83a 86a▷

```
int fat_get_free_dir_entry(int device, int cluster, int searchinroot);
```

Uses `device` 66a and `fat_get_free_dir_entry`.

implementiert. Die Funktion ermittelt im *Root-Verzeichnis* oder einem der Unterverzeichnisse in der *Datenregion* einen freien Verzeichniseintrag und gibt die Nummer des freien Eintrages zurück. Wurde kein freier Eintrag gefunden, gibt die Funktion `-1` zurück.

Aufgerufen wird die Funktion u.a. mit dem Parameter `searchinroot`, die Variable enthält den Wert einer der beiden Konstanten.

84c ⟨*fat constants* 49b⟩+≡ (89a) ◁82b

```
#define CLUSTERSEARCH 0
#define ROOTSEARCH 1
```

Defines:
`CLUSTERSEARCH`, used in chunk 78b.
`ROOTSEARCH`, used in chunk 78a.

Ein freier Eintrag ist entweder leer (`0x00`), oder er beginnt mit `0xE5`, siehe Kapitel 2.4.3.

Über den Funktionsparameter `searchinroot` wird unterschieden, ob die Funktion im *Root-Verzeichnis* oder in einem Cluster suchen soll. Die Variable `entrycount` zählt die durchsuchten Verzeichniseinträge mit.

85a ⟨*fat function implementations* 44c⟩+≡ (89b) ◁83b 86b▷

```
int fat_get_free_dir_entry(int device, int cluster, int searchinroot){
   int entrycount = 0;
   int i;

   if (searchinroot == 1){
      ⟨search dir entry in root 85b⟩
   } else {
      ⟨search dir entry in cluster 85c⟩
   }
   return -1;
}
```

Uses `device` 66a, `fat_get_free_dir_entry`, and `i`.

Bei der Suche im *Root-Verzeichnis* wird zunächst das gesamte *Root-Verzeichnis* eingelesen. Im anschließenden Schleifendurchlauf wird in 32-Byte-Schritten auf die Werte `0x0E` und `0x00` geprüft.

85b ⟨*search dir entry in root* 85b⟩≡ (85a)

```
int root_blocks[8] = {9,10,11,12,13,14,15,16};
char tmp_root[8*1024] = { 0 };

for (i = 0; i < 8; i++)
   readblock (device, root_blocks[i], tmp_root + i*1024);

int blockOffset = 512;

for (i = blockOffset ;i < (8*1024)-512; i += 32) {
   if (tmp_root[i] == 0x00 || tmp_root[i] == DIR_ENTRY_IS_FREE)
      return entrycount;
   entrycount++;
}
```

Uses `blockOffset`, `device` 66a, `DIR_ENTRY_IS_FREE` 50a, `i`, `root_blocks`, and `tmp_root`.

Sucht die Funktion in einem Cluster, wird das Daten-Cluster aus dem Parameter `cluster` eingelesen. Beim Einlesen von Daten-Clustern muss der `blockoffset` berücksichtigt werden.

85c ⟨*search dir entry in cluster* 85c⟩≡ (85a)

```
char dircluster[512] = { 0 };
int blockoffset;

if (cluster % 2 > 0){
   blockoffset = 0;
} else {
   blockoffset = 512;
}
readblock (device,(int)((31+(int)cluster)/2), (char *)fbuf);
memcpy (dircluster, &fbuf[blockoffset], 512 * sizeof(char));

for (i = 0 ;i < 512; i += 32){
   if (dircluster[i] == 0x00 || dircluster[i] == DIR_ENTRY_IS_FREE){
      return entrycount;
```

```
    }
    entrycount++;
}
```

Uses blockoffset 67b 71b, device 66a, DIR_ENTRY_IS_FREE 50a, dircluster, fbuf 43b, i, and memcpy 42a.

4.9.5. fat_write_dir_entry

Die Funktion `fat_write_dir_entry` schreibt einen Verzeichniseintrag auf den Datenträger.

86a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁84b 87b▷

```
int fat_write_dir_entry(int device, dirEntry dir,int cluster,
                        int dirno, int writeinroot);
```

Uses device 66a, dir 77, dirEntry 49a, and fat_write_dir_entry.

Dabei unterscheidet die Funktion anhand des Parameters `writeinroot`, ob die Datei in das *Root-Verzeichnis* oder in ein Daten-Cluster bzw. Unterverzeichnis geschrieben werden soll.

86b ⟨*fat function implementations* 44c⟩+≡ (89b) ◁85a 87c▷

```
int fat_write_dir_entry(int device, dirEntry dir,int cluster,
                        int dirno, int writeinroot){
    dirEntry *wdentry;

    if (writeinroot == 1){
        ⟨write dir entry in root 86c⟩
    } else {
        ⟨write dir entry in cluster 87a⟩
    }
    return 0;
}
```

Uses device 66a, dir 77, dirEntry 49a, fat_write_dir_entry, and wdentry 73.

Durch die Initialisierung des Dateisystems ist der erste Sektor des *Root-Verzeichnis*ses bekannt (`fat_FirstSectorOfRootDir`). Mit Hilfe der Formel

(512 ∗ fatMount.fat_FirstSectorOfRootDir)

wird der Start des *Root-Verzeichnis*ses ermittelt. Im nächsten Schritt wird die Byte-Position des zu schreibenden Verzeichniseintrages ermittelt und in `diroff` gespeichert. Dazu wird `dirno`, die Position des Eintrages, mit 32 multipliziert. Es folgt die Bestimmung des zu lesenden Blocks, das Lesen und die Berechnung des Offsets innerhalb des gelesenen Blocks. Anschließend wird der in `dir` übergebene Verzeichniseintrag mit `writeblock` geschrieben.

86c ⟨*write dir entry in root* 86c⟩≡ (86b)

```
int clusteroff = (512 * fatMount.fat_FirstSectorOfRootDir);
int diroff = clusteroff + (32 * dirno);
int reblock = diroff / 1024;

readblock (device,reblock, (char *)fbuf);
```

```
int blockoff = (int)(diroff%1024);

wdentry = (dirEntry*) &fbuf[blockoff];
*wdentry = dir;
writeblock (device, reblock, (char*)&fbuf);
```

Uses `blockoff` 73, `clusteroff`, `device` 66a, `dir` 77, `dirEntry` 49a, `diroff`, `fatMount` 43b, `fbuf` 43b, `reblock` 73, and `wdentry` 73.

Das Schreiben eines Unterverzeichniseintrages in ein Daten-Cluster ist ähnlich dem Schreiben ins *Root-Verzeichnis*. Damit das Cluster ermittelt werden kann, wird die `cluster`-Nummer übergeben. Mit Hilfe der Formel

$(31 + \text{clusternummer})/2)$

wird der Block ermittelt, aus dem gelesen bzw. in den geschrieben wird. Da nur ein 512-Byte-Cluster verändert werden soll, wird mit `blockoff` das entsprechende Cluster ermittelt.

87a ⟨*write dir entry in cluster* 87a⟩≡ (86b)

```
int blockoff;

if (cluster % 2 > 0){
   blockoff = (32 * dirno);
} else {
   blockoff = (32 * dirno) + 512;
}

readblock (device,(int)((31+(int)cluster)/2), (char *)fbuf);
wdentry = (dirEntry*) &fbuf[blockoff];
*wdentry = dir;
writeblock (device, (int)((31+(int)cluster)/2), (char*)&fbuf);
```

Uses `blockoff` 73, `device` 66a, `dir` 77, `dirEntry` 49a, `fbuf` 43b, and `wdentry` 73.

4.9.6. fat_delete

Mit Hilfe der Funktion `fat_delete()` werden Dateien gelöscht.

87b ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁86a 102a▷

```
int fat_delete (int device, const char *path);
```

Uses `device` 66a and `fat_delete` 87c.

Das Löschen von Dateien in FAT-Dateisystemen ist ein relativ einfaches Vorgehen. Es reicht aus, lediglich den Verzeichniseintrag (⟨*delete direntry* 88a⟩) und die Cluster-Kette in der FAT (⟨*delete fatentries* 88b⟩) zu entfernen. Die Daten-Cluster, welche die eigentlichen Dateiinhalte führen bleiben erhalten.

87c ⟨*fat function implementations* 44c⟩+≡ (89b) ◁86b

```
int fat_delete (int device, const char *path){
   ⟨delete direntry 88a⟩
   ⟨delete fatentries 88b⟩
return 0;
}
```

Defines:
`fat_delete`, used in chunk 87b.
`fat_filestat`, used in chunks 58c and 69b.
`int_fat_inode`, used in chunks 31b, 57, 62b, and 69b.
Uses `device` 66a.

Dieser Code-Chunk beschreibt wie ein Verzeichniseintrag gelöscht wird. Wie in Kapitel 2.4.3 beschrieben, reicht es aus das erste Byte des Verzeichniseintrages auf den Wert `0xE5` zu setzen. Damit wird der Eintrag als frei gekennzeichnet. Die restlichen Informationen bleiben unberührt.

88a ⟨*delete direntry* 88a⟩≡ (87c)
```
dirEntry fatdir, *wdentry;
int dirpos;

dirpos = fat_file_exists(device, path, &fatdir);

if (dirpos == -1){
   printf("Datei existiert nicht\n");
   return -1;
}

fatdir.DIR_shortName[0] = 0xE5;

int reblock = dirpos / 1024;
int blockoff = (dirpos)%1024;

readblock (device,reblock, (char *)fbuf);
wdentry = (dirEntry*) &fbuf[blockoff];
*wdentry = fatdir;
writeblock (device, reblock, (char*)&fbuf);
```
Uses `blockoff` 73, `device` 66a, `dirEntry` 49a, `dirpos` 69b 73, `fat_file_exists` 50c, `fatdir` 60c, `fbuf` 43b, `printf` 42a, `reblock` 73, and `wdentry` 73.

Nachdem der Verzeichniseintrag als frei markiert wurde, wird die Cluster-Kette von Anfang bis Ende durchlaufen. Bei jedem Schritt wird der Wert des aktuellen FAT-Eintrages auf `0x000` gesetzt und somit als frei gekennzeichnet.

88b ⟨*delete fatentries* 88b⟩≡ (87c)
```
unsigned short curclstno = fatdir.DIR_FirstClusterLow;
unsigned short nextclstno;

while(nextclstno != 0xFFF){
   nextclstno = read_fat_entry (device, curclstno);
   fat_write_fat_entry(device, curclstno, 0x000);
   curclstno = nextclstno;
}
```
Uses `curclstno`, `device` 66a, `fat_write_fat_entry`, `fatdir` 60c, `nextclstno`, and `read_fat_entry` 48b.

Die Daten-Cluster werden nicht überschrieben.

4.10. Zusammenführen des Sourcecodes

Der gesamte Sourcecode dieser Arbeit, wird in den folgenden zwei Code-Chunks zu einem ULIX-Modul zusammengeführt. Dabei entsteht eine Header-Datei `module.h` und eine Quelltext-Datei `module.c`.

Folgender Code-Chunk definiert die Header-Datei dieses ULIX-Modules. Darin werden u.a. Konstanten und die Funktionsprototypen aufgeführt.

89a ⟨*module.h* 89a⟩≡
⟨*fat constants* 49b⟩
⟨*fat type definitions* 38⟩
⟨*fat global variables* 43b⟩
⟨*fat function prototypes* 35a⟩

Die Implementierungen der Funktionen, die Modulinitialisierung und die Einbindung der `module.h`-Datei sowie die Tests sind Teil des Code-Chunk ⟨*module.c* 89b⟩.

89b ⟨*module.c* 89b⟩≡
⟨*module header* 41a⟩
⟨*init module* 41b⟩
⟨*fat function implementations* 44c⟩
⟨*Test Functions* 102b⟩

5. Tests

Dieses Kapitel widmet sich der Qualitätssicherung der Implementierung durch Tests. Dabei wird zuerst auf die Testumgebung sowie die Teststrategie eingegangen. Anschließend werden die definierten Testfälle beschrieben und bewertet. Testen ist ein Prozeß, ein Programm mit der Absicht auszuführen und Fehler zu finden (vgl. Myers & Pieper, 1995, S. 4).

5.1. Testumgebung

Als Basissystem für die Testumgebung dient ein Debian 6.0.1 (squezze) Linux. ULIX selbst läuft virtualisiert in einer QUEMU Umgebung. Die verwendete ULIX Version ist 0.10. Die ULIX-Programme `diff` und `hexdump` werden verwendet, um die Testergebnisse zu analysieren und validieren. Des Weiteren werden einige Testergebnisse mit Analyseprogrammen aus *The Sleuth Kit* Version 3.1.3 validiert. Darunter `fsstat`, `fls` und `istat`. Als Testgegenstände werden wie in Kapitel 4.1 beschrieben, Floppy-Images verwendet.

5.2. Funktionale Tests

Dieses Kapitel dokumentiert die durchgeführten Tests. Getestet wurde mit einem Systemtest. Dabei wird die fertige Software gegen die in Kapitel 3.1 definierten Anforderungen getestet (vgl. Liggesmeyer, 2009, S. 435).

5.2.1. Test Case: Auslesen Bootsektor

Testbeschreibung

Die Testfunktion `TC_printbootsektor` startet die Dateisystem-Initialisierung und gibt den Inhalt der Struktur `fatMount` aus.

Erwartetes Ergebnis

Der Bootsektor wird ausgelesen, alle Werte werden korrekt berechnet und der Struktur `fatMount` zugewiesen.

Ergebnis

Nach dem Aufruf, hat die Testfunktion folgendes Ergebnis geliefert:

```
FAT BOOTSEKTOR
Anzahl Bytes pro Sektor:                    512
```

```
Anzahl Sektoren pro Cluster:                 1
Anzahl reservierter Sektoren:                1
Anzahl FATs:                                 2
Anzahl Root Directory Eintraege:             224
Gesamt Anzahl aller Sektoren:                2880
Gesamt Anzahl aller Cluster in Data Region:  2847
Anzahl aller Sektoren in der Data Region:    2847
Anzahl Sektoren pro FAT:                     9
Anzahl Sektoren im Root Dir:                 14
Start Sektor des Root-Verzeichnisses:        19
Start Sektor der 1. FAT:                     1
Start Sektor der Data Region:                33
FAT Type:                                    12
```

Zum Vergleich wird das Floppy-Image mit `fsstat` aufgerufen. `fsstat` liefert u.a. folgendes Ergebnis:

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: FAT12

OEM Name: mkfs.fat
Volume ID: 0x0
Volume Label (Boot Sector):
Volume Label (Root Directory): ULIXFAT
File System Type Label:

Sectors before file system: 0

File System Layout (in sectors)
Total Range: 0 - 2879
* Reserved: 0 - 0
** Boot Sector: 0
* FAT 0: 1 - 9
* FAT 1: 10 - 18
* Data Area: 19 - 2879
** Root Directory: 19 - 32
** Cluster Area: 33 - 2879
...
```

Beurteilung

Test erfolgreich.

5.2.2. Test Case: Konvertieren in 8.3 Format

Testbeschreibung

Die Testfunktion `TC_convertto_short` konvertiert unterschiedliche `char`-Arrays zu einem 8.3-Format.

Erwartetes Ergebnis

Alle Strings werden in ein korrektes 8.3-Format konvertiert.

Ergebnis

Nach dem Aufruf hat die Testfunktion folgendes Ergebnis geliefert:

```
CONVERT TEST
Test 1:

ONE.TXT -> ONE     TXT

Test 2:

ONEONEONEONEONE.TXT -> ONEONE~1TXT

Test 3:

ONE -> ONE

Test 4:

ONEONEONEONEONE -> ONEONE~1

Test 5:

ONE.JS -> ONE     JS
```

Beurteilung

Test erfolgreich.

5.2.3. Test Case: Kopieren von Dateien

Testbeschreibung

In diesem Test Case werden Dateien von der Minix-Partition auf die FAT-Partition kopiert. Getestet wird mit unterschiedlichen Dateigrößen. Bei kleinen Dateien wird nur ein Cluster belegt und bei größeren Dateien muss eine verkettete Liste aufgebaut werden, welche die Cluster Reihenfolge definiert.

Die Dateien werden an unterschiedlichen Orten auf dem Dateisystem abgelegt. Im *Root-Verzeichnis* und in Unterverzeichnissen.

Ausgangspunkt ist ein Floppy-Image mit folgender Verzeichnisstruktur:

```
/
└── SIMON
    └── NICOLE
        └── FYNN
```

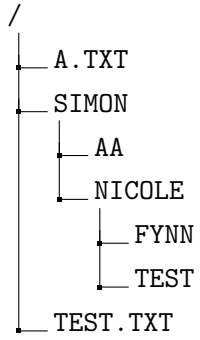
Unter ULIX befinden sich auf der Minix-Partition die Testdateien `a.txt` (31 Byte) und `test.txt` (1.680 Byte). Diese werden mit dem `cp` Programm auf das FAT-Dateisystem kopiert.

Erwartetes Ergebnis

Nach dem Kopiervorgang befinden sich die Dateien im *Root-Verzeichnis* und in den Unterverzeichnissen. Mit Hilfe des `diff`-Programms werden die Minix-Dateien mit den FAT-Dateien verglichen. Dabei dürfen die Dateien sich weder im Inhalt noch in der Größe unterscheiden.

Ergebnis

Nach dem Kopiervorgang befanden sich die Dateien im *Root-Verzeichnis* und in den Unterverzeichnissen:



Im Anschluss werden die kopierten Dateien mit den Ursprungsdateien verglichen:

```
diff a.txt /mnt2/A.TXT
diff test.txt /mnt2/TEST.TXT
diff a.txt /mnt2/SIMON/AA
diff test.txt /mnt2/SIMON/NICOLE/TEST
```

Alle 4 kopierten Dateien waren identisch. Es wurden auch nicht mehr Dateien angelegt.

Beurteilung

Test erfolgreich.

5.2.4. Test Case: Anwendung unter Linux

Testbeschreibung

Dieser Test soll zeigen das FAT-Dateisysteme, welche unter ULIX beschrieben worden sind, auch unter Linux funktionieren. Dabei wird das aus Test Case 5.2.3 beschriebene Dateisystem als Ausgangspunkt verwendet.

Das Dateisystem wird mit folgendem Befehl unter Debian eingebunden:

```
sudo mount -t vfat -o loop floppy.img /media/floppy
```

Erwartetes Ergebnis

Das Floppy-Image lässt sich unter Debian einbinden. Mit `ls` sind alle kopierten Dateien auffindbar. Mit dem Programm `cat` werden Dateien ausgegeben und mit `cp` wird eine Datei aus Debian kopiert. Anschließend wird ULIX gestartet und die aus Debian kopierte Datei ebenfalls mit `cat` ausgegeben.

Ergebnis

Das Dateisystem lässt sich einbinden. Die Dateien können mit `ls` dargestellt und mit `cat` fehlerfrei und vollständig ausgegeben werden.

Die unter Debian kopierte Datei (`~/ulix/Makefile`) konnte unter ULIX mit `cat` fehlerfrei dargestellt werden.

Beurteilung

Test erfolgreich.

5.2.5. Test Case: Ausgabe unter ULIX

Testbeschreibung

Die sequenzielle Ausgabe der Dateien von Anfang bis Ende wird in diesem Test überprüft. Die kopierten Dateien aus Test Case 5.2.3 werden mit `cat` ausgegeben.

Erwartetes Ergebnis

Egal auf welcher Verzeichnisebene die Dateien liegen, sie werden mit `cat` fehlerfrei und vollständig ausgeben.

Ergebnis

Die Ausgaben von

```
cat a.txt /mnt2/A.TXT
cat test.txt /mnt2/TEST.TXT
cat a.txt /mnt2/SIMON/AA
cat test.txt /mnt2/SIMON/NICOLE/TEST
```

waren alle fehlerfrei und vollständig.

Beurteilung

Test erfolgreich.

5.2.6. Test Case: Wahlfreie Ausgabe unter ULIX

Testbeschreibung

Die wahlfreie Ausgabe der Dateien wird mit dem Programm `hexdump` getestet. Dabei sollen n Byte von einer beliebigen Stelle in einer Datei ausgegeben werden. Validiert wird das Ergebnis mit der Ausgabe der Ursprungsdatei. Auch Ausgaben die Cluster-Grenzen (512 Byte) überschreiten werden berücksichtigt.

Erwartetes Ergebnis

Folgende Aufrufe erzeugen die exakt gleiche Ausgabe:

```
hexdump a.txt 5 10
hexdump /mnt2/A.TXT 5 10
hexdump test.txt 510 100
hexdump /mnt2/TEST.TXT 510 100
```

Ergebnis

Die Ausgaben sind identisch:

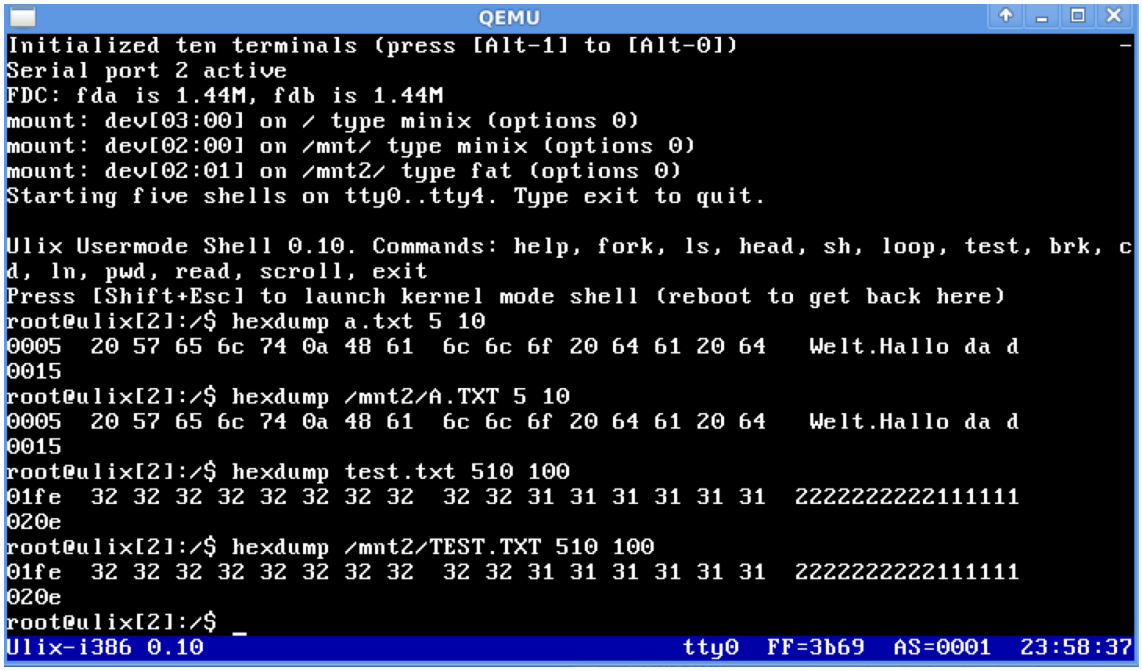


Abbildung 5.1.: Testergebnis wahlfreie Ausgabe

Beurteilung

Test erfolgreich.

5.2.7. Test Case: Wahlfreies Schreiben unter ULIX

Testbeschreibung

Wie im Test Case 5.2.6 das beliebige Lesen von Dateien getestet wurde, überprüft dieser Test Case das wahlfreie Schreiben in Dateien. Dabei soll eine Datei geöffnet werden und an eine beliebige Stelle das Wort LEA geschrieben werden. Für diesen Test wurde das Testprogramm `tcfatwrite` entwickelt, siehe A.4.

Erwartetes Ergebnis

Nach Ausführung des Testprogramms `tcfatwrite` ist das Wort LEA ab Position 10 in der Datei. Die Datei hat immer noch die gleiche Dateigröße wie vor dem Test.

Ergebnis

Nach Aufruf des Programms befindet sich LEA an Position 10.

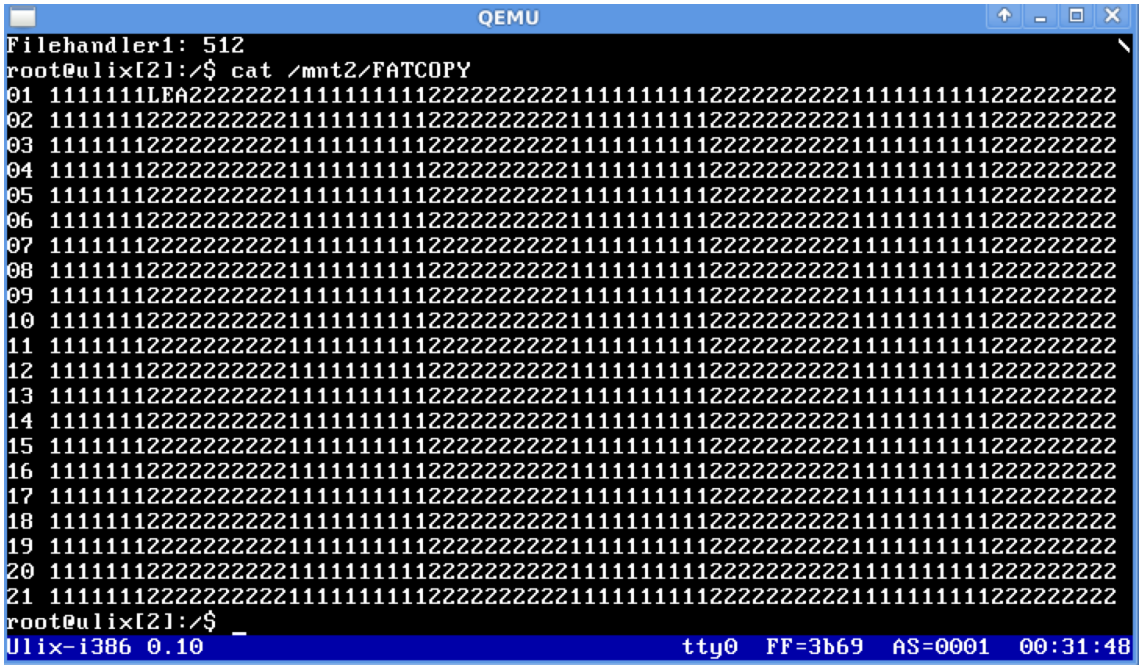


Abbildung 5.2.: Testergebnis wahlfreies Schreiben

Die Dateien unterscheiden sich im Inhalt:

```
root@ulix[2]:/$ diff test.txt /mnt2/FATCOPY
DIFF: 50 != 76
files differ (content)
```

Die Größe der Datei ist immernoch 1.680 Byte wie die Ausgabe von `istat` zeigt:

```
ulix@ulixdevel:~/ulix/bin-build$ istat -f fat floppy.img 42
Directory Entry: 42
Allocated
File Attributes: File, Archive
Size: 1680
Name: FATCOPY

Directory Entry Times:
Written:        Thu Jan  1 01:00:00 1970
Accessed:       Thu Jan  1 01:00:00 1970
Created:        Thu Jan  1 01:00:00 1970
```

Beurteilung

Test erfolgreich.

5.2.8. Test Case: Wahlfreies Schreiben über das Dateiende

Testbeschreibung

Dieser Test Case schreibt das Wort FYNN an eine Position 2.048 Bytes nach Dateiende. Dabei wird die Differenz zwischen Dateiende und Schreibposition mit `0`-Werten gefüllt. Das Testprogramm `tcfatwritebeyond` simuliert diesen Fall.

Erwartetes Ergebnis

Die Dateigröße erweitert sich um 2.048 Byte + 4 Byte (FYNN). Die Differenz zwischen Dateiende und Schreibposition ist mit `0`-Werten gefüllt.

Ergebnis

`tcfatwritebeyond` schreibt an Position 2.048 das Wort FYNN. Die Differenz ist mit Null (`0x20`)-Werten gefüllt. Die Ausgabe von `istat` zeigt, dass die Dateigröße wie erwartet um 2.052 Byte angewachsen ist.

```
ulix@ulixdevel:~/ulix/bin-build$ istat -f fat floppy.img 42
Directory Entry: 42
Allocated
File Attributes: File, Archive
Size: 3733
Name: FATCOPY

Directory Entry Times:
Written:        Thu Jan  1 01:00:00 1970
Accessed:       Thu Jan  1 01:00:00 1970
Created:        Thu Jan  1 01:00:00 1970
```

Beurteilung

Test erfolgreich.

5.2.9. Test Case: Zeitgleicher Dateizugriff

Testbeschreibung

Dieser Test Case öffnet die gleiche Datei zweimal, setzt mit dem ersten Deskriptor die Schreibposition an eine beliebige Stelle und schreibt das Wort LEA. Anschließend wird der Deskriptor geschlossen und der zweite Dateideskriptor setzt eine weitere Schreibposition. Bevor der zweite Deskriptor geschlossen wird, schreibt er das Wort FYNN an die definierte Stelle.

Erwartetes Ergebnis

Die beiden Wörter LEA und FYNN tauchen in der Datei an zwei unterschiedlichen Stellen auf.

Ergebnis

Die beiden Wörter wurden geschrieben:

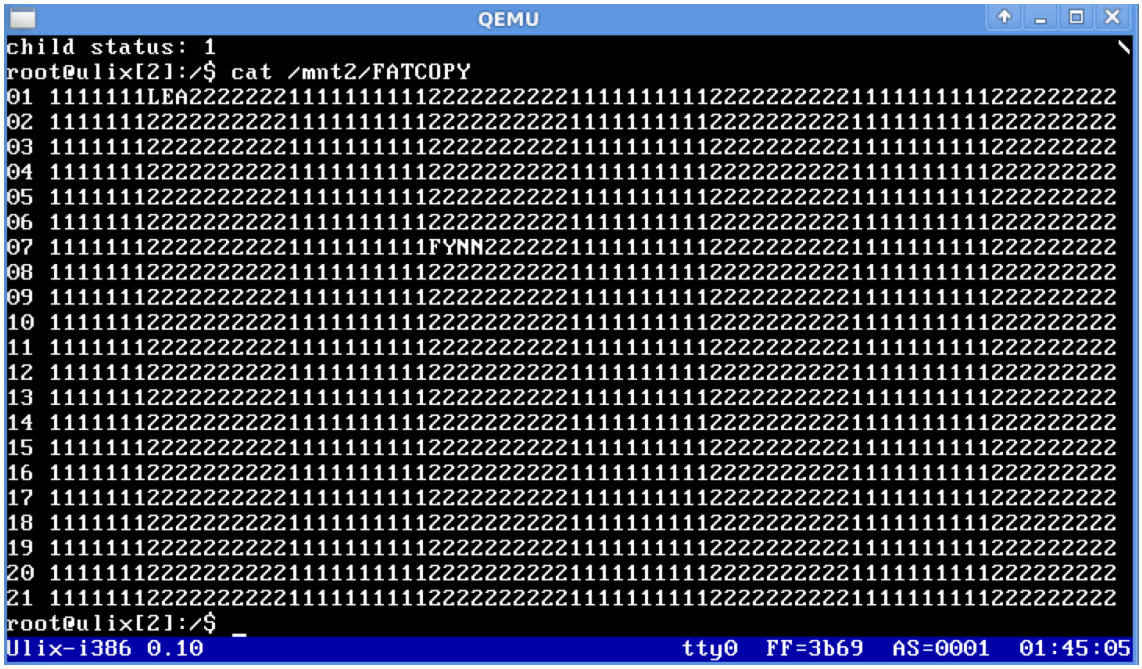


Abbildung 5.3.: Test Ergebnis zeitgleiches Schreiben

Beurteilung

Test erfolgreich.

5.2.10. Test Case: Löschen von Dateien

Testbeschreibung

Im Rahmen dieses Tests wird eine Datei auf das FAT-Dateisystem kopiert und anschließend wieder gelöscht. Zur Überprüfung, ob die allokierten FAT-Einträge wieder freigegeben werden wurde die Test-Funktion `TC_countfreecluster` implementiert. Für den Test wird die Funktion nach dem Kopieren und nach dem Löschen aufgerufen. Zur Überprüfung, ob der Verzeichniseintrag entfernt wurde, wird das Floppy-Image unter Debian gemountet.

Erwartetes Ergebnis

Die kopierte Datei befindet sich nicht mehr auf dem Datenträger und der Speicherplatz (Daten-Cluster) wurde freigegeben.

Ergebnis

Es wurden genau so viele Cluster wieder freigegeben, wie während des Kopiervorgangs auf den Datenträger geschrieben wurden. Nach dem das Floppy-Image unter Debian gemountet wurde, konnte der Verzeichniseintrag nicht mehr gefunden werden.

Beurteilung

Test erfolgreich.

6. Fazit / Ausblick

Zum Schluss dieser Arbeit wird in diesem Kapitel die Arbeit zusammengefasst. Zudem wird die Kritik an der eigenen Arbeit beleuchtet und ein Fazit gezogen. In Kapitel 6.4 wird auf den weiteren Forschungsbedarf eingegangen.

6.1. Zusammenfassung

Zu Beginn der Arbeit erfolgte eine intensive Einarbeitung in ULIX und dem ULIX-Sourcecode, im speziellen was die Umsetzung der Dateisysteme unter ULIX angeht. Dabei war die Herausforderung, die Integration von Dateisystemen in einem UNIX-artigen Betriebssystem wie ULIX zu verstehen. Ein zentrales Element an dieser Stelle war das Verständnis des virtuellen Dateisystems und die Funktionsweise der generischen Funktionen sowie der Abstraktion.

Darauf folgte die Auseinandersetzung mit dem FAT-Dateisystem. Die Einarbeitung in die Grundlagen und die Recherche zu verschiedenen FAT-Varianten, waren wichtige Komponenten in dieser Phase. Weiter wurden die in dieser Arbeit verwendeten Floppy-Image Dateien mit *Hexdumps* untersucht. Die Ausarbeitung der FAT-Grundlagen konzentrierte sich auf FAT12. Anschließend wurden erste Grobkonzepte zur Integration eines FAT-Dateisystemtreibers in ULIX erstellt. Dabei wurden Implementierungen von FAT-Treibern in Debian betrachtet. Nach der Einarbeitung in *Literate Programming* erfolgte die Implementierung eines ersten Prototyps.

Die Umsetzung des Prototyps und die darauf aufbauende Implementierung wurde unter der ULIX-Version 0.10 durchgeführt. Mit Hilfe des iterativen Vorgehens wurde die Implementierung stetig verfeinert und erweitert.

Die Tests während der Entwicklung und die Systemtests wurden in ULIX und Debian durchgeführt.

6.2. Fazit

Die vorangegangenen Recherchen und die Einarbeitung in dieses Thema lieferten keine Hinweise auf eine bereits existierende Implementierung eines FAT-Dateisystemtreibers in *Literate Programming*, was die Einzigartigkeit dieser Arbeit herausstellt.

Auch wenn FAT unter Windows nicht mehr das Standard Dateisystem ist, so wird es nach wie vor unterstützt und ist darüber hinaus weitverbreitet, vor allem bei portablen

Multimedia-Geräten und Speicherkarten. Somit ist das Forschungsthema *FAT-Dateisystemtreiber* immer noch aktuell. Es wird zum Beispiel an neuen FAT-Varianten zur Performanceverbesserung geforscht, um die Anforderungen zukünftiger Anwendungen zu erfüllen (vgl. Choi et al., 2008), oder es werden bestehende FAT-Varianten weiter verbessert (vgl. Munegowda et al., 2014).

Obwohl *Literate Programming* für den Autor eine neue Programmiermethode war, machte die Einarbeitung keinerlei Schwierigkeiten. Die Arbeit mit dieser Methode erwies sich während der Umsetzung als hilfreich. Die Code-Chunks förderten die Struktur und die Wiederverwendung. Es erwies sich als eine interessante Erfahrung im Vergleich zur klassischen Entwicklung. ULIX ist in C und Assembler geschrieben. Die Entwicklung der Komponente in C war eine größere Herausforderung. Wissen über Pointer und der Umgang mit Speicherplatz mussten erarbeitet werden. Zudem fand die Entwicklung auf einer Systemtiefe statt, in der es keinen Zugriff auf Standard Bibliotheken gab.

Im Rahmen dieser Arbeit wurden grundlegende Funktionalitäten eines FAT-Dateisystemtreibers erfolgreich implementiert und getestet. Dabei konnte aufgezeigt werden, wie sich weitere Dateisysteme in ULIX integrieren lassen und wie FAT-Dateioperationen in ULIX-artigen Betriebssystemen arbeiten. Die Realisierung des Treibers war ein sehr interessanter Teil, der nicht nur die Funktionsweise von FAT-Dateisystemen beleuchtet, sondern auch ein tieferes Verständnis vom Umgang mit Betriebs- und Dateisystemen erforderte. Der Autor erlangte dadurch einen völlig neuen Bilck auf Datei- und Betriebssysteme.

6.3. Kritik an der Arbeit

Ob die Implementierung der Komponente *FAT-Treiber* in Literate Programming, dem übergeordneten Ziel: Die Verbesserung der Vermittlung von Betriebssystemen dienen kann, lässt sich mit dieser Arbeit nicht beantworten.

Auch wenn grundlegende Funktionalitäten erfolgreich implementiert wurden, so fehlt die Unterstützung des *ls*-Programms. Das setzt die generischen Funktionen `u_getdent` und `u_stat` voraus, welche von der ULIX 0.10 nicht unterstützt werden. Aus diesem Grund wurde in dieser Arbeit die Implementierung simuliert.

Die aktuelle Implementierung kann als Grundlage zur Unterstützung weiterer FAT-Varianten dienen. Dazu müssen Teile der Implementierung generischer umgesetzt werden. So geht zum Beispiel der aktuelle Treiber, von einer festen Clustergröße von 512 Byte aus oder die Bereichsgrößen für FAT und das *Root-Verzeichnis* sind fest definiert.

6.4. Weiterer Forschungsbedarf

Der Funktionsumfang des in dieser Arbeit realisierten FAT-Dateisystemtreibers, geht auf die grundlegenden Prinzipien eines FAT-Dateisystems ein. Ausgehend von der Analyse des Dateisystems, über den Umgang mit der FAT bis zum Zugriff auf die Daten-Cluster.

Dieser Stand kann als Grundlage für einen weiteren Ausbau des FAT-Treibers dienen. So wäre es denkbar, weitere FAT-Varianten wie z. B. FAT32 oder Erweiterungen wie VFAT zu unterstützen. Auch Funktionalitäten wie `mkdir` oder die Unterstützung langer Dateinamen sowie die Verknüpfung der ULLX-Zugriffsrechte wären mögliche Ausbaustufen. Ebenfalls wäre eine Migration auf eine aktuelle ULIX-Version eine nächste Evolutionsstufe.

A. Anhang: Testprogramme

A.1. TC_printbootsektor()

102a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁87b 103a▷

```
void TC_printbootsektor(int device);
```

Uses `device` 66a and `TC_printbootsektor`.

102b ⟨*Test Functions* 102b⟩≡ (89b) 103b▷

```
void TC_printbootsektor(int device){

    fat_filesystem_init (device);

    printf ("FAT BOOTSEKTOR\n");
    printf ("Anzahl Bytes pro Sektor:\t\t\t%d\n",
            fatMount.fat_BytesPerSector);
    printf ("Anzahl Sektoren pro Cluster:\t\t\t%d\n",
             fatMount.fat_SecPerCluster);
    printf ("Anzahl reservierter Sektoren:\t\t\t%d\n",
             fatMount.fat_ReservedSectors);
    printf ("Anzahl FATs:\t\t\t\t%d\n",
             fatMount.fat_TotalCountOfFATs);
    printf ("Anzahl Root Directory Eintraege:\t\t%d\n",
             fatMount.fat_MaxRootDirEntries);
    printf ("Gesamt Anzahl aller Sektoren:\t\t\t%d\n",
             fatMount.fat_TotalCountOfSectors);
    printf ("Gesamt Anzahl aller Cluster in Data Region:\t%d\n",
             fatMount.fat_TotalCountOfDataClusters);
    printf ("Anzahl aller Sektoren in der Data Region:\t%d\n",
             fatMount.fat_TotalCountOfDataSectors);
    printf ("Anzahl Sektoren pro FAT:\t\t\t%d\n",
             fatMount.fat_SizeOfFATSectors);
    printf ("Anzahl Sektoren im Root Dir:\t\t\t%d\n",
             fatMount.fat_SizeOfRootDirSectors);
    printf ("Start Sektor des Root-Verzeichnisses:\t\t%d\n",
             fatMount.fat_FirstSectorOfRootDir);
    printf ("Start Sektor der 1. FAT:\t\t\t%d\n",
             fatMount.fat_FirstSectorOfFAT);
    printf ("Start Sektor der Data Region:\t\t\t%d\n",
             fatMount.fat_FirstSectorOfData);
    printf ("FAT Type:\t\t\t\t%d\n",
             fatMount.fat_FATType);
}
```

Uses `device` 66a, `fat_filesystem_init` 44c, `fatMount` 43b, `printf` 42a, and `TC_printbootsektor`.

A.2. TC_convertto_short()

103a ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁102a 103c▷

```
void TC_convertto_short();
```

Uses TC_convertto_short.

103b ⟨*Test Functions* 102b⟩+≡ (89b) ◁102b 104a▷

```
void TC_convertto_short(){
   printf ("CONVERT TEST\n");
   char Test1[30] = {'O','N','E','.','T','X','T','\0'};
   char Test2[30] = {'O','N','E','O','N','E','O','N','E',
                     'O','N','E','O','N','E','.','T','X','T','\0'};
   char Test3[30] = {'O','N','E','\0'};
   char Test4[30] = {'O','N','E','O','N','E','O','N','E',
                     'O','N','E','O','N','E','\0'};
   char Test5[30] = {'O','N','E','.','J','S','\0'};

   char shortstr[12];

   printf ("Test 1: \n\n");
   convertto_shortname(Test1, shortstr);
   shortstr[11] = '\0';
   printf ("%s -> %s \n\n", Test1, shortstr);

   printf ("Test 2: \n\n");
   convertto_shortname(Test2, shortstr);
   shortstr[11] = '\0';
   printf ("%s -> %s \n\n", Test2, shortstr);

   printf ("Test 3: \n\n");
   convertto_shortname(Test3, shortstr);
   shortstr[11] = '\0';
   printf ("%s -> %s \n\n", Test3, shortstr);

   printf ("Test 4: \n\n");
   convertto_shortname(Test4, shortstr);
   shortstr[11] = '\0';
   printf ("%s -> %s \n\n", Test4, shortstr);

   printf ("Test 5: \n\n");
   convertto_shortname(Test5, shortstr);
   shortstr[11] = '\0';
   printf ("%s -> %s \n\n", Test5, shortstr);
}
```

Uses convertto_shortname 54b, printf 42a, and TC_convertto_short.

A.3. TC_countfreecluster()

103c ⟨*fat function prototypes* 35a⟩+≡ (89a) ◁103a

```
void TC_countfreecluster(int device);
```

Uses device 66a and TC_countfreecluster.

104a ⟨*Test Functions* 102b⟩+≡ (89b) ◁103b

```
void TC_countfreecluster(int device){
   unsigned int freecount=0;
   unsigned int usedcount=0;
   int fatentry = 0;
   int i = 0;

   int fat_blocks[5] = {0,1,2,3,4};
   char tmp_fat[5*1024] = { 0 };

   for (i=0; i<5; i++)
      readblock (device, fat_blocks[i], tmp_fat + i*1024);

   i = 512;

   for (int j=0; j<fatMount.fat_TotalCountOfDataSectors; j++){
      fatentry = (tmp_fat[i+1]<<8) | tmp_fat[i];
      if (j % 2 > 0){
         if (fatentry >> 4 == 0x000)
           freecount++;
         else
           usedcount++;
         }else{
           if ((fatentry & 0x0FFF) == 0x000)
              freecount++;
           else
              usedcount++;
        }
      i++;
   }
   printf("usedcount: %d\n", usedcount);
   printf("freecount: %d\n", freecount);
}
```

Uses device 66a, fat_blocks 82c, fatentry 82d, fatMount 43b, i, printf 42a, TC_countfreecluster, and tmp_fat 82c.

A.4. tcfatwrite

104b ⟨*tcfatwrite* 104b⟩≡

```
#include "../ulixlib.h"
#define return exit(0);

#define O_RDONLY        0x0000     /* read only */
#define O_WRONLY        0x0001     /* write only */
#define O_RDWR          0x0002     /* read and write */
#define O_APPEND        0x0008     /* append mode */
#define O_CREAT         0x0200     /* create file */

int main (int argc, char* argv[]) {
    int fd1;
```

```
    char name1[3] = {'L','E','A'};

    fd1 = open("/mnt2/FATCOPY",O_RDWR);
    printf("Filehandler1: %d\n",fd1);

    lseek(fd1, 10,SEEK_SET);
    write(fd1,name1,3);

    close(fd1);
    return 0;
}
```

Uses main, O_APPEND 58a, O_CREAT 58a, O_RDONLY 58a, O_RDWR 58a, O_WRONLY 58a, printf 42a, and SEEK_SET.

A.5. tcfatwritebeyond

105a ⟨*tcfatwritebeyond* 105a⟩≡ 105b ▷

```
#include "../ulixlib.h"
#define return exit(0);

#define O_RDONLY        0x0000     /* read only */
#define O_WRONLY        0x0001     /* write only */
#define O_RDWR          0x0002     /* read and write */
#define O_APPEND        0x0008     /* append mode */
#define O_CREAT         0x0200     /* create file */

int main (int argc, char* argv[]) {
    int fd1;
    char name[4] = {'F','Y','N','N'};

    fd1 = open("/mnt2/FATCOPY",O_RDWR);

    lseek(fd1, 2048,SEEK_END);
    write(fd1,name,5);
    close(fd1);

    return 0;
}
```

Uses main, O_APPEND 58a, O_CREAT 58a, O_RDONLY 58a, O_RDWR 58a, O_WRONLY 58a, and SEEK_END.

A.6. tcfatmultiwrite

105b ⟨*tcfatwritebeyond* 105a⟩+≡ ◁ 105a

```
#include "../ulixlib.h"
#define return exit(0);

#define O_RDONLY        0x0000     /* read only */
#define O_WRONLY        0x0001     /* write only */
#define O_RDWR          0x0002     /* read and write */
```

```
#define O_APPEND        0x0008     /* append mode */
#define O_CREAT         0x0200     /* create file */

int main (int argc, char* argv[]) {
    int fd1, fd2;
    char name1[3] = {'L','E','A'};
    char name2[4] = {'F','Y','N','N'};

    fd1 = open("/mnt2/FATCOPY",O_RDWR);
    printf("Filehandler1: %d\n",fd1);
    fd2 = open("/mnt2/FATCOPY",O_RDWR);
    printf("Filehandler2: %d\n",fd2);

    lseek(fd1, 10,SEEK_SET);
    lseek(fd2, 510,SEEK_SET);
    write(fd1,name1,3);
    write(fd2,name2,4);

    close(fd1);
    close(fd2);
    return 0;
}
```

Uses main, O_APPEND 58a, O_CREAT 58a, O_RDONLY 58a, O_RDWR 58a, O_WRONLY 58a, printf 42a, and SEEK_SET.

B. Code-Chunks

⟨*absolute Byte Position* 35b⟩
⟨*Clusternummer* 35c⟩
⟨*conv ext* 55a⟩
⟨*conv noext* 55b⟩
⟨*Datenregion Zugriff* 37⟩
⟨*define parent dir* 77⟩
⟨*delete direntry* 88a⟩
⟨*delete fatentries* 88b⟩
⟨*fat constants* 49b⟩
⟨*fat function implementations* 44c⟩
⟨*fat function prototypes* 35a⟩
⟨*fat getdent* 31a⟩
⟨*fat global variables* 43b⟩
⟨*fat open* 60c⟩
⟨*fat read* 66b⟩
⟨*fat stat* 31b⟩
⟨*fat type definitions* 38⟩
⟨*fat write* 70⟩
⟨*fat write block 1* 71a⟩
⟨*fill gap* 72a⟩
⟨*get dates* 81d⟩
⟨*get fat value* 48d⟩
⟨*init* 44d⟩
⟨*init module* 41b⟩
⟨*little to big endian* 48c⟩
⟨*module header* 41a⟩
⟨*module.c* 89b⟩
⟨*module.h* 89a⟩
⟨*read loop* 68⟩
⟨*search dir entry in cluster* 85c⟩
⟨*search dir entry in root* 85b⟩
⟨*search free fat entry* 82d⟩
⟨*search loop* 51⟩
⟨*search loop data* 53⟩
⟨*search loop root* 52⟩
⟨*tcfatwrite* 104b⟩
⟨*tcfatwritebeyond* 105a⟩
⟨*Test Functions* 102b⟩
⟨*u_ open* 34⟩
⟨*ulix mount table* 33⟩
⟨*update direntry* 73⟩
⟨*update gapvalues* 72b⟩
⟨*update values* 75b⟩
⟨*write dir entry in cluster* 87a⟩
⟨*write dir entry in root* 86c⟩
⟨*write fat entry* 83c⟩
⟨*write file* 80⟩
⟨*write nbyte* 75a⟩
⟨*write second FAT* 84a⟩
⟨*writeinroot* 78a⟩
⟨*writeinsubdir* 78b⟩

C. Identifier

Literaturverzeichnis

Baumgarten, U., & Siegert, H. J. (2007). *Betriebssysteme*. Deutschland: Oldenbourg Verlag, 6. Aufl.

Carrier, B. (2005). *File System Forensic Analysis*. Addison Wesley Professional.

Choi, M., Park, H., & Jeon, J. (2008). Design and Implementation of a FAT File System for Reduced Cluster Switching Overhead. *Multimedia and Ubiquitous Engineering*.

Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley.

com! professional (2013). Dateisysteme NTFS, FAT & Co. `http://www.com-magazin.de/bilderstrecke/dateisysteme-ntfs-fat-co.-121101.html`. [Online; zugegriffen 29.12.2015].

Eßer, H.-G. (2015). *Design, Implementation and Evaluation of the ULIX Teaching Operating System*. Ph.D. thesis, FAU Erlangen-Nürnberg.
URL `https://opus4.kobv.de/opus4-fau/frontdoor/index/index/docId/6554`

Eßer, H.-G., & Freiling, F. C. (2014). *The Design and Implementation of the ULIX Operating System*. Unveröffentlichte Fassung.

FreeBSD (2006). Man page fs, inode. `https://www.freebsd.org/cgi/man.cgi?query=inode&manpath=FreeBSD+8.1-RELEASE+and+Ports`. [Online; zugegriffen 11.11.2015].

Friedman, M., & Pentakalos, O. (2002). *Windows 2000 Performance Guide*. O'REILLY.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). *Design Science in Information Systems Research*. MIS Quarterly, 28.

Hudson, D., Anvin, P., & Hodek, R. (2015). mkfs.msdos(8) – Linux man page. `http://linux.die.net/man/8/mkfs.msdos`. [Online; zugegriffen 22.11.2015].

Kappes, M. (2013). *Netzwerk- und Datensicherheit*. Springer Vieweg, 2. Aufl.

Kleuker, S. (2013). *Grundkurs Software-Engineering mit UML,*. Springer Vieweg, 3. Aufl.

Knuth, D. E. (1984). Literate Programming. *The Computer Journal Vol. 27 No. 2*.

Liggesmeyer, P. (2009). *Software-Qualität*. Spektrum Akademischer Verlag, 2. Aufl.

Microsoft (2000). Microsoft Extensible Firmware Initiative FAT32 File System Specification. Hardware White Paper, `http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc`. [Online; zugegriffen 26.04.2015].

Microsoft (2010a). Description of the exFAT file system driver update package. `https://support.microsoft.com/en-us/kb/955704`. [Online; zugegriffen 29.12.2015].

Microsoft (2010b). FAT File System. `https://technet.microsoft.com/en-us/library/cc938438.aspx`. [Online; zugegriffen 17.12.2015].

Microsoft (2010c). FAT File System (Windows Embedded CE 6.0). `https://msdn.microsoft.com/en-us/library/ee489982(v=winembedded.60).aspx`. [Online; zugegriffen 17.12.2015].

Microsoft (2010d). TFAT Overview. `https://technet.microsoft.com/de-de/sysinternals/aa915463`. [Online; zugegriffen 29.12.2015].

Microsoft (2015a). Detailed Explanation of FAT Boot Sector. `https://support.microsoft.com/en-us/kb/140418`. [Online; zugegriffen 20.12.2015].

Microsoft (2015b). TexFAT Overview (Windows Embedded CE 6.0). `https://msdn.microsoft.com/en-us/library/ee490643(v=winembedded.60).aspx`. [Online; zugegriffen 22.12.2015].

Munegowda, K., Raju, G. T., & Maninkandanraju, V. (2014). Adapting Endurance and Performance Optimization Strategies of ExFAT file system to FAT file system for embedded storage devices. *International Journal of Engineering and Technology*, *6*(1).

Myers, G. J., & Pieper, M. (1995). *Methodisches Testen von Programmen*. Deutschland: Oldenbourg, 5. durchges. Aufl.

Pate, S. D. (2003). *UNIX Filesystems*. Wiley.

Phillips, H. (2008). *New Perspectives on Microsoft Windows Vista for Power Users*. Cengage Learning.

Ramsey, N. (1994). Literate Programming Simplified. *IEEE Software Vol. 11 No. 5*.

Ritchie, D. M., & Thompson, K. (1974). *The UNIX Time-sharing System. Commun. ACM,*.

Saltzer, J. H., & Kaashoek, M. F. (2009). *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, .

Silberschatz, A., Galvin, P. B., & Gagne, G. (2008). *Operating System Concepts*. Wiley, 8. Aufl.

Smith, R. E. (2015). *Elementary Information Security*. Jones and Bartlett Learning, 2. Aufl.

Tanenbaum, A. S. (2009). *Moderne Betriebssysteme*. Deutschland: Pearson Studium, 3. Aufl.

vom Brocke, J., Simons, A., Niehaves, B., Riemer, K., Plattfaut, R., & Cleven, A. (2009). *Reconstructing the Giant: On the Importance of Rigour in Documenting the Literature*. MIS Quarterly, 28.

Webster, J., & Watson, R. (2002). Analyzing the Past to Prepare for the Future: Writing a Literature Review. *MIS Quarterly*, *26*(2).

Wikipedia (2015). File Allocation Table. `https://de.wikipedia.org/w/index.php?title=File_Allocation_Table&oldid=149057435`. [Online; zugegriffen 31.12.2015].

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde / Prüfungsstelle vorgelegen hat. Ich erkläre mich damit einverstanden/nicht einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

München, den 15. Januar 2016