Sandra Fuchs, BSc

Design and Analysis of a Transaction based Smartcard File System

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Masters degree programme: Software Development and Business Management

submitted to

Graz University of Technology

Supervisors

Ass.Prof. Dipl.-Ing. Dr.techn. Christian Steger
Institute for Technical Informatics
Head of Institute: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Dipl.-Ing. Martin Kaufmann, NXP Semiconductors Austria GmbH

Graz, May 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

....
date signature

Abstract

As smartcards often contain and exchange sensitive information, the storage of this data in the underlying file system and the related access control mechanisms are extremely important in order to guarantee information security. The goal of this thesis is to design a transaction based smartcard file system which allows to re-use memory.
In order to accomplish this, long-known file systems and memory management techniques as well as state-of-the-art approaches are investigated and compared. As common flash file systems are conceptualized for large memories, much adoptions on the system structure are necessary but anyhow their usage within smart cards turns out to be improperly and not efficient at all. Closer investigations show that structures based on a file allocation table or on the basic functionality of the *Extended File System* are suitable for adaptation for a smart card.

Several different file system concepts are developed and their efficiency in terms of performance and memory usage is evaluated and compared for the purpose of finding the most suitable one. The usage of reference transactions and careful comparisons of pro and contra arguments finally leads up to a FAT based smart card file system design in combination with a journaling mechanism being the most expedient one.

Keywords: smart card file system (SCFS), file system, card operating system (COS), non-volatile memory, memory management, journaling, EXT, FAT, YAFFS

Kurzfassung

Da Chipkarten sehr häufig sensible Informationen enthalten sowie diese mit Kartenlesern austauschen, ist die Art des Speicherns der Daten im Dateisystem auf der Karte sowie der dazugehörige Datenzugriffsmechanismus essenziell, um die Sicherheit der Daten zu gewährleisten. Das Ziel dieser Diplomarbeit ist der Entwurf eines transaktionsbasierten Dateisystems für Chipkarten, welches die Wiederverwendung von Speicher ermöglicht.
Um dies zu erfüllen, wurden mehrere altbekannte Dateisysteme, Speichermanagement-Techniken sowie die neuesten Forschungsergebnisse im Bereich der Chipkarten Dateisysteme untersucht und verglichen. Da bekannte Flash Dateisysteme eher für große Speichermedien konzipiert wurden, sind viele Anpassungen an der generellen Struktur notwendig, um sie für die Verwendung in Chipkarten passend zu machen. Trotzdem stellt sich heraus, dass ihre Anwendung in Chipkarten wenig effizient ist. Weitere Untersuchungen zeigen, dass Strukturen welche auf einem File Allocation Table (Dateizuordnungstabelle) oder auf der grundlegenden Funktionalität des *Extended File Systems* beruhen, passend für die Anforderung einer Chipkarte sind beziehungsweise erfolgreich adaptiert werden können.

Mehrere unterschiedliche Dateisystem-Konzepte werden in dieser Diplomarbeit entworfen und deren Effizienz in Bezug auf Leistungsfähigkeit, Funktionalität sowie Speicherverwendung wird ausgewertet und verglichen, um möglichst gut das am Besten passende Dateisystem heraus zu filtern. Die Verwendung von Referenz-Transaktionen und das sorgfältige Abwägen zwischen Pro- und Contra-Argumenten identifiziert schlussendlich das vorgeschlagene FAT-basierende Chipkarten Dateisystem in Kombination mit Journaling als das vielversprechendste.

Stichwörter: Chipkarten-Dateisystem, Dateisystem, Chipkarten-Betriebssystem, Speichermanagement, nichtflüchtiger Speicher, Journaling, EXT, FAT, YAFFS

Acknowledgements

I would like to express my appreciation and thanks to several people who supported me throughout the last months.

First and foremost, I have to thank my supervisors Martin Kaufmann from NXP Semiconductors Austria GmbH and Christian Steger from Graz University of Technology for their patience, support and guidance. Thank you for your advice, enlightening discussions, constructive criticism and review of draft versions of this thesis.

Second I would like to thank the Institute for Technical Informatics for making the cooperation with the company NXP Semiconductors Austria GmbH and therefore the realization of this thesis possible.

My thanks also go to my dear colleagues for their patience, help and efforts in providing useful answers to my countless questions.

Finally I owe special gratitude to my whole family for their continuous and unconditional support. I want to thank my parents, Christiane and Peter, for their confidence, encouragement and aid over the years of my academic studies. They both gave me motivation during tiring times and believed in me when I had doubts. Also my other half, Martin, encouraged and supported me and showed seemingly endless patience when discussing new ideas. Thank you for everything.

Sandra Fuchs

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The first chapter of this thesis is designated to provide an overview of the content of this document and to substantiate the motivation for discussing and examining the topic *Design of a Transaction based Smartcard File System.*

This thesis was written in cooperation with the company *NXP Semiconductors Austria GmbH.*

1.1 Motivation

The wish for a new file system design emerged out of the limitations on commonly used file systems, i.e. transaction based together with re-use of deleted memory. A linked list file system is taken as reference file system for this thesis. The two main components of this kind of file system are applications and files.

The problem of the reference solution is that a kind of garbage collection mechanism is not available and also would be very complex to realize.

Within the reference file system each file needs a shadow file connected to it in order to guarantee data safety and consistency during updating transactions. This means that twice the size of the file needs to be reserved in the memory to backup the whole file data. Finding a different kind of data backup mechanism which consumes less memory is desirable for a new, future design.

File or application deletion also is a very critical point. In case of a deletion, the memory is actually not erased and released but only marked as unused. Currently these unused memory blocks can not be reused and the space is lost.
This wasted space shall be eliminated in a new file system design. An effective mechanism which enables memory re-usage after file or application deletion needs to be introduced.

1.2 Objectives

The requirements for a new designed file system cover multiple points. First of all the basic functionalities, namely creation and deletion as well as read, write and update operations applied on files and applications need to be possible.
As smart cards operate transaction-based, a built-in transaction mechanism is desirable.

A big requirement is the power-fail-safe design of the file system. The system must remain in a consistent state all the time. Therefore a kind of backup management or mechanism has to be introduced. In case of a transaction abort it needs to be possible to roll back the changes and to return to the last stable file system state and consequently keep the file system in a working state.
Such a backup-oriented design is enormously important for a smart card‘s file system as a sudden power loss can happen anytime.

Memory reuse after file or application deletion is an additional extremely important aspect and the main goal of this thesis. The data management on the card shall be realized in a way that also free memory regions can be re-used effectively.

The new file system should save memory compared to the actual solution, if possible. This could probably be achieved through omitting the usage of shadow images and introducing a new data backup mechanism.

Another requirement is that the performance all in all stays the same or improves, it should definitely not get dramatically worse. Also less or the same number of write operations should be guaranteed by the new design, if possible.

1.3 Structure of the thesis

In this first chapter the motivation, objectives and main goals of the thesis have been described.

The second chapter explains all the basics that can be used to design and implement a new file system. Different methods and possibilities, like how to handle the file system‘s objects and how to do the memory management, are explained.

Chapter three gives an overview of a selection of the most common file systems. Both modern as well as older but well established file systems are quoted and explained in detail. In this chapter the practical usage of some of the basics which have been introduced in chapter two, can be traced. The stated file system designs herein also act as kind of guidelines for the elaborated new designs in the later following chapters.

In the following chapter, the outcomes of the state-of-the-art research are captured and discussed. The main points include memory management standards and different approaches regarding flash file system designs.

The actual proposals for a new file system design can be found in chapter five. Basically four different kinds of file systems are elaborated in detail. These include FAT, journaling, EXT and flash based designs.

A more concrete elaboration of the produced design approaches, with the focus lying on the functioning of concrete operations and transactions and the measurement of consumed write operations, is situated in chapter six.

In chapter seven a comparison of the made investigations and measurement results is carried out. The designed models are compared amongst each other and also to the reference file system with the aid of a reference transaction and general calculations. The goal of this chapter is the finding of the most qualified design solution.

The last chapter recapitulates the main points of this thesis, shortly summarizes the investigations that have been made and conclusions that have been drawn. In the end a short future outlook is given and room for improvement is stated.

Chapter 2

File System Implementation Methods

In this chapter multiple different ways, how a file system can be implemented, are discussed. In order to be able to make adequate design proposals which match the requirements of the new file system, all needed fundamentals and necessary components are explained. An important point when designing a new file system is the organization of directories and files what also includes block management, file allocation and free space management. Also metadata and therefore access rights, permissions and location of file contents need to be managed. To handle this, some different kinds of strategies and approaches are inspected in a more detailed way in the next sections.

First of all the two main components of each file system, the *file* and the *directory*, are outlined in sections 2.1 and 2.2. Following this, the different kinds of allocation methods (e.g. contiguous, linked and indexed allocation) are illustrated in section 2.3. In the last section, 2.4, the free space management techniques are presented.

2.1 File structure

A file is an abstract data type, it is a named collection of related information. It also is a logical storage unit and usually recorded on secondary non-volatile storage. A file is characterized through certain defined attributes, which are required by the underlying file system. Usually a file‘s attributes consist of name, identifier, type, size, access rights, modification times and much more. All the information concerning files is stored in the directory structure in form of directory entries [SGG05].

There are multiple operations which can be performed on a file: creating, writing, reading, deleting, updating. This is just the basic set of operations and can be extended to much more, e.g. a user can get and set various attributes of a file [SGG05].

An important point when designing a new file system is to pay attention on the support and recognition of file types. When a system recognizes the file‘s type, it can act in a certain way and operate on the file in reasonable ways. Depending on the file type, only

dedicated operations can be executed on the file and different kinds of information can solely be stored in a certain type of file [SGG05].

2.2 Directory structure

Directories are needed to keep track of the file system‘s files. Choosing the right directory-allocation and directory-management method can be a difficult decision as it affects efficiency, performance and reliability of the file system [SGG05].

In order to locate and open a file, its corresponding directory entry needs to be found. Therefore the root directory must be located first, what can be done through information blocks like the Superblock or the Master File Table, or either can be found at a predefined fixed position. After the successful location of the root directory, the desired directory can be found by searching through the directory tree. The found directory entry finally gives information concerning the location of the data blocks in the memory [Tan06].

2.2.1 Linear list

A simple realization method of a directory structure is the usage of a linked list containing pointers to data blocks, which can be seen in figure 2.1. This method is really easy to program and simple to understand but brings along one critically disadvantage: searching and finding in a linear list requires a linear search and therefore is time consuming. A possible solution concerning the linear time for searching could be the implementation of a sorted list. Unfortunately this approach complicates creating and deleting files, since the list needs to be updated after every change of directory information [SGG05].
The advanced version of the linked list is the double linked list. It has the advantage that the list can not only be traversed in one direction but in two, so each block contains a pointer to the next and also to the previous data block. This structure is depicted in figure 2.2.

Figure 2.1: Structure of a linked list

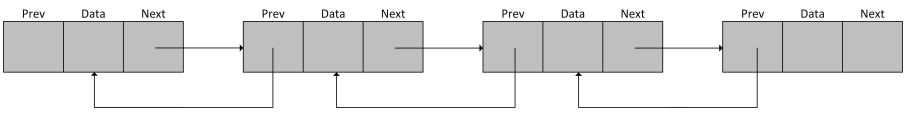

Figure 2.2: Structure of a double linked list

2.2.2 Hash table

Another approach for the implementation of a file directory is a hash table, which uses a linear list as well as a hash data structure. The hash table does the mapping from file name to file location on the disk. It accepts a hash value from a file name and then returns a pointer to the file name in the linear list. No directory search is needed any more and also insertion and deletion work uncomplicated [SGG05]. A picture of the model of a hash table can be seen in figure 2.3.

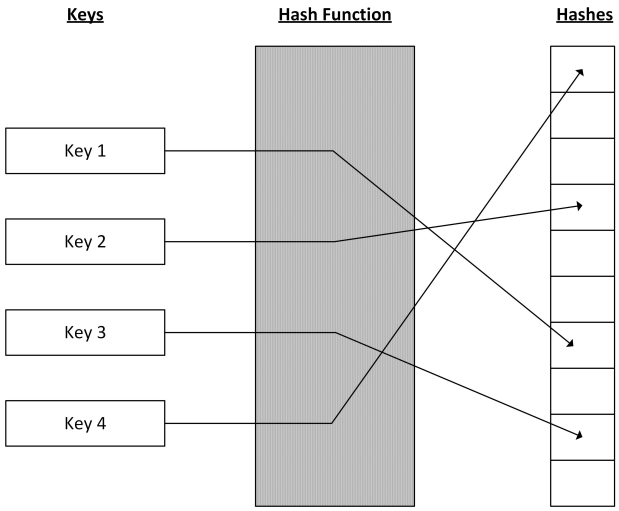

Figure 2.3: Functionality of an hash table

2.3 Allocation methods

An important point when implementing a file system is to allocate space for files, so that the memory is used in the most efficient way, and to keep track of which file is stored in which certain memory block. Three widespread methods of memory allocation are the contiguous, linked and indexed allocation which are described briefly hereafter.

2.3.1 Contiguous allocation

A simple allocation scheme is the contiguous allocation which demands that each file occupies a set of contiguous memory blocks. To find out, where a certain file is located in memory, only the address of the first block and the number of blocks used by the file need to be known. This contiguous placement in memory, which is depicted in figure 2.4, makes the read performance excellent because only one single operation is required. However, the contiguous allocation method has some major drawbacks. One problem is external fragmentation and the other one is finding space for a new file. To be able to reuse space, the maintenance of a free block list which shows the location and size of memory holes is necessary. Additionally to this also the final size of a new file needs to be known in advance in order to find a matching space [Tan06].

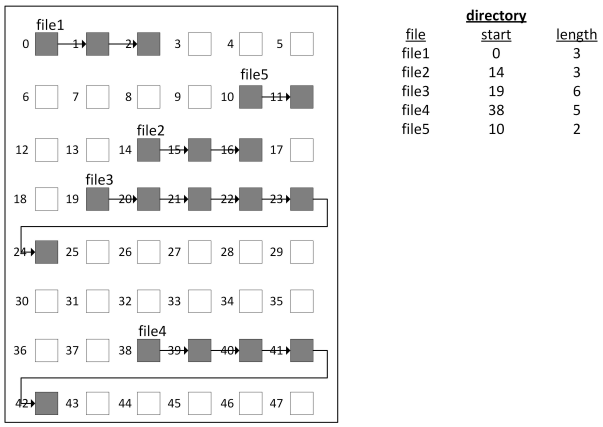

Figure 2.4: Contiguous block allocation

2.3.2 Linked allocation

This method of allocation gets rid of the problems of the contiguous allocation as every single disk block can be used and no space is lost due to external fragmentation. Every file is represented through a linked list of memory blocks. The parent directory contains a pointer to the first and to the last block of the file and each file block again contains a pointer to the next block. The size of the file does not need to be announced before file creation because a file is able to grow as long as free blocks are available. When extending the file, a free block needs to be found and written to and then linked to the end of the file [SGG05]. In figure 2.5 this allocation method can be considered closely.

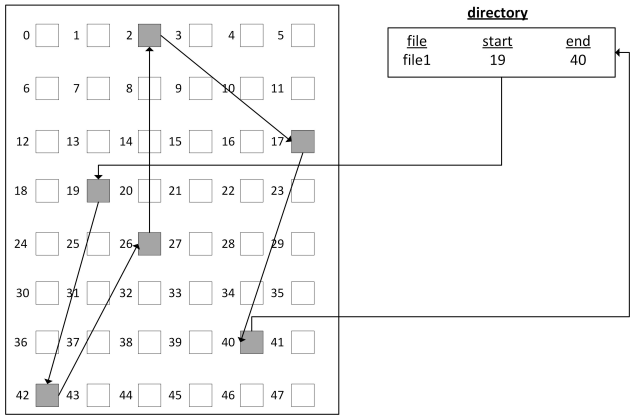

Figure 2.5: Linked allocation

A well-known and popular variation on linked allocation is a file allocation table (FAT). For the realization of a FAT a small section at the beginning of the volume is kept free in order to store the table. This table is set up in an increasing order, containing an entry for each memory block and a corresponding index [SGG05].

Basically a FAT works similar to a linked list. In order to find the location of a file, the directory entry gives information about the starting block number of the file and the corresponding FAT entry contains the block number of the next block of the file. This chain of blocks continues until one FAT entry contains a special value like NULL or NIL, what signals the end of the file [SGG05]. The schema of a FAT is illustrated in figure 2.6. Free block management is made easy with a FAT, because all free blocks can be identified quickly by their containing value (0x00).

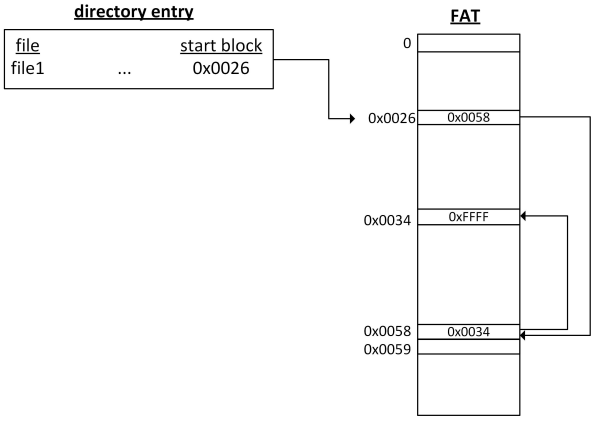

Figure 2.6: Linked allocation with a FAT

2.3.3 Indexed allocation

As linked allocation (without using a FAT) can not support efficient direct block access, another method which solves this problem is needed, namely the indexed allocation (portrayed in figure 2.7). This method uses one special structure: the index block, also called index node or INode. Each file has its own index block, which is a list of block addresses [SGG05].
At file creation all entries in the index block are set to NULL and only if a block is written, its address is placed in the corresponding entry in the index block. Direct access is supported out of the box, without the risk to suffer from external fragmentation. Unfortunately indexed allocation has a disadvantage concerning wasted space, because the pointer overhead is greater than the pointer overhead of a linked allocation scheme. Even if only one pointer is needed for a file a whole index block needs to be allocated [SGG05].

A variation of this method is the usage of multiple indirect blocks. Therefore only a limited number of pointers is kept in the file‘s INode (e.g. sixteen pointers per INode). The INode structure is also used by the EXT2 file system and therefore will be explained in detail in chapter 3.

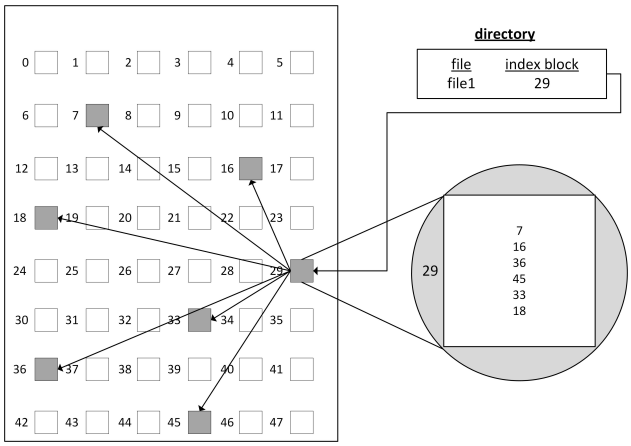

Figure 2.7: Indexed allocation

2.4 Free Space management

Memory management is a major concern because memory space is limited and there is a need to reuse space from deleted files. Most of the popular file systems split the whole memory up into blocks of a fixed size which are then available for directory and file storage.

When splitting up the memory into fix-sized blocks, an important decision concerning the size of the single allocation unit needs to be made. A large allocation unit means that every file (even a one byte file) reserves the whole block size of memory, whereas a small allocation unit means that big files will consist of really many blocks, and reading so much blocks will slow down the whole file system [Tan06].

In order to keep track of the free memory space respectively the free blocks, the file system needs to maintain some sort of structure which reflects the actual memory usage. The most common used techniques are the linked list and the bit vector.
The linked list simply links together all the free memory blocks, with each free block containing a pointer to the next free block. The free list is used from the head of the list, what means that if a free block is needed, simply the first block in the free list is taken [SGG05].

Another very often used management technique is the bit vector (or bitmap). In a bitmap each block is represented by one bit. This means that a volume with n blocks also requires a bitmap with n bits. If a block is free, the corresponding bit is set to 1, otherwise it is set to 0. Maintaining a bitmap requires only little space, is very simple to realize and efficient in usage [Tan06].

Methods which can also be used but are less popular, and therefore not specified in this thesis, include Grouping and Counting.

Chapter 3

Common File Systems

A file system is an important aspect of an operating system, as it provides the possibility for storage and access to data as well as programs of the operating system. It resides permanently on secondary storage, which is intended to store a large amount of data. Usually a file system is built up of two main parts namely a directory structure, which organizes files and gives information about them, and a file structure, which stores data and information. Because file systems are existent on many different kinds of devices and all of them have some special needs, file systems need to be optimized for many different platforms [SGG05].

This chapter gives an overview of the most common file systems nowadays. In the first section the functionality of the often used journaling file system is introduced. Following this, popular disk file systems like EXT, FAT12/16/32 and NTFS are described. In section 3.3 a selection of common flash file systems including YAFFS and JFFS is explained in detail.

3.1 Journaling file systems

In the first section a very often used kind of file system is described in detail. A journaling file system is a kind of file system which writes information about upcoming updates and changes to a journal before committing them. The journal usually is a circular log or a kind of buffer which contains log structured entries and is placed in the near of the tail of the file system. Journaling enables fast file system recovery after a crash and does not need scan-based recovery (e.g. via fsck), which is relatively slow. In comparison to non-journaled file systems, journaling file systems provide improved structural consistency, faster restart times, high-performance, better recovery mechanisms and generally come along with the features of a more advanced file system [Mag02].

Basically a journaling file system does not write upcoming data changes directly in the working file system structure, but stores the changes which are made during a transaction in the separated journal. Not until after a commit happens and the transaction was completed successfully, the changes are transferred to the file system. This mechanism guarantees that the file system is in a consistent state at every time.

The most popular journaling file systems are JFS, XFS, ReiserFS and EXT3. The detailed mode of operation of ReiserFS and EXT3 can be read in section 3.2.

3.2 Disk file systems

This section is dedicated to the most common secondary storage medium, the disk. Two important aspects make the disk to a useful and expedient medium for information storage: on the one hand a disk can be rewritten in place and on the other hand a disk can directly access every block of information at every time [SGG05].
A disk file system contains several kinds of information, like the number of total data blocks, the number of free blocks, the directory structure, file information and of course how to boot the stored operating system [SGG05].
Subsequent some of the nowadays most used disk file systems are presented and detailed.

3.2.1 EXT - Extended File System

The Extended File System with all its successors is the flagship of Linux and was especially designed for the Linux kernel. The first version, EXT, was the very first implementation that used the virtual file system and was soon superseded by EXT2.
EXT2 splits the disk up into blocks and groups them into block groups, which are simply contiguous groups of blocks. Each block group has the same size and consists of a copy of the superblock and a block group descriptor table, a block bitmap, an INode bitmap, an INode table and data blocks.

The superblock is a vital block as it contains necessary information about the file systems size, configuration and layout. It is read at mounting of the file system and is crucial for the whole boot process. Each block group is represented via a certain data structure, the group descriptor. It stores the location of the block bitmap, inode bitmap and the start of the inode table for every block group. And these, in turn, are stored in a block group descriptor table [Wan11].
In each block group the block bitmap provides information concerning the state of a block within that block group. Each bit of it represents one block in the block group. The INode bitmap works similar as the block bitmap. With each bit it represents the status of an INode in a block group [Poi11].

The INode (index node) is a certain structure which represents the EXT2 file system objects. An INode represents a single physical file, what means that each file or directory on the disk is associated with exactly one INode. The different types of physical files are: directory, regular file, symbolic link and special file. The INode structure contains pointers to the file system blocks which contain the actual data held in the object and also all of the objects metadata. The metadata of an object includes permissions, owner, flags, size, number of used blocks, modification times and some more essential attributes [Poi11].

As already said, each INode points to the location of its data on disk. Therefore one INode contains fifteen pointers to data blocks. The first twelve pointers point to direct blocks, what means that the file‘s data can be referenced directly and accessed quickly.

The thirteenth pointer points to an indirect block. This indirect block again contains pointers which point to data blocks. The fourteenth pointer points to a doubly-indirect block, which contains pointers to indirect blocks, and the fifteenth pointer points to a triple-indirect block, which contains pointers to double-indirect blocks. To get a better idea of this concept, the INode structure is depicted in figure 3.1. All these single INodes are stored in an INode table, which is available for every block group [Poi11].

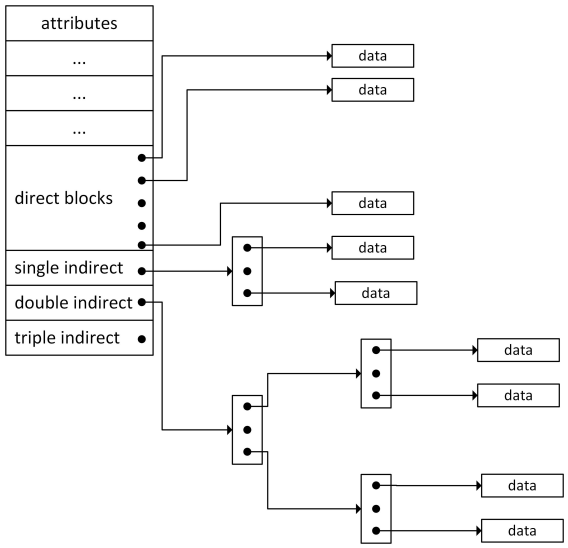

Figure 3.1: Structure of an INode

One disadvantage of the EXT2 file system is the enormously long recovery time after a crash. To get control of this issue, the new file system version EXT3 was developed on the basis of a journaling scheme. Journaling achieves fast file system recovery because at all times data which is potentially inconsistent has to be recorded in the journal. Resulting from this consequent recording, recovery can be obtained by scanning the journal and copying back all committed data into the main file system area [Twe98].

The journal is stored on the disk in form of a unique numbered INode and keeps record of three different types of data blocks: metadata, descriptor and header blocks. During the execution of transactions the journal has to record the content of the file system‘s new metadata blocks. After a successful commit of a transaction, the new file system blocks are located in the journal itself but nothing has been changed in the main file system and the blocks have not been synced to their actual location on disk. The old original blocks on disk need to keep unchanged until a commit of the total journal is performed. Once the journal has been committed, the data blocks in the journal can be written back to disk and the old disks version is not relevant any more [Twe98].

The EXT3 file system design offers some significant advantages compared to its predecessor. Availability as well as reliability of the file system are increased and of course crash recovery can be performed much quicker and should not cause much performance delay any more [Twe98].

3.2.2 FAT - File Allocation Table

The file system based on a FAT (a form of an index table) was implemented by Microsoft in three different types: FAT12, FAT16, FAT32. The different numbers in the names of the file system (12, 16, 32) are also reflected in the bit size of the entries in the FAT structure on the disk. FAT12 was initially developed as a file system for floppy drives, but in order to overcome the limitations of address length and volume size, soon FAT16 and FAT32 were released [Bha07]. A graphic which compares the three different FAT file systems can be inspected in figure 3.2.

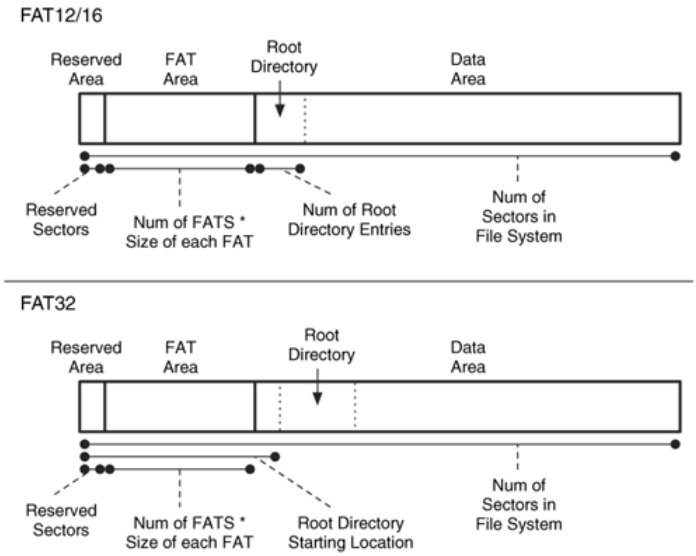

Figure 3.2: Layout of the three different FAT based file system types

Nowadays FAT is primarily used for removable media, such as floppy disks, flash memory, USB drives and digital cameras but is nearly not used on hard drives any more.

A FAT is a table containing entries for each cluster of the volume. Each entry of the FAT gives one of the following information about the data blocks and their usage:

- Cluster is free
- Volume is damaged at the clusters position
- Cluster is in use: Number of next cluster is given
- Cluster is in use: It is the last cluster in the chain

A FAT file system is structured into four basic regions [Cor05]:

- Reserved Region
- FAT Region (location of FAT and FAT copy)
- Root Directory Region (not existing on FAT32)
- File and Directory Data Region

In the reserved region, also called boot sector, the BPB (Bios Parameter Block) is located. The following sector is a very important data structure namely the FAT itself. A FAT basically defines a linked list of the clusters of a file and maps the data region of the volume by cluster number, with the first data cluster starting at cluster number two [Cor00].

A FAT directory simply is a regular file, which is characterized through a special attribute which indicates the directory type. The content stored in a directory can be other files as well as sub-directories. These directory contents are called directory entries and are data series with a size of 32 bytes. A directory entry is associated with one file and stores the cluster number of the first cluster of the file [Cor05]. An image of some FAT directory entries can be seen in figure 3.3.

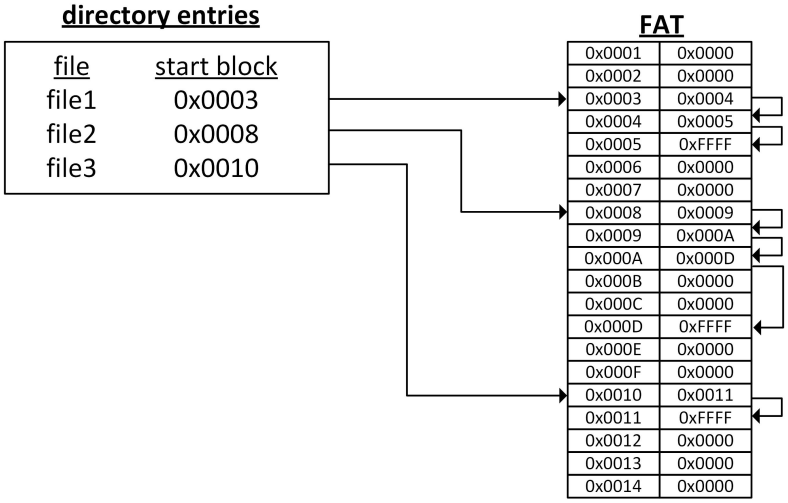

Figure 3.3: Structure of FAT directory entries

The root directory is a special directory and created during the initialization of the volume. It is present on every formatted volume and immediately follows the last file allocation table [Cor05].

The free clusters of the file system are managed through a list of all clusters containing the value 0x00 in their FAT cluster entry. This list of free clusters is not sorted and needs to be computed at every mounting of the volume.

3.2.3 NTFS - New Technology File System

NTFS is the default file system of the Windows operating system and was developed by Microsoft. It includes advanced features which can not be found in a FAT file system and was designed with focus on performance, reliability and security. Log files and checkpointing are two techniques used to restore the file system consistency after a crash or sudden power loss. NTFS supports large volumes and manages disk space efficiently by increasing the cluster size. Another benefit is the support of the Encrypting File System (EFS) technology which can be used to store encrypted files on the NTFS volume.
NTFS is the preferred file system for hard disks but can not be used on removable media. In order to format floppy disks or flash media, FAT file systems are used [Cor03].

The file system splits the disk up into clusters, with one cluster being the smallest amount of disk space that can be allocated to store a file. Clusters are numbered sequentially from the beginning of the partition, starting at cluster number zero, and each cluster again is partitioned into sectors. As the cluster size can be chosen when formatting a volume, every file system has a different maximum number of clusters it can support. The smaller the cluster size is, the more efficient the disk can be managed and store information. The supported cluster sizes range from 512 bytes to 4 kilobytes [Cor03].

A NTFS file system is structured into four basic regions:

- NTFS Boot Sector
- Master File Table (MFT)
- File System Data
- Copy of the Master File Table

The first sixteen sectors of a NTFS volume are allocated for the boot sector and the bootstrap code.

In the Master File Table all the necessary information for file retrieval is stored. It is a record table containing one entry (consisting of the file‘s name and attributes) for every file on the NTFS volume, including the MFT itself. In figure 3.4 the structure of the MFT can be regarded.
As the MFT stores information about itself, the first sixteen records of it are reserved for metadata files. These metadata files are vital for the correct functionality of the file system, with each of it being responsible for a different area. Emerging of the importance of the MFT, a backup copy of it is stored somewhere else on the disk, which can be read in case of corruption of the original MFT. Both data segment locations (for the MFT and the backup MFT) are stored in the boot sector [Cor03].

Each file of the file system typically is represented through one file record in the MFT. Only if a file has a very large number of attributes it maybe needs more than one record. Folder records store index information. Small folder records are entirely stored within the

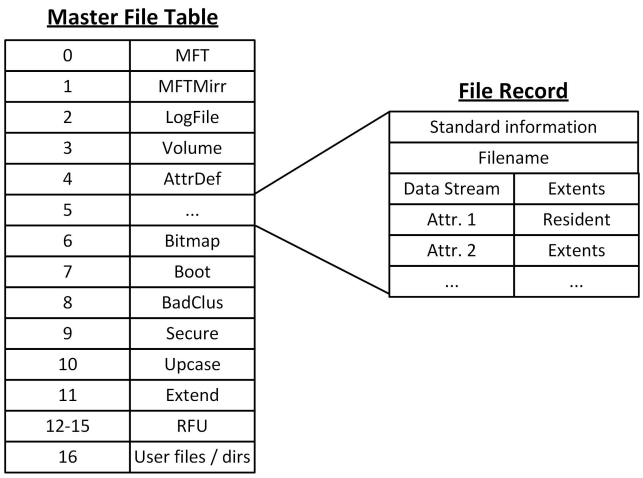

Figure 3.4: Architecture of the Master File Table in NTFS

MFT structure whereas large folders are organized as B-tree structure. For folders which are arranged as B-tree, the MFT contains records with pointers to external clusters which contain the actual folder entries. The B-tree structure offers a big benefit: when searching a particular file, NTFS outperforms FAT, because in large folders FAT must scan through all file names [Cor03].

3.2.4 ReiserFS

ReiserFS was developed for Linux by Hans Reiser and his company Namesys. It is a journaling file system and uses balanced trees (also called B+ tree) for file and directory storage [Buc03].

A ReiserFS volume is divided into memory blocks of a fixed size, which are ordered sequentially. The first file system block is the superblock. It contains important information like the block size, block numbers as well as journal and node information [Buc03].
After the superblock, a bitmap of free and used blocks follows, and subsequently the journal is situated. The size of the bitmap is relative to the block size of the file system [Buc03].

The whole Reiser file system is built up of a balanced tree composed of internal and leaf nodes, with each node being a memory block. Each file, directory or other item of the ReiserFS is associated with a unique key. This key can be compared to an index or INode number in other file systems [Buc03].
Keys in the ReiserFS are used to uniquely identify items, to locate them in the tree and also to keep track of item groupings (all items which belong to the same directory are grouped together). Each key is made up of four objects: a directory ID, object ID, object offset and type [Buc03].
Internal nodes consist of a block header, keys and pointers to child nodes, whereas leaf nodes are located at the lowest level of the tree and include data within them. They are made up of a block header, item headers and items. Items finally contain all the actual

data and are categorized into one of four different types: stat, directory, direct and indirect items. Files are made up of direct or indirect items [Buc03].

The ReiserFS journal is a continuous set of disk blocks and keeps track of transactions which are made to the file system. Before performing any changes to the file system directly, the belonging transactions are written into the journal. Only after a positive commit and a successful realization of all logged transactions, the journal can be flushed. The journal is a circular log and consists of a journal header and transactions of a variable length [Buc03].

A transaction describes an upcoming file system change in detail and writes all necessary information into the journal instead of directly modifying file system blocks. One log entry is built up of a transaction description block, which represents the type of transaction, a list of blocks affected by the transaction and a commit block, which signals the termination of a transaction [Buc03].

3.2.5 BTRFS - B-Tree File System

BTRFS is a Linux file system which is based on copy-on-write and uses B-trees as underlying data structure. Checksumming is used for integrity and reference counting for space reclamation. Instead of using memory blocks, BTRFS manages extents (a contiguous page-aligned on-disk area), which eliminates the need for a special block size [Rod13].

The B-tree structure of BTRFS is generic and only knows about certain kinds of data structures, namely keys, items and block headers. Block headers have a fixed size and keep track of important information fields like checksum, flags, and IDs. Internal tree nodes hold key pairs (in form of a block pointer) whereas leaf nodes hold an array of items (data pairs). Item data can have a variable size and type. Several items are usually grouped together to an object. The file system is made up of objects and its contained items are logically arranged in the B-tree [Rod13].

Small files which only need the space of one leaf block can be directly placed into the B-tree. Larger files are stored in the file system‘s extents, which only hold the user data without needing additional header blocks or formatting. An extent maps from a logical area in a file to a physical area in memory. When storing a file in a few large extents, then a file read can be efficiently performed with only a few disk operations, what is a very big benefit of the usage of extents [Rod13].
A directory contains multiple directory items, more precisely said two sorted lists of them. These two lists are used for path look-up and also for bulk directory operations. The directory item is an element, which contains a file name and a key [Rod13].

3.3 Flash file systems

Nowadays flash memory is often used in embedded systems and devices because it brings along many benefits in terms of data density, I/O performance and power consumption. Flash memories can be classified into two main categories: NOR flash, which is most used

for code execution, and NAND flash, which is commonly used for data storage and the main basic module for secondary storage systems in the embedded sector. Unfortunately NAND flash memories bring along some drawbacks: the memory cells have a limited lifetime, in-place data updates are not possible and erase/write operations are asymmetric. All these drawbacks bring along the need for a certain management of the flash memory, which can be achieved through special flash file systems [OBS12].

Data of a NAND flash memory is organized in a hierarchical way. The memory is divided into blocks and each block again is divided into pages, which contain the actual user data and metadata. The three key operations of flash memory are read, write (performed at page level) and erase (performed at block level). A flash file system has to get along with all the above mentioned drawbacks of flash memory, has to provide a sort of garbage collection mechanism, has to care about wear levelling and of course needs to introduce a bad block management mechanism [OBS12].

To get an insight into this topic, some widely used flash file systems are listed below and are explained in detail.

3.3.1 JFFS

JFFS (the Journaling Flash File System) is a log-structured file system, intended to be used on flash devices in embedded systems [Woo01]. It structures the memory as a circular log and therefore enforces wear levelling. Occurring changes to files and directories are written to the tail of the log in the form of a node. A node consists of a header, containing metadata, and file data. Nodes are chained together via pointers and can change their status to either be valid or obsolete.

The successor of JFFS, namely JFFS2, splits the memory into blocks and these blocks may be filled with nodes from bottom up, there is no circular log any more. In JFFS2 there is also a distinction between different types of nodes made. The three node types include the INode, the dirent-node and the cleanmaker-node. The INode contains all the metadata as well as a range of data belonging to the INode. The dirent-node contains a directory number, an INode number and a name. It either represents a directory entry or a link to an INode. The cleanmaker-node is used to show the success of an erase operation and is written to an erased block.

3.3.2 TFAT

The Transaction-Safe FAT File System (TFAT) is a file system designed by Microsoft in order to provide transaction-safety and a consistent file system without corruption in case of power loss or sudden removal of a storage device. The file system is available in three different versions: TFAT12, TFAT16 and TFAT32.

TFAT works with two copies of the file allocation table, a FAT0 table which represents a stable copy of the last known good FAT, and a FAT1 table in which current operations are recorded [Cor10]. In case of a transaction failure, the disk is set to the same state as it

has been before the transaction started, whereas in case of a successful completion of all transactions, the FAT1 table is copied to the FAT0 table and the file system reaches a new valid state. When modifying an existing file, a new cluster is allocated for the modified bits and the FAT is updated to include this new cluster. This mechanism is necessary to ensure the consistency of the original file, if a transaction fails to finish the file modification respectively to complete successfully [Cor10].

3.3.3 YAFFS

YAFFS (Yet Another Flash File System) is a robust open-source file system specifically designed to be high performing and suitable for the embedded use with flash memory (NAND and NOR flash). It is a log-structured file system, provides wear levelling and can be easily ported.
The advancement of the original version is called YAFFS2. Its concept is similar to YAFFS and shares most of the code. The benefit of YAFF2 is the support of the new NAND flash with 2 kilobytes page size instead of the old 512 bytes sized pages [Ltd02].

YAFF2 is a true log structured file system, meaning that it only writes sequentially without using deletion markers. The log entries either are a data chunk or an object header and each chunk again has a tag associated with it. The tag gives important information regarding the dependency of chunk and object, the position of a chunk within a file and the currency of the chunk. No file allocation tables or other mapping structures are needed by the YAFFS2 file system. This fact makes the system robust and additionally reduces the writing and erasing operations [Ltd02].

In order to work with the newer NAND types, YAFFS2 has to fulfill the objectives of zero overwrite and sequential chunk writing, and therefore has to make use of alternate mechanisms like the usage of the sequence number and the so-called shrink header marker [Ltd02].

All objects in the file system (files, directories, hard links, symbolic links and special objects) are represented in form of a *yaffs-object*. The main function of such an object is the storage of metadata and type-specific information. Therefore *yaffs-objects* store the object ID, the type, a pointer to the parent, permissions, access rights and other attributes. The YAFFS2 directory structure is built up by a tree of directory objects. This structure is necessary to quickly access a file system object by name and to perform operations on it. Each *yaffs-object* has a double linked list node to maintain its siblings inside a directory. Additionally to that, directory objects have a double linked list node to maintain their children (meaning all contained files and sub-directories) [Ltd02].

The file objects in the YAFFS2 file system are characterized through the following main values: file size, depth of the TNode tree, pointer to the top of the TNode tree. Each single file has one TNode tree in order to map the file position to the actual memory address. The TNodes are organized in levels and either point to other TNodes in a level beyond (takes effect for TNodes of level one and above) or directly to the data block‘s location in NAND memory (takes effect for TNodes of level zero) [Ltd02].

YAFFS2 keeps track of every memory chunk and its state in order to operate on it in a meaningful way. When a new free memory chunk is needed, it needs to be allocated from the so called allocation block. This block serves as available memory pool, whereof the chunks are allocated sequentially. In case this block is fully allocated, a new empty block is chosen to become the allocation block. As a fresh allocation block is defined after every power loss, this allocation mechanism slightly increases the amount of produced garbage but definitely improves robustness [Ltd02].

Chapter 4

State-of-the-Art in Smartcard File Systems

This chapter provides information regarding the current state of research in the field of smartcard file systems. Relevant approaches and ideas have been examined and are presented in the following sections.
Unfortunately this specific topic does not offer really much information and nearly no concrete design proposals are available at all. It was not possible to find any concrete guidelines or existing smartcard file system designs.
In order to anyhow give an overview of the topic range, related fields of research and comparable systems have been investigated.

The next few examples concentrate mostly on flash file systems respectively the memory organization and usage of EEPROM. The techniques and approaches which are used for organizing and maintaining flash memory and EEPROM can be taken as guiding principle and adapted to also be suitable for a tiny smartcard file system implementation.

4.1 ISO/IEC memory organization

According to the ISO/IEC 7816-4 technical reference, two main structures for applications and data are specified - the dedicated file *DF* and the elementary file *EF*. A dedicated file hosts applications, groups files and stores data objects. It can also be the parent of other files, which are situated directly under the DF. An elementary file stores data and can not be the parent of another file. An internal EF stores data which is interpreted by the card, and a working EF stores data which is not interpreted by the card [fSEC05b].

The logical organization of dedicated and elementary files can be done in two ways. On the one hand a hierarchical organization with a master file *MF*, and on the other hand a parallel organization without any apparent hierarchy of dedicated files, can be realized [fSEC05b].

4.2 Non-volatile memory management

In the paper *Non-volatile memory management methods based on a file system*, written by Shuichi Oikawa, several methods which enable the integration of the main memory and file system management for NV memory are presented. Three methods, namely *direct*, *indirect* and *mmap* are described in detail. The direct method directly utilizes the free blocks of a file system by manipulating its management data structures. The indirect method indirectly allocates blocks through a file that was created in advance and is dedicated for the use of main memory. The mmap method uses the mmap system call for block allocation though memory mapped files [Oik14]. The content of the paper is not directly relevant to the topic *smart card file system* as it does not explain anything about file system implementation directly. However, the mapping techniques and memory management methods introduced in the paper, give an overview of how memory organization could be implemented.

First of all the author examines some of the upcoming non-volatile memories, which enable high performance along with persistent data storage without the need of a power supply, like PCM, MRAM and ST-RAM. The major memory devices are currently DRAM and flash memory, which use electrical charge to memorize binary information. NV memory is considered as a candidate to replace or to be used along with DRAM and flash memory in order to get rid of problems like leakage or other power consumption problems. These new NV memory devices can maintain data without power supply and use resistance values instead of electrical charge [Oik14].

The main goal of the paper is the combination of the memory allocator which is used for DRAM and the file system which is utilized in NV memory to a big, virtual memory system and the creation of a link between DRAM and NV memory.
For this integrated memory management, the three already mentioned methods can be used to provide the linking between memory allocator and file system [Oik14].

4.3 Various examples of flash and smart card file system designs and proposals

As already mentioned, the research findings did not include any specific design proposal for a smart card file system. Therefore the next pages contain information about flash and other non-volatile file systems as well as the current progress in this area.

4.3.1 RIFFS - Reverse Indirect Flash File System

First of all I would like to present *RIFFS*, the *Reverse Indirect Flash File System*. RIFFS is a file system for flash memories with the focus on the characteristics and needs of this kind of memory. To overcome the limitations of flash, data is stored inside of the proper file and files are managed via a reverse-tree. RIFFS is designed as a competitor to the well-known and often used JFFS2 file system and also obtains better results in a direct comparison [PFm04].

The RIFFS project creates a special structure called file context, so altogether there are three structures within the file management: the *file*, the *file context* and the *logical data blocks*. The file context is responsible for adding control information to the file. It is implemented as a block inside the flash and contains among others a reference to the father context, which is necessary for the management of directories, and file attributes [PFm04].

Each logical block of a file contains a version that identifies the various parts of a file. When sorting it in ascending order, the organized data of the file can be accessed. This makes it possible to write the file blocks randomly to the memory and nevertheless guarantee its reconstruction at the start of the file system [PFm04].

The directory management of RIFFS is done through a *reversed linked tree*. The structures belonging to this tree contain all the information concerning itself, eliminating direct references in the system. The navigability of this structure is not adequate for a file system and therefore it is necessary to construct it in the RAM memory. A directory is implemented as a file, containing all its attributes. This way, each directory has a reference to its father, what successively characterizes a reverse tree [PFm04].

4.3.2 FRASH - Hierarchical File System for FRAM and Flash

Another approach, namely *FRASH*, is designed for byte-addressable NVRAM and NAND flash devices to enhance the efficiency of the file system in various aspects and also aims at exploiting physical characteristics of these two different kinds of memories [PFm04].

In FRASH, the *Hierarchical File System for FRAM and Flash*, the NVRAM is mapped into memory address space and contains file system metadata and file metadata. Metadata information is stored in FRAM and data information is stored in the NAND flash region. The consistency between NVRAM and NAND flash is guaranteed through transactions [kKSgJ07].

The objective of the paper "FRASH" is to resolve the file system mount latency issue and the overhead of metadata update while retaining the performance advantage of the log structure based file system. The proposed file system consists of the metadata and the data layer and uses YAFFS as a baseline. The metadata layer contains tag information and an object header. Tags are responsible for ordering the file data in the data layer, whereas the object header corresponds to the INode, which is familiar from Unix file systems, and stores all information regarding a file or directory: name, size, ownership. The tag also connects to an object header and is vital for building up the TNode tree [kKSgJ07].
When a scan operation occurs, the operating system scans the tag and the object header, validates the tag information and decides if the corresponding NAND pages in the data layer are valid or not. If everything is okay, the TNode tree can be created and used for the representation of the order of the file data in the data layer [kKSgJ07].

The FRASH file system designer's main goal is to achieve the best possible performance, combining FRAM and NAND memory. The file system can be directly built up using information in FRAM and the scan phase can be omitted, what has a great impact on speed

and performance. However, as FRASH is built up hierarchical, consistency across data storage can be guaranteed but produces a lot of overhead and therefore the performance goes down again [kKSgJ07].

4.3.3 FSOC - Flash Memory-based File System on Chip

In order to provide interoperability between portable storage devices and hosts, a storage device with embedded file system is presented in [ACL+07]. The paper describes *FSOC*, a *Flash Memory-based File System on Chip*, and focuses mainly on the easy interaction between a host system and a device and how an embedded file system implementation can be conductive on the arising needs.

The emerging benefits of FSOC according to the authors are both qualitative as quantitative. On the one hand, FSOC provides a high degree of interoperability, there is no need for host system developers to implement a file system, and FSOC improves the file system performance through optimizing it for a special purpose. On the other hand, the execution of the file system directly on the storage device offers advantages for the host. Data traffic can be reduced and also the energy consumption of the system goes down due to reduced data traffic [ACL+07].

The designers of this FAT-based file system incorporated three vital requirements into their design. As most important requirement, a quick recovery after power failure was figured out. For this reason, a journaling mechanism was added to the file system.
The second requirement was the efficient execution on low performance processors and the last requirement was the compact size of the code, due to memory limitations [ACL+07].

However, the main goal of the innovators was not situated in the elaboration or invention of a new file system, but the priority lies in improving the efficiency of the storage device and its interaction with the host [ACL+07].

4.3.4 Implementation of a Smart Card Operating System

In the paper *Design and Implementation of Smart Card COS*, the authors directly address the topic of this thesis. They explain what a *COS* (*Card Operating System*) is and which factors need to be considered when focusing on a new design respectively implementation.

Despite the COS architecture, working principle, transmissions and so on, also the file management is discussed shortly. The authors propose to use the ISO/IEC7816-4 logical structure for the smart card file system, which adopts the tree structure with hierarchical data management [YJXL10].
As the smart card file system is not strictly specified and has only the logical form of organization underlying, but no requirement for a specific physical implementation, a high flexibility can be achieved according to the authors [YJXL10].

In order to make a smart card faster, an efficient data structure is needed, as the access time on chip data depends on how the data actually is stored and how complex the

underlying data structure is organized in terms of space and time [JSPK13].

In [JSPK13] multiple data structures which can be applied to a smart card file system are examined and compared. The authors assume the usage of the file organization which has been presented in [YJXL10] and therefore use a differentiation between three types of files, which are the already known *master file*, *dedicated file* and *elementary file*.

Before the actual analysis is done, several types of referencing are introduced, namely file, data and record referencing methods:

- Files can be referenced by the file identifier, by the path (a concatenation of file identifiers, starting from the MF), by the short EF identifier or by the DF name [JSPK13].
- Data can be referenced either in the form of records, data object or data units. The data can be stored either in a single continuous sequence of records, or in a single continuous sequence of data units. For referencing a record or a data unit, the related EF needs to be selected and then any of the already mentioned file referencing methods can be used. After that, a particular record or data unit can be referenced either by the record number or by setting parameter bytes [JSPK13].
- Each and every record of a selected EF can be referenced with the record identifier or with the record number. When a reference is given with respect to the record identifier, an indication will also be there for specifying the logical position of the record [JSPK13].

In order to find the most efficient data structure the authors examine several different ones (array, linked list, double linked list, stack, queue, binary search tree BST, hash, heap) and come to the following conclusion:

- Regarding the *insertion operation*, nearly all structures need $O(1)$ time, respectively $O(n)$ time when the insertion needs to be done after at a specific position inside the structure, whereas BST and heap need $O(\log n)$ time [JSPK13].
- For the *deletion operation*, nearly all structures need $O(1)$ time, respectively $O(n)$ time when the deletion needs to happen at a specific position inside the structure, whereas BST and heap need $O(\log n)$ time [JSPK13].
- For performing the *search operation*, every data structure except the BST takes $O(n)$ time, whereas the BST needs $O(\log n)$ time [JSPK13].

According to [JSPK13] the search operation has been identified as the most vital one. The authors state that the usage time of a smart card is directly proportional to the time which is needed for searching of data that is stored on the smart card, and that the overall performance of operations is dependent on the time taken by the search process and not on the time which is needed for insertion or deletion of a record.

Out of this reasons [JSPK13] suggests the *binary search tree (BST)* as the best suitable data structure for performing smart card operations.

4.4 Conclusion

As already mentioned at the beginning of this chapter, the research in the area surrounding smartcard file systems proved itself to be difficult. It was not possible to get any detailed information regarding the internal design respectively the physical implementation. As also [JSPK13] observes, the logical form of organization and structuring data is specified through the ISO/IEC standard, but the further use of data structures is total flexible for the user. Numerous data structures including arrays, lists, heaps, trees and so on can be freely chosen by the implementer and used according to the special needs of a customized file system.

Although the authors of [JSPK13] suggest the binary search tree as the preferable data structure, they do not support their decision very well in my point of view. It might be the case that a BST fulfills the requirements of their file system implementation at the best, however it is not applicable in order to make an appropriate replacement of the reference file system. The reasons therefore are the extensive use of random access memory, what can not be provided from the card which is used as reference, additionally a high amount of write operations and time intensive re-structuring operations are needed by a BST.

Chapter 5

Design and Conception of potential Smartcard File Systems

This chapter presents some design proposals of potential smartcard file systems. Therefore already existing file systems have been examined in detail in chapter 3, in order to get a good understanding for the basic functionality and different ways of realization of such an important management system. Additionally, the underlying basic structure of multiple existing file systems was adapted in a way that it accomplishes the needs of a modern smart card file system and the results are represented on the following pages.

A smart card file system has to fulfill multiple requirements and has to cope with aggravated conditions like very limited space.
As contactless memory cards support the usage of multiple applications, the file system has to administrate a variable number of applications as well as the corresponding files, whereby files can be of different types and sizes. When designing a file system, the overall memory overhead should be kept as small as possible and the whole memory management needs to be done quite carefully. In case of application or file deletion the allocated memory shall be releasable and reusable.
The file system should also be based on a power-fail-save design, meaning that it has to remain in a consistent state at every time and has to support some kind of backup or rollback mechanism. Overall goals also include fast data access and a small number of needed write operations.

In the first section of this chapter the two approaches of journaling file systems are explained. One of the suggestions uses the well-known standard journaling way of work, namely writing everything into a buffer before actually modifying the last good, working data. The other, *reverse* one was invented for this thesis, and illustrates a slightly modified version.
Following this, two different proposals based on a design using file allocation tables are discussed: a double FAT approach and one proposal based on a FAT in combination with journaling.
The next suggested file system is similar to the extended file system, which is used by Linux and uses INodes as the main management structure.

In the fourth section a design based on the flash file system YAFFS is made. This file system is based on the usage of a log-structure, categorizing the memory blocks into chunks. Finally the tree based file systems are discussed shortly.

5.1 Design decisions (based on the reference file system)

As all of the file system designs which are presented in the following sections are structured in a block oriented way, a common block size is required as well. Depending on the total size of the user memory, a matching block size needs to be figured out. Possible values are for example 32 bytes, 64 bytes, 96 bytes, etc.
As the reference file system, which is discussed in detail in chapter 7, uses a block size of 32 bytes and also to be able to provide a certain degree of comparability, a block size of 32 bytes is chosen as default value for the file system designs within this thesis. (This is only an exemplary assumption and can of course be changed when effectively using one design.)

5.2 Journaling file system design

In this section no file system design is presented directly, but the mode of operation of normal and reverse journaling is explained and analysed in detail. How a journaling file system basically works can be read in chapter 3.

At the beginning of the user memory, a management block is situated. This is followed by the actual memory blocks and the journal region. In a journaling based approach the user can choose how much space he wants to reserve for the journal.

The journal is empty in the beginning and as the rest of the memory split in blocks of 32 bytes. Each time a memory block will be written or modified, an entry in the journal will be made.

Writing data to the journal

The journal does not only store the plain user data but also very important information that is necessary to later on store the data at the right position in the file system.
Writes to the journal are made using journal entries. Each entry contains the block number of the data block which should be modified, the offset in bytes, the length of the data and the actual data.

The exact structure of a journal entry and a short description can be seen in section **??**.

Basically there are two ways of using a journal, which are explained in the next two sections. Both do normally not use any other kind of indexing methods like trees or a FAT, which is the only thing that is used is the journal. When realizing a whole file system without any indexing structure and only using journaling, the data blocks which belong to one data structure (like a file) need to be written sequentially.

This journaling-only approach therefore is not examined any further and only the basic principles of journaling are explained in this chapter, so that they can be reused for a combined file system design.

5.2.1 Normal journaling

In the *normal* mode, everything that is written to the card is not directly placed into the concerned memory blocks in the real file system, but first in the journaling area. This makes it possible, that nothing is changed immediately in the file system and that the new data is made available in the journal and offered for further use.

After writing to the journal and committing the transaction successfully, the new written data is completely available in the journal and can be synced to the actual destined memory blocks.

When the transaction can not be finished or is disrupted in any way, the data in the journal is not completely available and therefore not synced to the memory blocks in the file system. The old state of the file system remains active and is not changed at all. In this case, no rollback or any recovery mechanism is used, because the original file system state stays unmodified.

Procedure after successful commit

When a transaction was finished without disruption, and a commit was executed, the new data can be written from the journal to the actual position in the file system. The old data in the memory blocks in the file system can simply be overwritten, as the newer version is backed up in the journal.

Unfortunately this copy operation takes a lot of time and write effort after the commit action happened. As the commit needs to be fast and the card needs to be ready for new transactions, there is no time available for a complex write operation like this.
Copying it before a new transaction can start or even during the commit operation is impossible and therefore this normal journaling mechanism can not be used for a smartcard file system.

5.2.2 Reverse journaling

In the *reverse* mode, everything that is written to the card is written directly in the file system, into the concerned data block. In order to introduce a security mechanism, the file system is not modified immediately, but the original data which is located in the block that is going to be modified is copied to the journal. After making this backup copy, the destined memory blocks can be overwritten with the new data.
So before overwriting or deleting any data of the working file system, the concerned block or piece of data is copied to the journal in form of a journal entry. After securing the original data, the new data can be directly written into the file system.

When the transaction was successful and the commit action happened, the file system is immediately in the right state. As the data writes were executed directly in the user memory blocks, all data is up to date and the journal is not needed any more. After a successful commit, the journal can be emptied.

In case of a failure during a transaction, everything which was already written into the file system needs to be reversed. After a disruption, the file system is in an inconsistent state and the original data blocks which are located in the journal need to be copied back to the user memory.
This means, that all changes which were made are obsolete. After the whole content of the journal is copied back to its original location, the old, stable state of the file system is re-established and it can be used again. After copying everything back to the right memory blocks, the journal also is not used any more and its content can be deleted.

5.2.3 Comparison of both solutions

The first, *normal* journaling proposal offers a great protection for the working file system. Through first writing all upcoming changes to the journal, no data in the actual user memory blocks is corrupted. However, this solution needs to write all data from the journal to the file system during a commit operation, respectively at the end of a transaction.
Such a big write operation is infeasible and therefore this proposal can not be used.

The second, *reverse* journaling proposal also guarantees the consistency of the file system through saving the original data in the journal.
This idea is especially faster when writing to empty blocks, as not the whole block needs to be written to the journal. Therefore the amount of bytes which need to be written is less in comparison to the other solution.
Another advantage is the actuality of the file system. The new data is directly written to the right block and the old data immediately saved in the journal. This guarantees an up-to-date file system at all times, without any need to copy journal contents at the end of a transaction. Therefore a quick commit operation is ensured.
A little disadvantage of this solution is the rollback mechanism in case of transaction failure. Everything that is saved in the journal needs to be written back to the file system, what again is really time consuming but acceptable.

5.3 Proposals based on the usage of a file allocation table

In this section two possible file system solutions which make use of a file allocation table are presented. The functionality and structure of a FAT is explained in detail in chapter 3, in section 3.2.2.

The first presented solution makes use of two FATs whereas the second solution utilizes one FAT and a journal to keep the data consistent.

5.3.1 File system using two FATs (Double FAT approach)

In this solution the whole user memory is split up into multiple essential parts. At the beginning, a management block is situated, which keeps track of important information concerning the file system.
Subsequently a FAT table as well as its copy are located. Following the FAT section, a structure called application list and its copy are held in the memory. This application list, which is depicted in figure 5.1, does nothing more than listing references to all applications which are available on the card. The application list is optional, there is no explicit need to store this separate list as the applications could also be linked through the application header blocks. Its a design decision, whether an application list should be used or not. After this optional list, the data section starts and occupies the whole left over memory. The overall memory layout can be seen in the pictures 5.2 and 5.3. Depending on which design is chosen either an application list is included or not.

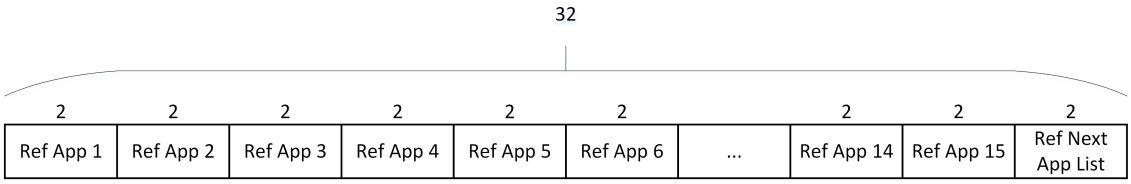

Figure 5.1: Structure of a separate application list

The management blocks, which are also diverse, depending on the usage of a separate application list, can be regarded in the figures 5.4 and 5.5.

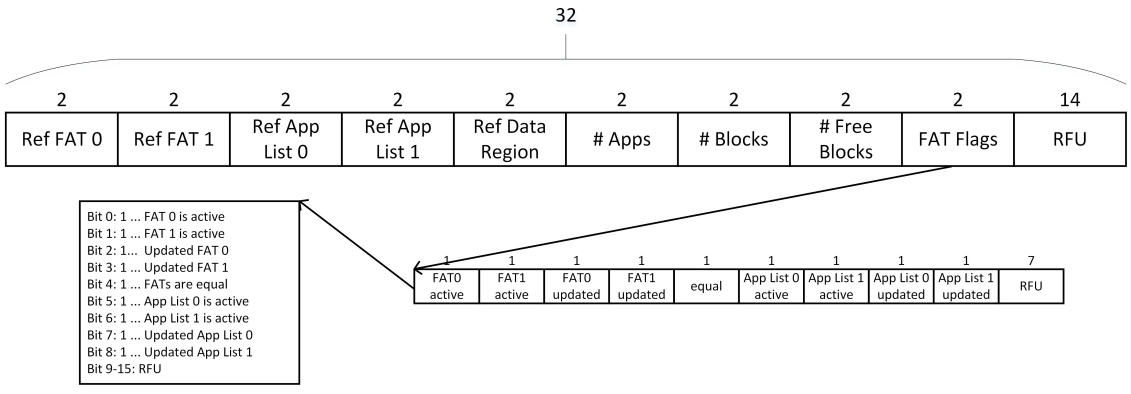

Figure 5.4: Management block of a FAT based file system including an application list

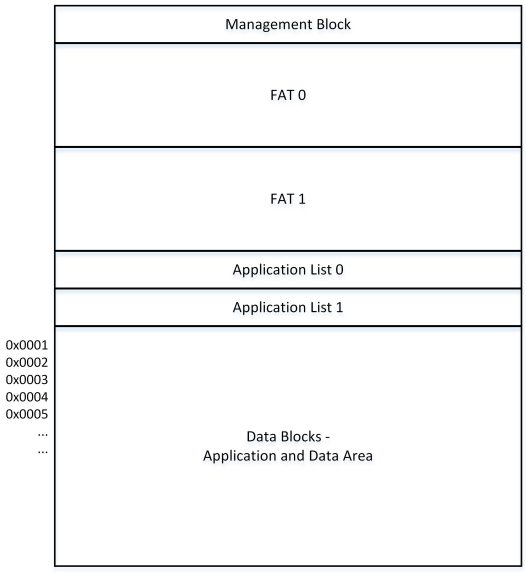

Figure 5.2: Memory layout including application lists

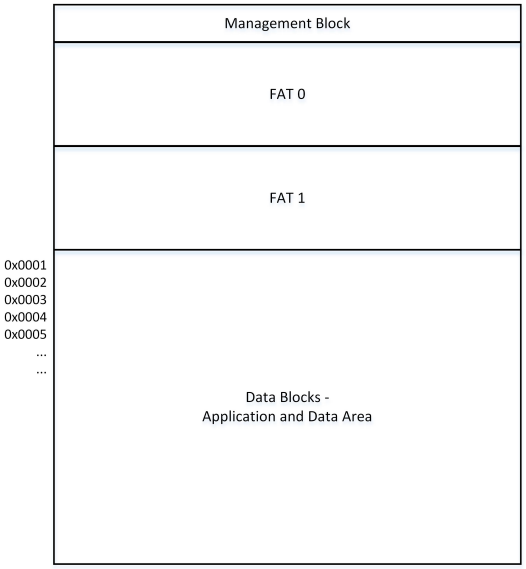

Figure 5.3: Memory layout without application lists

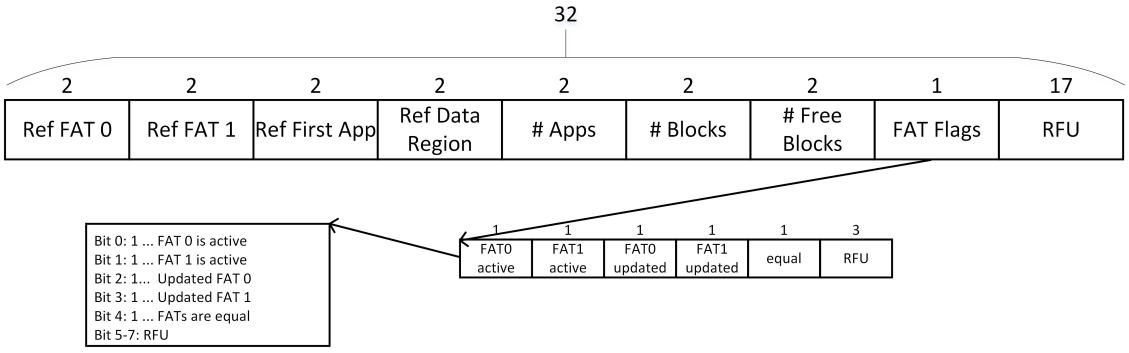

Figure 5.5: Management block of a FAT based file system without an application list

The disk organization is done in blocks: the whole user memory is split up into memory chunks of the same size, namely 32 bytes. Every block is addressed through a 2 byte address. For each memory block, one FAT entry exists. This FAT entry consists of a 2 byte sized entry, namely the usage of the memory block, so the whole FAT contains the usage of each memory block of the file system. The index in the FAT represents the block number and the entry at this index gives information about its usage and the following block, if available.

Depending on the information in the FAT one block can either be free, damaged or used:

- `0x0000`: the block is free
- `0xFFFE`: the file system is damaged at the blocks position
- `0x0001` - `0xFFFD`: the block is in use and the number of next block is given
- `0xFFFF`: the block is in use and that it is the last block in the chain

Memory consumption

As one FAT entry only needs 2 bytes of space and the memory is separated into blocks of 32 bytes, one memory block can contain a total of 16 FAT entries, one after another. The structure of a FAT entry is depicted in figure 5.6.

The size of one total FAT therefore makes up $\frac{1}{16}$ of the whole user memory. With consideration, that two FAT tables are needed, the required space doubles to $\frac{1}{8}$.
Example: In order to address a user memory consisting of 8 kilobytes, a FAT table with only 16 blocks (each containing 16 FAT entries) is needed. This results in a size of 512 bytes per FAT table.

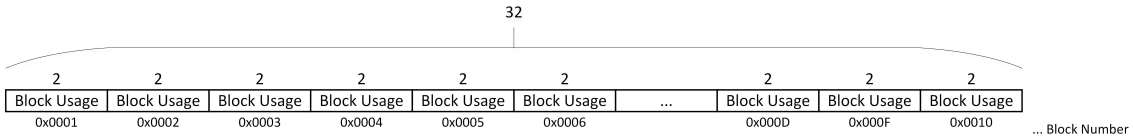

Figure 5.6: Structure of a FAT entry

Additional to that, the space for the application list makes up at least 32 bytes. Depending on the amount of available applications, also more blocks could be needed. Each application needs a field of 2 bytes for the application pointer inside the list.

In order to translate the block number to the real block address of the file system, a little calculation has to be done. The block number one is the data block located at the address of the data region which is contained in the management block. So the formula for calculating the block address is:

$$\textit{block address} = \textit{address of data region} + \textit{block number} - 1 \quad .$$

Design of the transaction-safe structure

In order to guarantee transaction safe modifications on a file system, two copies of a FAT (FAT0 and FAT1) are stored internally. In the beginning both copies hold the same content and during the transaction process, one of them always is kept as kind of a stable backup version, while the other one is modified according to the changes which are made

during a transaction. The one which represents the latest good status is the *active FAT* whereas the one which is modified during the transaction is the *inactive FAT*.

The stable file system version is indicated via a certain flag, the *active flag*. If this flag is set, the FAT is the last known good one and represents the latest working version of the file system.

All modifications on the file system are retained in the *inactive FAT*. After modifying the file system and successfully completing the modification, several flags need to be set to indicate the changes of the file system.

If a transaction is disrupted through power loss or something else, the active flag will not be toggled and therefore the already made changes to the file system will be lost and the working version (active FAT) will not be influenced at all.

The setting of the *active* flag as well as the setting of the *updated* flag is included in an atomic operation which concludes an entire transaction and commits it.

After the *active flag* has been toggled and a change of the active FAT as well as the application and block counters was made, the new modified file system is active. However, the two FAT images have a different content after switching the active FAT, and need to be synchronized before a new transaction can start.

If the two FAT images are not synchronized, the inactive FAT will still contain the status of an old file system version and new upcoming changes will bring the whole file system in an inconsistent state. Therefore it is extremely important that the active FAT is copied to the inactive FAT after a successful modification and commit, so that both are the same and represent the latest working file system version. The synchronization actually is done when starting a new transaction, that means that before doing any new modification to the file system, all changes are copied and both versions are synchronized. After successfully synchronizing both FAT images, the *equal* flag needs to be set. These two flags signalize the equality of both FATS.

As already said, a check of the FAT management flags needs to be done in order to guarantee file system consistency, when a new transaction starts.

Application and file creation

When creating an application in the file system, the first free data block is allocated for the 32 byte sized application header. This block number is then stored in the application list (if available), in order to collect all available applications and provide a method for quick application access. Additionally the application counter which is located in the management header, is increased.

The application header consists of multiple fields, which can be inspected in figure 5.7.

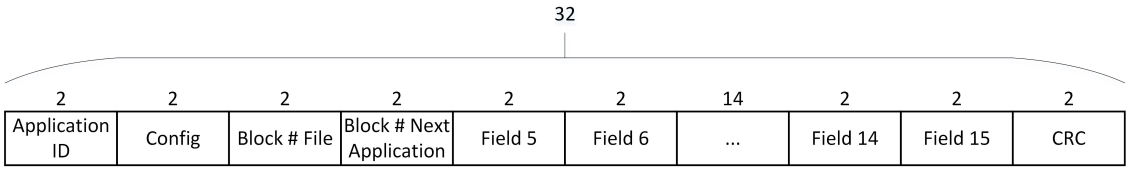

Figure 5.7: Application header of a FAT based file system

The fields *Bock# File*, *Block# Next Application*, etc. are no references but contain block numbers of the blocks, where the according information is stored. If one of these information is not needed (e.g. empty application which has no file associated with it), `0x0000` is written into the field and no FAT entry and no data allocation is necessary.

When a file is created, first of all a check whether the necessary free space is available is made. If there are enough blocks available for the file content and its 32 byte sized header, then the file can be written to the file system.

Important Note: Data is always written in blocks of 32 bytes. Even if only 1 byte is needed, a whole block needs to be allocated and the remaining 31 bytes stay unused.

If there is enough space available, the file header is written. The structure of a file header is depicted in figure 5.8.

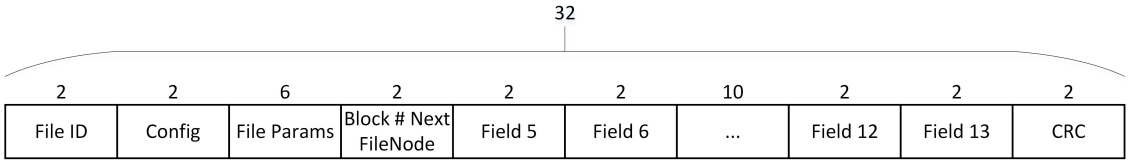

Figure 5.8: File header of a FAT based file system

Basically a file header contains the configuration settings of a file, its size, the block number where the file data starts, various access right sets as well as the block number of the next file node.

After the file header has been written, the file data is written to memory. The data is written to any free block in the whole memory. If the file is bigger than 32 bytes and therefore needs multiple blocks, it is preferable that these blocks reside continuously in the memory, but if this is not possible, they can also be placed in distributed locations. The management of such distributed blocks is done through the FAT. He cares for the correct linking of one data block to another and so builds up whole files and their contents.

The file node only needs to be linked to its application or the predecessor file, if one exists. If it is the first file of an application, the block number of the file node needs to be written into the field *Block# File* of the application header. Otherwise this number needs to be written into the field *Block# Next FileNode* of the previous file node.

Application deletion/file deletion

The big benefit of this file system structure is the possibility to easily delete files and applications and to reuse the new available space.

In order to delete a single file, the block number of the file node has to be erased from the whole file system. That means that the block number may not be referenced from an application header nor from another file.
Is the file which shall be deleted the first one of an application, the file node block number of the next file, or simply `0x0000` if there are no more files in the application space, needs to be written in the application header. Else if the file is not the first one, the previous file header needs to be updated in a way that it contains the block number of the file next after the actual one. In this way the actual file is left out and the files of an application are chained up anew.
Besides this also the FAT needs to be updated in order to update the block usage and to signal that all the previously used blocks are free.

If a whole application shall be deleted, the application ID needs to be removed from the application list and the application counter has to be decreased. After that, all the files that belong to this application need to be marked as free in the FAT.

5.3.2 File System using a FAT in combination with journaling

The file system design which is proposed in this section is very similar to the basic structure of the previously presented design in section 5.3.1.
Like in the previous proposal, the whole memory is split up into blocks of 32 bytes each, which are addressed through a 2 byte address. The overall memory layout looks similar but has one important additional part included, namely the Journal. This can be seen in figure 5.9.

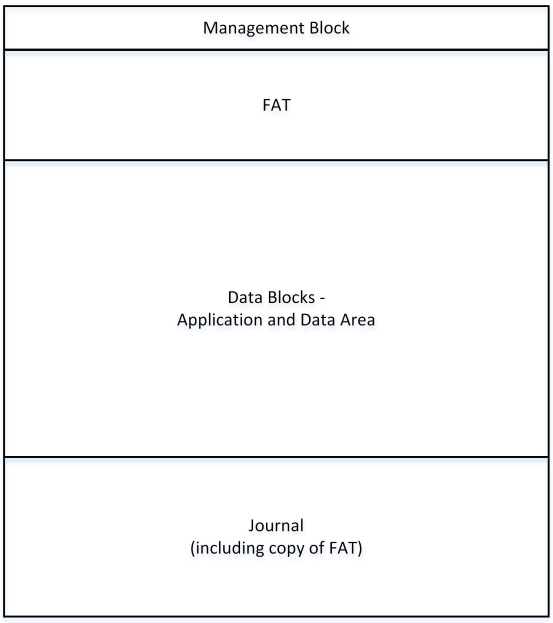

Figure 5.9: Memory Layout of a FAT/journaling combination

Instead of relying on two FAT images for safe file system modifications, transaction safety is guaranteed through a separate memory area called the *journal*.

The journal is a memory section, where all affected data blocks are stored before doing any upcoming changes to the file system. The size of the journal can be defined by the user, he specifies how much memory the file system should reserve for backup reasons. When defining only a small section for the journal, the usable user memory area remains bigger and therefore more data can be stored on the card. Otherwise, if the journaling area is defined bigger, more data can be backed-up and secured. The option which requires the most memory but provides perfect security is splitting the available memory in two halves, one for the user data and one for the journal. This makes it possible to make a backup of the whole data that is stored on the card.
However, normally this will not be necessary and therefore the user can choose, how much memory he will reserve for the journal. The only limitation that emerges is that the journal needs to be at least big enough to store a copy of the FAT, to be able to restore the block linkage in case of a crash.

In the first part of the memory, the management block is located. It looks different than in the previous solution and can be seen in figure 5.10.

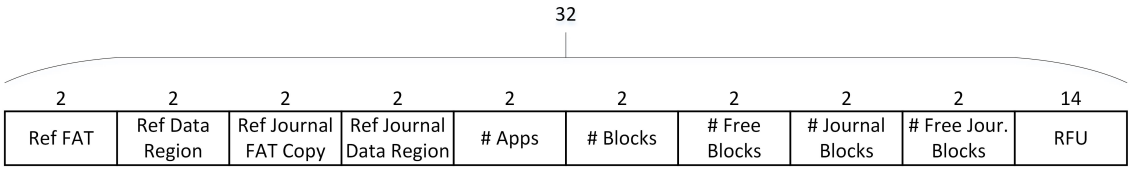

Figure 5.10: Management block of a FAT/journaling combination

The following parts are the FAT and the data blocks. The last memory section contains the journal, which internally includes a copy of the FAT.
The journal is empty in the beginning and, as the whole memory, split into blocks of 32 bytes. Each time a file or an application will be written or modified, an entry in the journal will be made.
The valid FAT will be copied to the journal in order to secure it and to backup the latest working version of the file system. So before modifying the original file system in its working state, a copy of the data which is going to be modified will be made and written to the journal in form of a journal entry. After writing the journal entry and consequently securing the original data, the changes to the file system can be made.

Writing to the journal

Every time a new transaction starts, the journal is cleared completely. All journal entries look the same. First of all the block number of the data block which is going to be modified is written to the journal. Next the offset inside the block and then the length of the data which is going to be modified in bytes is added. Finally the actual data is appended.

The structure of a journal entry is illustrated in figure 5.11. Each single entry consists of the already named parts (block number, offset, length, data).

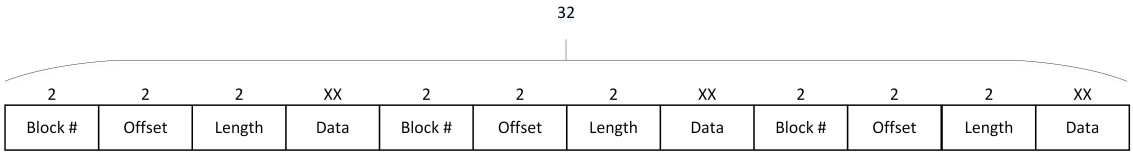

Figure 5.11: Structure of a journal entry

The journaling mechanism makes it possible to backup every kind of data, that is available in the file system. It does not matter if it is about metadata, header data, user data or something else. Everything can be stored, only the block number and data length need to be known.

In order to illustrate the structure of a journal entry, one example can be seen in figure 5.12.

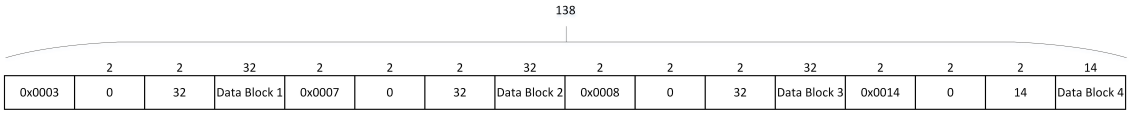

Figure 5.12: Example of a journal entry

Memory consumption

As already mentioned in section 5.3.1 one file allocation table needs $\frac{1}{16}$ of the whole user memory.
The space which is required for the journal can be specified by the user and is therefore not known beforehand. It needs at least to be big enough, to store one FAT as a secure copy and to offer some memory which can be taken for journaling purposes.
All the remaining memory is available for the user data.

5.4 Design approach based on the EXT file system structure

Based on the basic functionality of the *Extended File System* which is explained in chapter 3, in section 3.2.1, a possible file system solution is designed and explained in this section.

5.4.1 EXT based file system in combination with journaling

The basic structure for managing metadata that is used is the INode. The INode stores all information which is connected to one file, except the actual user data itself. Its modified structure is designed similar to the one which is used original by the EXT file system, but instead of using 15 pointers, the proposed solution uses less fixed pointers but rather a dynamic solution.

For an EXT based file system memory layout, the overall organization of the data structure is done in blocks. The whole user memory is split up into blocks of the same size, namely 32 bytes. The first block of the file system is the superblock. It contains important management information, like the amount of blocks and free blocks, and pointers to the block usage bitmap, the data area and the journal. It can be seen in figure 5.13.

Subsequently the data area follows. It contains all user data as well as the later on explained INodes. This section is followed by the journal. The size of the journal is not fixed and can be chosen by the user at file system initialization.
The memory layout is depicted in figure 5.14.

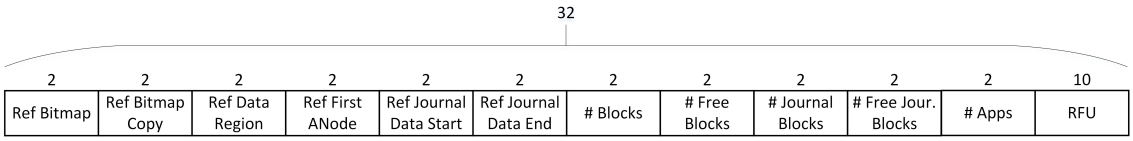

Figure 5.13: Superblock of the EXT file system

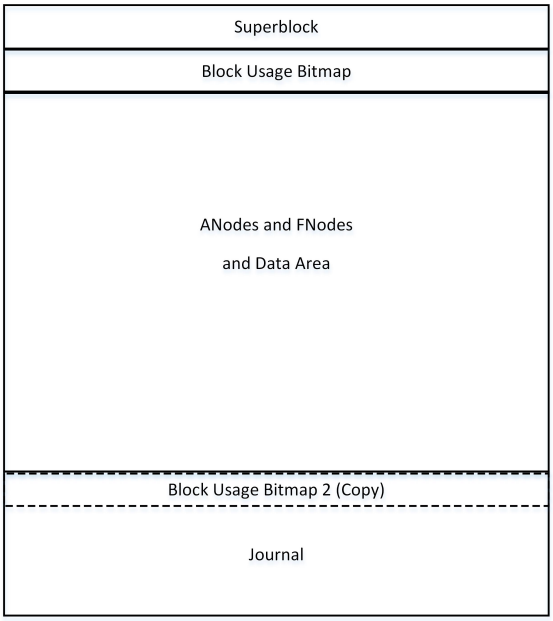

Figure 5.14: Memory layout of an EXT/journaling combination

ANodes and FNodes

First of all a distinction between two important types, namely application-INodes and file-INodes (further on called ANode and FNode), needs to be made. Applications contain one up to multiple files and therefore probably need to store more meta information than a single file does. Both types (applications as well as files) are represented through an inode, but the structure and size of the inode slightly differs.

The ANode contains important information concerning the application itself as well as a pointer to a so-called *file list*. The file list contains 15 reserved fields which are intended to store pointers to files of the application. Initially they are all set to NULL and only get changed if there are files added to the application. If one application needs more than 15 files, the sixteenth field can be used to point to a new file list, which again offers space for pointers to 15 new files. This chaining of file lists can be done as long as there is space available on the card. A picture of an ANode and the mentioned file list can be inspected

beyond in figure 5.15.

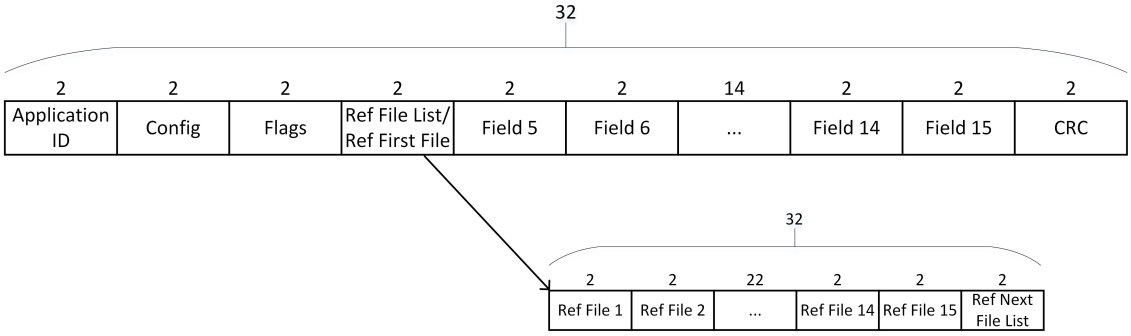

Figure 5.15: Application node (ANode) with a file list

If an application needs more than the 15 designated files, a new list can be allocated and the sixteenth pointer can be used to point to its location.

The FNode is basically structured in the same way as the ANode. It contains important information concerning the file and a *data block list* instead of a *file list*, as the ANode does. This means that the FNode does not point to file headers but contains pointers to data blocks. The organization of the pointer list is done in the same way like it is done for the ANode. This *data block list* however only is needed, if the single data blocks of one file are located fragmented in the user memory. If it is possible to write the data into continuous memory blocks (what should be preferred) only one pointer to the first data block is needed. Then no additional memory block needs to be used and a single block pointer is sufficient.
Whether the data can be written sequentially or not needs to be recorded in the configuration settings of the FNode. In the *Config* field the *Block List* bit needs to be set to true if a separate data block list is needed, and to false in case no list but only one data pointer is needed.

An example of the layout of an FNode can be seen in figure 5.16.

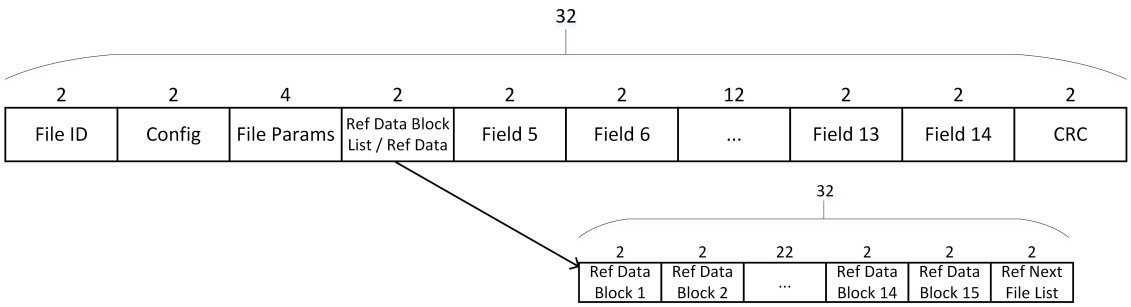

Figure 5.16: FNode with an optional list containing pointers to data blocks

Journaling mechanism

The data backup mechanism which has been selected for this file system design is the reverse journaling, as it is described in the subsection 5.2.2. The principle workflow looks similar to the proposed solution in combination with a FAT, which is explained in subsection **??**. However, instead of using a FAT, no special kind of management data structure, except of a bitmap, is needed within this approach.

Also in this design the user can specify the size of the memory section that he wants to use for the journal himself. Journal entries are structured in pretty the same ways as they are structured in the FAT-based design. The only big difference to the previous presented journaling version is, that no block numbers are used any more. Instead of numbering each memory block, their address is taken. This address is also written to the journal entry to identify which memory block is going to be modified. The other fields which represent the offset and length stay the same. A journal entry and the small changes can be seen in figure 5.17.

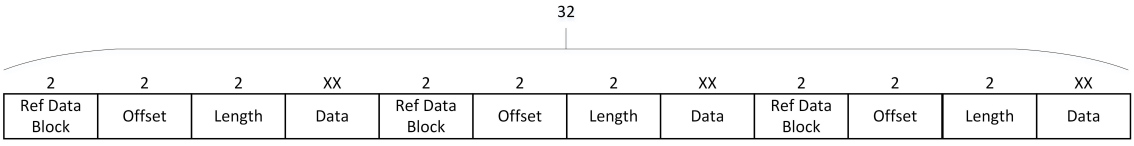

Figure 5.17: Structure of a journal entry

Memory consumption

The memory consumption for this EXT based design approach is quite low. 32 bytes are needed for the Superblock, 1 bit for each block in the data section is needed for the block

usage bitmap and the size of the journal can be specified by the user and is therefore unknown beforehand. The journal contains one copy of the block usage bitmap, so it needs to have at least its size.

Although the memory consumption is quite low for management structures, the actual realization of the file system might tend to waste a bit of space. The *file list* and also the *block lists* are (if needed) allocated in form of a whole block and therefore it might be the case, that they might waste space if not all available fields in one block are needed.

5.5 Design approach based on the YAFFS file system

Based on the basic functionality of *Yet Another Flash File System* which is explained in chapter 3, in section 3.3.3, a possible file system solution is designed and explained in this section.

The structure which is used by the YAFFS files system is a log structure. One entry in the file system is called a chunk, which basically is a memory block. Each chunk consists of a memory region and some tags which are used for management reasons. Originally YAFFS1 has a modified log structure whereas YAFFS2 has a true log structure. A true log structure means that writes to the file system only are done sequentially.

YAFFS1 uses deletion markers and serial numbers within each chunk. The serial number is incremented every time, a chunk is replaced and deletion markers are used to track the status of each chunk. This means that old chunks are re-accessed and marked as deleted or unused, and therefore an overwrite or write-back occurs in a non-sequential manner. YAFFS2 follows the philosophy of zero overwrites, meaning that the usage of deletion markers is forbidden and write operations have to be performed strictly sequential. YAFFS2 has a better write performance than YAFFS1. However, as YAFFS2 does not use deletion markers, scanning is significantly more complex and takes more time for this file system version. A big disadvantage is that the chunks have to be ordered each time the file system is accessed and that the scan has to be performed backwards, meaning in reverse chronological order.

5.5.1 Mixture between YAFFS1 and YAFFS2 design

As in the previous proposals, also in this suggestion the user memory can be split up into blocks of 32 bytes but for later on explained reasons the block size actually is set to 40 bytes.

In the presented file system the user memory is structured as one big log. No management block or any directories are needed at all. The log structured memory contains all kind of user data, including headers, key data and so on. Both application and file information as well as the connected headers are stored in this part of the memory.
The log structured memory is maintained in a log based chronological order. No matter which kind of data is written to the memory, writing is done chronological starting from the top. When a new data block is written, the first next free block is allocated and

used. This simple basic layout can be modified in order to get a quicker access to free memory blocks. When maintaining an optional additional block usage bitmap, the first free memory block can be found very quickly. The big benefit of keeping a block usage bitmap in memory is that searching the next free block is faster than traversing each block of the log structured area until a free one is found. On the other hand the drawback is the additional memory consumption as for each memory block one additional bit is needed for maintaining the usage list. The basic idea of the layout of the log structured file system can be seen in figure 5.18.

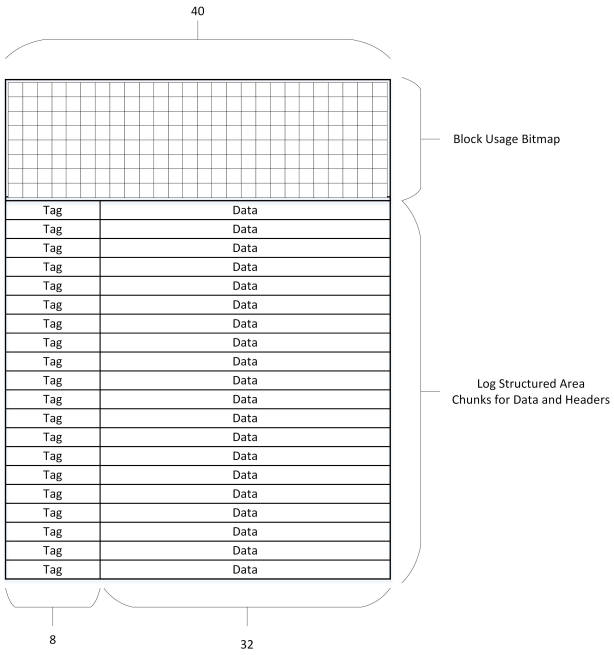

Figure 5.18: YAFFS based file system layout

Chunks of the log structured memory

A memory block is called chunk and is 40 bytes big. Each chunk consists of a tag and a data region. In the data region the user data is written and in the tag region some information which is necessary for the log management is stored. This means that not the whole 40 bytes can be used for the plain user data because some bytes are used for tagging. A chunk with a size of 40 bytes contains a 8 byte sized tag and 32 bytes are reserved for user data. A chunk is depicted in figure 5.19.

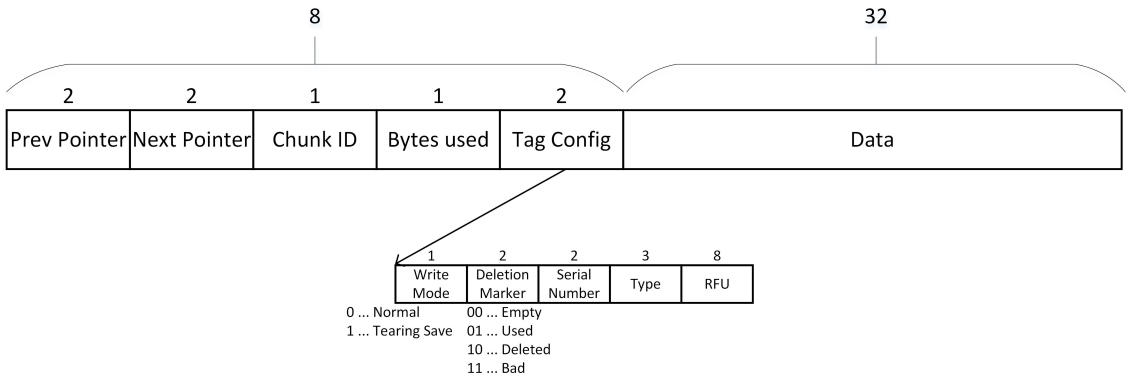

Figure 5.19: Chunk with tag and data region

A chunk is characterized through its tags. The tags comprise the following important fields:

- Prev Pointer: This pointer is used to point to the previous chunk or to the parent application, if no previous chunk is available.
- Next Pointer: This pointer is used to point to the next chunk.
- Chunk ID: This ID is used to specify the chunk sequence, if more than one chunk is used to represent the user data.
- Bytes used: It reflects how many bytes of the data region of this chunk are used. It contains a number between `1` and the size of the data region.
- Tag Config: This field contains the write mode, deletion marker and the serial number.

The deletion marker gives information about the status of a chunk. Each chunk may be in one of the following states:

- EMPTY `0x0`: This chunk has nothing in it and can be allocated. In the beginning, when the card is initialized, each chunk is in this state. When it is allocated it moves to the USED state.
- USED `0x1`: This chunk is allocated and in use. If it is not used any more and replaced by a newer version it comes into the DELETED state.
- DELETED `0x2`: This chunk is not used any more and its content can be deleted. After actual erasure its state is set to EMPTY.
- DEAD `0x3`: This chunk is corrupted or bad. This is a final state.

The serial number is a two bit counter, starts with the value `0x0` and goes up to `0x3`. It is modified each time a chunk is changed and re-written as a new chunk. When the limit `0x3` is reached, counting restarts from the beginning.

The type specifies, which kind of data is stored in the data region of the chunk. The type is also essential for the writing mode of the chunk. Depending on the type writing/updating either is done tearing safe or in the standard way.

Memory consumption

The current proposed solution combines each 32 byte sized data block with a 8 byte sized tag.
Example: In order to address a user memory consisting of 8 kilobytes it first is split up into blocks of 40 bytes. As the result is not even, 204 blocks can be addressed and the rest of the memory unfortunately stays unused. Therefore only 8160 bytes out of 8192 can be used. To build up a bitmap for this memory structure, 204 bits are used, that are 25 bytes and 4 bits, which can be round up to 26 bytes. So the amount of space which is required by a bitmap is really small.

B-Tree design based on original YAFFS2 using a true log structure

The original YAFFS file system uses no pointers or other structures at all in the non-volatile memory, it only consists of the log and nothing else. For easier access to directories and files, a binary tree is built up when mounting the file system and is updated each time a change is done. This tree is normally held in RAM.
When approaching the design of the card file system in exactly this way, without using references to chunks at all, the benefit is, that every chunk can be rewritten without doing any update on the actual block and therefore no tearing safe write operations are needed.

However building and maintaining a binary tree on the smart card is infeasible due to space issues and timing reasons. With every little change in the log also a change in the tree would be necessary and all pointers of the tree would have to be chained anew.
Emerging of this, the linear search has to be used to make data access possible without using a binary tree. The huge drawback is, that reading and accessing an object (application or file) in the right order is slow, because the chunks are not sorted and every part needs to be searched separately. Searching through a list takes linear time, that means that in the worst case, all elements of the list need to be traversed. The linear search is proportional to the number of elements in the list, what makes the performance really bad.

Design using pointers inside the log structure

In order to get rid of the need of a binary tree and to avoid the linear search, the structure of the YAFFS file system has been modified and also pointers have been included in the log entries. The references inside the chunks point to the previous and next chunk and make data access easier but unfortunately complicate the management of the chunks.

In the depicted chunks in figures 5.20 and 5.21, which represent application and file headers, references to other chunks are used. This makes the log structured write to the file system a bit more complicated.

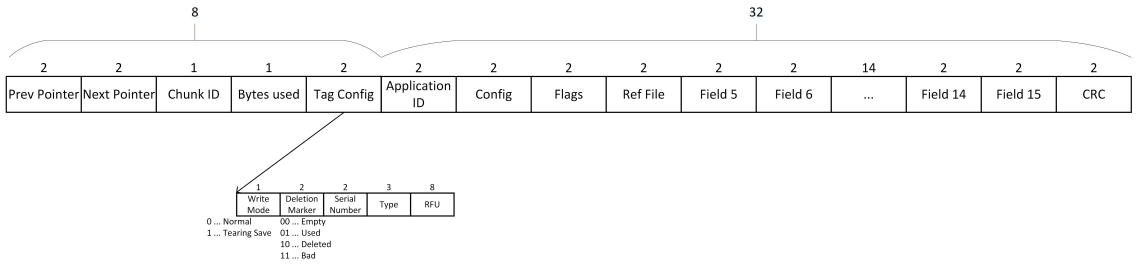

Figure 5.20: Chunk with tag and application header

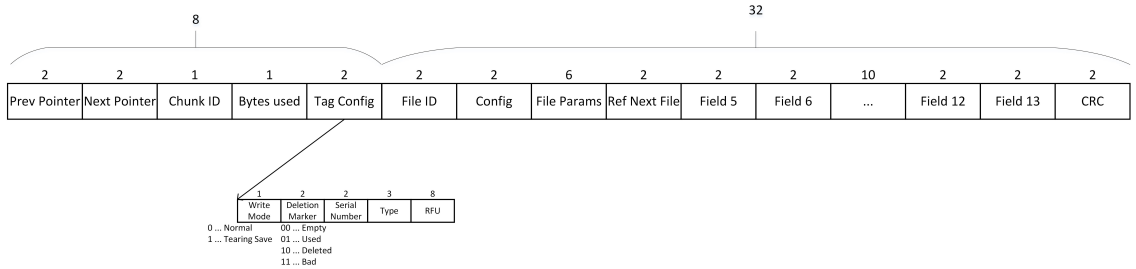

Figure 5.21: Chunk with tag and file header

When writing a new object nothing special has to be considered and the write operation can be performed in a normal way. This means, that for each data block a new chunk is allocated and tag as well as data section are written in the freshly allocated chunk. However, when doing an update operation and the data therefore is written to a new chunk, also the pointer references in the chunk before and the chunk following the currently updated one, need to be updated in order to keep the file system consistent. These pointer updates need to be done tearing safe, as the modification is done directly in the original block.

The only type which is updated using a new allocated chunk, copying the data into it and appending it to the log, is the file data. Updating all other kinds of data is done tearing-safe directly in the concerned chunk. The usage of pointers makes the file system to a false log structured one, allowing write-backs and overwrites.

An example of a practical use case is given in the table 5.1. In this example the transactions happened in the same order as the data is written to the memory. First of all an application is created and the application written to block `0x0000`. An application header only needs one chunk, therefore the fields Prev Pointer and Next Pointer in the tag section

are set to the value `0xFFFF`. As the application is going to have an additional application information block and one file connected to it, the data section of the application header chunk contains pointers to these addresses. Subsequently the three additional application info blocks are written. After this, the file header and the file data is written to the file system. The data is 82 bytes big and therefore needs three chunks.

Chunk Adr	*Prev Pointer*	*Next Pointer*	*Chunk ID*	*Bytes used*	*Tag Config*	*Data*
0x0000	0xFFFF	0xFFFF	0	32	10100000	Application Header
0x0028	0x0000	0x0050	0	32	10100001	Add App Info
0x0050	0x0028	0x0078	1	32	10100001	Add App Info 2
0x0078	0x0050	0xFFFF	2	32	10100001	Add App Info 3
0x00A0	0x0000	0xFFFF	0	32	10100100	File 1 Header
0x00C8	0x00F0	0x0140	1	32	00100101	File 1 Block 1
0x00F0	0x0118	0x0168	2	32	00100101	File 1 Block 2
0x0118	0x0140	0xFFFF	3	20	00100101	File 1 Block 3
0x0140	0	0	0	0	0	0
0x0168	0	0	0	0	0	0
0x0190	0	0	0	0	0	0

Table 5.1: Example for a log structured write to the file system

All objects which belong directly to an application are also connected to this one via a pointer. The first chunk of these objects uses the Prev Pointer (which normally is not used in the first chunk of a new object and would point to `0xFFFF`) as Parent Pointer which points to the address of the belonging application.
In all other cases the Prev Pointer is used to point to the previous chunk and to connect chunks which belong to each other, in order to build a sequential double-linkage of data.

The usage of the Parent Pointer makes it easier to manage object dependencies inside the log structured file system and provides a possibility which allows a quick navigation to the parent application starting from any object.

When updating the data in chunk `0x0118` the chunk is not overwritten, but a new version of it is appended to the log. When the operation succeeds and the new chunk is written successfully, the pointer of the previous chunk is updated tearing safe and set to the new written chunk. Also the deletion marker of the old chunk can be set to DELETED in a tearing safe operation. The table 5.2 has been updated and shows the made changes.

Changes in the tag region of a chunk are always done tearing safe directly in the concerned chunk. A new chunk only is allocated and written, if an update of the data region of a file occurs, or if totally new information is written.

Chunk Adr	*Prev Pointer*	*Next Pointer*	*Chunk ID*	*Bytes used*	*Tag Config*	*Data*
0x0000	0xFFFF	0xFFFF	0	32	10100000	Application Header
0x0028	0x0000	0x0050	0	32	10100001	Add App Info
0x0050	0x0028	0x0078	1	32	10100001	Add App Info 2
0x0078	0x0050	0xFFFF	2	32	10100001	Add App Info 3
0x00A0	0x0000	0xFFFF	0	32	10100100	**File 1 Header, tearing safe pointer update in data region**
0x00C8	0x00F0	0x0140	1	32	**0100000**	**File 1 Block 1, old version, tearing safe configuration update**
0x00F0	0x0118	0x0168	2	32	00100101	File 1 Block 2
0x0118	0x0140	0xFFFF	3	20	00100101	File 1 Block 3
0x0140	**0x00F0**	**0x0140**	**1**	**32**	**00101101**	**File 1 Block 1 updated, new block written**
0x0168	0	0	0	0	0	0
0x0190	0	0	0	0	0	0

Table 5.2: Example for a log structured data update

The proposed solution makes it possible to do write operations of new objects and updates of file data in a log structured way. For all other kinds of updates the already existing chunks are modified in a tearing-safe way.
When updating file data and therefore writing a new chunk, deletion markers are used to mark the old chunk as deleted. The data inside this chunk is not erased immediately, the label only signals that it has to be deleted. It is not erased immediately during the commit operation because this would take too much time and the commit operation shall be as fast as possible. Therefore it is only marked as deleted and the actual erasure is done each time before a new transaction starts. Before starting a transaction, the file system is checked quickly and the content of all chunks which are marked as DELETED is erased, their deletion marker is set to EMPTY and so the chunk can be reused again.

As old unused chunks are marked as deleted and consequently erased, holes with unallocated memory emerge in the log structure. These holes provide space for new chunks and can be allocated and used for new data blocks. The reuse of holes is possible as coherent data blocks do not need to be written contiguously. However this reuse leads to fragmentation.

5.6 Tree based file system proposals

As file systems which are based on B-trees rank among the most used disk file systems, they also have been taken into consideration regarding their possible usage on smartcards. A brief explanation of two common tree-structured file systems, namely ReiserFS and BTRFS, is given in chapter 3 in sections 3.2.4 and 3.2.5.
Additional to the already provided information, more details on B-trees in general and the associated dictionary operations are discussed in this section.

5.6.1 Definition and mode of operation of B-trees

A B-tree is a dynamically updatable, balanced tree-like index structure, also specified as multiway search tree. The B-tree can be seen as an extension to the well-known balanced binary tree, as a B-tree is always balanced [Lz09].

In the B-tree every node corresponds to one disk block. An internal node stores a list of keys and a list of pointers and a leaf node stores a list of records, each containing a key and a value. Every node except the root node has to be at least half full. If the root node is an internal node it must have at least two child pointers. [Lz09].
The keys of an internal node act as separation values which divide the sub-trees. This leads to the fact that the number of branches or child nodes of a node will be one more than the number of keys stored in the node [Lz09].

B-trees support the insertion and deletion operations in $\mathcal{O}(\log_p n)$ time on a hard disk, with n being the number of records in the tree and p being the page capacity in number of records [Lz09].

5.6.2 Dictionary operations

The B-tree supports two kinds of queries, the exact-match query and the range query. This means that on the one hand searches for a record with a certain key and on the other hand searches for records whose keys belong to a range, can be executed. Search operations always start at the root [Lz09].

In order to insert a new record into the tree, an exact-match query has to be performed first, in order to locate the leaf node in which the record should be stored. If there is enough space available, the record is stored in the node, and if not, the leaf node is split and a new one is allocated. After a split, the records with the larger keys of the overflowing node are moved to the new node. This may also cause the parent node to overflow and so on. In the worst case, all nodes along the insertion path are split [Lz09].

The deletion of a record from the B-tree also uses the exact-match query as first step to locate the leaf node which contains the record, and afterwards the record is underflowing removed from the found leaf node. If this node is at least half full, the operation is finished, otherwise the node and the contained records need to be re-distributed to a sibling. If the redistribution is not possible, the node needs to be rotated or is merged with a sibling. After deletion, the tree needs to be rebalanced. Rebalancing starts from a

leaf node and proceeds towards the root [Lz09].
As the deletion of a key implicates expensive operations like redistribution, rotation, merging and rebalancing, it is a much more complex operation than the insertion.

5.6.3 B-tree inside the smartcard

The usage of a B-tree brings along certain kinds of necessities. Besides the main memory, which is separated into blocks, an additional area needs to be reserved for the organization of the tree. The structure of the whole tree is built up through pointers. Each node except the leafs contains some own information, keys and pointers to child nodes. The leaf nodes contain the actual information.

As the accumulation and chaining of pointers requires much space and a complex way of storage, the effect of quick search and insert operations is already marred. Through the B-tree a data block can not simply be accessed over one pointer and often a traversal of a long pointer chain is necessary until the desired block is reached. This possibly long pointer chains slow down the access time and may unnecessarily lengthen read operations on the smart card.

Additional to this, the deletion of a file or single data block can really be horrible for the performance. It is infeasible for a smartcard to completely re-structure a whole B-tree in case of erasing a node. Rotating and rebalancing consume much time and write operations. As many pointers need to be set anew in the worst case, countless write operations would have to be performed, what simply is unacceptable.

Based on the facts that a B-tree based file system is memory consuming as well as totally non-performant with respect to write operations and timing issues, it is rather not suitable for smart cards. As a result, this kind of file system is not examined any further and regarded as unqualified.

Chapter 6

Comparison of the different suggested File System Concepts

This chapter deals with the comparison of the in chapter 5 presented different file system approaches. In order to find out the most appropriate file system, which provides the best performance and memory utilization, the key operations which are performed on a smartcard are examined in detail.

In the following sections, each presented file system approach is investigated thoroughly by means of the operations *write, read, update, delete, commit* on the smart card objects *application* and *file*.

The results of the investigation give an insight in the required write operations and memory usage.

6.1 General clarifications

In the following sections different types of file systems and design approaches are explained and compared in detail, what requires a short discussion of general requirements and an explanation of different ways to approach these.

6.1.1 Fragmentation

As the majority of the suggested designs is block oriented, it might be said that a block-oriented design is most efficient if blocks which belong together are also situated next to each other in the memory. A continuous placement of adjacent memory blocks is a big benefit for read as well as for write operations. A low fragmentation decreases access times and makes it possible to do one longer, bigger operation instead of multiple smaller ones, which are costlier. A high internal fragmentation leads to a high distribution of data blocks which would belong together and therefore increases read and write times.

Emerging from this fact it is desired to write belonging blocks continuously as far as possible. Therefore an additional field in the file header and application header of the file

system designs is available, which can distinguish between two types of data positioning: fragmented or continuous. When file data or any other object data is written continuous and that is also marked in the belonging header, the read operation can be performed with knowing only the starting block and data length, no further information about block linkage or block association is needed.

6.1.2 Handling of security relevant data (e.g. key data)

Each application can have some kind of security relevant data connected to it, for example its own application key file. A security object can be accessed through the application, as the reference to it is always stored in the application header.

If an application key file or other kind of security relevant object is needed, it is always created together with the application. During the creation of an application the object needs to be initialized.

As this kinds of objects are very sensitive and managed by an independent instance, they need to be written continuously into the memory, meaning that the blocks need to be placed next to each other. A distributed storage of e.g. key blocks is not possible and therefore a check which searches through the file system that tries to find a big enough memory chunk needs to be made before creating an application.

Depending on the number of keys, which need to be specified in order to create one application according to the reference file system, the number of blocks which are required for one key file can differ. As all keys need to be written sequentially, the free blocks need also to be found successively in the memory. If not enough adjacent free memory blocks can be found, the key file can not be written and therefore also the application can not be created. This kind of limitation needs to be made in order to ensure the sensitive handling of security relevant data as well as to guarantee the unproblematic access to the data from hardware elements. Security objects often are accessed from and treated by hardware and therefore require this kind of protection which prohibits the usage of splitted of memory blocks.

6.1.3 Linking of applications and files

Basically two different kinds of ways, how applications and files can be linked to each other in order to apply a kind of directory structure are available. On the one hand the approach of the reference file system can be used, which basically is the linking of one application to another through a pointer in the application header. And on the other hand a separate structure, the application list, can be used. The application list is basically a memory block which contains only pointers (respectively block numbers) which refer to all applications that are available in the file system.

Both solutions are well suitable for the new file system, however both bring along certain drawbacks.
The linked list of applications is really fast when accessing an application in the first section of the list. The longer the list grows, the slower it gets. In contrast to that, the

application list performs better when accessing an application at the end of the block.

The application list wastes memory, when only a few applications are stored on the card. As a whole block needs to be allocated it is a very inefficient solution when not using many applications. The linked list therefore does not waste any memory. The 2 byte sized field, which is required to store a reference or a block number, is available in the application header. No additional block is required and the linking can be done out of the box. Another drawback of the application list is the limitation to sixteen applications. In case more applications are needed, a second block needs to be allocated and linked to the first one, what again wastes expensive memory.

Based on the discussed reasons it can be said, that the linked list approach basically is the preferable one. However, the application list can be the better performing solution sometimes and also is required in order to realize some design approaches. Therefore both presented ways are used in this thesis.

6.2 Examination of the double FAT approach with application lists

In this section the file system approach based on two FATs with the help of using application lists is explained in detail. The standard operations like write/update/delete are explained in detail and the needed write operations are counted.

The measurement results of the standard operations applied on the FAT file system using two application lists can be seen in table 6.1 at the end of this section.
In figure 6.1 a diagram that depicts the workflow of an transaction, can be inspected.

6.2.1 Create and write operations

In order to create an application in the file system, first of all the management block needs to be read. This needs to be done to check if the *double FAT* system is consistent, namely if the two FAT tables are equal and reflect the same file system status. Also the amount of available empty blocks needs to be checked in order to guarantee that there is enough space left and the new application can be written.

The file system state is okay and equal for both FATs, if the flags are set in the following way: *equal* == 1, *FAT0/FAT1 updated* == 0, *AppList0/AppList1 updated == 0.*
In case the flag *equal* is set to zero, some synchronization steps need to be done, which are explained in the subsection 6.2.5. If the equality of the two FATs is guaranteed and there is enough space left on the card, the write operation can begin.

In order to write an application, the following steps need to be executed:
First of all, the first free block of the file system needs to be found. As all free blocks are characterized through a `0x0000` entry in the FAT, this can be done by starting from the top and traversing all FAT entries until the first free one is found. Additionally to one free

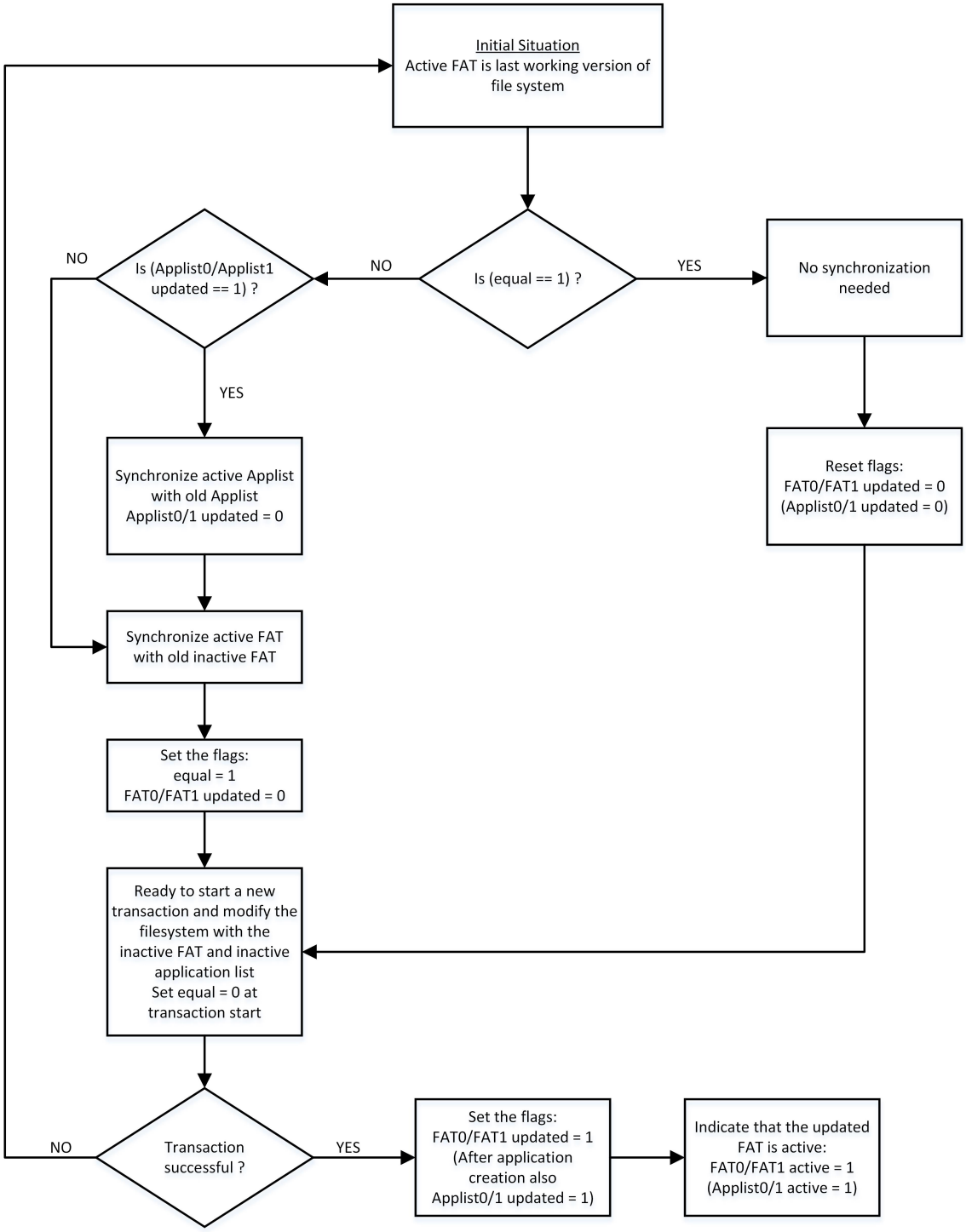

Figure 6.1: Workflow of one transaction using the FAT file system design

block, which is required for the application header, also free blocks for the application key file need to be found.

When the required free blocks have been found, the transaction can be started. The content of the application header can be written into the data region and the blocks need to be marked as used in the inactive FAT. After this, the number of the block where the application header was written to, needs to be written to the inactive application list in order to have a connection to the new application.

After this has been done, several fields and flags in the management block need to be modified in one tearing-safe write operation. The free block counter needs to be decreased and the inactive application counter needs to be increased. The *FAT0/FAT1 updated* flag needs to be set to true, the equal flag needs to be set to false and the *active* flag needs to be set to the freshly updated FAT.

For this transaction, three normal write operations (write application header, write to FAT, update application list) and one tearing-safe write operation (update management header) are needed.

When creating a file in the file system, again a free block needs to be found in the FAT. When it was found and if there are enough free blocks for the file‘s content, the file header can be written to the file system. After writing the file header and initializing the blocks for the file data with zeros, the FAT needs to be updated accordingly. In the best case, namely when enough blocks are available sequentially and not fragmented, only 1 write operation is needed for writing the header and initializing the file‘s data blocks.

If the previous mentioned steps succeeded, the created file still needs to be linked either to the parent application (if it is still empty) or to the previous file. Subsequently, the data can be written to the file.

Instructions for **creating an application**:

1. TS-Write: Update the management header and modify flags (equal)
2. Write: Write the application header
3. Write: Update the inactive FAT
4. Write: Store the application reference in the inactive application list
5. TS-Write: Update the management header and modify flags (updated FAT, active FAT, active App List, updated App List)

Instructions for **creating a file**:

1. TS-Write: Update the management header and modify flags (equal)
2. *(1 to n)* * Write: Write the file header and initialize the blocks of the data area
3. Write: Update the inactive FAT
4. TS-Write: Store the file reference in the previous file header or the parent application
5. TS-Write: Update the management header and modify flags (updated FAT, and active FAT)

In order to write data to a file, the affected block or blocks are written anew and need to be linked correctly in the FAT. So the original data is not touched but copied to another block and modified there. This new block then needs to be linked correctly after writing it, so that a file‘s content stays usable.

Instructions for **writing file data**:

1. TS-Write: Update the management header and modify flags (equal)
2. Depending on the case, different operations need to be executed. If only certain bytes inside a block need to be modified:
   (a) Write: Copy the data block which should be modified to a new, free block
   (b) Write: Write the new data bytes in the new block, where the copied data content was placed
3. If whole data blocks need to be modified:
   (a) *(1 to n)* * Write: Write the new data to *1 to n* new, free blocks
4. Write: Link the blocks in the inactive FAT, instead of the old ones now the new data blocks are integrated

Instructions for **writing file data (without backup)**:

1. *(1 to n)* * TS-Write: Write the new data directly into the *1 to n* blocks which should be updated, without security protection. Then also no commit operation is needed afterwards.

Instructions for the **commit operation (after writing file data)**:

1. *(1 to n)* * TS-Write: Update the file header(s) (update the file pointer if the first block is moved and eventually disable the contiguous flag)
2. TS-Write: Update the management header and modify flags (updated FAT, and active FAT)

6.2.2 Update operations

If an update operation needs to be done, it depends on which kind of object the update should be executed. If a header should be updated, the update is done through a tearing-safe write operation directly in the concerned block. No update on the FAT needs to be done and no new linking is required.

Instructions for **updating an application or file header**:

1. TS-Write: Update the desired block through a tearing-safe write operation

6.2.3 Delete operations

A delete operation does not really modify the file system but more or less only updates the FAT accordingly. If a memory block is not used any more and shall be erased, the related FAT entry is set to unused, namely `0x0000`. This signalizes that the block is free and available for further use. Additionally, the *free block counter* needs to be increased.

The update of the FAT is done with a normal write operation and the update of the management block is done tearing-safe. Therefore one normal and one tearing-safe write operation is needed.
If an application is deleted, additionally the application list needs to be updated, respectively the application entry needs to be deleted, with 1 normal write operation.
As also all the files which are contained in one application need to be freed, all the blocks which are used by the files of an application and by the application itself need to be marked as unused in the FAT.

When deleting a single file, the reference to this file needs to be eliminated. Therefore either the header of the previous file or the application header (if it only contained one file) needs to be modified tearing-safe. The pointer to the erased file needs to be overwritten, so that the field contains a pointer to the next file or if the application is empty, it points to `0xFFFF`. This is done tearing safe.

Instructions for **deleting an application**:

1. TS-Write: Update the management header and modify flags (equal)
2. Write: Set the blocks which shall be deleted to unused `0x0000` in the inactive FAT
3. Write: Remove the application reference from the inactive application list
4. TS-Write: Update the management header and modify flags (updated FAT, active FAT, updated App List, active App List)

Instructions for **deleting a file**:

1. TS-Write: Update the management header and modify flags (equal)
2. Write: Set the blocks which shall be deleted to unused `0x0000` in the inactive FAT
3. TS-Write: Remove file pointer from the previous file header or the parent application, if it was the first file
4. TS-Write: Update the management header and modify flags (updated FAT, and active FAT)

6.2.4 Read operations

In order to read a file, the matching application has to be found in the application list. Therefore the application list has to be pursued until the right application is found. When this has been done, the application header can be read and through the pointer to the first file, the first file header can be read. Each file header contains a pointer to the ensuing file, what makes it possible to read one file header after the other until the desired one is reached.

6.2.5 Synchronization of FATs

Before a new transaction can begin, it has to be guaranteed that the file system is in a valid state and that the two FATs, the application lists and the fields of the management block are synchronized.

In order to be sure that the file system is in a synchronous state, the flags *equal*, *FAT0/FAT1 updated*, *FAT0/FAT1 active*, *AppList0/1 active* and *AppList0/1 updated* need to be examined.

The file system state is okay, if the flags are set in the following way: *equal* == 1, *AppList0/1 updated* == 0, and *FAT0/FAT1 updated* == 0.

If the flag *equal* is set to 0, some of the following steps need to be executed:

1. If the flag *FAT0/FAT1 updated* is set to 1, this FAT was updated during the last transaction and also is the active one now. The active FAT needs to be copied to the inactive one and the *FAT0/FAT1 updated* needs to be set to 0.
2. If the flag *AppList0/1 updated* is set to 1, this App list was updated during the last transaction and also is the active one now. The active App List needs to be copied to the inactive one and the *AppList0/1 updated* needs to be set to 0.
3. Finally, if all synchronization steps have been done, the *equal* flag needs to be set to 1 again.

For the whole synchronization, two normal write and one tearing-safe write operations are needed (two normal write operations for copying the FAT and the application list and one

tearing-safe write for flag updates in the management block).

The synchronization is not done during or after the commit, but immediately before a new transaction starts. The flags indicate, which data needs to be copied and synchronized. Before doing any modification to the file system, the status always needs to be checked and synchronization has to be done, if necessary. If this is not done, a new transaction can not start.

Instructions for **synchronization** in case *equal* == 0, *AppList0/1 updated* == 1, *FAT0/FAT1 updated* == 1:

1. Write: Copy the whole updated FAT to the inactive one
2. Write: Copy the updated application list to the inactive one
3. TS-Write: Update the management header, reset flags and set the equal flag

Instructions for **synchronization** in case *equal* == 0, *AppList0/1 updated* == 0, *FAT0/FAT1 updated* == 1:

1. Write: Copy the whole updated FAT to the inactive one
2. TS-Write: Update the management header, reset flags and set the equal flag

6.2.6 Measurement of standard operations

In the subsequent table 6.1, the write operations which are required to execute some standard operations are listed.

standard operations	*best case*	*worst case*
create application	2 TS-Write + 3 Write	
create file	3 TS-Write + 2 Write	3 TS-Write + (n + 1) Write
delete application	2 TS-Write + 2 Write	
delete file	3 TS-Write + 1 Write	
update application header	1 TS-Write	
update file header	1 TS-Write	
write file data (without backup)	1 TS-Write	n TS-Write
write file data (with backup)	1 TS-Write + 2 Write	1 TS-Write + (n + 1) Write
commit	2 TS-Write	(n + 1) TS-Write
synchronization (after modification)	1 TS-Write + 1 Write	1 TS-Write + 2 Write
synchronization (after crash)	2 Write	

Table 6.1: Measurement results of the write operations needed for the standard operations in the FAT approach with additional separate applist

6.3 Examination of the double FAT approach without separate application lists

In this section the file system approach based on two FATs without using separate application lists is explained.

Most of the operations work nearly in the same way as they do for the approach using application lists, which was presented in section 6.2. Only small changes are applied, as the updates to the application list are not needed any more and the management header therefore requires tearing-safe writes.

6.3.1 Create and write operations

As for the previously presented example also for this one, the first step is the check of the management block in order to guarantee that enough space is free, and a file system consistency check needs to be done. If both FATs are equal, everything is okay, otherwise a synchronization has to be performed.

If the equality of the two FATs is guaranteed, and there is enough space left on the card, the write operation can begin.

In order to write an application, the following steps need to be executed:
First of all, the first free block of the file system needs to be found. As all free blocks are characterized through a `0x0000` entry in the FAT, this can be done by starting from the top and traversing all FAT entries until the first free one is found. As already explained in the previous section, also enough adjacent blocks which are required for the key store file, need to be found.

When the free blocks have been found, the transaction can be started. The content of the application can be written into the data region and the blocks need to be marked as used in the inactive FAT.

After this has been done, two tearing-safe writes need to be executed. One is used to link the application to the previous application and the second is used to update several fields and flags in the management block. In the management block, the free block counter needs to be decreased by one, the application counter needs to be increased, the *FAT0/FAT1 updated* flag and the *FAT0/FAT1 active* flag need to be set.

For this write operation of an application, two normal write (write application header, write to FAT) and one to two tearing-safe write operations (update management header and optionally link the application) are needed.

File creation works in the same way as it does for the previous presented file system example.

Instructions for **creating an application**:

1. TS-Write: Update the management header and modify flags (equal)
2. Write: Write the application header
3. Write: Update the inactive FAT
4. TS-Write: Link the application into the linked list of applications, therefore update the previous application header if necessary (otherwise do it together with the next write operation and update the management block if it is the first application in the file system)
5. TS-Write: Update the management header and modify flags (updated FAT, active FAT)

Instructions for **creating a file**:

1. TS-Write: Update the management header and modify flags (equal)
2. *(1 to n)* * Write: Write the file header and initialize the blocks of the data area
3. Write: Update the inactive FAT
4. TS-Write: Store the file reference in the previous file header or the parent application
5. TS-Write: Update the management header and modify flags (updated FAT, and active FAT)

In case of a data write, the affected block or blocks are written anew and need to be linked correctly in the FAT. So the original data is not touched but copied to another block and modified there. This new block then needs to be linked correctly during the commit action, so that a file‘s content stays usable.

Instructions for **writing file data**:

1. TS-Write: Update the management header and modify flags (equal)
2. Depending on the case, different operations need to be executed. If only certain bytes inside a block need to be modified:
   (a) Write: Copy the data block which should be modified to a new, free block
   (b) Write: Write the new data bytes in the new block, where the copied data content was placed
3. If whole data blocks need to be modified:
   (a) *(1 to n)* * Write: Write the new data to n new, free blocks
4. Write: Link the blocks in the inactive FAT, instead of the old ones now the new data blocks are integrated

Instructions for the **commit operation**:

1. *(1 to n)* * TS-Write: Update the file header(s) (update the file pointer if first block is moved and eventually disable the contiguous flag)

2. TS-Write: Update the management header and modify flags (updated FAT, and active FAT)

Instructions for **writing file data (without backup copy)**:

1. *(1 to n)* * TS-Write: Write the new data to the n blocks which should be updated, without security protection. Then also no commit operation is needed afterwards.

6.3.2 Update operations

If an update operation needs to be done, it depends on which kind of object the update should be executed. If a header should be updated, the update is done through a tearing-safe write operation directly in the concerned block.

Writes to the management block as well as linking of new files or applications are done tearing-safe every time.

Instructions for **updating an application or file header**:

1. TS-Write: Update the desired block through a tearing-safe write operation

6.3.3 Delete operations

A delete operation does not modify the file system at all but only updates the FAT accordingly. If a memory block is not used any more and shall be erased, the related FAT entry is set to unused, namely `0x0000`. As all the files which are contained in one application need to be deleted, all the blocks which are used by the files of an application and by the application itself need to be marked as unused in the FAT. This signalizes that the blocks are free and available for further use. Additionally, the *free block counter* needs to be increased. Also the application key store file needs to be deleted, if present. It does not suffice to mark the blocks of the key store file as unused, because the whole content needs to be removed completely and the keys need to be overwritten.

Additionally to the FAT update also the link to this application must be eliminated. So the previous application header or the pointer to the first application in the management block needs to be deleted and substituted by the following application pointer.

The update of the FAT is done with a normal write operation, the update of the management block is done tearing-safe and also the new linking is done tearing-safe. Therefore one normal and two tearing-safe write operations are needed.

If only one single file is deleted, also the reference to this file needs to be eliminated. Therefore either the header of the previous file or the application header (if it only contained one file) needs to be modified tearing-safe. The pointer to the erased file needs to be overwritten, so that the field contains a pointer to the next file or if the application is empty, it contains `0xFFFF`. Therefore one normal and two tearing-safe write operations are needed.

Instructions for **deleting an application**:

1. TS-Write: Update the management header and modify flags (equal)
2. Write: Set the blocks which shall be deleted to unused `0x0000` in the inactive FAT
3. TS-Write: Remove/Replace the application pointer from previous application header or the management block, if it was the first application
4. TS-Write: Update the management header and modify flags (updated FAT, and active FAT)

Instructions for **deleting a file**:

1. TS-Write: Update the management header and modify flags (equal)
2. Write: Set the blocks which shall be deleted to unused `0x0000` in the inactive FAT
3. TS-Write: Remove/Replace file pointer from previous file header or the parent application, if it was the first file
4. TS-Write: Update the management header and modify flags (updated FAT, and active FAT)

6.3.4 Read operation

In order to read a file, the matching application has to be found through the linked list of application headers. Therefore one after another application header has to be perused until the right application is found. When this has been done, the file header can be read. Each file header contains a pointer to the ensuing file, what makes it possible to read one file header after the other until the desired one is reached.

To read the data content of a file or another object which needs more than one block, the FAT is used, because it links one block to another and forms a chain of blocks. The starting block of the content of one file always is known and the FAT simply provides linking to the following one.

6.3.5 Synchronization of FATs

Basically the synchronization is executed nearly in the same way as it is done in section 6.2. The only difference is, that the application list does not need to be copied and synchronized, as there is no application list available in this design. The synchronization is not done during or after the commit, but immediately before a new transaction starts.

The flags indicate, which data needs to be copied and synchronized. Before doing any modification to the file system, the status always needs to be checked and synchronization has to be done, if necessary. If this is not done, a new transaction can not start.

Instructions for **synchronization** in case *equal* == 0, *FAT0/FAT1 updated* == 1:

1. Write: Copy the whole updated FAT to the other one
2. TS-Write: Update the management header, reset the *FAT0/FAT1 updated* flag and set the *equal* flag

In case of transaction failure, also a synchronization needs to happen before any new transaction can be executed. This is necessary in order to get rid of differences which might have occurred during the last transaction. Therefore the following operations need to be done:

1. Write: Copy the whole active FAT to the inactive FAT

6.3.6 Measurement of standard operations

In the subsequent table 6.2, the write operations which are required to execute some standard operations are listed.

standard operations	*best case*	*worst case*
create application	2 TS-Write + 2 Write	3 TS-Write + 2 Write
create file	3 TS-Write + 2 Write	3 TS-Write + (n + 1) Write
delete application	3 TS-Write + 1 Write	
delete file	3 TS-Write + 1 Write	
update application header	1 TS-Write	
update file header	1 TS-Write	
write file data (without backup)	1 TS-Write	n TS-Write
write file data (with backup)	1 TS-Write + 2 Write	1 TS-Write (n + 1) Write
commit	2 TS-Write	(n + 1) TS-Write
synchronization (after modification)	1 TS-Write + 1 Write	
synchronization (after crash)	1 Write	

Table 6.2: Measurement results of the write operations needed for standard operations in the FAT approach

6.4 Examination of the FAT file system with journaling combination

In this section a file system which uses a FAT in combination with journaling is examined in detail. This special composing of two structures makes the whole workflow of the file system complicated and difficult to realize. However the solution guarantees great protection for the file system‘s data and provides consistency.

6.4.1 Standard journaling with FAT

The normal journaling mode basically is explained in section 5.2.1. As already explained, the data is not written directly into the file system but into the journal. After writing to the journal, when the transaction was completed successfully, the new written data is completely available in the journal and can be synced to the actual memory blocks.

Unfortunately, this solution is not applicable as it does not fill the requirements of a modern smartcard. After writing to the card and finishing a transaction, a commit operation is executed. As stated in the requirement section, the commit has to be really fast and should not contain any costly operations.
The presented solution is not practicable, as the commit operation would include one large write operation, what is in contrast to the specified requirements. The whole content of the journal would need to be written to the file system during the commit, what would take far too much time.

Due to the impracticability of the proposed approach, the *normal* journaling mechanism is not investigated any further.

6.4.2 Reverse journaling with FAT

In this section the approach, which is based on the *reverse* journaling that is explained in section 5.2.2, is investigated and explained in detail.

In figure 6.2 the whole workflow of a transaction to the journaling file system can be inspected and is described in the following paragraphs.

6.4.3 Create and write operation

The first step that needs to be done is to check the management block in order to guarantee that enough space is free. Only if there is enough space left on the card, the write operation can begin.

When writing a new block to the file system, a free one needs to be searched in the FAT. As all free blocks are characterized through a `0x0000` entry in the FAT, this can be done by starting from the top and traversing all FAT entries until the first free one is found.

Before starting a transaction, a status check is done. This is basically necessary, in order to guarantee a synchronized state between file system and journal. The requirements and

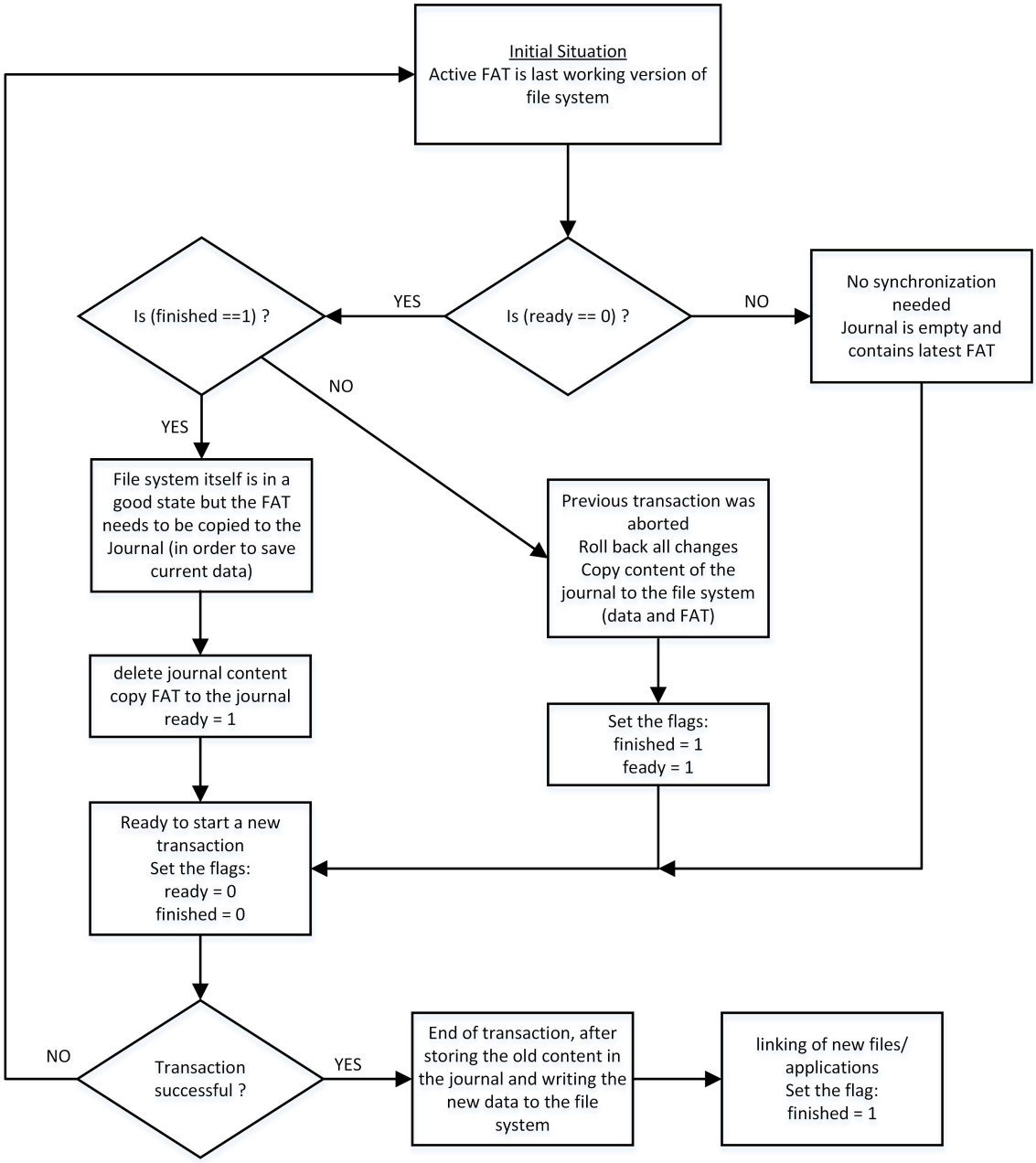

Figure 6.2: Workflow of a transaction in a FAT/reverse journaling combined file system

process of this synchronization is described later on. After this step, when a stable file system status can be guaranteed, the actual write operation starts.

In this journaling mode the upcoming data is directly written into the file system at the desired position. As this approach implicates that the old (valid) data is simply overwritten in case of an update, this data therefore needs to be backed up in the journal.

A backup is necessary in case of a transaction failure, meaning that the new content can not be written completely to the file system and therefore it is left in an inconsistent state. In this case, the old data needs to be available so that the old file system state can be restored.
Additionally to the data blocks also the FAT needs to be secured in the journal, for being able to restore the linking of data blocks and file system structures.

When writing totally new data to an unused block, there is no need to store the old content in the journal, as there is no content which could be stored. Therefore writing new data is basically twice as fast as updating data. When writing new data to unused file system blocks, no damage can be done at all. Only if the new written blocks are linked in the file system, inconsistencies can arise.
After writing new data blocks and also updating the FAT accordingly, they need to be integrated into the file system. This is done at the end of each transaction.
Depending on which kind of object was written, different kinds of actions are required. If an application was written, it needs to be put into the chain of applications: a reference to the application needs to be written into the previous application header or into the management block, if it is the first one in the whole file system.
If a file header was written, it needs to be put into the chain of files of one application: a reference to the file needs to be written into the previous file header or into the application header, if it is the first file of an application.

When everything was written successfully, also the *finished* flag needs to be set. After this has been done, the new data is integrated into the file system and switched to active. In order to use the journal again, as already said, a synchronization step needs to be executed before starting a new transaction.

For the create application command, two normal write operations (write application header, write to the FAT) and three tearing-safe write operations (update management header in the beginning and end and link the application) are needed.

For the create file command at least two normal write (write file header, initialize data blocks, write to FAT) and three tearing-safe write operations (update management header in the beginning and end and link the file) are needed.

Instructions for **creating an application**:

1. TS-Write: Set the ready and finished flag to zero
2. Write: Write the application header
3. Write: Update the FAT
4. TS-Write: Link the application into the linked list of applications, therefore update the previous application header or the management block if it is the first application in the file system
5. TS-Write: Update the management header and set the finished flag

Instructions for **creating a file**:

1. TS-Write: Set the ready and finished flag to zero
2. *(1 to n)* * Write: Write the file header and initialize the blocks of the data area
3. Write: Update the FAT
4. TS-Write: Store the file reference in the previous file header or the parent application
5. TS-Write: Update the management header and set the finished flag

In contrast to the already presented create operations, which only write to unused blocks, the write data operation is more complex. When updating existing data, the content which is going to be replaced has to be secured before overwriting it. The storage of the old data is done via writing it into the journal as one journal entry. A journal entry consists of the block number and the actual data. Just after copying the old data to the journal, the new data can be written directly into the file system. At this point no update of the FAT is necessary, as write data operations are always executed on already linked blocks.

As preliminary to actually writing to the file system the old data needs to be written to the journal, the write operation is basically needed twice and therefore the whole process takes more or less twice as much time.

When a transaction can be finished successfully, the *finished* flag as well as the *ready* flag need to be set. The *ready* flag can instantly be set because no changes were made on the FAT during an update operation.

Instructions for **writing file data**:

1. TS-Write: Set the ready and finished flag to zero (only at transaction start)
2. *(1 to n)* TS-Write: Copy block numbers and data contents of the blocks which are going to be replaced to the journal
3. *(1 to n)* Write: Write the new data to the destined blocks in the file system

Instructions for the **commit operation**:

1. TS-Write: Update the management header and set the finished flag and ready flag

Instructions for **updating file data (without backup)**:

1. *(1 to n)* * TS-Write: Write the new data to the n blocks which should be updated, without security protection through the journal

6.4.4 Update operations

If an update operation needs to be done, it depends on which kind of object the update should be executed. If a header should be updated, the update is done through a tearing-safe write operation directly in the concerned block. No update on the FAT needs to be done and no new linking is required.

Instructions for **updating an application**:

1. TS-Write: Update the application through a tearing-safe write operation

Instructions for **updating a file header**:

1. TS-Write: Update the file header through a tearing-safe write operation

6.4.5 Delete operations

In case a file or an application is not needed any more, the corresponding blocks need to be marked as unused in the FAT and the links to the object need to be removed from the previous file header respectively application header.

When deleting an application, the optional application key store file needs to be deleted permanently from the memory through overwriting its contents.

The update of the FAT is done with a normal write operation and following this, the update of the management block and also the new linking is done tearing-safe during the commit operation.

Instructions for **deleting an application**:

1. TS-Write: Set the ready and finished flag to zero
2. Write: Set the blocks which shall be deleted to unused `0x0000` in the FAT
3. TS-Write: Remove/Replace the application pointer from previous application header or the management block, if it was the first application
4. TS-Write: Update the management header and set the finished flag

Instructions for **deleting a file**:

1. TS-Write: Set the ready and finished flag to zero
2. Write: Set the blocks which shall be deleted to unused `0x0000` in the FAT
3. TS-Write: Remove/Replace the file pointer from previous file header or the parent application if it was the first file
4. TS-Write: Update the management header and set the finished flag

6.4.6 Read operation

Reading is done in the same way as it is done for the previously explained double-FAT approach and can be read in detail in section 6.3.

6.4.7 Synchronization process

Each time a new transaction is started, the file system state has to be checked. The two flags *finished* and *ready* which are situated in the management block give information about the state of the whole system.

The positive *finished* flag means that the last transaction was successful. Additionally, a positive *ready* flag signalizes that the file system is ready for upcoming modifications, the journal is empty and the FAT is in a synchronized state. If only the *ready* flag is not set, this means that the *clean-up/update* of the journal was interrupted.

When everything is okay, the new updated FAT needs to be copied to the journal in order to keep it up-to-date before a new transaction can start. Also the rest of the journal content needs to be deleted. This is necessary for having a new and current journal available and to be ready for upcoming operations.

In case the transaction fails, nothing is changed in the management block and the flags *finished* and *ready* remain unchanged and still are set to `0`. This means that a roll-back needs to be performed and the old data which was securely stored in the journal needs to be copied back to the file system. The secured FAT which is available in the journal needs to be synced with the file system and of course all data blocks need to be copied back to their original position.

Instructions for **synchronization** in case *finished == 1* and *ready == 0*:

1. Write: Copy the whole FAT to the journal
2. TS-Write: Update the management header and set the ready flag and the journal to empty

Instructions for **synchronization** in case *finished == 0* and *ready == 0*:

1. *n* * Write: Copy the data content of one journal entry after the other back to the correct block in file system
2. Write: Copy the FAT from the journal to the file system
3. TS-Write: Update the management header and set the finished flag, the ready flag and set the journal to empty

6.4.8 Measurement of standard operations

In the subsequent table 6.3, the write operations which are required to execute some standard operations are listed.

standard operations	*best case*	*worst case*
create application	2 TS-Write + 2 Write	3 TS-Write + 2 Write
create file	3 TS-Write + 2 Write	3 TS-Write + (n + 1) Write
delete application	2 TS-Write + 1 Write	3 TS-Write + 1 Write
delete file	3 TS-Write + 1 Write	
update application header	1 TS-Write	
update file header	1 TS-Write	
write file data (without backup)	1 TS-Write	n TS-Write
write file data (with backup)	2 TS-Write + 1 Write	(n + 1) TS-Write + n Write
commit	1 TS-Write	
synchronization	1 TS-Write + 1 Write	1 TS-Write + (n + 1) Write

Table 6.3: Measurement results of the write operations needed for the standard operations in the FAT/journaling approach

6.5 Examination of the EXT file system approach

This section dedicates itself to the extended file system design proposal which was discussed in section 5.4.

The fundamental decision that needs to be made when using an EXT file system is, if an application list in a separate memory block should be used or not. Depending on that, two different designs of the later used application node (ANode) are possible.

When using an application list, the ANode has two bytes more of empty space which can be used for an additional file list pointer, which makes it possible to access all 32 files directly from the ANode. When no application list is used, only one file list pointer can be used and the file list needs to be linked internally. The information regarding whether an application list is used or not can be found in the superblock, in the *Config* field.

The following detailed explanation of the workflow of the file system is done for the version which does not use a separate application list.

6.5.1 Create and write operations

The first step that needs to be done is to check the superblock in order to guarantee that enough space is free. Only if there is enough space left on the card, the write operation can start.

When writing a new block to the file system, a free one needs to be found in the file system. Therefore the bitmap needs to be traversed, starting from the beginning. All free blocks are characterized through a `0`.

Before starting a transaction, a status check is done. This is basically necessary, in order

to guarantee a synchronized state between file system and journal. The requirements and process of this synchronization is described later on. After this step, when a stable file system status can be guaranteed, the actual write operation can start.

In this journaling mode the upcoming data is directly written into the file system at the desired position, as already explained in the example of the FAT and journaling combination, that was presented in section 6.4. As this approach implicates that the old (valid) data is simply overwritten in case of an update, this data therefore needs to be backed up in the journal beforehand. Additionally to the data blocks also their original storage position needs to be secured in the journal, for being able to restore the linking of data blocks and file system structures.

When creating a new file, it is desirable to allocate contiguous blocks for the data content. If this is possible, no data block list is necessary and only a pointer to the first data block needs to be written into the file header.

When writing totally new data to an unused block, there is no need to store the old content in the journal, as there is no content which could be stored. Therefore writing new data is basically twice as fast as updating data. When only writing new data to unused file system blocks, no damage can be done at all. Only if the new written blocks are linked in the file system, inconsistencies can arise. After writing new data blocks and also updating the bitmap accordingly, they need to be integrated into the file system.

When everything was written successfully, also the *finished* flag needs to be set. After this has been done, the new data is integrated into the file system.
In order to use the journal again, as already said, a synchronization step needs to be executed before starting a new transaction.

Instructions for **creating an application**:

1. TS-Write: Set the ready and finished flag to zero
2. Write: Write the application header
3. Write: Update the bitmap
4. TS-Write: Link the application into the linked list of applications, therefore update the previous application header or the management block if it is the first application in the file system
5. TS-Write: Update the management header and set the finished flag

Instructions for **creating a file**:

1. TS-Write: Set the ready and finished flag to zero
2. Write: Write the file header (and eventually the data block list)
3. *(1 to n)* * Write: Initialize the blocks of the data area
4. Write: Update the bitmap
5. TS-Write: Link the file header into the linked list of files, therefore update the previous file header or the parent application header if it is the first file
6. TS-Write: Update the management header and set the finished flag

Also for the EXT based file system the write data operation is complex. In order to secure the old data, which is going to be overwritten, it is written to the journal in the form of a journal entry. A journal entry consists of the reference to the original block and the actual (old) data.

Just after copying the old data to the journal, the new data can be written directly into the file system. At this point no update of the bitmap is necessary, as write data operations are always executed on already reserved and linked blocks.

As preliminary to actually writing to the file system the old data needs to be written to the journal, the write operation is basically needed twice and therefore the whole process takes twice as much time.

When a transaction can be finished successfully, the *finished* flag as well as the *ready* flag need to be set. The *ready* flag can instantly be set because no changes were made on the FAT during an update operation.

Instructions for **writing file data**:

1. TS-Write: Set the ready and finished flag to zero (only at transaction start)
2. *(1 to n)* * TS-Write: Copy block pointers and data contents of the blocks which are going to be replaced to the journal
3. *(1 to n)* * Write: Write the new data to the destined blocks in the file system

Instructions for the **commit operation** :

1. TS-Write: Update the management header and set the finished flag and ready flag

Instructions for **writing file data (without backup)**:

1. *(1 to n)* * TS-Write: Write the new data to the destined blocks in the file system without security protection through the journal

6.5.2 Update operations

If an update operation needs to be done, it depends on which kind of object the update should be executed. If a header should be updated, the update is done through a tearing-safe write operation directly in the concerned block. No update on the FAT needs to be done and no new linking is required.

Instructions for **updating an application or file header**:

1. TS-Write: Update the desired block through a tearing-safe write operation

6.5.3 Delete operations

As already sufficiently explained in previous sections, a delete operation does not delete the content of the file system (except of the security relevant data) but changes the management structure, which is in this approach the block usage bitmap, and the linking of data blocks.

In case a file or an application is not needed any more, the corresponding blocks need to be marked as unused in the bitmap and the links to the object need to be removed from the previous file header respectively application header.

The update of the bitmap is done with a normal write operation and following this, the update of the management block and also the new linking is done tearing-safe.

Instructions for **deleting an application**:

1. TS-Write: Set the ready and finished flag to zero
2. Write: Set the blocks which shall be deleted to unused `0` in the block usage bitmap
3. TS-Write: Remove/Replace the application pointer from previous application header or the management block, if it was the first application
4. TS-Write: Update the management block and set the finished flag

Instructions for **deleting a file**:

1. TS-Write: Set the ready and finished flag to zero
2. Write: Set the blocks which shall be deleted to unused `0` in the block usage bitmap
3. TS-Write: Remove the file pointer from the file list in the parent application
4. TS-Write: Update the management block and set the finished flag

6.5.4 Synchronization process

The synchronization is done immediately before a new transaction starts. The flags indicate, which data needs to be copied and synchronized. Before doing any modification to the file system, the status always needs to be checked and a synchronization has to be done, if necessary. If this is not done, a new transaction can not start because it would lead to an inconsistent state and the data content would not be readable any more.

Instructions for **synchronization** in case *finished* == 1 and *ready* == 0:

1. Write: Copy the whole active bitmap to the inactive one (bitmap 2 in the journal region)
2. TS-Write: Update the management block, specify that the journal is empty and set the ready flag

Instructions for **synchronization** in case *finished == 0* and *ready == 0*:

1. *(1 to n)* * Write: Copy the data content of one journal entry after the other back to the correct block in file system
2. Write: Copy the FAT from the journal to the file system
3. TS-Write: Update the management block and set the finished flag, the ready flag and set the journal to empty

A detailed example of how a transaction can be executed and which synchronization steps need to be done can be inspected in the figure 6.3.

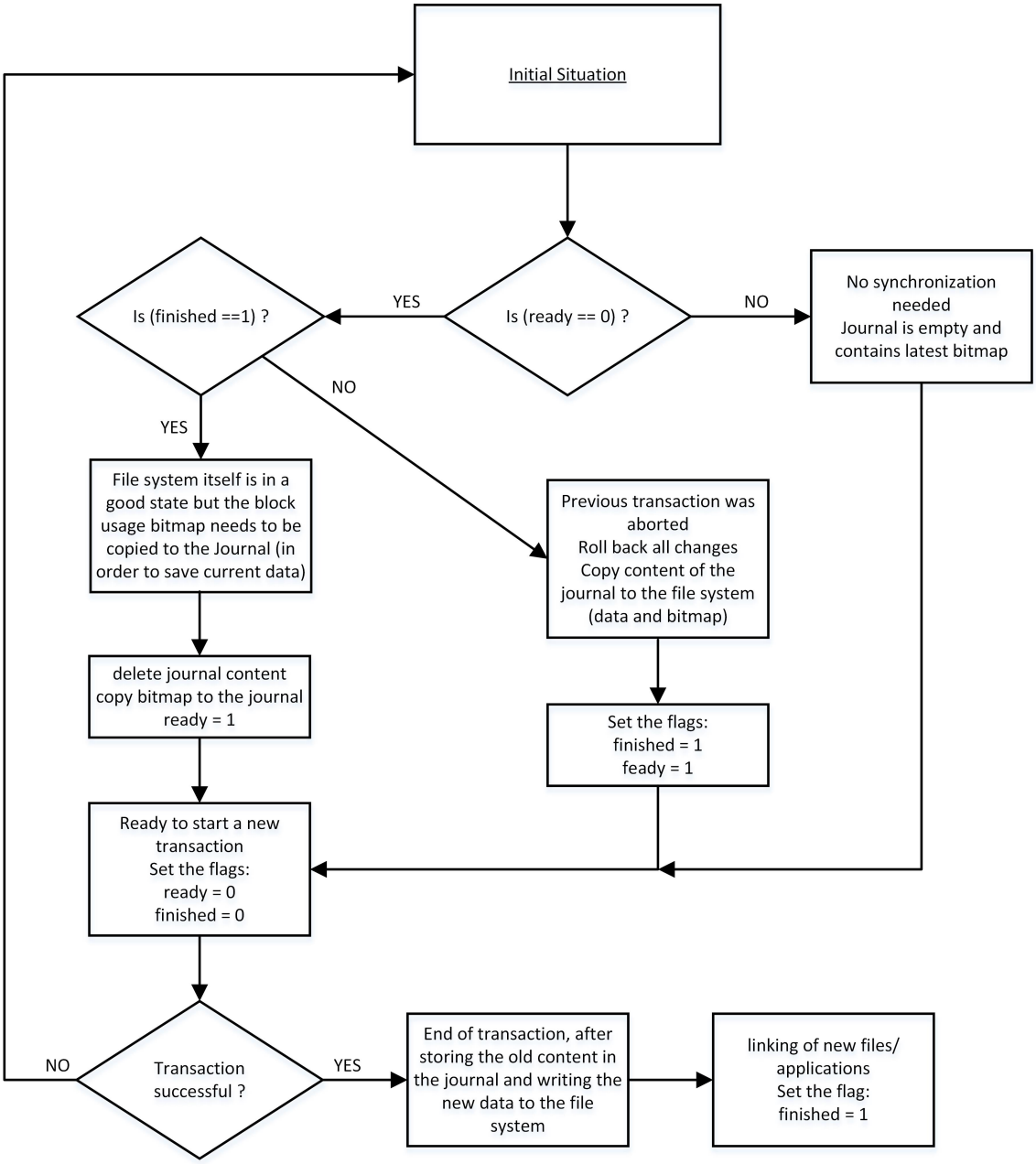

Figure 6.3: Workflow of a transaction in an EXT based file system

6.5.5 Measurement of standard operations

In the subsequent table 6.4, the write operations which are required to execute some standard operations are listed.

standard operations	*best case*	*worst case*
create application	2 TS-Write + 2 Write	3 TS-Write + 2 Write
create file	3 TS-Write + 2 Write	3 TS-Write + (n + 2) Write
delete application	2 TS-Write + 1 Write	3 TS-Write + 1 Write
delete file	3 TS-Write + 1 Write	
update application header	1 TS-Write	
update file header	1 TS-Write	
write file data (without backup)	1 TS-Write	n TS-Write
write file data (with backup)	2 TS-Write + 1 Write	(n + 1) TS-Write + n Write
commit	1 TS-Write	
synchronization	1 TS-Write + 1 Write	1 TS-Write + (n + 1) Write

Table 6.4: Measurement results of the write operations needed for the standard operations in the EXT approach

6.6 Examination of the YAFFS file system approach

This section explains the YAFFS file system design more precisely and presents the measurement results regarding number of write operations and performance.

6.6.1 Create and write operations

As this file system design does not use a management block or a similar management structure, the status of the file system has to be determined via iterating through the whole file system‘s log and reconstructing its state. The whole checking-process is described in a later following subsection.

In order to write a new block to the file system, an empty block has to be found in the block usage bitmap. All free blocks are characterized through a `0` in the bitmap. Depending on which kind of data is written, it may be the case, that multiple blocks are needed for one object.

When writing new data to the file system to a free memory chunk, the data simply is written to the data section of the chunk. After doing that, also the corresponding tag needs to be updated. The fields *Chunk ID*, *Bytes used*, and *Tag Config* need to be set accordingly. Also the pointer fields need to be updated in order to mark connections between single chunks.

At the end of one transaction the new written chunks are integrated in the file system:

a new application is linked to the previous application and a new file is linked to the previous file. In order to do this, the tag section of the chunk which needs to include a pointer to the new object needs to be modified tearing-safe. Additionally to this also the block usage bitmap needs to be updated accordingly in order to keep track of used and unused memory chunks.

A detailed overview of the exact work flow of a write operation can be seen in figure 6.4.

Instructions for **creating an application**:

1. TS-Write: Write application to the data section of the selected chunk and update its tag section
2. TS-Write: Update the bitmap
3. TS-Write: Link the application into the linked list of applications, therefore update the previous application header

Instructions for **creating a file**:

1. *(1 to n)* * TS-Write: Write the header to the data section of the selected chunk and update its tag section, also allocate the chunks for the data blocks and update their tag sections
2. TS-Write: Update the bitmap
3. TS-Write: Link the file header into the linked list of files, therefore update the previous file header or the parent application if it is the first file

When writing file data, the data section is rewritten and updated to a new, empty chunk. Also the tag is written to the new tag section and its fields are updated accordingly. Everything needs to be written tearing-safe because the file system check needs to be able to read through every chunk.

Instructions for **writing file data**:

1. *(1 to n)* * TS-Write: Copy the chunk(s) to a new empty chunk
2. *(1 to n)* * TS-Write: Do the data update in this copied chunks and also update the tag sections, increase the serial number and maybe set the pointers anew
3. TS-Write: Update the pointer of the previous chunk so that the new updated chunk(s) is integrated in the file system
4. *(1 to n)* * TS-Write: Update the tag section of the old chunks, set the deletion marker to deleted in order to signalize that the old chunks are not used any more
5. TS-Write: Update the bitmap

Instructions for **writing file data (without backup)**:

1. *(1 to n)* * TS-Write: Update the data directly in the original chunk and update its tag sections, increase the serial number

6.6.2 Update operations

The update operation is more complex than the write operation, as it overwrites respectively re-writes data.
Updates of any object that is no file data are always done tearing-safe directly into the containing chunk. Also updates to the tag region are always done in a tearing-safe way. The only update that is done in another way is the update of file data.

Instructions for **updating an application or file header**:

1. TS-Write: Update the desired chunk through a tearing-safe write operation

6.6.3 Delete operations

In case a file or an application is not needed any more, the corresponding blocks need to be marked as unused in the bitmap and the links to the object need to be removed from the previous file header respectively application header. Also the deletion marker of the tag section of all chunks which shall be deleted need to be set to *Deleted*.

Instructions for **deleting an application header**:

1. TS-Write: Update the bitmap: Set the block which shall be deleted to `0` in the block usage bitmap
2. TS-Write: Remove/Replace the application pointer from previous application header
3. *(1 to n)* * TS-Write: Set the deletion markers of the chunks which are going to be deleted to *Deleted*

Instructions for **deleting a file**:

1. TS-Write: Update the bitmap: Set the blocks which shall be deleted to `0` in the block usage bitmap
2. TS-Write: Remove/Replace the file pointer from the previous file header or from the application header if it is the first file
3. *(1 to n)* * TS-Write: Set the deletion markers of the chunks which are going to be deleted to *Deleted*

In case the delete operation can not be finished, a system-check needs to continue the deletion before a new transaction starts. When the operation already fails before updating the bitmap, nothing is changed at all and the whole transaction needs to be done again.

Otherwise, if the update of the bitmap was finished and the operation failed later on, the deletion needs to be continued, what can be done in the following way:
As the bitmap is always updated tearing-safe, it represents the valid state of the file system at any time and therefore each block that is marked as unused in the bitmap can never be allocated or used in the file system. This means that by iterating through the bitmap and comparing the block usage with the entry in the tag section of the chunk, each mismatch can be found and removed. All deletion markers of unused chunks need to be set to *Deleted* and also all references to unused chunks need to be superseded.

6.6.4 Read operation

Reading a file works through finding the parent application in the linked list of applications and from there on having access to all its files. When the right application is found, all files which belong to it can be read by reading one file header after the other, because each file header points to the next one.

In order to read the data content of a file, the pointer to the data which is stored in the file header needs to be read. This pointer directs to the first data chunk. If more chunks are needed for the content of the object, the next ones can be reached through the tag section of one chunk.

6.6.5 File system check

The file system check needs to be done before a new transaction can be started. The block usage bitmap is the main reference for examining the file system status. The bitmap is always updated tearing-safe and timely after a data write or before a data deletion. If a transaction fails, the bitmap nevertheless reflects the right, valid file system state.

Chunks which are marked as unused in the bitmap are also not used in the file system. If a pointer points to a block which is marked as unused in the bitmap, it needs to be erased from the whole file system. It needs to be excluded from the linked pointer lists (applications and files) and also marked as deleted in the deletion marker.

A mismatch of the entry in the bitmap and the chunk‘s deletion marker can arise, when a deletion happened and it could not be finished. Then the leftovers need to be eliminated from the file system. Another possible reason is the creation of a new object or the allocation of a new block and its incomplete integration into the file system. A new block could have been written already, but not linked correctly, what means that it needs to be erased as no relation to other objects can be determined at all.

Instructions for the **file system check**:

1. Go through the whole bitmap and compare the block usage with the deletion marker of the according chunk in the file system
2. If they reflect the same usage, nothing has to be done. This is the case when the bitmap entry is set to `0` and the corresponding chunk‘s deletion marker is set to *Empty* or when the bitmap entry is set to `1` and the corresponding chunk‘s deletion marker is set to *Used*.
3. If the bitmap entry is set to `0` and the corresponding chunk‘s deletion marker is set to *Deleted*:
   (a) Remove the content of the data section
   (b) Reset the tag section and set the deletion marker to *Empty* in order to be reused again
4. If the bitmap entry is set to `0` and the corresponding chunk‘s deletion marker is set to *Used*:
   (a) If necessary, remove all references to this field from the file system. Exclude it from an application/file linking.
   (b) Set the deletion marker to *Deleted*

In the worst case, the file system check needs n tearing-safe write operations to set the deletion marker to a correct value and additionally n tearing-safe write operations to update applications or file headers.

6.6.6 Measurement of standard operations

In the subsequent table 6.5, the write operations which are required to execute some standard operations are listed.

standard operations	*best case*	*worst case*
create application	3 TS-Write	
create file	3 TS-Write	(n + 2) TS-Write
delete application	3 TS-Write	(n + 2) TS-Write
delete file	3 TS-Write	(n + 2) TS-Write
update application header	1 TS-Write	
update file header	1 TS-Write	
write file data (without backup)	1 TS-Write	n TS-Write
write file data (with backup)	5 TS-Write	(3n + 2) TS-Write
synchronization	2 TS-Write	2n TS-Write

Table 6.5: Measurement results of the write operations needed for the standard operations in the YAFFS based file system design

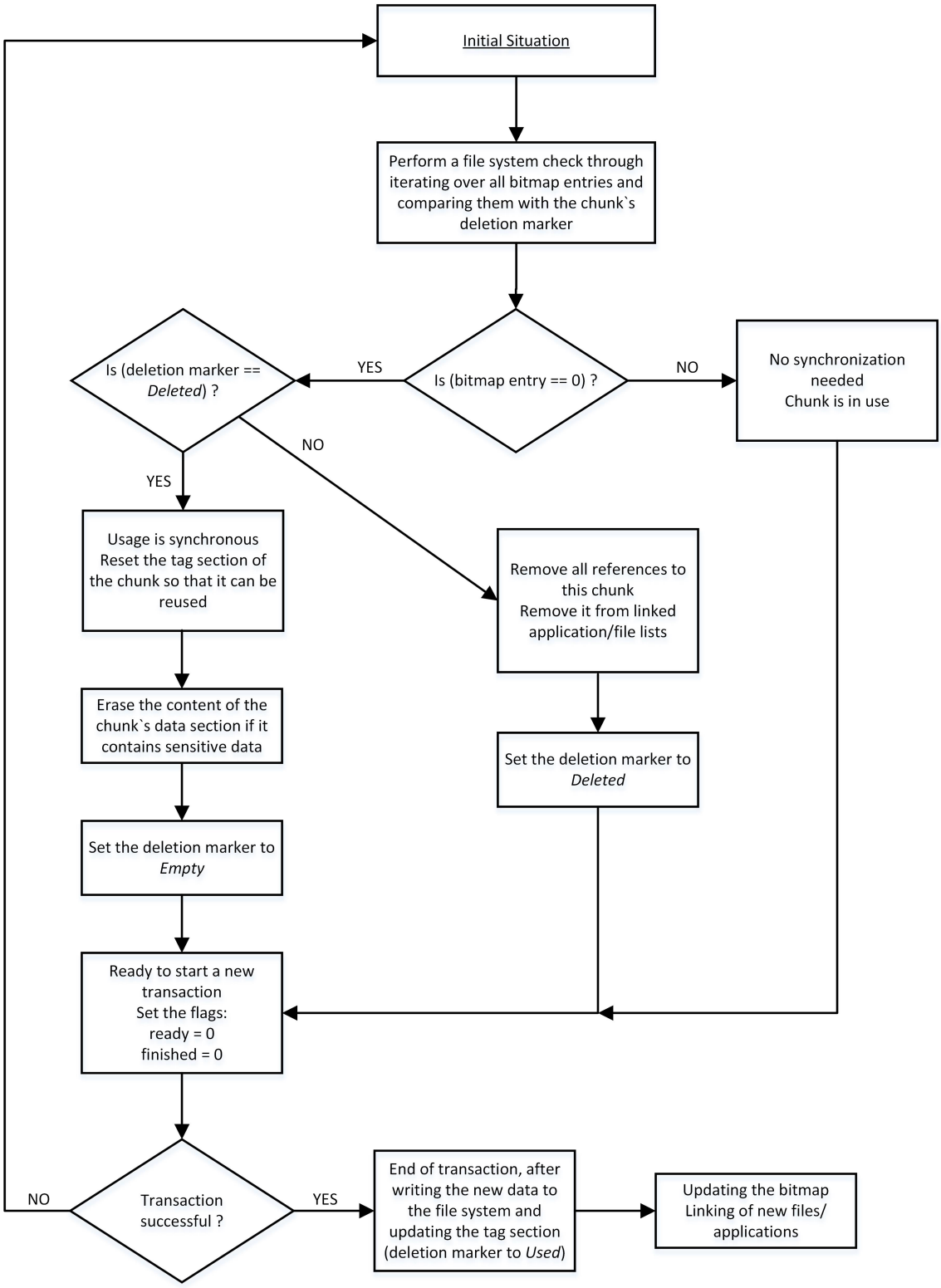

Figure 6.4: Workflow of a transaction in the YAFFS file system design

Chapter 7

Evaluation of Results and Comparison with the Reference File System

This chapter examines the results of the detailed investigation of the proposed file system approaches which were made in chapter 6 and compares them to the reference file system.

First of all the reference file system is described in detail. Therefore the workflow of some standard operations is explained and visualized, and also the requirements for a new file system are recapitulated shortly. In order to provide a basis for comparison, the number of access operations is measured for standard use cases as it has been done for all other file system proposals in the previous chapter.

7.1 Detailed analysis of the reference file system

This file system is, as already mentioned shortly in chapter 1, realized in form of a linked list conforming to the *ISO 7816-4* standard. The two main components of the file system are applications and files.

Applications are structured as a linked list and chained to each other.
Files are also structured as a linked list and the reference to this file list is stored in the corresponding application node. The file size always is a multiple of the block size and different kinds of files are supported.

A picture of the file system structure is depicted in figure 7.1.

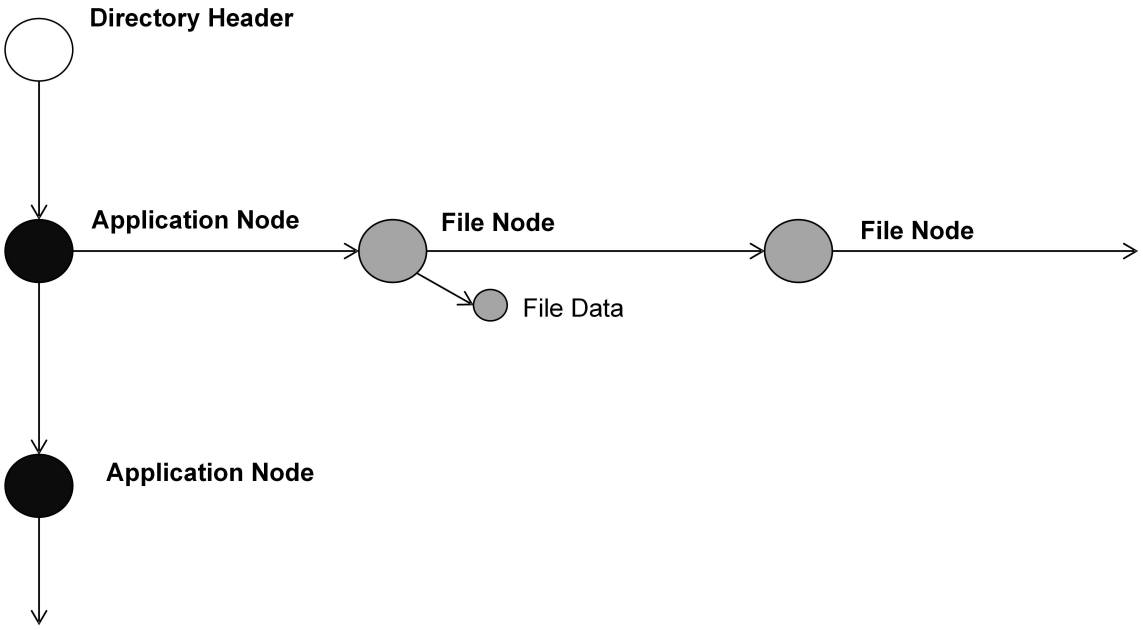

Figure 7.1: Layout of the reference file system

The reference file system supports two different ways how data can be written to the memory

- Write data without security protection
- Write data with a security protection after making a secure copy or guaranteeing the recoverability of the old content (shadow image needed for this approach)

The secure way of writing data makes use of a shadow image. This shadow image is needed for guaranteeing data integrity in case of tearing. Basically using a shadow image means that twice the space is needed because critical data needs to be copied to a second identical-sized memory location, called the shadow image. All updates or write operations are executed on this inactive shadow image. Only if the operation succeeds, data validity can be ensured and the shadow image can be switched to active.

This security mechanism on the one hand provides perfect data security, but on the other hand is kind of wasting memory, as really much data needs to be copied and therefore is available twice on the card.

The main point which needs to be improved at the reference solution is the handling of deleted memory. After file or application deletion the memory which becomes available is not really erased and released but only marked as unused. These unused memory blocks can not be reused at the moment and their space is wasted.

Emerging of these facts, the requirements for the new file system design emerge. Highest

importance has the re-usability of deleted memory chunks. Another important point is to minimize the write operations. As write operations are the most expensive operations which can be performed, they have to be used carefully and restricted. Also the development of a new data backup mechanism, which consumes less memory than the reference file system, is desirable for a future design.

In the following sections the performance and functionality of the standard operations when applied to the file system is measured and explained clearly.

Create an application (visualized in figure 7.2)

- Write: Write the ANode
- Depending on the file system state: If the ANode is the first one in the whole system:
  - TS-Write: Link the ANode to the previous ANode, therefore update the previous ANode header
- TS-Write: Write the configuration

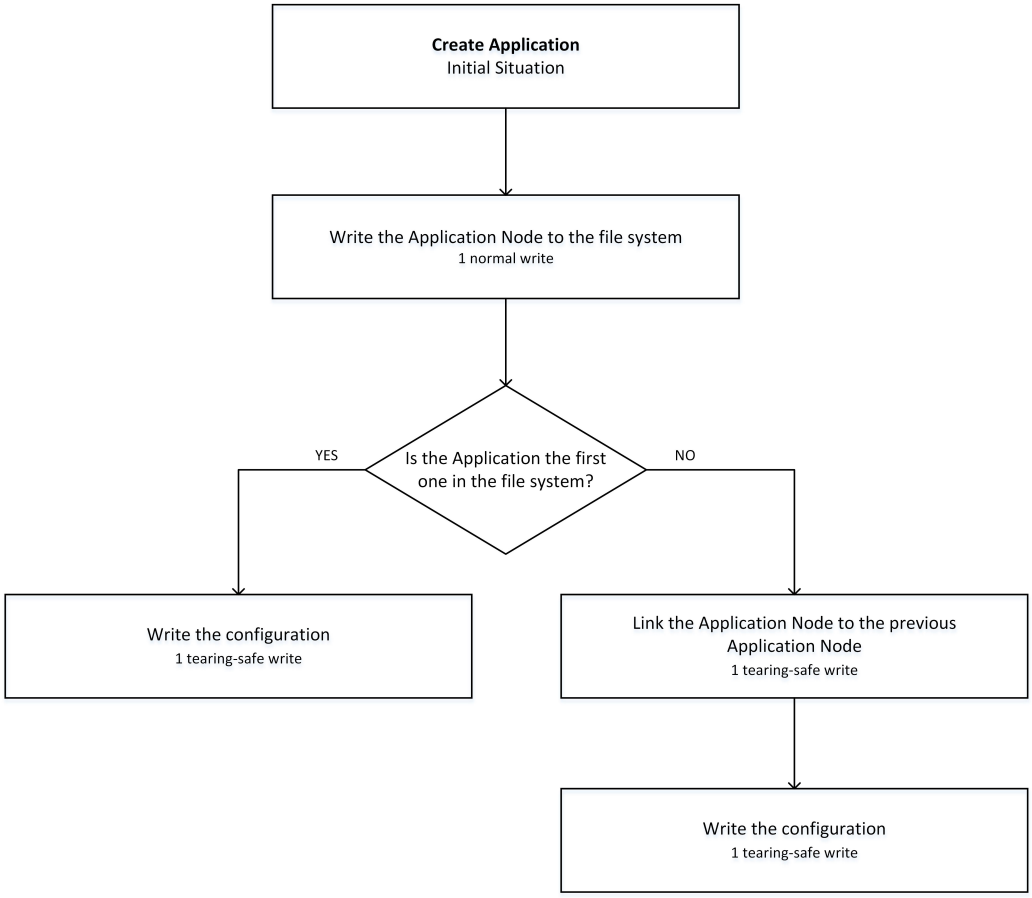

Figure 7.2: The reference *Create Application* operation

Create a file (visualized in figure 7.3)

- Write: Write the FNode
- TS-Write: Update the FNode reference in the ANode if the FNode was the first one or link it to the previous FNode
- TS-Write: Write the configuration

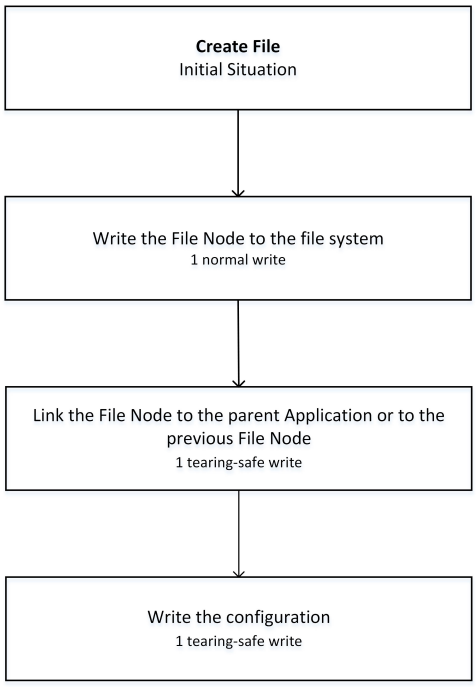

Figure 7.3: The reference *Create File* operation

Delete an application (visualized in figure 7.4)

- TS-Write: Mark the application as unused, therefore set a flag in the *Config* field of the application header to false

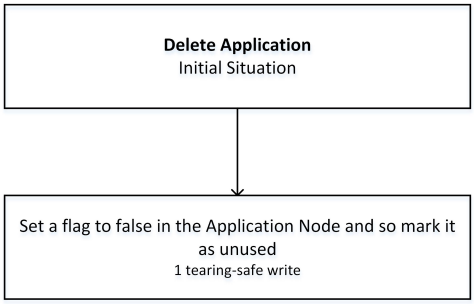

Figure 7.4: The reference *Delete Application* operation

Delete a file: (visualized in figure 7.5)

- TS-Write: Mark the file as unused, therefore set a flag in the *Config* field of the file header to false

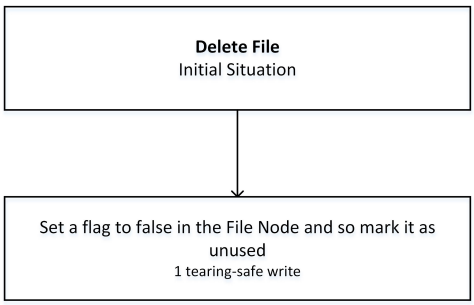

Figure 7.5: The reference *Delete File* operation

Write data (without backup):

- TS-Write: Write data to the desired block(s)

Write data (with backup):

- Write: Copy the content of the active image to the inactive image in order to synchronize the content
- Write: Write the new data to the inactive image

Commit:
The command *Commit Transaction* validates all write access of the ongoing transaction

- TS-Write: Update some flags in the ANode

The write operations as well as the following commit operation are visualized in figure 7.6.

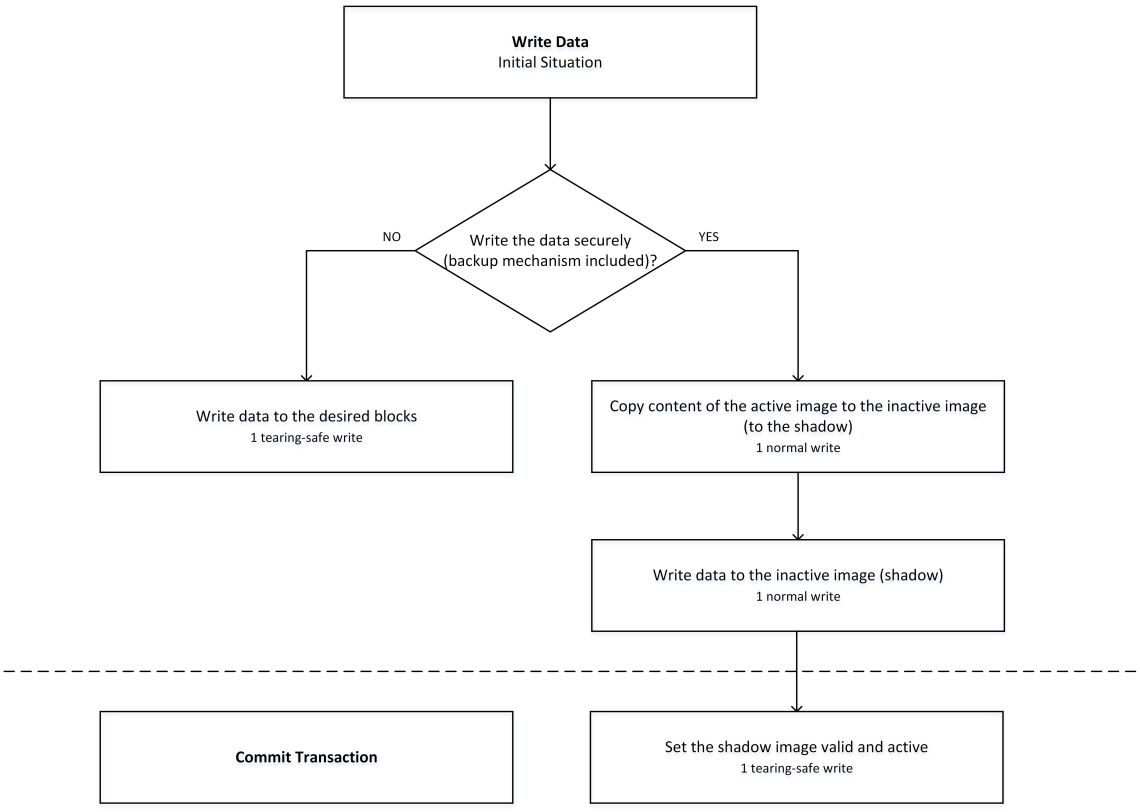

Figure 7.6: The reference *Write Data* operation

Measurement of standard operations

In the subsequent table 7.1, the write operations which are required to execute some standard operations are listed.

reference standard operations	*best case*	*worst case*
create application	1 TS-Write + 1 Write	2 TS-Write + 1 Write
create file	2 TS-Write + 1 Write	
delete application	1 TS-Write	
delete file	1 TS-Write	
update application header	1 TS-Write	
update file header	1 TS-Write	
write data (without backup)	1 TS-Write	
write data	2 Write	
commit	1 TS-Write	

Table 7.1: Measurement results of the write operations needed for the reference file system standard operations

7.2 Evaluation of measurement results

In the following tables the write operations which are required from the different file system designs are compared. Table 7.2 shows the amount of write operations which are needed in the best case, table 7.3 shows how many of them are needed in the worst case.

best case comparison	*reference file system*	*double FAT with application lists*	*double FAT without application lists*	*FAT with journaling*	*EXT design*	*YAFFS design*
create application	1 TS + 1 N	2 TS + 3 N	2 TS + 2 N	2 TS + 2 N	2 TS + 2 N	3 TS
create file	2 TS + 1 N	3 TS + 2 N	3 TS + 2 N	3 TS + 2 N	3 TS + 2 N	3 TS
delete application	1 TS	2 TS + 2 N	3 TS + 1 N	2 TS + 1 N	2 TS + 1 N	3 TS
delete file	1 TS	3 TS + 1 N	3 TS + 1 N	3 TS + 1 N	3 TS + 1 N	3 TS
update application header	1 TS	1 TS	1 TS	1 TS	1 TS	1 TS
update file header	1 TS	1 TS	1 TS	1 TS	1 TS	1 TS
write file data (without backup)	1 TS	1 TS	1 TS	1 TS	1 TS	1 TS
write file data	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
commit	1 TS	2 TS	2 TS	1 TS	1 TS	
synchronization of file system		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
sum	9 TS + 4 N	17 TS + 11 N	18 TS + 9 N	17 TS + 8 N	17 TS + 8 N	22 TS

Table 7.2: Comparison of write operations needed in the best case scenario in the different file system design approaches

worst case comparison	*reference file system*	*double FAT with application lists*	*double FAT without application lists*	*FAT with journaling*	*EXT design*	*YAFFS design*
create application	2 TS + 1 N	2 TS + 3 N	3 TS + 2 N	3 TS + 2 N	3 TS + 2 N	3 TS
create file	2 TS + 1 N	3 TS + (n + 1) N	3 TS + (n + 1) N	3 TS + (n + 2) N	3 TS + (n + 2) N	(n + 2) TS
delete application	1 TS	2 TS + 2 N	3 TS + 1 N	3 TS + 1 N	3 TS + 1 N	(n + 2) TS
delete file	1 TS	3 TS + 1 N	3 TS + 1 N	3 TS + 1 N	3 TS + 1 N	(n + 2) TS
update application header	1 TS	1 TS	1 TS	1 TS	1 TS	1 TS
update file header	1 TS	1 TS	1 TS	1 TS	1 TS	1 TS
write file data (without backup)	1 TS	n TS	n TS	n TS	n TS	n TS
write file data	2 N	1 TS + (n + 1) N	1 TS + (n + 1) N	(n + 1) TS + n N	(n + 1) TS + n N	(3n + 2) TS
commit	1 TS	(n + 1) TS	(n + 1) TS	1 TS	1 TS	
synchronization of file system		1 TS + 2 N	1 TS + 1 N	1 TS + (n + 1) N	1 TS + (n + 1) N	2n TS
sum	10 TS + 4 N	(2n + 15) TS + (2n + 9) N	(2n + 17) TS + (2n + 7) N	(2n + 17) TS + (3n + 7) N	(2n + 17) TS + (3n + 7) N	(9n + 13) TS

Table 7.3: Comparison of write operations needed in the worst case scenario in the different file system design approaches

7.3 Practical utilization of the proposed file system designs on the basis of reference transactions

In order to be able to compare the suggested designs and to draw a conclusion from the different possibilities, reference transactions are used. A reference transaction can be carried out by each file system candidate and is used to show the number of needed write operations and the consequential performance. After utilization of the transactions and collection of all results, the numbers can easily be compared and a potential final file system candidate can possibly be figured out.

In order to get informative results, two reference transaction are carried out.
The first one, further on called Example 1, is a fictional one, which has been designed to reflect the *Write* and *Update* operations and their impact on the overall performance in the best possible way.
The second reference transaction, Example 2, is a real one which is normally used for ticketing.
The sequence of operations of the first reference transaction is depicted in figure 7.7 and the second example can be seen in figure 7.8.

The focus of both examples lies in the relevant operations *Read Data* and *Write Data*. The other types of operations, like *Create Application*, *Create File*, *Delete File* and *Delete Application* are negligible because they are not executed in the electric field (e.g. only at card personalization) and therefore are not directly transaction-relevant.

The assumption that is made for both examples is that all files do already contain data, so all write operations that are carried out in fact need to be performed as an update operation.

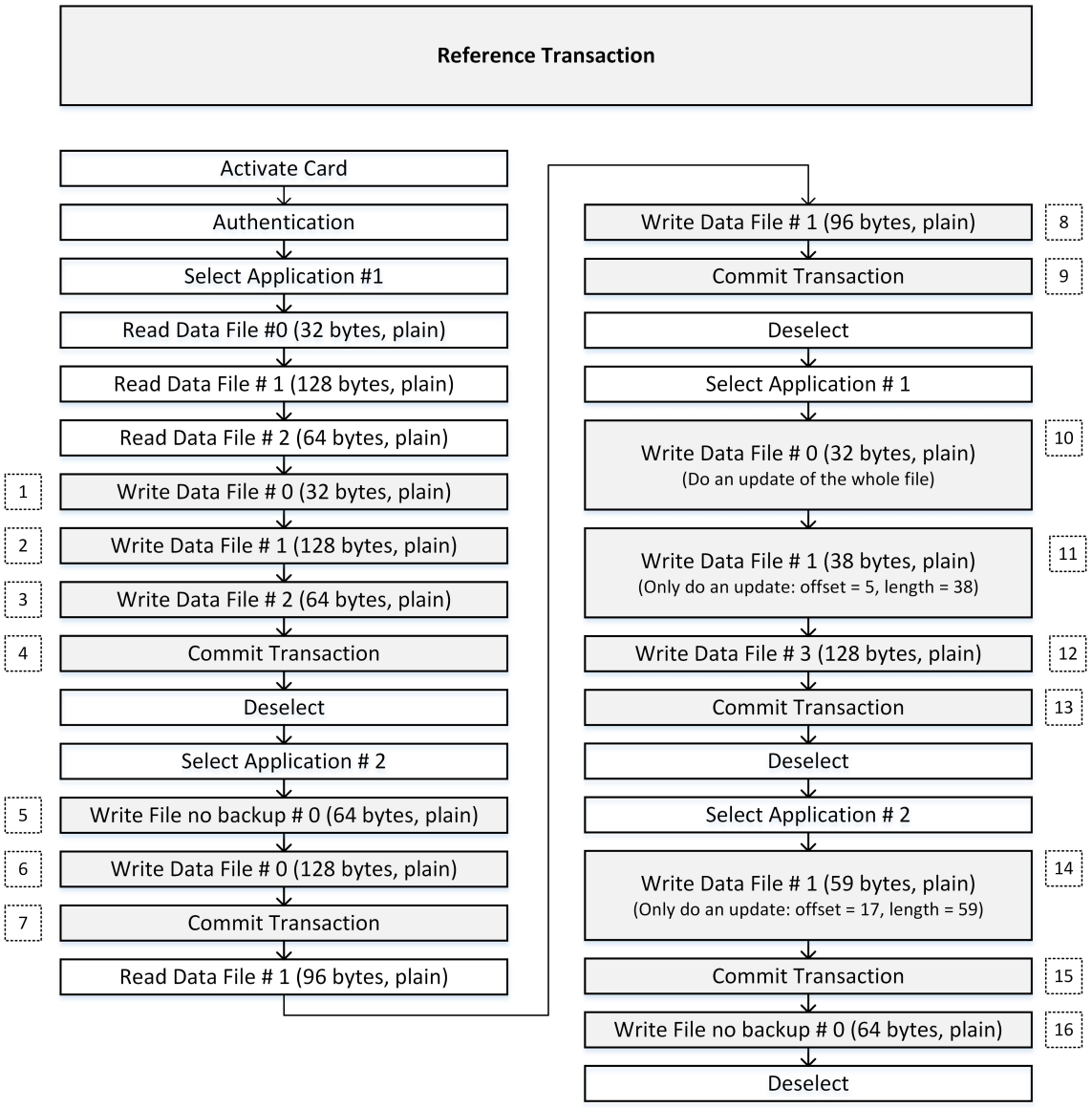

Figure 7.7: *Example 1* - An exemplary reference transaction

	reference file system	*double FAT with applist*	*double FAT*	*FAT with journaling*	*EXT design*	*YAFFS design*
1	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
2	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
3	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
4	1 TS	4 TS	4 TS	1 TS	1 TS	
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
5	1 TS	1 TS	1 TS	1 TS	1 TS	1 TS
6	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
7	1 TS	2 TS	2 TS	1 TS	1 TS	
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
8	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
9	1 TS	2 TS	2 TS	1 TS	1 TS	
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
10	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
11	2 N	1 TS + 3 N	1 TS + 3 N	2 TS + 1 N	2 TS + 1 N	5 TS
12	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
13	1 TS	4 TS	4 TS	1 TS	1 TS	
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
14	2 N	1 TS + 3 N	1 TS + 3 N	2 TS + 1 N	2 TS + 1 N	5 TS
15	1 TS	2 TS	2 TS	1 TS	1 TS	
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
16	1 TS	1 TS	1 TS	1 TS	1 TS	1 TS
sum	18 N + 7 TS	25 N + 30 TS	25 N + 30 TS	14 N + 30 TS	14 N + 30 TS	57 TS

Table 7.4: Measurement results of the reference transaction *Example 1*

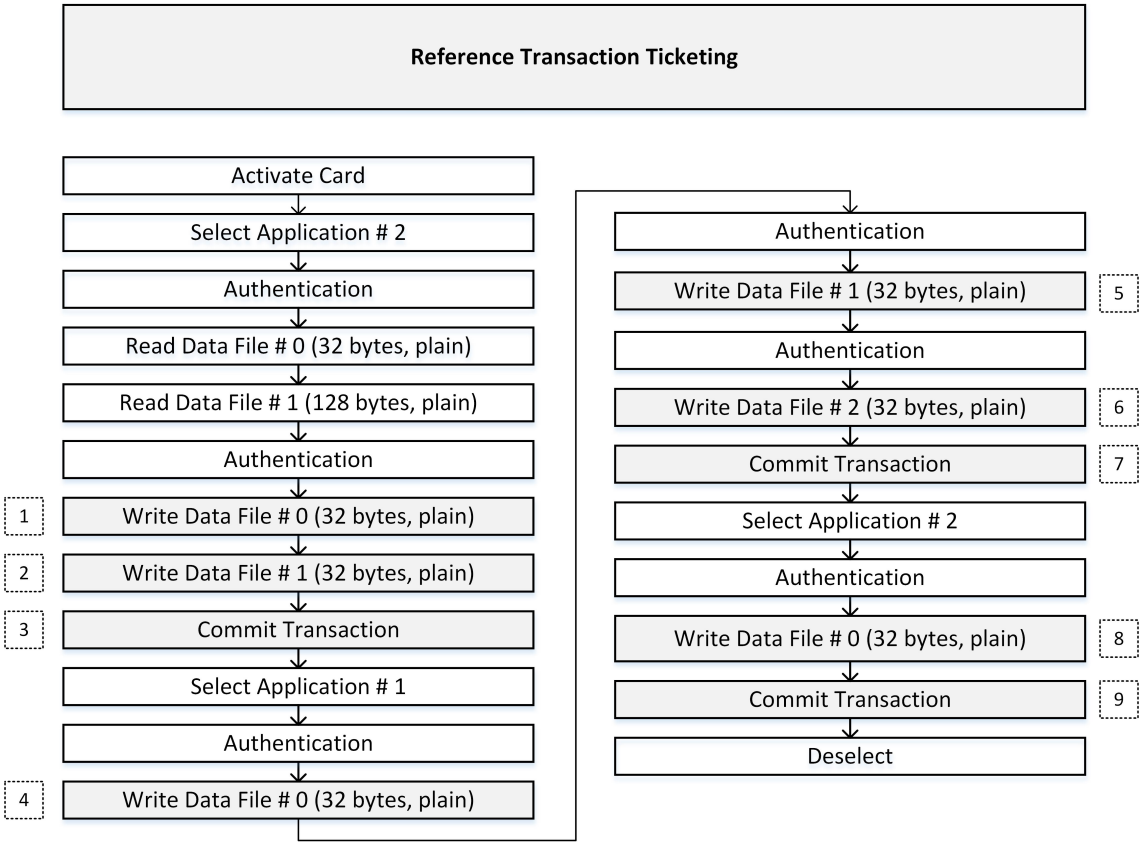

Figure 7.8: *Example 2* - A reference transaction with common ticketing functionality

	reference file system	*double FAT with applist*	*double FAT*	*FAT with journaling*	*EXT design*	*YAFFS design*
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
1	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
2	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
3	1 TS	3 TS	3 TS	1 TS	1 TS	
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
4	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
5	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
6	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
7	1 TS	4 TS	4 TS	1 TS	1 TS	
sync		1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	1 TS + 1 N	2 TS
8	2 N	1 TS + 2 N	1 TS + 2 N	2 TS + 1 N	2 TS + 1 N	5 TS
9	1 TS	2 TS	2 TS	1 TS	1 TS	
sum	12 N + 3 TS	15 N + 18 TS	15 N + 18 TS	9 N + 18 TS	9 N + 18 TS	36 TS

Table 7.5: Measurement results of the reference transaction *Example 2*

As the measurement results of the two reference transactions which can be seen in table 7.4 and table 7.5 show, the file system designs which use a journaling mechanism are performing best in comparison to the other ones.

These results prove, that on the one hand the *FAT with journaling* and on the other hand the *EXT* design could be best suitable for a new smartcard file system.
However, when comparing their performance results with the number of write operations which are needed by the reference file system, a huge gap can be determined, as can be seen in table 7.6.

	reference file system	*FAT with journaling*	*EXT design*
Example 1	18 N + 7 TS	14 N + 30 TS	14 N + 30 TS
Example 2	12 N + 3 TS	9 N + 18 TS	9 N + 18 TS

Table 7.6: Performance difference between the reference solution and the proposed designs

The comparison of the needed amount of write operations clarifies that the reference file system is performing far better than the new design proposals. Apart from that fact, the new design proposals offer other benefits like more efficient memory management and the functionality to reuse released memory.

7.4 Determination of the ideal suitable file system

For the purpose of figuring out the ideal file system solution, all positive and negative aspects of the two journaling based designs and the reference file system implementation are contrasted.

All in all the memory consumption of the currently implemented reference file system is really low. Nearly no management structures are needed at all, as the whole system is build up through a linked list and arranged with the aid of pointers and block linking. The user memory area is bounded through two pointers which are situated in a configuration block, outside of the user memory itself. Therefore the customer really has the whole empty user memory at his disposal.

The drawback of the actual solution nevertheless is the impossibility of freeing memory after data erasure. This fact makes it impossible to reuse memory which is not needed anymore and therefore wastes a lot of useful space after delete operations were carried out.

Both journaling solutions enable the memory reuse after data deletion and therefore provide a distinct advantage over the currently used solution.

The combination of *FAT with journaling* uses multiple blocks of the user memory for system management purposes:

- 32 bytes: Management Block
- $\frac{1}{16}$ of the user memory: FAT
- $\frac{1}{16}$ of the user memory: Copy of the FAT, situated in the Journal region
- defined by user: Size of the Journal region, needs to be at least big enough to store the copy of the FAT and some additional blocks

As the journal size can be defined by the user, its hard to predict, how much space will be used for it. It should at least be big enough to store the copy of the FAT (if it is kept in the user memory) and to backup the content of one backup data file. In the ideal case it is big enough to store the content of all files which are modified during one transaction, but of course also this size differs from case to case.

The management block is not fully stretched and only 18 out of the reserved 32 bytes are effectively needed and the rest is reserved for future use. If it is possible, it would be a good opportunity to swap the fields of the management block out to free fields in the non-volatile memory. This would reduce the allocation of the user memory and free one block.

As the FAT and its copy in total make use of $\frac{1}{8}$ of the whole user memory, it would be worth considering storing the copy of the FAT somewhere outside the user area. If there is space for it anywhere in the non-volatile memory, it would reduce the user memory occupancy drastically and consequently offer more free space to the customer.

The *EXT* based design requires the following amount of memory blocks for management purposes.

- 32 bytes: Management Block
- 1 Bit for each block: Block Usage Bitmap
- 1 Bit for each block: Copy of the Block Usage Bitmap
- 32 Bytes: each file list that is needed (exact memory consumption is unpredictable)
- 32 Bytes: each data block list that is needed (exact memory consumption is unpredictable)
- defined by user: Size of the journal region, needs to be at least big enough to store the copy of the bitmap and some additional blocks

As for the FAT file system design, also here the journal size can be defined by the user, and therefore it is hard to predict how much space will be needed for it. It should at least be big enough to store the copy of the bitmap and to backup the content of one backup data file.

The management block is also in this design proposal not fully stretched, only 20 out of the reserved 32 bytes are effectively needed and the rest is reserved for future use. The same principle applies here: if it is possible, it would be a good opportunity to swap the fields of the management block out in order to free one block in the user memory.

The block usage bitmap requires 1 bit of space for each memory block. For an 8 kilobyte sized user memory, 256 blocks can maximally be used and therefore 32 bytes (exactly one block) are needed for the block usage bitmap. Obviously the bitmap is not using much space and very memory efficient.

Additionally to the bitmap the file system structure needs more blocks, depending on its use. Applications are connected through one field in the application header, the *Ref ANode*, which simply points to the next application and builds up a linked list of applications.

Files can be connected through different ways: Either the application itself points to a file list, which basically is one block containing further references to files, or the application points to the first file and this file again points to the next file and again builds up a linked list of files. Depending on the utilization, the application either requires one or two memory blocks at application creation time. The second block would be needed for the already mentioned file list.
Both possibilities can be used. The method with the separate file list needs one memory block more, but makes the deletion of one file very simple, as it just needs to be eliminated from the said list and no other file header needs to be accessed and modified. The second method with one file header pointing to the next one saves one memory block, but when deleting a file, the files need to be linked anew and therefore the previous file header needs to be updated.

The organization and linking of the data blocks again can be done in two different ways. If there is enough memory space free to write the file data contiguously into blocks, no separate data block list is needed and only one pointer (to the start of the connected data blocks) needs to be stored in the file header.
Otherwise, if the data blocks can not be written contiguously, a separate memory block needs to be allocated, which needs to store the so called data block list. The data block list is one block which contains references to the blocks where the file‘s data is stored.
Consequently one file header either requires one or two memory blocks. If the data can be written contiguously only one block is needed, otherwise a second one needs to be allocated.

One benefit of the EXT based design is the fact, that nearly no blocks need to be reserved for management structures. Only the management block and the two bitmaps are required, all other memory blocks can be used more or less autonomously by the customer. If additional blocks are needed (second block for file list respectively data block list), they are only allocated when they are needed and not from the beginning, like it is the case when using a FAT.

For reasons of clarification, a comparison of positive and negative aspects of the two finalists and the reference file system is arranged in table 7.7.

reference file system	*FAT with journaling*	*EXT design*
+ whole user memory can be used by customer + linked list structure is really easy to realize + quick switching between active and inactive image + no fragmentation can appear at all + no management structure needed at all	+ journal size can be defined by customer + FAT structure is easy to realize + FAT makes release of unused blocks easy + fragmentation is avoided as long as possible + management block and copy of FAT can eventually be swapped out of the user memory + memory can be reused after data deletion + data can also be written in non-adjacent blocks if necessary + memory consumption is predictable	+ journal size can be defined by customer + EXT structure is really easy to realize + bitmap makes release of unused blocks easy + fragmentation is avoided as long as possible + management structures do not need much space (only management block and bitmap are needed) + memory can be reused after data deletion + data can also be written in non-adjacent blocks if necessary + bitmap only needs 1 bit of space per memory block
- memory can not be reused after data deletion - realization of a garbage collector is too difficult at the moment - data can only be written sequentially	- one FAT needs $\frac{1}{16}$ of whole user memory	- data block list will waste space, if not fully utilized - consumption of memory is not predictable (need for a data block list is not known beforehand)
4 positives, 3 negatives	*8 positives, 1 negative*	*8 positives, 2 negatives*

Table 7.7: Positive and negative aspects of the journaling file system designs and the reference file system

Both the measurement results of the reference transactions and the pros and cons which have been illuminated in table 7.7, show that the two approaches *FAT in combination with journaling* and *EXT based design* are qualified for becoming an efficient new smartcard file system.

The designs show equally good performance in terms of write operations and both make use of journaling and the same security mechanism.
In terms of memory consumption, the *FAT* based design however has an edge over the other approach. Although the solution requires more memory blocks for management structures, these can be swapped out of the user memory if possible, what makes the overall memory usage smaller than for the *EXT* design approach. A big disadvantage of the *EXT* design is the unpredictability of the usage of data block lists. The exact amount of required memory blocks is not known beforehand and depends on the size of a data file. This fact makes it difficult to guarantee the optimal usage of the user memory and therefore the *FAT* approach is favorable.

For the named reason I consider the *FAT in combination with journaling* based file system design as the most promising one and suggest to take its realization and implementation into consideration.

When comparing the *FAT in combination with journaling* design to the reference file system, the second one outperforms the *FAT* design easily when doing operations in the electric field. Nevertheless I believe that the very efficient use of the user memory and the possibility to reuse memory blocks after erasure will offer a tremendous advantage in the future.

Chapter 8

Conclusion and Future Work

In this thesis multiple different kinds of existing file systems have been examined, analysed, reused and partly combined in order to create a new efficient transaction based smart card file system.

Implementation techniques, which are needed for the whole structure and realization of a new file system have been introduced and also commonly used disk-, flash-, and further popular file systems have been analysed and interpreted in great detail. After having laid the foundation through exposing file system basics, latest state-of-the-art subjects gave a deeper insight into smartcard specific research findings. Especially the concepts which are used in current flash file systems can be taken over and be used for the EEPROM organization of a smartcard.

Based on the made perceptions, a closer selection of file systems and techniques which came into question for further elaboration, has been made. This selection included the *Extended File System*, the *File Allocation Table* management structure, the *Journaling* backup mechanism and *Yet Another Flash File System*. Through reuse, alteration and further development of the mentioned technologies, new customized file system designs have been created. These approaches were adapted according to smartcard-specific needs and requirements like a strongly limited available memory, the need for a small implementation of file and application structure and a matching block size in order to loose as few memory space as possible. Furthermore low transaction times and consequently fast data access as well as an appropriate error recovery mechanism have been taken into consideration. In chapter 5 the conception of five different file system drafts can be investigated.

The design drafts have been refined and compared in terms of needed write operations during a transaction in chapter 6. This differentiation together with the observation of the memory consumption was essential for drawing conclusions and also comparing the individual approaches among themselves and also to the reference file system.

After using reference transactions for retrieving realistic measurement results and a careful consideration of pro and contra arguments, I suggested the *FAT in combination with journaling* design approach as the most promising one. In comparison to the reference file system, the proposed one is not as fast in writing data because the journaling mechanism

requires some additional security-relevant writes and flag updates. The advantages of the file system are noticeable in terms of memory consumption and management and a lot of benefits arise through the possibility to reuse memory.

8.1 Future Work

In the future probably tree-based file system designs can become more and more worthwhile for the application inside smart cards. As binary trees offer great performance for insert and search operations, namely $O(\log n)$, the time which is needed for data access could be reduced and transaction times could be kept very low. At the moment a tree-based file system can not be used for a realization, as the tree structure certainly requires a lot of maintenance and re-structuring operations and the little RAM space on the smart card is not suitable for this kind of requirements.

Bibliography

[ACL+07] Seongjun Ahn, Jongmoo Choi, Donghee Lee, Sam H. Noh, Sang Lyul Min, and Yookun Cho. *Design, Implementation, and Performance Evaluation of Flash Memory-based File System on Chip.* Journal of Information Science and Engineering 23, 2007.

[Bha07] Pramod Chandra P. Bhatt. *An Introduction to Operating Systems: Concepts and Practice, Second Edition.* PHI Learning Private Ltd., 2007.

[Buc03] Florian Buchholz. *The structure of the Reiser file system.* 2003.

[Car05] Brian Carrier. *File System Forensic Analysis.* Addison-Wesley Professional, 2005.

[Cor00] Microsoft Corporation. *Microsoft Extensible Firmware Initiative. FAT32 File System Specification. FAT: General Overview of On-Disk Format.* December 2000.

[Cor03] Microsoft Corporation. *NTFS Technical Reference.* `http://technet.microsoft.com/en-us/library/cc758691%28v=ws.10%29.aspx`, 2003.

[Cor05] Microsoft Corporation. *Microsoft FAT Specification.* 2005.

[Cor10] Microsoft Corporation. *Transaction-Save FAT File System.* `http://msdn.microsoft.com/en-us/library/aa911939.aspx`, 2010.

[FB] Florian Frank and Jrn Bruns. *Journaling Dateisysteme.* SelfLinux-0.12.3.

[fSEC05a] International Organization for Standardization/International Electrotechnical Commission. *ISO/IEC 7810: Identification cards - Physical characteristics.* Technical report, January 2005.

[fSEC05b] International Organization for Standardization/International Electrotechnical Commission. *ISO/IEC 7816: Identification cards - Integrated curcuit cards. Part 4: Organization, security and commands for interchange.* Technical report, January 2005.

[JSPK13] Shalini Jain, Anupan Shukla, Bishwajeet Pandey, and Mayank Kumar. *Efficient Data Structure Based Smart Card Implementation.* 2013.

[kKSgJ07] Eun ki Kim, Hyungjong Shin, and Byung gil Jeon. *FRASH: Hierarchical File System for FRAM and Flash.* 2007.

[Ltd02] Aleph One Ltd. *YAFFS Yet Another Flash File System.* `http://www.yaffs.net/`, 2002.

[Lz09] Ling Liu and M. Tamer zsu. *Encyclopedia of Database Systems.* Springer US, 2009.

[Mag02] Linux Magazine. *Journaling File Systems. Advanced Linux file systems are bigger, faster, and more reliable.* Linux Magazine, 2002.

[Man12] Charles Manning. *How Yaffs Works.* Technical report, March 2012.

[Mic08] Sun Microsystems. *System Administration Guide: Devices and File Systems.* Sun Microsystems, 2008.

[NML09] Gap-Joo Na, Bongki Moon, and Sang-Won Lee. *In-Page Logging B-Tree for Flash Memory.* 2009.

[OBS12] Pierre Olivier, Jalil Boukhobza, and Eric Senn. *Performance Evaluation of Flash File Systems.* CoRR, abs/1208.6390, 2012.

[Oik14] Shuichi Oikawa. *Non-volatile main memory management methods based on a file system.* 2014.

[PFm04] Marcelo Trierveiler Pereira, Antonio Augusto Froehlich, and Hugo marcondes. *RIFFS: Reverse Indirect Flash File System.* 2004.

[Poi11] Dave Poirier. *The Second Extended File System, Internal Layout.* 2001 - 2011.

[Rod13] Ohad Rodeh. *BTRFS: The Linux B-Tree Filesystem.* ACM Transactions on Storage, Vol. 9, August 2013.

[SGG05] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, Seventh Edition.* John Wiley & Sons, 2005.

[Tan06] Andrew S. Tanenbaum. *Operating Systems Design and Implementation, Third Edition.* Pearson Education, Inc., 2006.

[Twe98] Stephen C. Tweedie. *Journaling the Linux ext2fs Filesystem.* 1998.

[Wan11] Paul S. Wang. *Mastering Linux.* Taylor & Francis Group, 2011.

[Woo01] David Woodhouse. *JFFS: The Journalling Flash File System.* Ottawa Linux Symposium, 2001.

[YJXL10] Chen Yuqiang, Guo Jianlan, Hu Xuanzi, and Liu Liang. *Design and Implementation of Smart Card COS.* 2010.