Data Recovery on Android Phones

By Baiyi Huang

B.A. in Electronic Commerce, May 2009, Beijing University of Posts and Telecommunications

A Thesis submitted to

The Faculty of
The School of Engineering and Applied Science of The George Washington University
in partial fulfillment of the requirements for the degree of Master of Science

August 31, 2011

Thesis directed by

H. Howie Huang
Assistant Professor of Engineering and Applied Science

Acknowledgment

I would like to thank H. Howie Huang, Nan Zhang, Tian Lan, Guru Prasadh Venkataramani, Sea Mao and Leaf Ye for their many helpful comments and discussions regarding this work. I would like to thank my family for their love and support.

Abstract

Data Recovery on Android Phones

In my thesis research, I develop a data recovery program for Android-based mobile phones that can recover over 90% of modified files without using a backup strategy. I also design a new tool called NandDump that can be used to dump all of the information in NAND Flash from Android phones to image files.

Most data recoveries for Android or NAND flash are based on a backup strategy, which means that the system needs to consume flash space to store the current state of the system. And when a user requests data recovery, the system will first check if there has any checkpoints and then try to recover with historical data. In this method, flash space is needed to store the recovery information, and the recovery time must be coordinated with checkpoints. In comparison, my program is the first recovery program for Android phones, which allows one to set any points of time that one wants to recover the system to the extent of what is available on the flash.

I also design the Android analysis tool NandDump that can successfully copy the data in NAND Flash, such as valid, invalid or even the bad blocks contents. By cross-compiling through PC to Android phones, the tool provides a complete copy of devices information on Android phones through MTD (Memory Technology Device) level. It can be used to dump many file systems in Android as long as it goes through the MTD..

With these two effective tools, one can recover all the data that is not erased by the garbage collection. We can see from the results that in some cases it can successfully recover more than 90% of the deleted or updated data.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, there are two types of flash memories: NAND Flash and NOR Flash [1]. The NOR Flash is mostly used for execution in portable devices such as BIOS, mobile phones, and the networking memory. On the other hand, the NAND Flash is designed for high density data storage, such as cell phones main memory, solid-state disk drives.

The flash memory is different from the traditional hard drives in data storage and even the strategy it uses to manage the old data. Therefore, specific file systems have been designed for NAND Flash. YAFFS [2] stands for Yet Another Flash File System that has been used by different operating systems. Android has set YAFFS as their default file system for the mobile phones [3].

YAFFS2 follows the “write once” strategy, which means once a chunk is written, it cannot modify any information unless it performs an erase operation by garbage collection. Because the performance of garbage collection is determined by the erase operations, all of the file systems cannot erase the unused blocks right away. When an object is modified, a new header and data need to be written in the flash. Therefore some old headers are left in the flash that allows us to recover data from flash.

The main contributions of this thesis are twofold:

1. We design and develop the NandDump tool and the recovery program on the NAND Flash Memory.

2. We conduct a comprehensive study on the data recovery algorithms on the Android phones. The result shows that in some cases it can recover over 90% of your deleted or modified data in NAND Flash.

The rest of the thesis is organized as follows. Chapter 2 provides the background about the NAND Flash and its physic features. Chapter 3 describes the detailed design and analysis about the recovery program. Chapter 4 presents the test and recovery rules of the programs. Chapter 5 presents the conclusion.

Chapter 2

Background

In this chapter, Section 2.1 has an overall introduction about the YAFFS File System, such as the data structure about the YAFFS Spare Area, Tags Space, and the Tnode structure. Section 2.2 describes the relationship between YAFFS and Android.

2.1 YAFFS File System

YAFFS stands for Yet Another Flash File System, which is designed for NAND flash. It will write a page that includes the file metadata such as the file name, parent's id, file size, and even the path on the spare area. Each new file will be assigned to a unique ObjectID, and every data chunk in this file will also be assigned to a unique chunkID. YAFFS has a tree structure in memory to indicate the physical location of these chunks. When a file is deleted, YAFFS will write a new object header at the end of the file to indicate that the file has been deleted. After that it can be erased by the garbage collection.

YAFFS marks each newly written block with a unique sequence number that is monotonically increasing through the lifetime of the flash. The order of the chunks can be calculated from the block sequence number and the chunk offset in the block [3]. So

it means when YAFFS begins a start-up scan, it can tell the difference between two chunks that have the same ObjectID and ChunkID.

For page deletions, YAFFS decides the system state by scanning the blocks in their allocation orders. One can say that YAFFS performs “soft” deletes on the files. The file is not really erased until the garbage collector selects the block and performs the erase operation.

When the YAFFS file system starts, the YAFFS_ScanBackwards() is called to build the file system by scanning every block on the flash. As we mentioned before, YAFFS assigns each used block a sequence number. The latest used block owns the largest sequence number.

First, YAFFS scans the whole flash to collect the blocks that need to scan. Second, it sorts the blocks by sequence numbers. At last, the scanning begins from the block with the largest sequence number. Moreover in one block the chunk with the largest physical chunk index is also scanned at first. Because of the scan strategy and soft deletion, we can recover the flash by selecting the right header for each file and rebuild the system based on the Tnode stored in memory.

2.1.1 YAFFS Spare Area and Tag Space

In YAFFS, the data is stored in data area and the metadata is saved in spare area. The struct YAFFS_ExtendedTags is used to save the above data.

Name	Length(bits)	Start(bits)	End(bits)
ObjectID	16	0	15
ChunkID	16	16	31
extraIsShrinkHeader	1	32	32
extraShadows	1	33	33
extraObjectType	4	34	37
serialNumber	2	38	39
firstChunkValid	8	40	47
byteCount	11	48	58
sequenceNumber	29	59	87
extraHeaderInfoAvailable	1	88	88
extraParentObjectID	16	89	104
extraFileLength	21	105	125
Reserved	2	126	127
extraEquivalentObjectID	16	0	15
eccForTag	16	16	31
Reserved	8	32	39
secondChunkValid	8	40	47
eccFor1stHalfPage	24	48	71
eccFor2stHalfPage	24	72	95
Reserved	32	96	127

Table 1.The YAFFS data structure [4]

YAFFS does not write data in flash specific to the files, the file data is written in a sequential log. The entries in the log can hold two types of chunk, one is data chunk and the other is object header.

Object Header is a descriptor for an object. It contains information about the file, such as the parentsID, the path, the time stamp, the fileSize and so on.

ObjectID indicates which objects the chunk belongs to. Every file in YAFFS have a unique ObjectID, it stores in a hash table indexed by ObjectID.

ChunkID indicates the place where in the file this chunk belongs to. A ChunkID of zero means that this chunk is an objectHeader. ChunkID one means this is the first data chunk of this file. ChunkIDs are reset to zero for every new object. If the two chunks have same ChunkIDs and ObjectIDs, we can only tell their difference by comparing the sequence numbers.

Sequence numbers is used to determine block allocation order. The sequence number allows the allocation of 1Mbytes per second for nearly 12 years, which is longer than the lifetime of a flash device.

2.1.2 Tnode Structure

For YAFFS, each file has a Tnode tree to provide the translation from file address to physical chunk address. There always is a pointer to the top of the Tnode tree [4].

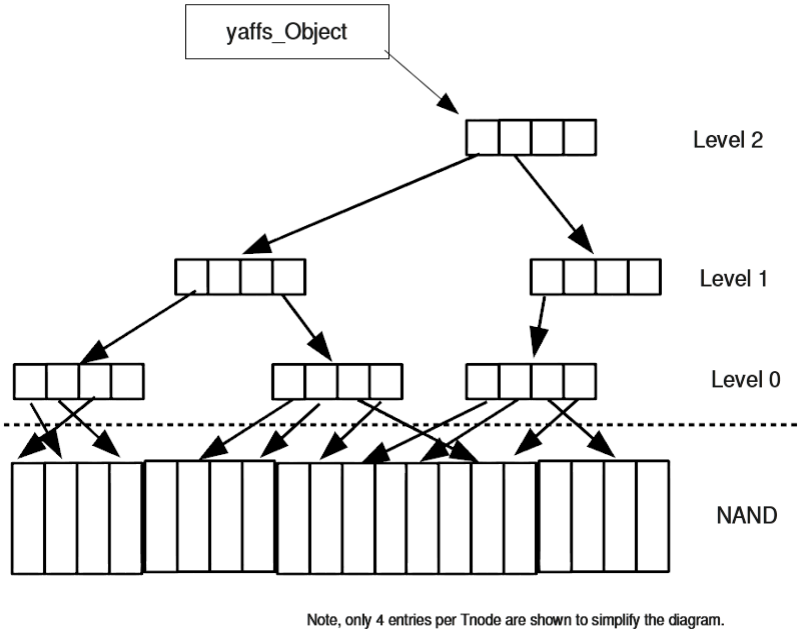


Figure 1. The Tnode Structure for YAFFS2 [5]

YAFFS Tnode tree is built in different pointers. At the level 0, one node has 16 ChunkIDs which can point to the chunk's logical location in memory. Every Tnode has 8 pointers to point to the other nodes in the next level. We can use YAFFS_FindLevel0Tnode() to find the position of any physical chunk [3]. The tree must update those chunks that have been updated.

When the YAFFS first mounts flash memory, it will scan all the blocks in all pages, and sort them by block sequence number. After that, it will scan from the largest sequence

number to the smallest to build the YAFFS_Object in the RAM. With the different in Spare Area, there will be different approaches [4]. When the YAFFS scans a header which is deleted, it will remove all of the data belongs to this file to the deleted directory.

2.1.3 Log-Structured File system

YAFFS is a robust log-structured file system which means every operation has been recorded in the flash. In the next example, we make an assumption that one block have 4 pages.

Step 1: First we create a file.

Block	Chunk	ObjectID	ChunkID	Contents
0	0	100	0	Object Header

Step 2: We write some data chunks to the file.

Block	Chunk	ObjectID	ChunkID	Contens
0	0	100	0	Object Header
0	1	100	1	Data
0	2	100	2	Data
0	3	100	3	Data

Step 3: Then we close the file. This will lead a new object header in the file. Therefore the previous object header is useless.

Block	Chunk	ObjectID	ChunkID	Contens
0	0	100	0	Object Header
0	1	100	1	Data
0	2	100	2	Data
0	3	100	3	Data
1	0	100	0	New object header(length n)

Step 4: Next we open the file and overwrite a part of the data chunk in the file. The replace data and object header chunks become useless.

Block	Chunk	ObjectID	ChunkID	Contents
0	0	100	0	Object Header
0	1	100	1	Data
0	2	100	2	Data
0	3	100	3	Data
1	0	100	0	Obsolete object header
1	1	100	1	New data chunk
1	2	100	0	New object header

Step 5: Now we want to resize the file to zero. It will has a new object header with length 0, set the isShirnk mark as true and make the remaining data invalid.

Block	Chunk	ObjectID	ChunkID	Contents
0	0	100	0	Object Header
0	1	100	1	Data
0	2	100	2	Data
0	3	100	3	Data
1	0	100	0	Obsolete object header
1	1	100	1	New data chunk
1	2	100	0	New object header
1	3	100	0	New object header (length 0).

When all of the pages in this block become invalid, which means that block 0 does not contains useful information so we can erase block 0 and reuse the block

Step 6: We rename the file. To do this we write a new object header for the file. Block 0 now contains only deleted chunks and can be erased and reused.

Block	Chunk	ObjectID	ChunkID	Contents
0	0			Erased
0	1			Erased
0	2			Erased
0	3			Erased
1	0	100	0	Useless object header
1	1	100	1	Deleted
1	2	100	0	Useless object header
1	3	100	0	Useless object header
2	0	100	0	New object header with new name

Table 2. Log-structured file system

2.2 Android and YAFFS

Because Android [5] uses the Linux-kernel, we can use any Linux supported file systems. But because most mobile phones use Nand flash, we can assume that they use YAFFS file system. Android phones will mount three YAFFS images: userdata, system, cache.

/system is a YAFFS2 file system and contains information about the system detail. It uses the first Nand chip and df command shows that it has 64MB capacity, and it is nearly 30% used. From test we can see that a normal user should never need to update data on this image, but we know that over-the-air updates would be stored on this image.

The /data file system is very similar to /system, and uses the YAFFS [6]. This is the place where user applications are installed and saved data from. The android phones control the changes made to the /data is saved in ~/.Android/userdata.img.

Chapter 3

Recovery Algorithm

In this chapter, Section 3.1 presents the importance of sequence number and isShrink Header. Section 3.2 presents a basic idea about the garbage collection. Section 3.3 provides the NandDump tool and section 3.4 presents the detail about data recovery.

3.1 Sequence Number and isShrink Header

YAFFS marks every newly written block with a unique sequence number that is increased through the lifetime of the flash. When YAFFS scans the flash and finds multiple chunks that have the same ObjectIDs and ChunkNumbers, it can only tell the difference by the sequence number. In this way older pages can be overwritten without braking the write once rule.

We cannot modify the page status into invalid on flash. So the situation should be considered that if there is no checkpoint on the flash, we have to scan all of the blocks at mount time. Furthermore, if there is no sequence number, and we have two chunks with same objected and ChunkID, we cannot distinguish them by any method. As we mentioned before, YAFFS has write once rule, so every page must be written once before an erase operation has performed on this block. When the YAFFS performs the

startup scan, it will first read the latest written header and the remaining header without the isShrink mark will be treated as invalid and put into deleted directory. Therefore, the isShrink mark is used for telling the system that before this header, the file has a resize operation which is decreasing the file size.

3.2 Garbage Collection

When a block state is full and contains useless chunks, we should perform an erase operation on this block and reuse it in future. This process is called garbage collection. However, in some cases, there are still some useful chunks in that block so we cannot erase those blocks for it would lose valid data. Once all chunks in this block have been deleted, the block can be erased and ready for reuse.

If the remaining blocks are quite enough, then YAFFS does not need to work hard on the garbage collection, but will try to erase blocks with very few valid chunks. This is called passive garbage collection. However, if the remaining free blocks are not enough, YAFFS will perform a garbage collection on every block to recover more space and will erase each block even it contains only one invalid page. This is called aggressive garbage collection.

Theoretically, if there is no missing sequence number which means no garbage collection performed, we can recover every version of specific file. However, once the

system reserved block number is smaller than some value, YAFFS will call the garbage collection process. So this is the reason why we have missing sequence number blocks.

3.3 NandDump

A NAND dump is a copy of the NAND Flash that stores all the information in the chips. NandDump uses the API provided by the MTD (Memory Technology Device) layer. If NandDump uses a different API than the present one, there will have an IO error.

Because every manufacturer has its own strategy to deal with bad blocks, it is impossible to make the recovery program run under every different flash. Therefore, I modified the NandDump tool to make it can be run under Android OS. Meanwhile, I also make the NandDump to call manufacturer's bad block check function through MTD to directly check if the block is bad, and then mark it and fill up the data into 0xFF.

I compile Nandump with the gcc cross-compiler. After that I push it with ADB (Android Debug Bridge) shell to the sdcard and make it can execute binaries programs. Because we need to run the NandDump on Android phones, the most challenging task is how to deal with the bad blocks. The file system must save those bad blocks in BBT (bad-block table) since we cannot distinguish the factory-bad and worn-out-bad blocks

from the marker in the OOB (Out of Band) area [7]. The whole process is shown below:

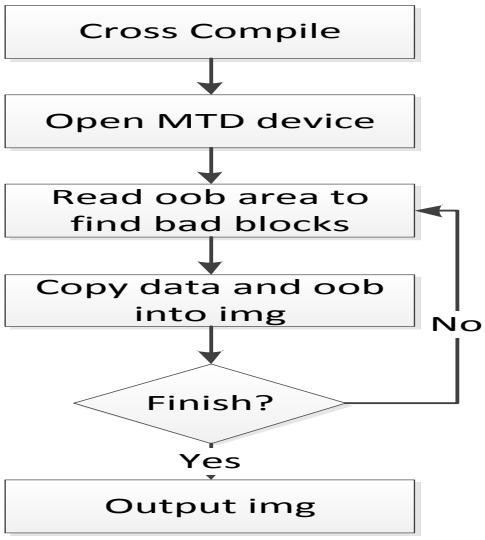


Figure 2. Chart flow for NandDump

The different Nand flash types have different policies to mark the block as bad, and the bad block table for each type of flash is also totally different. So here is the problem, in my program, I cannot design solutions for each type of chips. Therefore, I modify the NandDump tool. In detail, I make the NandDump in Android phones to call each chips own methods to decide whether if a block is bad or good. This will completely solve the problem as I mentioned before. So after bad block self-check, we just fill in 0xFF to the block when it marks as bad and then copies it to our output image.

3.4 Recovery Detail

In this section, the recovery algorithms will be described in detail. First we discuss the basic idea about the overall recovery. Then we will talk about some specific situations such as how to deal with those files that missed the header or data chunk.

3.4.1 Basic Idea

Suppose we have a pattern like this:

Block	Chunk	ObjectID	ChunkID	Sequence Number	Content
1	0	100	0	3	Header(Old)
1	1	100	1	3	Data
1	2	100	2	3	Data
1	3	100	3	3	Data
2	0	100	0	4	Header(Old)
2	1	100	1	4	Data
2	2	100	3	4	Data
2	3	100	0	4	Header(Old)
3	0	100	4	5	Data
3	1	100	2	5	Data
3	2	100	1	5	Data
3	1	100	0	5	Header(New)

This is the ideal situation, all blocks and chunks are not erased by garbage collection. So, we use “back” search from the last chunk to the first chunk, and merge those chunks which have not same ChunkID. As shown below:

Block	Chunk	ObjectID	ChunkID	Sequence Number	Content
1	0	100	0	3	Header(Old)
1	1	100	1	3	Data
1	2	100	2	3	Data
1	3	100	3	3	Data
2	0	100	0	4	Header(Old)
2	1	100	1	4	Data
2	2	100	3	4	Data
2	3	100	0	4	Header(Old)
3	0	100	4	5	Data
3	1	100	2	5	Data
3	2	100	1	5	Data
3	1	100	0	5	Header(New)

We merge them into a new file and save. Next, we delete a header and all of the data under that header. In the example, we delete the block 3. And we continue doing the "back" search. As shown:

Block	Chunk	ObjectID	ChunkID	Sequence Number	Comment
1	0	100	0	3	Header(Old)
1	1	100	1	3	Data
1	2	100	2	3	Data
1	3	100	3	3	Data
2	0	100	0	4	Header(Old)
2	1	100	1	4	Data
2	2	100	3	4	Data
2	3	100	0	4	Header(Old)

As the same, those left chunks made of the new version of the deleted file. And here we continue:

Block	Chunk	ObjectID	ChunkID	Sequence Number	Content
1	0	100	0	3	Header(Old)
1	1	100	1	3	Data
1	2	100	2	3	Data
1	3	100	3	3	Data
2	0	100	0	4	Header(Old)

Table 3. Basic idea for recovery pattern

3.4.2 Recovery Processes

The scan process contains two parts. The first time it scans the OOB space from the beginning to the end, marks the bad blocks and invalid blocks, and sorts the block by the sequence number. The second time it scans from the highest sequence number to the smallest number and builds the Tnode structure in the memory. Here is a simple flow chart for the recovery.

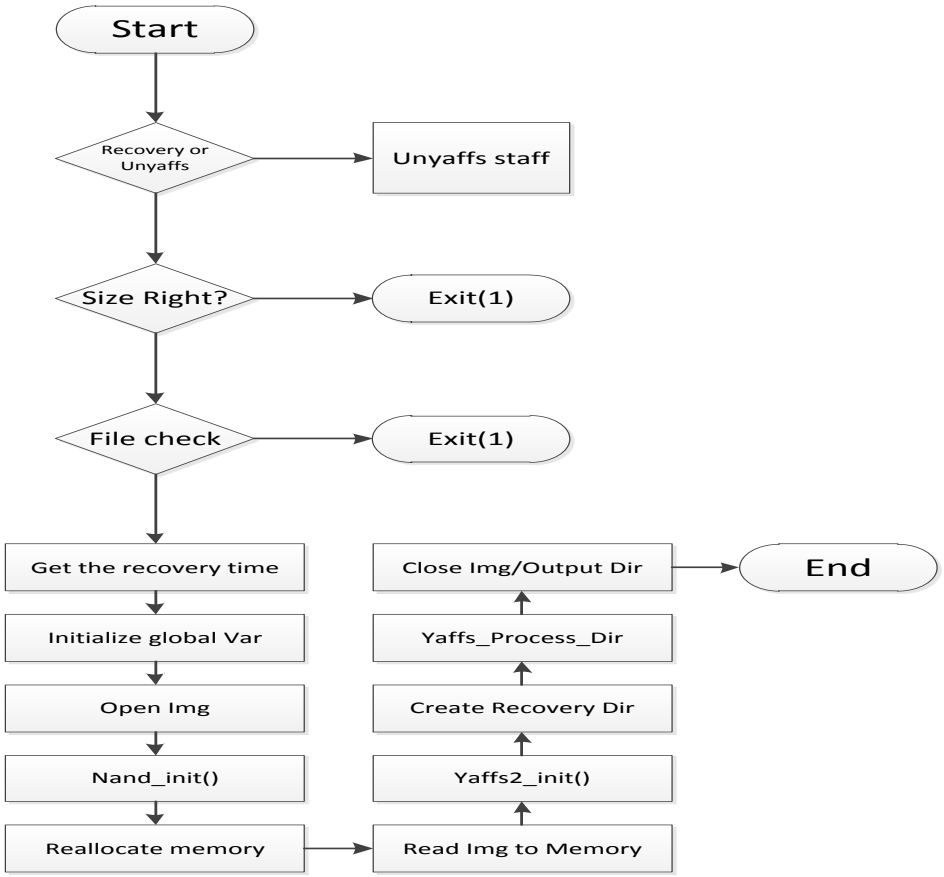


Figure 3. Chart Flow for Main Function

Step One: Scan the Whole Flash

We scan from the first block to the last block, and keep reading the sequence number. If

the sequence number falls among 0x10000 between 0xEFFFF00, it means this block is a normal used block, stored the sequence number and the block number. After scanning the entire image, we need to sort the sequence number and find the last block.

There are two kinds of data stored in the Nand flash: one is object header; another is data chunk. Because of “write once”, YAFFS will have many object headers for the same object on flash. Every modify operation will lead to a new object header written on the flash. Obviously, every new object header will make the old one invalid. It will be erased after the garbage collection. File operation will either enlarge or reduce the file size. If the file size is reduced it will add isShrink mark in the new header.

The algorithm scans from the highest sequence number to the smallest. First read every page‘s OOB tags. If the sequence number is 0xFFFFFFFF, it means that page is not valid. The possible situation for this is that this page is either not in use or has writing error. For those two situations we just ignore the pages.

When the ChunkID is larger than 0, it means this page is data chunk. Then we search if the object is existed in the memory by ObjectID. If not, we create the Tnode, otherwise we will check if the object file type is a regular file. If not, we abandon the page. If the page offset is smaller than the shrinkSize, it means this page is valid, and then we assign a Tnode to the YAFFS_object so as to save the physics ChunkID. If the file size is

reduced, we will mark isShrink in the object header, and store the reduced offset size in the shrinkSize. During the go back scan, if we reach the object header which contains the isShrink marker, it means some data in the next may not be valid anymore. Meanwhile we mark the invalid offset in the shrinkSize. If the scan does not meet the shrink object header, the shrinkSize is set to be 0xffffffff. In most time when we perform go back scan, we always reach the object header before the data chunk.

When the ChunkID equals to 0, it means this page is an object header. Then we read the data space and obtain the file information. We search the YAFFS_object from Tnode in memory and check the time. If it is not found and the time is correct, we build the Tnode in memory. If the corresponding YAFFS_object valid mark is true, it means we already found one before this header, so we delete this header.

If the header is a file header, we must check if the header contains the shrink mark. If yes, we know that there exists some data chunk for this file that may not be valid anymore. So we save the shrinkSize and ignore those data that is not in the shrinkSize offset.

If the object header valid mark is not true, it means that we do not find the object header, so we must save some useful information to object. If it is a directory we only need to save the mtime, ctime, atime, etc. Otherwise we must find the parent

YAFFS_object so as to link the child to the parents. If this page is data then we must recalculate the file size and renew the object header. Here is the YAFFS initialization chart:

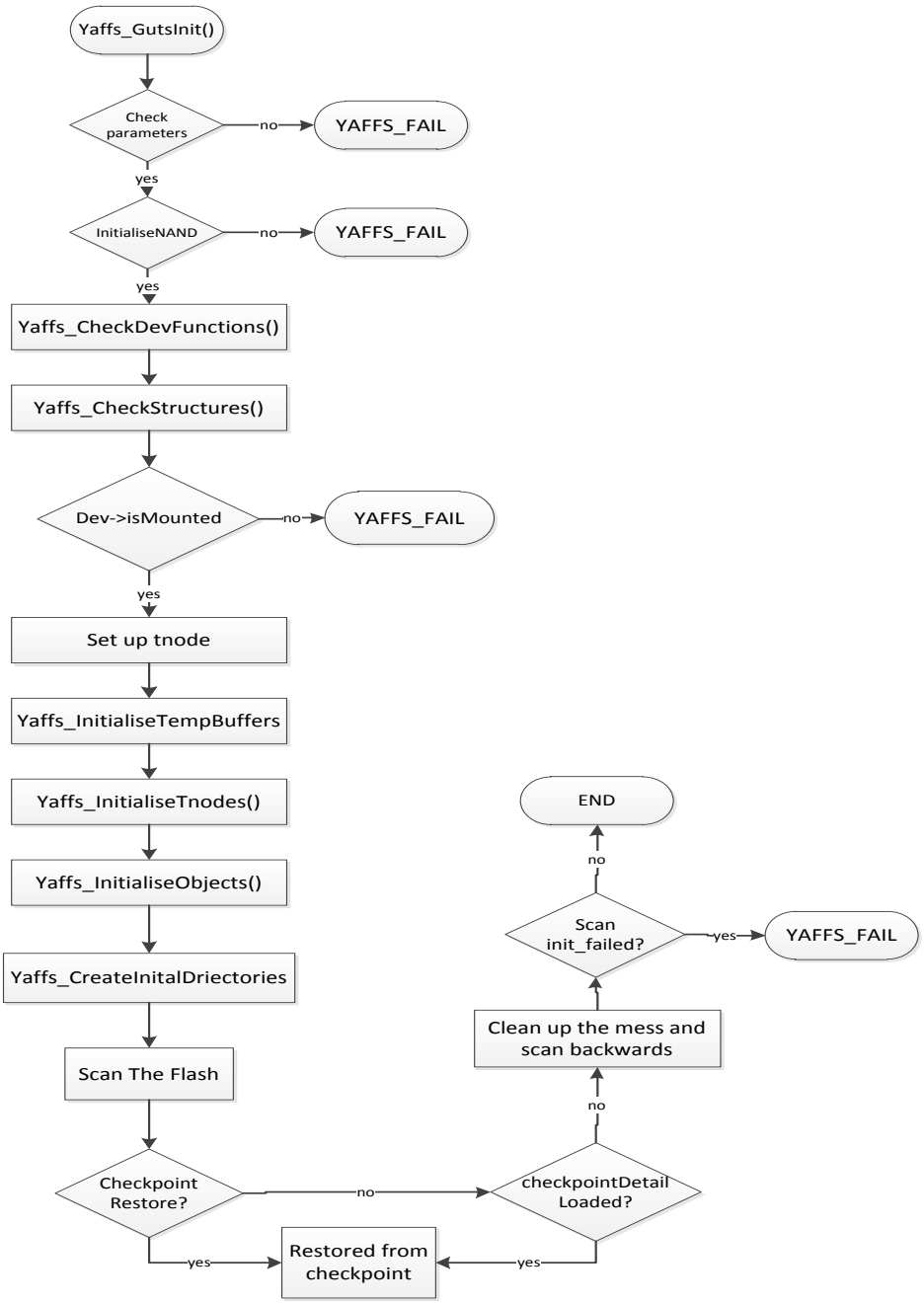


Figure 4.YAFFS Initialization

Step Two: Backwards Scanning

After scanning the whole image, we create a directory tree in memory. First we start from the root directory, which is built since the YAFFS initialized virtual directory. The difference between virtual directory and real directory is that the object header in virtual directory does not contain physics chunk. If the mtime and ctime master with the recovery time, we put this header into the Tnode, otherwise we ignore the header and all of data chunks.

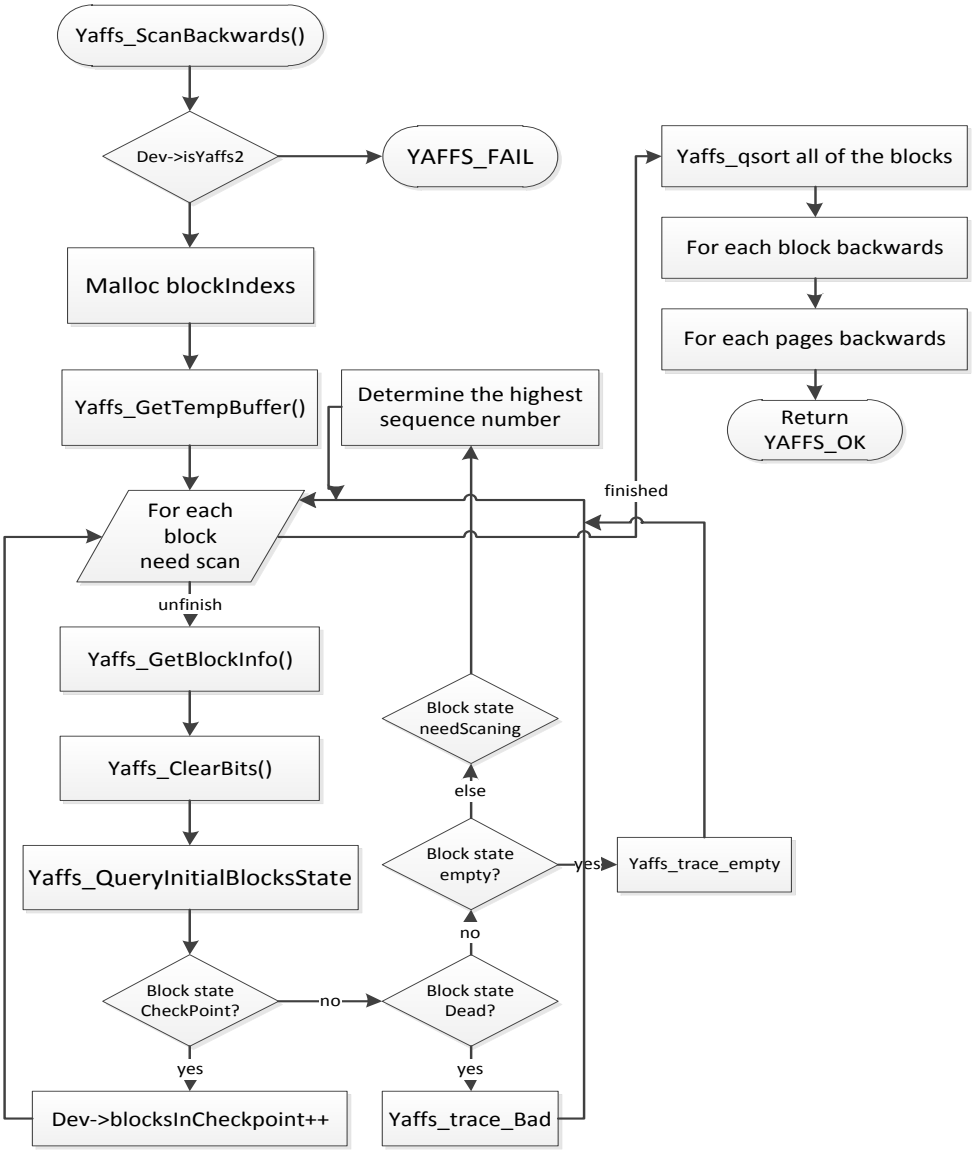
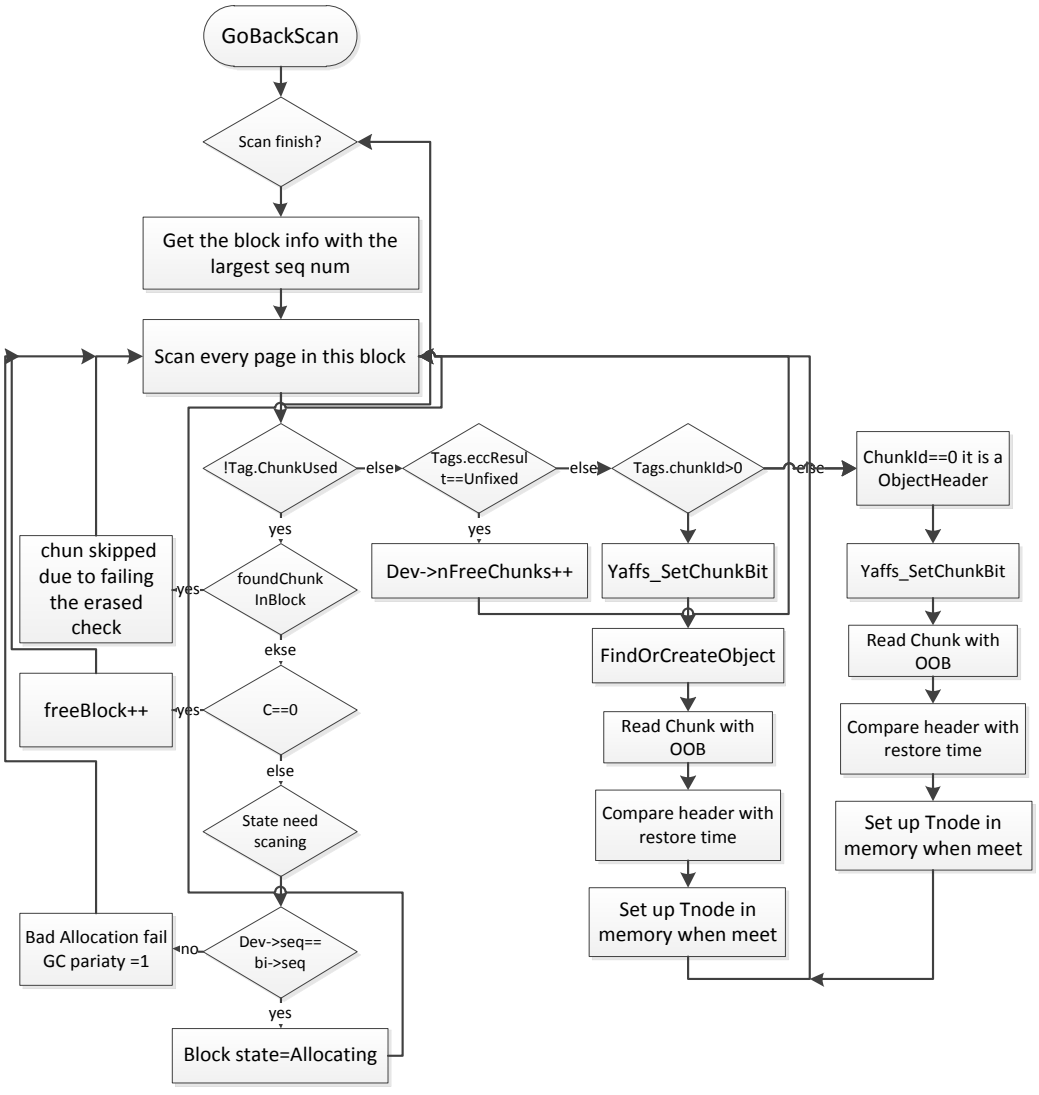


Figure 5. YAFFS Backwards Scanning

Step Three: Recover the File

First we will check if the object is a directory and then check every child directory and file. If it is the file, we compare the recovery time from user input with the timestamp for that file. If the recovery time is larger than the mtime and ctime then create a file and calculate the file size. Repeat this process until all of the files and directories are done.



Fugure 6. Recovery Detail

One can see that once the header or the data chunk does not meet the requirement of the recovery, it is directly removed from the memory. Because if we put them into the Tnode, the information must be updated then the unexpected data or header should be also added into the Tnode.

3.4.3 Recovery Operations

There are two operations, delete and update, which can make a chunk invalid and qualified for garbage collect. When we perform a deletion operation, YAFFS will first move this object to a directory named “unlinked” and check if the object is ready to be deleted. If so, it will move the object to another directory “deleted”. Files under this folder are treated as “deleted” even though they are not erased on flash [7]. Garbage collection will perform this operation later. We know that in real flash, we must consider the situation that the flash is used for a long time, so garbage collection will erase many blocks and write new data on the block, making the data on the flash become more separate. We must deal with all of the situations. So, only for the delete operation, if we ignore the last deletion header and try to recovery the object from the last header, we may have the following situations, which are always aimed to the currently recovery header:

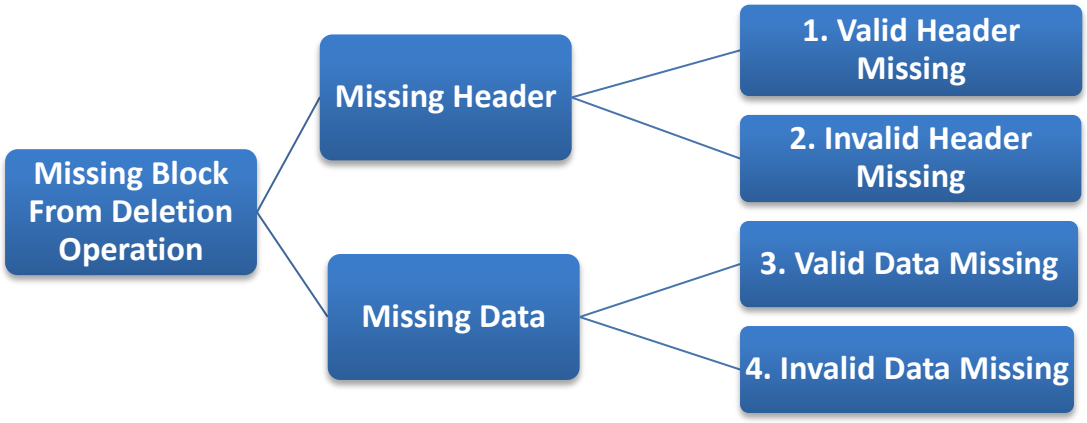


Figure 7. Missing Block Operation

1. Valid Header Missing: Valid header means it is the current recovery header. In this situation, suppose we delete a file and already write a deletion header in the end of the file. As we discussed in the garbage collection, YAFFS will collect and erase the useless block. If not it will chose the earliest full block to erase. So, when a file is deleted, it is possible that the last valid header is erased. If it happens, the recovery program will keep scan the next invalid header (means it is not the original one the deletion operation affected) and set it as the valid header, and try to recover the data belongs to that one. For example, consider the pattern:

1.2.3.4	5.6.H.x	1.2.3.4	H.x.x.x	H(d).x.x.

If the last recovery header is missing, YAFFS will first read the data chunk. If there is no Tnode in the memory, then it will create a new one and keep inserting remaining data to the Tnode. So the next recovery pattern would be:

1,2,3,4	5,6,H,x	1,2,3,4	Erased	H(d),x,x,
1,2,3,4	5,6,H,x	1,2,3,4	Erased	H(d),x,x,

2. Invalid Header Missing: Invalid header means it is the header that has the same ObjectID with the current recovery header. For example, if a file is updated many times, it contains invalid header. If only those headers are missing we can still recover the whole file from the remaining data chunk by comparing their sequence numbers. For example, consider the pattern:

1,2,3,4	5,6,,7,8	H,x,x,x	1,2,3,H	H(d),x,x,

If the second invalid header is erased by garbage collection, it has no impacts on the recovery file. The next pattern would be:

1,2,3,4	5,6,,7,8	Erased	1,2,3,H	H(d),x,x,
1,2,3,4	5,6,,7,8	Erased	1,2,3,H	H(d),x,x,

3. Valid Data Missing: Valid data means it belongs to the current recovery header. This means the valid data belongs to the valid header is erased by the garbage collection. So if this is happened, we cannot recover a whole file. Moreover, the loss of the valid data may lead to those invalid data become the valid and insert them into recovery file. In this problems, after those valid data is erased, it always leads to the invalid data become the valid data. For example:

1,2,3,4	5,6,H,x	X,5,6,7	H,x,x,x	H(d),x,x,
1,2,3,4	5,6,H,x	Erased	H,x,x,x	H(d),x,x,
1,2,3,4	5,6,H,x	Erased	H,x,x,x	H(d),x,x,

4. Invalid Data Missing: Invalid data means it does not belong to the current recovery header. This means some invalid data for the recovery header is erased by garbage collection. Some times when we update a file, some of the used data chunk is useless, so the YAFFS will collect those blocks and erased them for future use.

For example, consider this pattern:

1,2,3,4	5,6,H,x	X,5,6,7	H,x,x,x	H(d),x,x,
1,2,3,4	Erased	X,5,6,7	H,x,x,x	H(d),x,x,
1,2,3,4	Erased	X,5,6,7	H,x,x,x	H(d),x,x,

So these situations may happen at the same time, making the total possibilities 15. Here is the list of all of the possibilities. Some of the combination may affect the final result for the recovery file. In this list we suppose that the flash is for daily use, so we ignore the very extremely situations.

Possible Mistakes	Affect the Recovery File	Possibility
1	No	Very Small
2	No	Very Small
3	Yes	High
4	No	High
1,2	No	Very Small
1,3	Yes	Normal
1,4	No	Very Small
2,3	Yes	Very Small
2,4	No	Very Small
3,4	Yes	High
1,2,3	Yes	Very Small
1,2,4	No	Very Small
1,3,4	Yes	Very Small
2,3,4	Yes	Very Small
1,2,3,4	Yes	Very Small

Table 4. Possibility of Missing Data

As shown above, if valid data is missing, at most time it will affect the recovery file. If we delete a file and wait a long time and then try to recovery the file, it has no doubt it will increase the possibility of failing to rebuild the file, even part of it. From the list we can see that the missing valid data, or missing valid data and invalid data gets high possibility and has a great effects on the recovery file. We will discuss them in details.

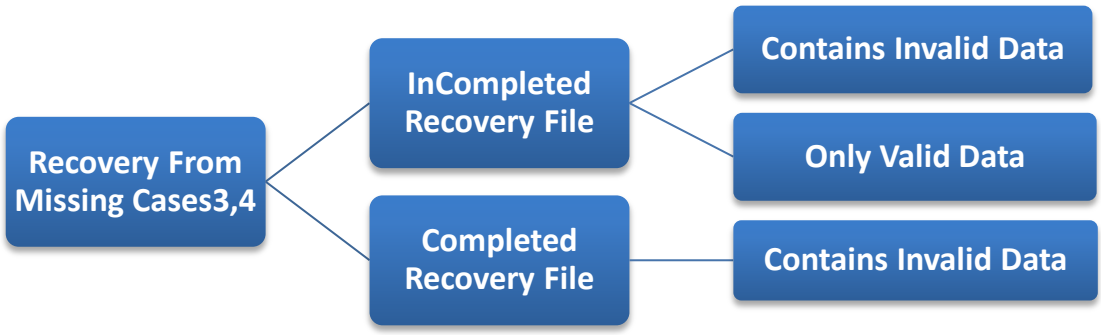


Figure 8. Recovery from Missing Cases 3, 4

Consider, for example, we have the pattern like:

X,x,x,1	2,3,4,x	5,6,H,x	1,2,3,4	5,H,x,H(d)

And when we delete the file, the fourth block must be the dirtiest block for the file, so it must be erased first. And suppose we also have another erased block. We have:

X,x,x,1	2,3,4,x	5,6,H,x	Erased	5,H,x,H(d)

So when we try to recover this file after the erase operation, the invalid data will become useful when we rebuild the file.

X,x,x,1	2,3,4,x	5,6,H,x	Erased	5,H,x,H(d)

For this part, only valid data is in the final recovery file, so it is much easier. The bad news is that we can never recover those erased data from the flash while the good news

is that we can tell that how much of the file is recovered. Under this circumstance it is very similar to the incomplete recovery file. For this situation we cannot know exactly what exist in the recovery file, such as does it contain the invalid data for this header? Do we miss a recovery header? However, there are some rules we can follow which will be presented in the next chapter.

It is obvious that recovery file from update operation is similar to the deletion operation. What we do is to ignore the last valid header and its data, and make the next header become the valid header and start scan the flash. The only situation I want to explain is the recovery from shrink header. As we discuss before, the shrink header marker is used to mark object headers that are written to shrink the size of a data file.

Consider a file that has been through the following [8]:

First we create a file and write 5MB of data on it, and then we truncate the file to 1MB. After this, we set the file access position to 3MB and write 1MB of data. Therefore, the last object header must use isShrink Mark to indicate that there is 1MB hole on this file.

YAFFS must make sure the isShrink Header can only be erased after all of the blocks with the smaller sequence number have been erased. In YAFFS file system, the file name is stored in the object header. So when an operation has changed the object name, all that it needs to do is to write a new header for this file. And the previous object

header is set to be useless that means it is ready for the garbage collection. But for a name change the previous object header only have one invalid page changed, while the garbage collection always chose those dirtiest block first. In other words, it is rarely happened that one name change can directly lead to an erase operation. So we can say that those old header will save always on the flash for some times. In conclusion, once we get those headers, we can recover them from flash easily.

3.4.4 Time Window

In Android 2.1, the YAFFS has a problem that is when a deletion operation performs on one file, it will first write two object headers (one is unlink, the other is deleted). Then the file can be erased by garbage collection. However, during this operation, YAFFS just copies the latest object header's information to the unlinked and deleted headers. It means that the time stamp saved in unlink or deleted header is exactly the same as the last valid object header. Therefore, when a user performs a delete operation, we have three exactly same object headers at the end of the file. Therefore, when we perform a recovery, we need to set a time T, the system will recover all of the files which deleted during this time window. There is also another option that one can disable this function, making it is exactly operated by the time stamp saved in object header.

Chapter 4

Experiment and Recovery Rules

The test Android phone is HTC G1. It is the Google first Android OS mobile phone in 2008. The main memory is 256MB. After the NandDump I obtain two images, one is user.data and the other is system.img. Here is the result I get from those images. Because it is very hard to calculate the exactly updated and deleted file. The percentage represents the ratio between the recovery file's real size compared with the fileSize in the corresponding header.

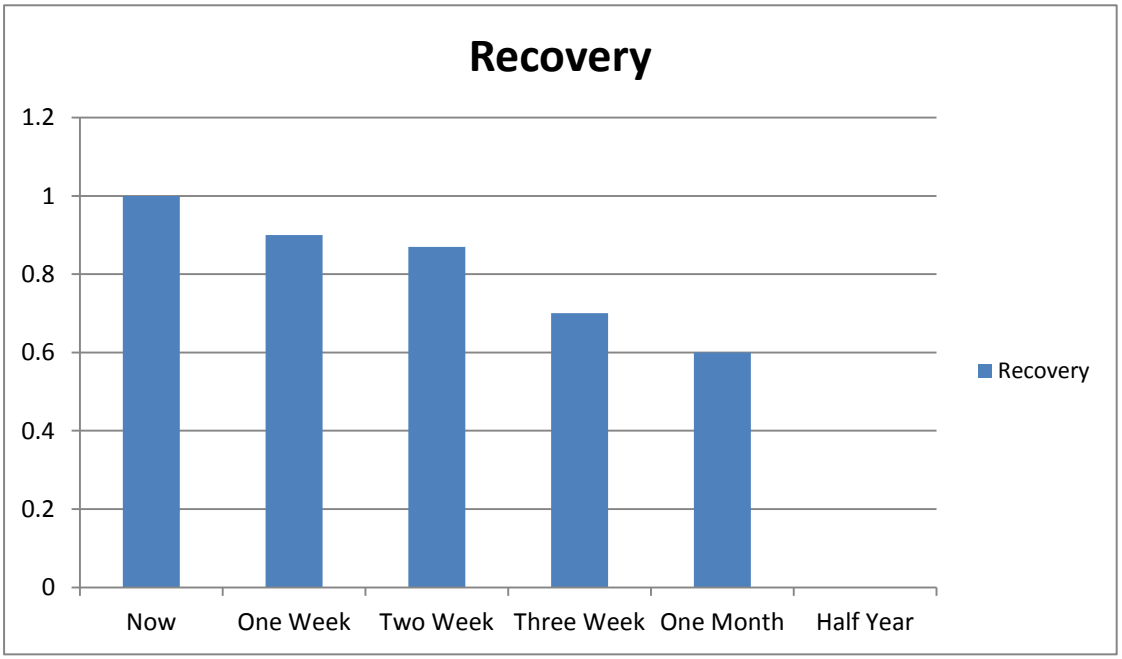


Figure 9. Recovery result for images

One thing I want to point out here is the result of recovery from half year ago. The

reason why it stays nearly 0% is, before that time, the device must have performed a whole aggressive garbage collection. It means that the whole flash had reached a limited point that the remaining block is not enough, so every block in flash will come to one cycle and all of the unused or invalid pages will be erased. After the device has performed an aggressive garbage collection, only valid data stay on the device. So for this time, there must be aggressive garbage collection performed on the flash few months ago, so I cannot get any useful data for recovery. The YAFFS can disable garbage collection to obtain a better performance [10]. After the recovery from the dump image, we must use mkyaffs tool to rebuild the recovery file and make it into YAFFS model so as to remount it on the Android phone.

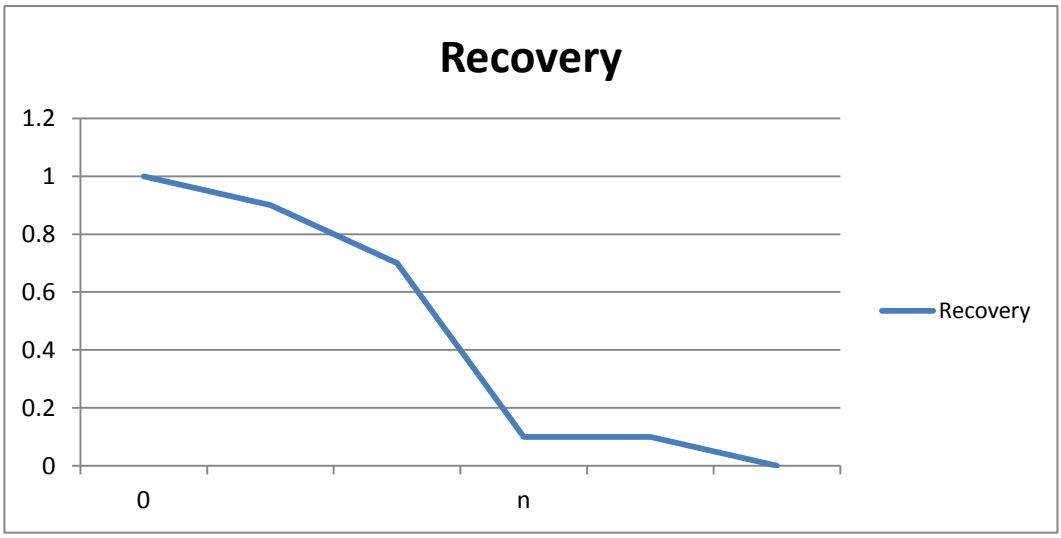


Figure 10.Recovery from garbage collection

This picture shows the picture that I obtain from the analysis of the code and simulator. The X axis indicates the number of used block for the whole flash. N means the reserved blocks numbers for the device to perform an aggressive garbage collection. Therefore, the chart means with the number of used block increase, there must also

exists many blocks to be set invalid. So the recovery percentage must decrease in this case. Once the free blocks have reached the specific number n, then a global garbage collection will be performed. So after this operation, there will be no invalid pages on the device at that moment. Therefore, the recovery ratio will be sharply reduced after this time. Many factors can affect the recovery result, including the free blocks number, the garbage collection degree, whether the garbage collection is off. In the following, I draw some conclusions according to the source code and the currently images.

Recovery Rules:

1. The fewer garbage collections, the better recovery result we can get.

2. We can have a better recovery result for those recently updated file.

3. The fewer changes in one file, the better result we can get.

4. The more free blocks we have the better result we can get.

While it is obvious that we can recover much more data from the image if the garbage collection had not erased those blocks, if there is no garbage collection, we can recover any time stamp of the system, all thanks to the log-structured YAFFS file system. For the latest YAFFS, one can do this by disabling the garbage collection. For those recently updated information, for the sequence number is larger than those old blocks. So we can assume that the possibility that a block will erase is smaller than the old blocks. For instance, if a deleted file contains many completed blocks, those blocks will become

totally useless for those blocks only contain that file. In most time it will always lead to a garbage collection, because the whole block is invalid. On the contrary, those files that contain few pages can survive during this situation. Because there may be other useful information in this block, garbage collation will not select those blocks to erase.

For YAFFS garbage collection, it will erase those blocks that have much more invalid pages. It means that if a block has only few pages changed, it will not erase when there are many free blocks. Those blocks can only be erased when the reserved blocks are not enough, so a device aggressive garbage collection will be performed. Every useless data will be erased at this time. So if a block has made few changes, it is more likely to keep it stay on the flash. Flash memory will perform garbage collection when there has no enough space for new data. But the garbage collection is expensive, which means it has a great effect on the performance of the flash itself. Therefore, the garbage collection will not perform at any time. YAFFS has strategy to calculate the frequency of garbage collection. If the free space is enough, there is no need to perform the garbage collection. So the data can be saved as long as possible.

Chapter 5

Conclusion

Because of low cost and high performance, almost all modern flash memory devices use NAND flash memory. Besides, NAND flash chips are extremely compact and capable of fast read/write operations.

In this thesis, I develop the process of flash memory data recovery that restores data from primary storage media when it cannot be accessed normally. It is a flash memory file recovery service that restores corrupted and deleted data. I show that in many cases more than 90% of lost data can be restored.

For the future work, we need to consider different cases of missing sequence numbers and make the recovery program to support large size pages.

Bibliography

[1]. Wai-Ming Cheung, “How YAFFS Works”, www.YAFFS.net/files /HowYAFFSWorks.pdf, 2005

[2]. Android Developer, http://developer.Android.com/sdk/index.html

[3]. Wai-Ming Cheung, “YAFFS Documents”, www.YAFFS.net/files /, 2007

[4]. Wang Wei, “Build your own Linux kernel”, blogimg.chinaunix.net/ upfile2/100123192128.pdf

[5].Charles Manning, “Yet another Flash File System” http://zh.wikipedia.org/ YAFFS

[6]. Cheng Wei,”The process about how to program in Linux MTD deriver”, 2010

[7]. Android Developer, http://developer.Android.com/sdk/index.html

[8]. “NAND bad blocks”, http://wiki.openmoko.org/wiki/Bad_blocks

[9]. Gang Yunming,”YAFFS start-up analysis for ARM11”, 2011

[10].”YAFFS: A NAND File System”, elinux.org/images/e/e3/YAFFS.pdf,2008

[11]. “Unix System Call”http://www-asc.di.fct.unl.pt/~jcc/ASC2/material/syscalls.txt

[12]. “Garbage collection control to yaffs2” http://www.balloonboard.org/lurker/ 20100315.231830.b8c8f776.en.html