The FS/Z File System

Specification and On Disk Format

Version 1.0

2021

Copyright

MIT licence

Table of Contents

Preface

"Keep it simple, stupid."

/ Kelly Johnson /

This is a new file system which was designed for the future. It's goal is to be unlimited in capacity, expandable, being backwards compatible at the same time. Features are optional, a basic driver is extremely easy to implement, but a fully featured driver is comparable to commercial grade solutions.

It is built on the assumption that modern storage devices has no moving mechanical parts any more therefore they do not suffer from the seek penalty. Also they do use linear sector addressing mode exclusively (also known as LBA mode). This is most certainly true for flash drives, USB sticks, SD cards and SSD disks. It is assumed that future storage will be like that too, not having mechanical parts and keeping linear addressing.

If that assumption stands, then file systems can be significantly simplified. There's no need for cylinders, bands nor block groups, neither pre-allocation, because blocks with related data doesn't need to be located closely any more. The one and only priority left is using as few space as possible for the housekeeping data without sacrificing the effectiveness of interpreting that information.

There's also no need for tables with limited capacities. Even though those are usually allocated on format knowing the storage's size, it's a common task that operating system administrators create a backup of the file system and restore the backup to a larger storage. With fixed sized tables, a file system can't really take advantage of the bigger volume (unless a slow and risky resizefs operation is performed). Not to mention how annoying it is when a user runs out of available i-nodes for example when there's plenty of free space on the storage.

File systems of the past always suffered from limited field widths. A file system which is designed for the future should always overestimate the size of each field considerably, and then a driver could always decide to implement fewer bits.

FS/Z is a file system which takes all of these considerations into practice.

Baldaszti Zoltán Tamás

Revision History

Currently this is the very first version, FS/Z 1.0 so no changes.

Introduction

We all get used to the fact that disks and removable media is capable of storing files. However on the lowest level this isn't so. Storage devices can only understand sectors, so there's a need to somehow translate a list of arbitrary length files organized in a hierarchy into a list of fixed sized chunks.

This is where a file system comes into play. So far many file systems has been developed, but at the end of the day all what they do is bookkeeping how files are represented as fixed sized blocks.

In general all file systems share the same build-up:

- **superblock**: is a structure that's unique to the file system, usually has magic bytes to allow identification and records all the necessary information to locate other parts of the file system.
- **block size** or **allocation unit**: is the size in which the file system keeps track of things. Also called logical sector size or cluster size. It cannot be smaller than the physical sector's size, and it is common practice that more consecutive physical sectors make up one logical block.
- **allocation info**: used to keep track of free blocks and which block belongs to which file.
- **meta data**: is the way how a file system stores file properties like the file's parent directory, name, size, type, access rights, last modification date etc. Some file systems consider allocation a file property, so they store it here along with the other meta data. Others only store a pointer to the first allocation record, and keep the actual allocation info separated.
- **file data**: is what you can see when you open a file in an application. It is a general expectation that all the other meta data should be kept small so that bigger part of the volume can be used for storing the actual file data.

In what file systems are different, is how the above gets implemented. Choosing the format wisely is very important, because it defines what algorithms can be used, therefore having a huge impact on the file system's overall performance, and how big part of the volume is "wasted" for storing the meta data.

The act of determining the allocation unit size and writing the necessary meta data without any file data is called "**to format a disk**". If this operation doesn't zero out file data, then that's called quick format.

Finally, file systems can be placed on the entire storage, or those can be divided into variable sized non-overlapping areas, called partitions. In this case a file system only manages the contents of one partition, and each partition may use different file systems.

Page left blank intentionally

Terms and Definitions of FS/Z

- The terms **must, must not, should, should not** and **may** are to be interpreted as described by RFC 2119.
- **physical sector** is the smallest input output unit a storage device can handle, usually 512 bytes (most common), 2048 bytes (some discs) or 4096 bytes (most notably SSDs and some newer hard disks).
- **LBA**: linear block address is a physical sector address, expressed with a numerical value. Currently 48 bit wide most of the time, but storage with 64 bit sector addresses exists. The very first physical sector of the storage is addressed by LBA 0.
- **RAID**: redundant array of inexpensive disks, is a method to create logical LBA addressing over multiple storage to increase redundancy or expand volume capacity beyond one disk.
- **partitions**: are non-overlapping areas of the storage, expressed by LBA starting address and number of physical sectors. Partitions are defined by a partitioning table, called GUID Partitioning Table, as defined by the EFI specification. Partitions are usually started on 2048 physical sector aligned addresses, so file systems start on a 1 Megabyte boundary.
- **logical sector:** (or simply **block**) is the building block of the file system, always multiple of physical sector. The smallest possible size is 2048 bytes, but most commonly it is chosen to be 4096 bytes. It's maximum size isn't limited in practice, could be 2^{266} (exponent is $2^8 + 11$).
- **LSN**: logical sector number, is a numerical value, stored on a 128 bit wide integer. The file system's first allocation block is LSN 0. Without partitions, LSN 0 equals to LBA 0. If a partition is not logical sector size aligned, then logical sector size modulo physical sector size must be added when converting. LSN *x* = LBA *x * logical sector size / physical block size + start physical address % (logical sector size / physical sector size).* For example, assuming 8 physical sectors make up a logical sector, if the file system starts on the disk, then LSN 0 = LBA 0, LSN 1 = LBA 8, LSN 2 = LBA 16 etc. If that file system is moved to a partition which starts at LBA 3, then LSN 0 = LBA 3, LSN 1 = LBA 11, LSN 2 = LBA = 19 etc. It is expected that partitions should start on a LBA address which is multiple of logical sector size, but not mandatory.
- **integer**: is a numerical value, which can be 8, 16, 32, 64, 96 or 128 bits wide. On disk always stored in a *little endian format*, for example 0x01020304 as the series of bytes 0x04, 0x03, 0x02, 0x01. A driver might choose the implement less, for example up to 64 bits only.

- **i-node**: a logical sector which stores all the properties of a file, except its name.
- **fid**: file identifier, an LSN which points to a logical sector with i-node structure in it.
- **directory**: is used to organize files into a hierarchy. It is a list of file names, and a directory might contain other directories. On disk they are stored just as any other regular file, but with special content, with a list of fixed sized directory entries.
- **directory entry**: a pair of fid and file name.
- **root directory**: is the top of the file hierarchy. All files and directories in the file system must be descendants of the root directory.
- **timestamps**: date and time is stored as number of *micro*seconds since 1970. jan. 1 00:00:00 UTC, without any timezone nor daylight saving applied. AD 1970 to AD 586912.
- **UUID**: a universally unique identifier, stored on 16 bytes.
- **ACE**: access control entry, a UUID without the last byte which identifies the principal, and access rights and permissions stored in the last byte.
- **ACL**: access control list, a list of ACEs. The first entry in the list identifies the owner (also called the control ACE), the one with the right to modify other ACE entries in the list.
- **CRC**: cyclic redundancy check, a 32 bit checksum calculated from the contents to validate integrity. Multiple methods exist, FS/Z uses the Castagnoli method with polynomial 0x1EDC6F41, the same used in network packets, and different to the ANSI CRC which is used in the EFI GPT. The reason for this choice is, many architectures have hardware accelerated calculation capabilities for the Castagnoli CRC32c variant, but not for the ANSI CRC32a.

Summary of File System Limitations

Technically the file system is unlimited, because limits are so high that in practice they never reached.

Property	Maximum allowed size
Logical sector size (allocation unit) maximum	2^{266} bytes
File system (partition) size	2^{128} logical sectors
Largest file size	2^{128} bytes
Maximum number of entries in a directory	$2^{121} - 1$ entries
Maximum number of logical sectors in one extent	2^{96} logical sectors
File name maximum length (UTF-8 encoded)	111 bytes

Overall On Disk Layout

The overall layout describes how file system meta data is organized on disk. The Logical Sector Number (LSN) indexes allocation blocks, and if the file system is on the entire disk, LSN *0* = LBA *0*, otherwise LSN *0* = first block of the partition. All logical sector addresses are relative to the file system's first block, which is therefore LSN *0*.

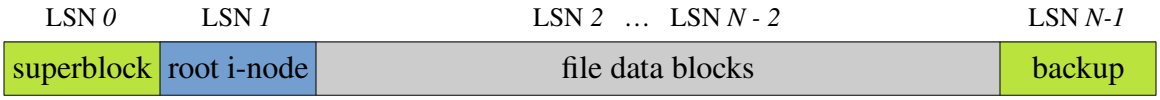


Figure: block layout of an FS/Z formatted partition or storage

The **superblock** is always stored in the very first block at LSN *0*. Optionally in the very last block, there's a **backup** copy of the superblock at LSN *N - 1*. Everything else is free to be used as file data blocks, but there must be at least one block that stores the **root i-node** of the file system. Its position is recorded in the superblock, and usually (but not necessarily) comes right after the superblock.

The backup superblock is optional, it is not an error if it's missing, however drivers should place it whenever possible and they should warn the user that file system might be truncated if it's missing.

The FS/Z file system was designed in a way that in lack of a valid superblock, other blocks can be parsed to reconstruct it entirely. However this is a slow operation, hence the superblock backup.

This layout means that quick formatting with FS/Z is indeed very quick, requires writing only 3 blocks to the storage, no more. Resizing is also quick and easy, one just have to move the superblock backup.

In case FS/Z is used to maintain a partition and not the entire storage, the partition type should reflect what that partition is used for, and not what the partition is formatted with. But in case you really don't know the former, you can use a jolly joker UUID in the GUID Partitioning Table for the latter:

{ 0x5A2F534F, 0x0000, 0x5346, { 0x2F,0x5A,0x00,0x00,0x00,0x00,0x00,0x00} }

Or with GUID notation:

5A2F534F-0000-5346-2F5A-000000000000

But as it was pointed out, partition type should reflect how the partition is used (for storing applications, for user's private data, for shared data etc.) instead.

Page left blank intentionally

The Superblock

The very first block of the file system is the superblock, regardless the size of the block, or how many physical sectors give a block. It is always at the beginning of the first logical sector at address LSN *0*.

Related fields are not necessarily grouped together, it was more important that each field must be aligned on an offset which is multiple of its size. All integer numbers are stored in *little endian* format.

Offset	Size	Field	Description
0	512	loader	reserved, exists for historical reasons
512	4	magic	magic bytes, ‘F’, ‘S’, ‘/’, ‘Z’
516	1	version_major	file system version
517	1	version_minor	file system version
518	1	logsec	logical sector size in power of two minus 11
519	1	enctype	encryption alogirthm if used
520	4	flags	file system feature flags
524	2	maxmounts	maximum number of mounts allowed before next fsck
526	2	currmounts	current number of mounts counter
528	16	numsec	total number of logical sectors
544	16	freesec	first free logical sector
560	16	rootdirfid	root directory i-node’s logical sector number
576	16	freesecfid	free area pseudo file i-node’s logical sector number
592	16	badsecfid	bad sector pseudo file i-node’s logical sector number
608	16	indexfid	search index directory i-node’s logical sector number
624	16	metafid	meta labels file i-node’s logical sector number
640	16	journalfid	journal pseudo file i-node’s logical sector number
656	16	journalhead	pointer to journal’s head logical sector number
664	16	journaltail	pointer to journal’s tail logical sector number
672	16	journalmax	number of logical sectors in the journal
680	28	encrypt	encryption key mask
708	4	enchash	password CRC
712	8	createdate	file system creation date and time (format time)

720	8	lastmountdate	date and time of last mount
728	8	lastumountdate	date and time when superblock was written
736	8	lastcheckdate	date and time when fsck was last run on this volume
744	16	uuid	file system unique identifier (same as GPT entry UUID)
760	256	reserved	reserved for future use, must be zero
1016	4	magic2	second magic bytes, ‘F’, ‘S’, ‘/’, ‘Z’
1020	4	checksum	CRC checksum of bytes at 512 - 1020
1024	1024	raidspecific	RAID specification

Legacy Loader

For historical reasons, the first 512 bytes are reserved for boot loader code. It isn’t used on modern machines, so it is just filled up with zeros. Old systems stored the MBR partitioning table here, which is not used any more. Its successor, the GPT partitioning scheme starts at the 512th byte.

File System Identification Magic Bytes

These 4 magic bytes being “FS/Z” are used to identify that the volume is using FS/Z. When a storage is partitioned, then exactly at this position there must be an “EFI PART” magic. This conflict guarantees that it can be determined if the file system is using the entire disk of if the disk is partitioned.

File System Version

The current version is 1.0 which is stored as version_major 1 and version_minor 0.

Logical Sector Size

Defines how big allocation unit is used for this file system, in power of two minus 11. Smallest possible value is 2048 bytes (encoded as 0), and the most common is 4096 bytes (encoded as 1). On big file servers, 65536 bytes (encoded as 5) is also probable. When not specified otherwise, logical sector size should be set to 4096 bytes by default. Setting the right size is very important, because having it small increases the required amount of meta data, and having it too large will waste a lot of storage space at the end of each file.

Encryption Algorithm

When encryption is used (enchash non-zero), then this field defines what cipher algorithm is used. Values 2 to 255 are reserved for future use.

Value	Define	Algorithm
0	FSZ_SB_EALG_SHACBC	SHA256 and XOR based cyclic block cipher
1	FSZ_SB_EALG_AESCBC	AES 256 cyclic block cipher

File System Feature Flags

It is a bitmask which defines the file system's features. Bits 4 - 31 are reserved for future use.

Bit	Define	Description
0	FSZ_SB_BIGINODE	I-node structures require 2048 bytes
1	FSZ_SB_JOURNAL_DATA	File data is also journaled, not just the meta data
2	FSZ_SB_SOFTRAID	If software RAID isn't used, this bit must be cleared
3	FSZ_SB_ACCESSDATE	Store last access timestamp in i-nodes

Mount Counters

When a file system is about to be used for the first time in a session (gets mounted on a directory or first referenced as an X: driveletter), the current number of mounts in currmounts must be incremented. If this counter reaches the value specified in maxmounts, then it must be set to zero, and a complete file system check must be performed before the file system could be taken into use. With both values set to zero it means no file system check forced and no mount counter feature is used.

Volume Capacity

Determined by the numsec and freesec fields. Both are 128 bits integer numbers expressed in logical sectors. The numsec contains the total number of logical sectors available on the volume minus 1. The logical sector pointed by numsec should contain the backup copy of the superblock. When the file system is defragmanted, then freesec contains the number of used sectors, otherwise it is the last used sector plus 1. Note that freesec isn't storing the number of free sectors, rather it points to the first surely available free logical sector. The value of freesec can't be larger than the value of numsec. When freesec is zero, that means the file system is in streaming or tape archive mode:

numsec should point to the backup but in this case (and only in this case) could be zero, the file system must be defragmented, file data must be either inlined or expressed using extents only, there's no free space available, and the stream is terminated by the backup copy of the superblock.

Location of the Root Directory

The field rootdirfid points to the logical sector with the root directory's i-node. It is usually (but not necessarily) right after the superblock, so it is 1. Root directory is a standard directory, except it's type in the i-node is not "dir:" but "dir:fs-root" to be recoverable.

Free Sectors Registry

When the file system is defragmented, the field freesecfid must be zero, and the first available free sector is determined by the freesec field. Otherwise freesecfid must point to an i-node with type "int:fs-free-sectors". That pseudo file's allocation records are marking the free logical sectors of the file system from the superblock to freesec. The free sectors after the last used sector always determined by the freesec field.

Bad Sectors Registry

Similarly to free sectors, bad sectors are kept track using a pseudo-file. When there are no bad logical sectors on the media, badsecfid field must be zero. Otherwise it must point to a i-node with type "int:fs-bad-sectors". That pseudo file's allocation records are marking the unusable, bad or faulty logical sectors in the file system.

Search Index

If this feature is used, then indexfid must point to an i-node with type "dir:fs-search-index". This directory must contain entries with i-node type names (like "textplain/", "imagjpeg/") pointing only files with i-node type of "int:fs-search-index". Those files contain a list of fids and full paths for quick look-up.

Meta Label Keys

When extended attributes feature is enabled in the file system, then key-value meta labels are stored (not to be confused with the file system meta data, file meta labels are just part of that). The meta label database consist of two parts: meta keys are UTF-8 encoded strings, each can be 255 bytes in size, there can be 65535 of them and they are stored in a pseudo file pointed by metafid. That i-node must have

the type “int:fs-meta-labels”. In contrast, the metasec field in the i-nodes points to the data with the meta key id value pairs. When this feature is not used, metafid must be zero.

Journaling Information

If journaling is not used, then all related fields in the superblock must be zero. Otherwise journalfid must point to an i-node with the type “int:journal” or “int:journal-data” (the latter is used if FSZ_SB_JOURNAL_DATA flag is set. This is a special file that must be pre-allocated using a single extent, cannot have versions, and its data must be placed right after its i-node. The fields journalhead and journaltail implements a circular buffer in this pre-allocated area. The size of the area is also stored in journalmax (as well as in the i-node), and if either journalhead or journaltail reaches journalfid + journalmax + 1, they must be wrapped around to journalfid + 1. When journalhead equals to journaltail, that means the journal log is empty.

Encrypted Volumes

The use of encryption is indicated by enchash not being zero, and that alone. If this feature is used, then encrypt must not be zero, and the field enctype tells the cipher to be used. The same CRC of the password is stored in enchash, regardless to the algorithm. This allows drivers to verify the password in a generalized way. The other field, encrypt contains the encryption mask or initialization vector, and its exact contents depends on which cipher was selected. With encryption, all the logical sectors except the superblock and its backup copy must be encrypted, and unused free space should be filled up with random bytes.

Creation, Last Mount, Check and Change Timestamps

All timestamps in FS/Z are stored as number of microseconds (one millionth of a second) since 1970. jan. 1 00:00:00 UTC, not using timezones nor daylight savings. Good up till AD 586912.

The createdate is set to the time of the formatting. It should not change.

The lastmountdate is set and the lastumountdate is cleared when the file system is used for the first time in a session (gets mounted on a directory or first referenced as an X: driveletter). The lastumountdate must be kept zero during a session.

When the file system is unmounted (or storage ejected), then lastumountdate is set to that time. This way a zero lastumountdate indicates that the file system wasn’t unmounted cleanly, so it might have errors.

The lastcheckdate indicates the last time when file system check was performed on the volume.

Volume Unique Identifier

This value is unique to the file system. If file system is stored on a partition, then it must be the same as the partition UUID value. It can be used to detect removable media change, and to identify RAID disks.

Reserved Area

256 bytes after that, and before the second magic is reserved for future use, must be set to zero.

Ending Magic and Checksum

The superblock is ended in the second magic2 bytes, using the same values ("FS/Z"). As it is located on another physical sector than the first magic, this makes sure of it that the entire superblock was read into memory. After that comes the checksum, which is the CRC checksum for the area between the two magics (magic bytes included), from offset 512 to the byte at 1019 inclusive.

RAID Specific Information

If software RAID feature is used, indicated by the FSZ_SB_SOFTRAID flag being set, then the field raidspecific, bytes from 1024 to 2047 are used to describe the configuration. RAID is using more devices to be seen as one, and the same superblock and raidspec is copied to all of the devices, except they all have a different unique volume identifiers. Because this a flexible configuration, RAID specification is encoded in a serialized byte stream. The specification starts with a small header, which includes the number of definitions:

Offset	Size	Field	Description
1024	4	magic	magic bytes, 'F', 'S', 'R', 'D'
1028	4	checksum	checksum from 1032 to the end of the records
1032	1	numdef	number of definition records

This header is followed by numdef times varying length records.

Offset	Size	Field	Description
0	1	type	either 0, 1, 5 or 0x80, 0x81, 0x85
1	1	numdisk	(n) number of disks in this definition record
2	n / n * 16	disks	n * 1 byte (if type & 0x80) or n * 16 bytes UUIDs

If the RAID type byte has its most significant bit set, then each disk device is stored as one byte, an index to one of the previous RAID definition records (starting from 0). If most significant bit is cleared, then each device is stored as 16 bytes, with their corresponding unique volume identifier.

In an asymmetric arrangement, one might need to wrap a UUID in a record so that it could be referenced by a record index. For that RAID0 with one device can be used. Otherwise RAID0 allows any number of disks, RAID1 must have at least two disks and should have no more, and RAID5 must have exactly 3 disks.

Type could mean concatenation (RAID0) in striped mode, which uses the following scheme: LBA n is on device n modulo number of devices, the sector n / number of devices. So for example with two devices LBA 0 is device A's first sector, LBA 1 is device B's first sector, LBA 2 is device A's second sector etc.

Another option is mirroring (RAID1) of devices. Here all sectors are duplicated on all devices: LBA 0 maps to the first sector on both device A and B, LBA 1 is the second sector on both device A and B, etc.

Parity bit can be saved with XORed blocks (RAID5). This requires exactly three devices. Two stores the data, the third the XORed block. Which one is used for the XORed block is rotated. For example LBA 0 is device A's first sector. LBA 1 is device B's first sector. The XORed block of these two is stored in device C's first sector. LBA 2 is device B's second sector, LBA 3 is device C's second sector, then the XORed block is stored in device A's second sector. LBA 3 is device A's third sector, LBA 4 is device C's third sector, the XORed block is in device B's third sector. Then the permutation repeats: LBA 5 is on device A, LBA 6 is on device B, XORed block on device C, etc.

In addition to these, all combinations can be used too. For example a RAID1 mirrored disk can be used in a RAID0 concatenation, or more RAID5 disks can be mirrored with RAID1. Note that while the RAID specification encoding allows any combinations, not all makes sense.

Page left blank intentionally

The I-Node

This is the most important structure of the FS/Z file system. One common format is used to describe all the different file system meta data types, as well as files and directories.

The i-node structure is placed at the beginning of a logical sector, and its size is independent to the logical sector size. It is 1024 bytes unless FSZ_SB_BIGINODE flag is set, in which case it's 2048 bytes. The remaining bytes of the logical sector are used for inlining data.

Offset	Size	Field	Description
0	4	magic	magic bytes, 'F', 'S', 'I', 'N'
4	4	checksum	CRC checksum of bytes 8 to 1024 (or 2048)
8	4	filetype	first 4 bytes of the main mime type
12	60	mimetype	first 60 bytes of the sub mime type
72	8	createdate	i-node creation timestamp
80	8	changedate	i-node last status change timestamp
88	8	accessdate	i-node last access timestamp
96	8	numblocks	number of allocated blocks for this i-node
104	8	numlinks	number of references to this i-node
112	16	metasec	logical sector number of the meta label block
128	64	version5	file version (oldest)
192	64	version4	file version
256	64	version3	file version
320	64	version2	file version
384	64	version1	file version
448	16	sec	file data LSN for current (or only) version
464	16	size	file size for current version
480	8	modifydate	file modification timestamp for current version
488	8	flags	file allocation flags for current version
496	16	owner	file owner for current version, control ACE
512	512 / 1536	groups	access control list (512 bytes unless big inode used)

Magic and Checksum

The i-node always starts with a 4 bytes magic “FSIN”. This is followed by the checksum, which is the CRC of bytes 8 to the end of the i-node, byte at 1023 (or 2047) inclusive.

I-Node Type

The main part is in filetype. For files the 4th byte is never a ‘:’, and for FS/Z specific i-nodes it is.

filetype	Description
text	text file
imag	image file
vide	video file
audi	audio file
appl	application or other binary file
boot	boot loader application (same as “appl”, but must not be relocated)
dir:	directory
uni:	directory union
lnk:	symbolic link
pip:	named pipe (FIFO)
dev:	device file
sck:	socket file
int:	internal FS/Z meta data

The mimetype field contains the sub mime type, like “html”, “jpeg” etc. For FS/Z special i-nodes:

mimetype	Description
fs-root	only with “dir:”, marks the root directory in the file system
fs-free-sectors	only with “int:”, marks the free logical sectors
fs-bad-sectors	only with “int:”, marks the bad or unusable logical sectors
fs-search-index	only with “dir:” or “int:”, mime type search index
fs-meta-labels	only with “int:”, file listing the available meta label keys
fs-journal	only with ”int:”, journal for meta data
fs-journal-data	only with “int:”, journal for both meta and file data

I-Node Creation, Change and Access Timestamps

Uses the common timestamp format like the superblock, number of microseconds since 1970. jan. 1 00:00:00 UTC, not using timezones nor daylight savings.

The createdate field is set when the i-node is created, and it should not change.

The changedate field is set when the i-node is modified (and not when file data is modified, see modifydate for that).

The accessdate field is set when the i-node is accessed. This is an optional feature, this field could be left as zero. Only used when FSZ_SB_ACCESSDATE in superblock flags is set.

Number of Allocated Logical Sectors

The field numblocks contains the total number of all additional logical sectors allocated for this i-node, including the sector directories, sector lists and data sectors for all versions, but not including the i-node itself nor the gaps in sparse files.

Number of Links

The field numlinks counts the number of references to this i-node in the directory entries or in the superblock. Special FS/Z i-nodes always have this field as 1. The root directory also starts with one.

Meta Label Values

When extended attributes feature is enabled in the file system, then key-value meta labels are stored for this i-node on the logical sector pointed by metasec. The keys are described in a pseudo-file pointed by the metafid field in the superblock. When that's zero, metasec in i-nodes should also be.

File Versions

The FS/Z file system is capable to store historical records of files, up to 5 old versions. This feature can be turned on and off on a per i-node basis, by using the FSZ_IN_FLAG_HIST flag.

When used, older version records are copies of the fields sec to owner (from offset 448 to 511, 64 bytes), but with their own sec and flags value. When a new version is saved, all the versions are shifted by 64 bytes towards the beginning of the i-node. If the oldest version is not zero, then logical sectors allocated for that are freed, except if those are also used in newer version's allocation. The latest, which is the current (or only) version starts at offset 448.

File Modification Timestamp

The time when the file's content was modified is stored in the modifydate field.

File Allocation Flags

The way how allocation is taken care of, and how file data is translated into LSNs, stored in the flags and in the sec fields. Flag is a bitset, in which the least significant byte selects the translation. The bits 10 to 63 are reserved for future use.

Bit	Define	Description
0 - 7	FSZ_IN_FLAG_INLINE FSZ_IN_FLAG_DIRECT FSZ_IN_FLAG_SD0 FSZ_IN_FLAG_SD*x* FSZ_IN_FLAG_SECLIST0 FSZ_IN_FLAG_SECLIST*x*	inlined data direct reference inlined sector directory level *x* indirect sector directory inlined sector list level *x* indirect sector list
8	FSZ_IN_FLAG_HIST	old versions of the file are kept
9	FSZ_IN_FLAG_CHKSUM	file has content data checksums too

Permissions and Access Control List

Depending on the i-node size, the field groups can store either 32 or 96 entries, each being 16 bytes.

These are UUIDs, universally unique identifiers of the principals, without the last byte. That byte contains the permission bits, and also one bit identifies the UUID as a group, or as an individual user.

Even though the field owner is stored along with the file versions, and copied with them, the owner field of the current version is also the very first entry in the access control list. As such, it does not only specify the file permission bits for the owner, but also specifies who has the right to modify the access control list, hence also called the control ACE. When there are exactly two elements in the groups field, and the first being a group ACE, and the second being a group ACE with a special "*" principal **0000002A-0000-0000-0000-0000000000*xx*** (which stands for catch all mask), then FS/Z i-node mimics POSIX access permissions, owner's permission stored in byte 511, group permissions in byte 527 and world permissions in byte 543. Otherwise a file could belong to multiple groups, or different users could have different access permissions to it.

An ACE with full zeros (no matter the permission bits) always terminates the list, otherwise there can be as many ACE as the size of the i-node allows. The i-node must be padded with zeros.

By default, only the file owner has any access to the file, and any kind of access to others is denied (default deny policy). He is also the one who can change permission bits in the owner field, and who can add or more ACE entries in the groups field.

The permissions are stored in the last (16th) byte of the UUID.

Offset	Size	Field	Description
0	4	Data1	first part of the UUID
4	2	Data2	second part of the UUID
6	2	Data3	third part of the UUID
8	7	Data4	fourth part of the UUID, without the last byte
15	1	access	permission bits, see below

Bit	Define	Description
0	FSZ_READ	read access
1	FSZ_WRITE	write access
2	FSZ_EXEC	execution on files or list directory on directories and unions
3	FSZ_APPEND	only allows extending the file, but not modifying already existing data
4	FSZ_DELETE	permission to remove the file or directory
5	FSZ_GROUP	is set if the ACE's principal is a group and not an individual user
6	FSZ_SUID	set user on execution, file creation on directories
7	FSZ_GUID	set groups on execution, inherit (copy) ACL on directories

Permissions FSZ_WRITE and FSZ_APPEND are mutually exclusive. Normally whether a file can be deleted or not depends on the write permission of the parent directory. However it might be needed that a certain user should be allowed to remove a file (a lockfile for example), without giving write access to the entire directory. That's what FSZ_DELETE permission is for. When FSZ_EXEC is granted on files, it means that the file is executable. When granted to directories, it means the principal is allowed to list the directory's contents, otherwise they can only access files for which they know the filename. On executable files, FSZ_SUID sets the process' owner to the file's owner. On directories this means that all newly created files and sub-directories will inherit the owner of the parent directory. Finally FSZ_SGUID is similar, but it means the process or the newly created files will inherit the ACL.

Page left blank intentionally

Allocation

There are three translation schemes that can be used to translate file offsets into logical sector blocks. Each file version define these independently, and there are three fields in each version that control allocation: size (the size of the file), sec (pointer to allocation data) and flags (file allocation flags).

Allocation Structures

Sector List

Used to implement extents and marking free sector areas on disks. A list item contains a starting LSN and the number of logical sectors in that area, describing varying length contiguous areas on disk. Unused entries must be set to zero. Independent to the FSZ_IN_FLAG_CHKSUM flag.

Offset	Size	Field	Description
0	16	sec	LSN, pointer to the next level (up to 2^{128} logical sectors)
16	12	numsec	number of logical sectors in this extent (up to 2^{96} sectors)
28	4	checksum	CRC of the data in the pointed logical sector(s)

Sector Directory

A logical sector that has *logical sector size / 16* entries. Unused entries must be set to zeros. The building block of memory paging like translations. Unlike sector lists, sector directories describe fix sized, non-contiguous areas on disk.

When FSZ_IN_FLAG_CHKSUM is clear, then on each level:

Offset	Size	Field	Description
0	16	sec	LSN, pointer to the next level (up to 2^{128} logical sectors)

When FSZ_IN_FLAG_CHKSUM is set:

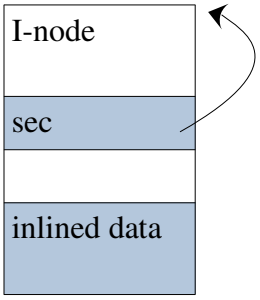
Offset	Size	Field	Description
0	12	sec	LSN, pointer to the next level (up to 2^{96} logical sectors)
12	4	checksum	CRC of the data in the pointed logical sector

Translations

Inlined Data

Can only be used for the current (or only) version, when the file size is less than or equal to *logical sector size* minus the *i-node structure's size* (that is 1024 bytes, unless FSZ_SB_BIGINODE flag is set, in which case it's 2048 bytes).

Here appropriate version's sec points to the i-node, and in this case the appropriate version's flag is FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_INLINE (0xff).

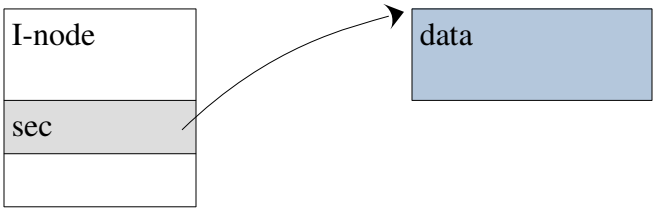


Direct Data

This can be used by all versions, but only if file size is smaller than or equal to *logical sector size*.

Here sec points to the data sector directly, and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_DIRECT (0).



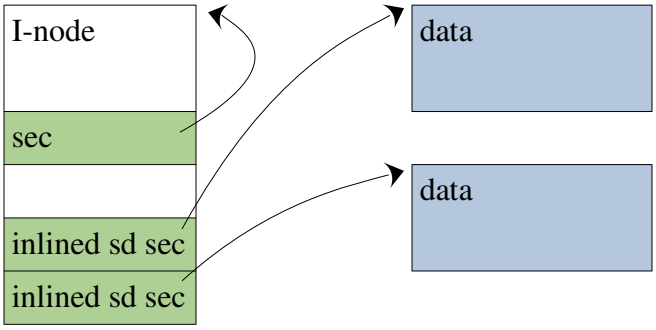
Inlined Sector Directory

Can only be used for the current (or only) version, when the file size is less than or equal to

((logical sector size minus the *i-node structure's size) / 16) * logical sector size*.

Here version's sec points to the i-node, each sec in the inlined sector directory points to data sectors directly, version's flag is

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SD0 (0x7F).



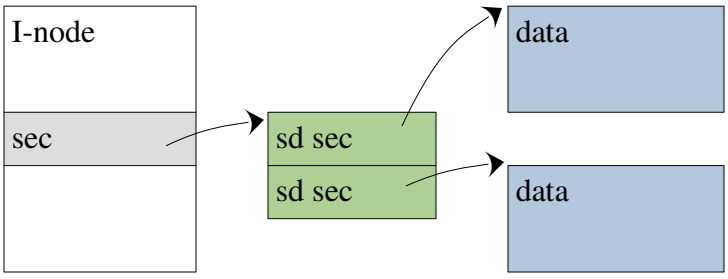
Indirect Sector Directory

This can be used by all versions, but only if file size is smaller than or equal to

*(logical sector size / 16) * logical sector size*.

Here sec points to the sector directory, in which each sec points to the data sectors and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SD1 (1).



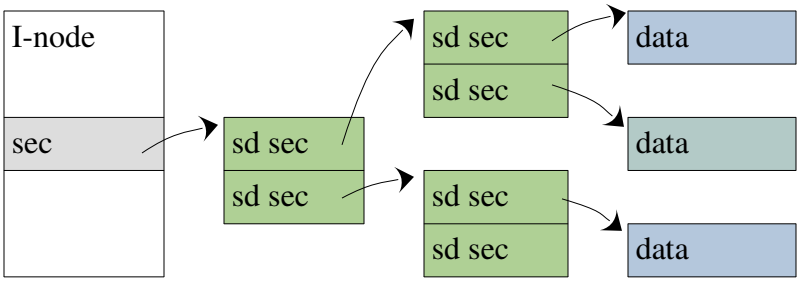
Double Indirect Sector Directory

This can be used by all versions, but only if file size is smaller than or equal to

(*logical sector size / 16)* ^{2} * *logical sector size*.

Here sec points to sector directory, where each sec points to another sector directory in which sec points to the data sectors and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SD2 (2).



Triple Indirect Sector Directory

This can be used by all versions, but only if file size is smaller than or equal to

(*logical sector size / 16)* ^{3} * *logical sector size*.

Here sec points to the sector directories in three levels, and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SD3 (3).

Quadriple Indirect Sector Directory

This can be used by all versions, but only if file size is smaller than or equal to

(*logical sector size / 16)* ^{4} * *logical sector size*.

Here sec points to sector directories in four levels, and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SD4 (4).

This scheme with indirect sector directories can go up to 11 levels, when file size is not larger than

(*logical sector size / 16)* ^{11} * *logical sector size*.

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SD11 (11).

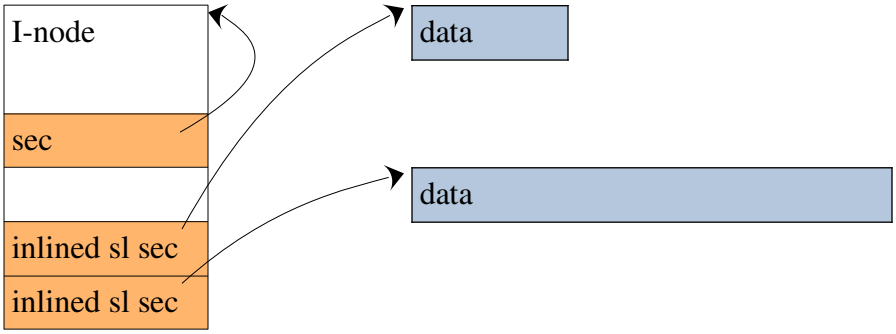
Inlined Sector List

Can only be used for the current (or only) version, when there are no more extents than

(logical sector size minus the *i-node structure's size) / 32*.

Here version's `sec` points to the i-node, and each `sec` in the sector list points to data sectors directly, version's flag is

`FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SECLIST0` (0x80).



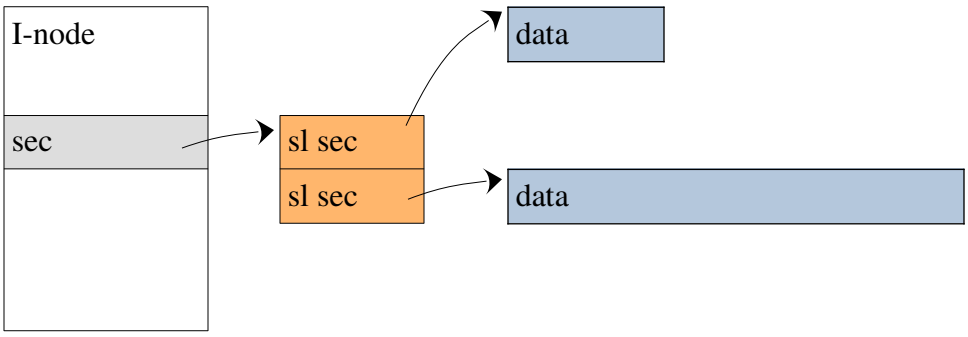
Normal Sector List

This can be used by all versions, when there are no more extents than

(logical sector size / 32).

Here version's `sec` points to a logical sector with sector list, and each `sec` in the sector list points to data sectors directly, version's flag is

`FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SECLIST1` (0x81).



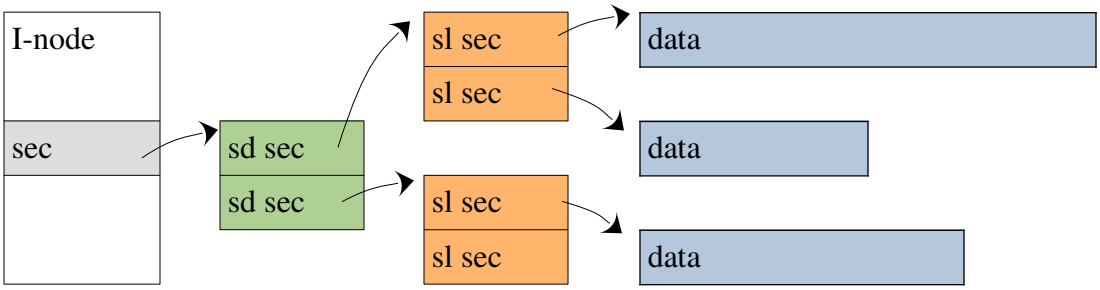
Indirect Sector List

This can be used by all versions, when there are no more extents than

(*logical sector size / 16) * (logical sector size / 32)*.

Here sec points to sector directory, where each sec points to a sector list in which sec points to the data sectors and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SECLIST2 (0x82).



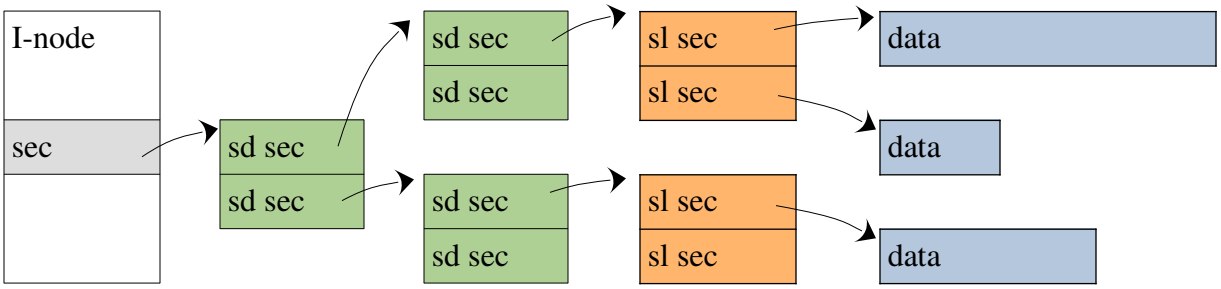
Double Indirect Sector List

This can be used by all versions, when there are no more extents than

(*logical sector size / 16)* ^{2} * *(logical sector size / 32)*.

Here sec points to sector directory, where each sec points to another sector directory where each sec points to a sector list in which sec points to the data sectors and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SECLIST3 (0x83).



Triple Indirect Sector List

This can be used by all versions, when there are no more extents than

(*logical sector size / 16)* ^{3} * *(logical sector size / 32)*.

Here sec points to sector directory in three levels, where last level's sec points to a sector list in which sec points to the data sectors and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SECLIST4 (0x84).

Quadriple Indirect Sector List

This can be used by all versions, when there are no more extents than

(*logical sector size / 16)* ^{4} * *(logical sector size / 32)*.

Here sec points to sector directory in four levels, where last level's sec points to a sector list in which sec points to the data sectors and

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SECLIST5 (0x85).

This scheme can go up to 9 levels, when there are no more extents than

(*logical sector size / 16)* ^{8} * *(logical sector size / 32)*.

FSZ_FLAG_TRANSLATION(flags) = FSZ_IN_FLAG_SECLIST9 (0x89).

Sparse Files

These are files with gaps in them. If either a sector directory or a sector list references an LSN 0, that means a block full of zeros. For sector directories the block's size depends on the level where the LSN 0 appeared (at least one logical sector size), and for sector lists the size is given in the extent in numsec.

There are two possible ways how to create a sparse file with gaps. Either an application seeks over the file size before it writes something, or when a logical sector aligned data in multiple of logical sector size full of zeros are written to a file. A driver must always check for the latter too, because empty logical sectors must not be included in the i-node's numblocks field.

Bookkeeping Free Space

The way how free space is recorded is pretty simple. No special structures, no bitmaps, it uses exactly the same i-node structure as normal files.

There are two fields in the superblock to keep track of free space:

The freesec field always points to the last used logical sector plus one. This is the first available free sector's LSN.

The freesecfid field must be zero if the file system is defragmented, and free of external fragmentation. Otherwise it must point to an i-node with the special type "int:fs-free-sectors". This pseudo file isn't actually used, rather it's allocation info covers the free gaps between the superblock and the freesec LSN.

When a file or directory gets deleted, which isn't the very last in the file system, then it's allocation info is copied into the pseudo file's allocation info pointed by freesecfid. Otherwise the number of freed logical sectors are simply subtracted from freesec.

When a new logical sector is allocated, then first freesecfid must be checked. If it's not zero, then the block must be taken away from the pseudo file's allocation info, and added to whereever the block is needed. It is not mandatory, but for effectiveness strongly recommended that the free sectors pseudo file should use sector list translation. When all the logical sectors are taken away, then the pseudo file's i-node should be freed too, and if done, then the freesecfid field must be cleared in the superblock (not always possible, because freeing the free sector pseudo file's i-node might create a new gap. In that case it is okay to have an empty free sectors pseudo file).

If the pseudo file is empty, then freesec must be checked if it's smaller than numsec. If not, that means a "No space left on device" error. Otherwise the logical sector pointed by freesec must be returned and freesec incremented by one. If the returned LSN is recorded in the bad sectors pseudo file, then the whole process must be repeated to return another logical sector instead.

Bookkeeping Bad Sectors

If there are bad, faulty or otherwise unusable logical sectors on the storage, then the badsecfid field in the superblock must point to an i-node with the special type "int:fs-bad-sectors". This pseudo file isn't actually used, rather it's allocation info covers the bad sectors on the volume.

It is very similar to free sector's pseudo file, but because bad sectors are often scattered through the disk, for effectiveness it is recommended that the bad sector pseudo file should use sector directory translation.

Logical sectors that are recorded in the bad sectors allocation info, must not appear anywhere else in the file system's meta data.

The File Hierarchy

The Root Directory

Files are grouped together in directories. Those directories can contain other directories, and by so creating a hierarchy. It starts with a root directory, who's i-node is pointed by the rootdirfid field in the superblock. The root directory always must exists, and its i-node type is different to the rest, so that it can be recovered, it's not "dir:", but "dir:fs-root".

Otherwise directories are just like files, they use exactly the same allocation, and they use exactly the same permissions and access control list. Their contents can be inlined in the i-node just like files, so a minimal root directory needs one logical sector, a combined i-node and file list.

There are two things in directories differ from files: they cannot have versions (there's only a current version of them), and their file contents consist of fixed sized records. A directory always has a header record and may have other records, therefore it's file size is 128 bytes at minimum, and it is always multiple of 128 bytes.

Directory Header

The directory header has the same size as the entry records, 128 bytes and looks like this:

Offset	Size	Field	Description
0	4	magic	magic bytes 'F', 'S', 'D', 'R'
4	4	checksum	CRC of the directory entries, from byte 16 to end
8	1	display_type	GUI preferences: display type (icon, list, detailed etc.)
9	1	sorting_order	GUI preferences: the sorting order of the directory
10	6	reserved	GUI: preferences: reserved for future use
16	16	numentries	number of entries in this directory
32	16	fid	back reference to the directory's i-node
48	79	reserved	reserved for future use, must be zero
127	1	flags	directory flags

The main purpose of the header is make the directories recoverable.

Otherwise its fields aren't really used, except GUI file managers can store their user preferences for the directory here. Note how the GUI preferences **aren't part of the checksum**.

If the directory entries are different to the standard, then directory flags can be used to indicate that.

Bit	Define	Description
0	FSZ_DIR_FLAG_UNSORTED	the entries aren't lexicographically sorted (shouldn't happen)
1	FSZ_DIR_FLAG_HASHED	the entries are stored using a hash algorithm

Directory Entries

After the header comes a list of entries, always lexicographically sorted. This means a bit slower directory creation, but allows extremely fast O($\log_2$) look-ups using libc's bsearch (binary search). Note that sorting_order field in the header is just a GUI preference for a displaying option, it does not influence how entries are stored on disk.

There are *file size / 128* entries, or as indicated in the header's numentries field. Those two must always match. The size of an entry is always 128 bytes, padded with zeros.

Offset	Size	Field	Description
0	16	fid	the pointed i-node
16	112	name	a zero terminated and padded, UTF-8 encoded filename

Unlike other file systems, in FS/Z the current directory "." and the parent directory ".." are **not stored** as an entry. The current directory's fid is recorded in the header for recovery, but otherwise not used. The parent directory can't be determined as with symlinks and directory unions multiple parents may exist. A driver therefore must always do string operation on the path to get the parent's path and look up that in their internal cache to get the correct i-node for the parent directory in a certain path.

Filenames are UTF-8 encoded, must be non-empty, and must not contain the characters zero, '/' (slash, directory separator) and ';' (semicolon, version separator). They must not start with '/' either, but if the fid in the directory entry points to another directory record, then the filename must be ended in '/'. This way directories are visually distinguished from other file types. The character ';' (semicolon) is reserved for indicating the file versions in paths: ";0" refers to the current version (can be omitted), ";-1" is the version before, ";-2" version before that, etc. up to ";-5" which refers to the oldest version (similar to FILES11 and ISO9660). The length limit of 111 might seem small compared to other file systems' 255 limit, but in reality more than enough, you'll never run out of it with every-day file names. For example, "*The hundred years old man who climbed out of the window and disappeared.mp4*" is 75 bytes long.

Special Files

Named Pipes (FIFO)

I-node type “pip:”. Named pipes has no file content.

Symbolic Links

I-node type “lnk:”. Symbolic links are stored the same way as any file, their file content being the link target path.

Directory Unions

I-node type “uni:”. Are symbolic like constructs, paths are listed as zero terminated UTF-8 strings, ended in a zero byte. For example: “/bin(zero)/usr/bin(zero)(zero)”.

Device Files

I-node type “dev:”. Their file content is always inlined, and looks like this

Offset	Size	Field	Description
0	16	major	the device major number
16	16	minor	the device minor number
32	1	type	0 for character devices, 1 for block devices

Socket Files

I-node type “sck:”. They contain a 8 bytes long numeric id.

Search Index

I-node type “dir:fs-search-index”. This directory must contain entries with i-node type names (like “textplain/”, “imagjpeg/”), but names starting with “dir:” or “int:” are not allowed. The fid field in the directory entries pointing only files with i-node type of “int:fs-search-index”. Those files contain a list of fids and full paths for quick look-up.

I-node type “int:fs-search-index”. Is just a simple file containing records. Each record starts with a 16 bytes fid, followed by a zero terminated, UTF-8 full path, starting from the root directory, without the leading ‘/’. There’s no limit how long a path can be.

Meta Label Keys

I-node type “int:fs-meta-labels”. Encoded the same way as directory unions, zero terminated, UTF-8 encoded keys ended with a zero. Keys can’t be longer than 255 bytes, and there can be maximum 65535 different keys on a file system.

Meta Label Values

These doesn’t have an i-node type because they are pointed by metasec field in the i-nodes. Instead they have a header. They are allocated on contiguous sectors, and counted in the i-node’s numblocks field. When they grow beyond logical sector size, and the sector right after them isn’t free, then these must be relocated, and the metasec field must be updated to point to the new starting logical sector.

They start with a small, fixed sized header for recoverability:

Offset	Size	Field	Description
0	4	magic	magic bytes ‘F’, ‘S’, ‘M’, ‘T’
4	4	checksum	CRC of the data, from byte 8 to end
8	2	numentries	number of key-value pairs
10	2	size	size of the entire meta block, including this header
12	4	reserved	reserved for future use
16	16	fid	back reference to the i-node

This is followed by numentries variable length records

Offset	Size	Field	Description
0	2	keyid	meta label key id
2	2	length	(n) length of the (probably binary) data value
4	n	value	meta label value

The Journal File

I-node type “int:fs-journal” or “int:fs-journal-data” (the latter is used if FSZ_SB_JOURNAL_DATA superblock flag is set. This is a special file that must be pre-allocated using a single extent, cannot have versions, and its data must be placed right after its i-node. The fields journalhead and journaltail implements a circular buffer in this pre-allocated area. Each write to this circular buffer starts with a transaction record, followed by data sectors.

Offset	Size	Field	Description
0	4	magic	magic bytes ‘F’, ‘S’, ‘T’, ‘R’
4	4	checksum	CRC of the sector list, from byte 8 to end
8	8	numentries	number of extents in the transaction
16	8	transdate	transaction’s timestamp
24	8	reserved	reserved for future use, must be zero

This transaction header is followed by numentries times extents in the same format as in sector list, padded to be logical sector size, followed by the logical sectors with the data in order. The padding and the data sectors are not included in the checksum, however extents have their own checksums for the data sectors.

If the type is “int:fs-journal” and FSZ_SB_JOURNAL_DATA superblock flag is not set, then only meta data sectors are written to the journal, but not file data sectors. If type is “int:fs-journal-data”, and FSZ_SB_JOURNAL_DATA superblock flag is set, then both meta data and file data sectors are written.

The CRC32c Calculation Algorithm

The following code can be used to calculate the checksums on an FS/Z file system.

```
/* precalculated CRC32c lookup table for polynomial 0x1EDC6F41 (castagnoli-crc) */
uint32_t crc32c_lookup[256]={
    0x00000000L, 0xF26B8303L, 0xE13B70F7L, 0x1350F3F4L, 0xC79A971FL, 0x35F1141CL, 0x26A1E7E8L, 0xD4CA64EBL,
    0x8AD958CFL, 0x78B2DBCCL, 0x6BE22838L, 0x9989AB3BL, 0x4D43CFD0L, 0xBF284CD3L, 0xAC78BF27L, 0x5E133C24L,
    0x105EC76FL, 0xE235446CL, 0xF165B798L, 0x030E349BL, 0xD7C45070L, 0x25AFD373L, 0x36FF2087L, 0xC494A384L,
    0x9A879FA0L, 0x68EC1CA3L, 0x7BBCEF57L, 0x89D76C54L, 0x5D1D08BFL, 0xAF768BBCL, 0xBC267848L, 0x4E4DFB4BL,
    0x20BD8EDEL, 0xD2D60DDDL, 0xC186FE29L, 0x33ED7D2AL, 0xE72719C1L, 0x154C9AC2L, 0x061C6936L, 0xF477EA35L,
    0xAA64D611L, 0x580F5512L, 0x4B5FA6E6L, 0xB93425E5L, 0x6DFE410EL, 0x9F95C20DL, 0x8CC531F9L, 0x7EAEB2FAL,
    0x30E349B1L, 0xC288CAB2L, 0xD1D83946L, 0x23B3BA45L, 0xF779DEAEL, 0x05125DADL, 0x1642AE59L, 0xE4292D5AL,
    0xBA3A117EL, 0x4851927DL, 0x5B016189L, 0xA96AE28AL, 0x7DA08661L, 0x8FCB0562L, 0x9C9BF696L, 0x6EF07595L,
    0x417B1DBCL, 0xB3109EBFL, 0xA0406D4BL, 0x522BEE48L, 0x86E18AA3L, 0x748A09A0L, 0x67DAFA54L, 0x95B17957L,
    0xCBA24573L, 0x39C9C670L, 0x2A993584L, 0xD8F2B687L, 0x0C38D26CL, 0xFE53516FL, 0xED03A29BL, 0x1F682198L,
    0x5125DAD3L, 0xA34E59D0L, 0xB01EAA24L, 0x42752927L, 0x96BF4DCCL, 0x64D4CECFL, 0x77843D3BL, 0x85EFBE38L,
    0xDBFC821CL, 0x2997011FL, 0x3AC7F2EBL, 0xC8AC71E8L, 0x1C661503L, 0xEE0D9600L, 0xFD5D65F4L, 0x0F36E6F7L,
    0x61C69362L, 0x93AD1061L, 0x80FDE395L, 0x72966096L, 0xA65C047DL, 0x5437877EL, 0x4767748AL, 0xB50CF789L,
    0xEB1FCBADL, 0x197448AEL, 0x0A24BB5AL, 0xF84F3859L, 0x2C855CB2L, 0xDEEEDFB1L, 0xCDBE2C45L, 0x3FD5AF46L,
    0x7198540DL, 0x83F3D70EL, 0x90A324FAL, 0x62C8A7F9L, 0xB602C312L, 0x44694011L, 0x5739B3E5L, 0xA55230E6L,
    0xFB410CC2L, 0x092A8FC1L, 0x1A7A7C35L, 0xE811FF36L, 0x3CDB9BDDL, 0xCEB018DEL, 0xDDE0EB2AL, 0x2F8B6829L,
    0x82F63B78L, 0x709DB87BL, 0x63CD4B8FL, 0x91A6C88CL, 0x456CAC67L, 0xB7072F64L, 0xA457DC90L, 0x563C5F93L,
    0x082F63B7L, 0xFA44E0B4L, 0xE9141340L, 0x1B7F9043L, 0xCFB5F4A8L, 0x3DDE77ABL, 0x2E8E845FL, 0xDCE5075CL,
    0x92A8FC17L, 0x60C37F14L, 0x73938CE0L, 0x81F80FE3L, 0x55326B08L, 0xA759E80BL, 0xB4091BFFL, 0x466298FCL,
    0x1871A4D8L, 0xEA1A27DBL, 0xF94AD42FL, 0x0B21572CL, 0xDFEB33C7L, 0x2D80B0C4L, 0x3ED04330L, 0xCCBBC033L,
    0xA24BB5A6L, 0x502036A5L, 0x4370C551L, 0xB11B4652L, 0x65D122B9L, 0x97BAA1BAL, 0x84EA524EL, 0x7681D14DL,
    0x2892ED69L, 0xDAF96E6AL, 0xC9A99D9EL, 0x3BC21E9DL, 0xEF087A76L, 0x1D63F975L, 0x0E330A81L, 0xFC588982L,
    0xB21572C9L, 0x407EF1CAL, 0x532E023EL, 0xA145813DL, 0x758FE5D6L, 0x87E466D5L, 0x94B49521L, 0x66DF1622L,
    0x38CC2A06L, 0xCAA7A905L, 0xD9F75AF1L, 0x2B9CD9F2L, 0xFF56BD19L, 0x0D3D3E1AL, 0x1E6DCDEEL, 0xEC064EEDL,
    0xC38D26C4L, 0x31E6A5C7L, 0x22B65633L, 0xD0DDD530L, 0x0417B1DBL, 0xF67C32D8L, 0xE52CC12CL, 0x1747422FL,
    0x49547E0BL, 0xBB3FFD08L, 0xA86F0EFCL, 0x5A048DFFL, 0x8ECEE914L, 0x7CA56A17L, 0x6FF599E3L, 0x9D9E1AE0L,
    0xD3D3E1ABL, 0x21B862A8L, 0x32E8915CL, 0xC083125FL, 0x144976B4L, 0xE622F5B7L, 0xF5720643L, 0x07198540L,
    0x590AB964L, 0xAB613A67L, 0xB831C993L, 0x4A5A4A90L, 0x9E902E7BL, 0x6CFBAD78L, 0x7FAB5E8CL, 0x8DC0DD8FL,
    0xE330A81AL, 0x115B2B19L, 0x020BD8EDL, 0xF0605BEEL, 0x24AA3F05L, 0xD6C1BC06L, 0xC5914FF2L, 0x37FACCF1L,
    0x69E9F0D5L, 0x9B8273D6L, 0x88D28022L, 0x7AB90321L, 0xAE7367CAL, 0x5C18E4C9L, 0x4F48173DL, 0xBD23943EL,
    0xF36E6F75L, 0x0105EC76L, 0x12551F82L, 0xE03E9C81L, 0x34F4F86AL, 0xC69F7B69L, 0xD5CF889DL, 0x27A40B9EL,
    0x79B737BAL, 0x8BDCB4B9L, 0x988C474DL, 0x6AE7C44EL, 0xBE2DA0A5L, 0x4C4623A6L, 0x5F16D052L, 0xAD7D5351L };
uint32_t crc32_calc(unsigned char *start,int length) {
    uint32_t crc32_val=0;
    while(length--) crc32_val=(crc32_val>>8)^crc32c_lookup[(crc32_val&0xff)^(unsigned char)*start++];
    return crc32_val;
}
```

Examples

Superblock HexDump

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200  46 53 2f 5a 01 00 01 00  00 00 00 00 ff 00 00 00  |FS/Z............|
00000210  00 10 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000220  05 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000230  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000240  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000002c0  00 00 00 00 00 00 00 00  80 a9 ab 02 32 be 05 00  |............2...|
000002d0  80 a9 ab 02 32 be 05 00  80 a9 ab 02 32 be 05 00  |....2.......2...|
000002e0  00 00 00 00 00 00 00 00  3d 3f 63 19 b5 92 e2 03  |........=?c.....|
000002f0  08 05 67 60 71 96 c7 e7  00 00 00 00 00 00 00 00  |..g`q...........|
00000300  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000003f0  00 00 00 00 00 00 00 00  46 53 2f 5a 0d 17 e1 16  |........FS/Z....|
```

loader magic version logsec enctype flags maxmounts currmounts numsec freesec rootdirfid checksum

Logical sector size 4096 bytes (1 << (logsec + 11)), number of total logical sectors 0x1000, first free logical sector 5, root directory's i-node at logical sector 1.

I-Node HexDump

```
00001000  46 53 49 4e fd fa ac 97  64 69 72 3a 66 73 2d 72  |FSIN....dir:fs-r|
00001010  6f 6f 74 00 00 00 00 00  00 00 00 00 00 00 00 00  |oot.............|
00001020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001040  00 00 00 00 00 00 00 00  c0 95 b4 36 32 be 05 00  |...........62...|
00001050  80 a9 ab 02 32 be 05 00  00 00 00 00 00 00 00 00  |....2...........|
00001060  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
00001070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000011c0  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000011d0  80 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000011e0  80 a9 ab 02 32 be 05 00  ff 00 00 00 00 00 00 00  |....2...........|
000011f0  72 6f 6f 74 00 00 00 00  00 00 00 00 00 00 00 17  |root............|
00001200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

magic checksum mimetype createdate changedate accessdate numblocks numlinks sec size modifydate flags owner permissions

Mime type root directory, number of allocated blocks 0 (inlined data), number of links 1, data sector 1 (inlined), file size 0x180 bytes, flags 0xFF (translation FSZ_IN_FLAG_INLINE). Owner is 756F6F72-0000-0000-0000-000000, has read, write, execute and delete permissions.

Directory HexDump

```
00001400  46 53 44 52 c4 ff f1 e2  00 00 00 00 00 00 00 00  |FSDR............|
00001410  02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001420  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001430  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001480  02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001490  61 2f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |a/..............|
000014a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001500  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00001510  62 2f 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |b/..............|
00001520  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

magic checksum GUI properties numentries fid (back reference) fid (1st entry's) filename (1st entry's) fid (2nd entry's) filename (2nd entry's)

No displaying user preferences for the directory, has 2 entries, its i-node is at LSN 1.

The first entry's i-node is at LSN 2, and its name is "a/" (so it is a directory).

The second entry's i-node is at LSN 4, and its name is "b/" (is a directory).

(Hah, 42 pages in total, that cannot be a coincidence!)