

The Second Extended File System

Internal Layout

Dave Poirier

`instinc@users.sf.net`

The Second Extended File System: Internal Layout

by Dave Poirier

Copyright © 2001-2002 by Dave Poirier

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license can be acquired electronically from <http://www.fsf.org/licenses/fdl.html> or by writing to 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Table of Contents

About this book	i
1. Disk Organisation	1
1.1. superblock	2
1.1.1. s_inodes_count	3
1.1.2. s_blocks_count	3
1.1.3. s_r_blocks_count	4
1.1.4. s_free_blocks_count	4
1.1.5. s_free_inodes_count	4
1.1.6. s_first_data_block	4
1.1.7. s_log_block_size	4
1.1.8. s_log_frag_size	4
1.1.9. s_blocks_per_group	5
1.1.10. s_frags_per_group	5
1.1.11. s_inodes_per_group	5
1.1.12. s_mtime	5
1.1.13. s_wtime	5
1.1.14. s_mnt_count	5
1.1.15. s_max_mnt_count	6
1.1.16. s_magic	6
1.1.17. s_state	6
1.1.18. s_errors	6
1.1.19. s_minor_rev_level	6
1.1.20. s_lastcheck	6
1.1.21. s_checkinterval	7
1.1.22. s_creator_os	7
1.1.23. s_rev_level	7
1.1.24. s_def_resuid	7
1.1.25. s_def_resgid	7
1.1.26. s_first_ino	7
1.1.27. s_inode_size	8
1.1.28. s_block_group_nr	8
1.1.29. s_feature_compat	8
1.1.30. s_feature_incompat	8
1.1.31. s_feature_ro_compat	8
1.1.32. s_uuid	8
1.1.33. s_volume_name	8
1.1.34. s_last_mounted	9
1.1.35. s_algo_bitmap	9
1.2. Group Descriptor	9
1.2.1. bg_block_bitmap	10
1.2.2. bg_inode_bitmap	10
1.2.3. bg_inode_table	10
1.2.4. bg_free_blocks_count	10
1.2.5. bg_free_inodes_count	10
1.2.6. bg_used_dirs_count	10
1.2.7. bg_pad	10

1.2.8. bg_reserved.....	10
1.3. Block Bitmap	10
1.4. Inode Bitmap.....	11
1.5. Inode Table.....	11
1.5.1. i_mode	12
1.5.2. i_uid.....	13
1.5.3. i_size.....	13
1.5.4. i_atime	13
1.5.5. i_ctime	13
1.5.6. i_mtime.....	13
1.5.7. i_dtime	14
1.5.8. i_gid.....	14
1.5.9. i_links_count	14
1.5.10. i_blocks.....	14
1.5.11. i_flags	14
1.5.12. i_osd1	14
1.5.13. i_block	15
1.5.14. i_generation	15
1.5.15. i_file_acl	15
1.5.16. i_dir_acl.....	16
1.5.17. i_faddr.....	16
1.5.18. i_osd2	16
1.6. Data Blocks	18
2. Directory Structure.....	19
2.1. Directory File Format.....	19
2.1.1. inode	19
2.1.2. rec_len	19
2.1.3. name_len.....	19
2.1.4. file_type	19
2.1.5. name	20
2.2. Sample Directory	20
2.3. Indexed Directory Format	21
2.3.1. Index Structure	22
2.3.2. Lookup Algorithm	23
2.3.3. Insert Algorithm	23
2.3.4. Splitting	24
2.3.5. Key Collisions	24
2.3.6. Hash Function.....	24
2.3.7. Performance.....	25
3. Inodes, file identifiers.....	27
3.1. Inode Number	27
3.2. Locating the Inode structure	27
3.3. Locating the Inode Table.....	28
4. File Attributes	29
4.1. Standard Attributes.....	29
4.1.1. SUID, SGID and -rwxrwxrwx.....	29
4.1.2. File Size	29

4.1.3. Owner and Group	29
4.2. Extended Attributes.....	29
4.2.1. Attribute Block Header.....	30
4.2.2. Attribute Entry Header	31
4.3. Behaviour Control Flags	32
4.3.1. EXT2_SECRM_FL - Secure Deletion	33
4.3.2. EXT2_UNRM_FL - Record for Undelete.....	33
4.3.3. EXT2_COMPR_FL - Compressed File	33
4.3.4. EXT2_SYNC_FL - Synchronous Updates.....	33
4.3.5. EXT2_IMMUTABLE_FL - Immutable File	33
4.3.6. EXT2_APPEND_FL - Append Only	33
4.3.7. EXT2_NODUMP_FL - Do No Dump/Delete.....	33
4.3.8. EXT2_NOATIME_FL - Do Not Update .i_atime	34
4.3.9. EXT2_DIRTY_FL - Dirty	34
4.3.10. EXT2_COMPRBLK_FL - Compressed Blocks	34
4.3.11. EXT2_NOCOMPR_FL - Access Raw Compressed Data.....	34
4.3.12. EXT2_ECOMPR_FL - Compression Error	34
4.3.13. EXT2_BTREE_FL - B-Tree Format Directory.....	34
4.3.14. EXT2_INDEX_FL - Hash Indexed Directory.....	34
4.3.15. EXT2_IMAGIC_FL -	35
4.3.16. EXT2_JOURNAL_DATA_FL - Journal File Data.....	35
4.3.17. EXT2_RESERVED_FL - Reserved	35
A. Credits.....	36

List of Tables

1-1. EXT2_ERRORS values	6
1-2. EXT2_OS values	7
1-3. EXT2 revisions	7
1-4. EXT2_*_INO values	12
1-5. EXT2_S_I values	12
2-1. EXT2_FT values	20
4-1. Behaviour Control Flags	32

List of Figures

1-1. floppy disk meta-data layout	1
1-2. 20mb partition meta-data layout.....	1
1-3. superblock structure	2
1-4. group_desc structure.....	9
1-5. inode structure	11
1-6. inode osd2 structure: Hurd	16
1-7. inode osd2 structure: Linux	17
1-8. inode osd2 structure: Masix	18
2-1. directory entry	19
2-2. Sample Directory Data Layout.....	20
2-3. Performance of Indexed Directories.....	25
3-1. Sample inode computations.....	27
4-1. ext2_xattr_header structure	30
4-2. ext2_xattr_header structure	31

About this book

The latest version of this document may be downloaded from <http://www.freesoftware.fsf.org/ext2-doc/>

This book is intended as an introduction and guide to the Second Extended File System, also known as Ext2. The reader should have a good understanding of the purpose of a file system as well as the associated vocabulary (file, directory, partition, etc).

Trying to implement ext2 drivers isn't always an easy task, the most difficult issue is unfortunately the documentation available. It seems like most of the documentation on the net about the internal layout of Ext2 was written to complement the Linux sources rather than be a complete document by themselves.

Hopefully this document will fix this problem, may it be of help to as many of you as possible.

Unless otherwise stated, all values are stored in little endian byte order.

Chapter 1. Disk Organisation

The first aspect of using the Second Extended File System one has to grasp is that all the meta-data structures size are based on a “block” size rather than a “sector” size. This block size is variable depending on the size of the file system. On a floppy disk for example, it is 1KB (2 sectors), while on a 10GB partition, the block size is normally 4KB or 8KB (8 and 16 sectors respectively).

Each block is further sub-divided into “fragments”, but I have yet to see a file system which fragment size doesn’t match block size. Although my guts tells me that there must be some folks out there using different sizes for fragments and blocks.

Except for the superblock, all meta-data structures are resized to fit into blocks. This is something to remember when trying to mount any other file system than one on a floppy. The “Inode Table Block” for example will contain more entries in a 4KB block than in a 1KB block, so one will have to take that into account when accessing this particular structure.

The next major aspect is that the file system is split into “block groups”. While a floppy would contain only one block group holding all the blocks of the file system, a hard disk of 10GB could easily be split into 30 of such block groups; each holding a certain quantity of blocks.

At the start of each block group are various meta-data structures detailing the location of the other, more informative, meta-data structures defining the current file system state. Here’s the organisation of an ext2 file system on a floppy:

Figure 1-1. floppy disk meta-data layout

offset	# of blocks	description
-----	-----	-----
	0	1 boot record
		-- block group 0 --
(1024 bytes)		1 superblock
2		1 group descriptors
3		1 block bitmap
4		1 inode bitmap
5		23 inode table
28		1412 data blocks

And here’s the organisation of a 20MB ext2 file system:

Figure 1-2. 20mb partition meta-data layout

offset	# of blocks	description
-----	-----	-----
	0	1 boot record
		-- block group 0 --


```

(1024 bytes)      1 superblock
                  2      1 group descriptors
                  3      1 block bitmap
                  4      1 inode bitmap
                  5      214 inode table
                219      7974 data blocks
                -- block group 1 --
            8193      1 superblock backup
            8194      1 group descriptors backup
            8195      1 block bitmap
            8196      1 inode bitmap
            8197      214 inode table
            8408      7974 data blocks
                -- block group 2 --
        16385      1 block bitmap
        16386      1 inode bitmap
        16387      214 inode table
        16601      3879 data blocks

```

The layout on disk is very predictable as long as you know a few basic information; block size, blocks per group, inodes per group. This information is all located in, or can be computed from, the superblock structure.

Without the superblock information, the disk is useless; therefore as soon as enough space is available, one or more superblock backups will be created on the disk.

The block bitmap and inode bitmap are used to identify which blocks and which inode entries are free to use. The data blocks is where the various files will be stored. Note that a directory is also seen as a file under Ext2, we will go in more detail about that later on.

While all ext2 implementations try to be compatible, some fields in the various structures have been customized to fit the requirements of a specific operating system. Where such differences are known, they will be indicated in proper time.

1.1. superblock

The superblock is the structure on an ext2 disk containing the very basic information about the file system properties. It is layed out in the following form:

Figure 1-3. superblock structure

offset	size	description
0	4	s_inodes_count
4	4	s_blocks_count

```

8      4 s_r_blocks_count
12     4 s_free_blocks_count
16     4 s_free_inodes_count
20     4 s_first_data_block
24     4 s_log_block_size
28     4 s_log_frag_size
32     4 s_blocks_per_group
36     4 s_frags_per_group
40     4 s_inodes_per_group
44     4 s_mtime
48     4 s_wtime
52     2 s_mnt_count
54     2 s_max_mnt_count
56     2 s_magic
58     2 s_state
60     2 s_errors
62     2 s_minor_rev_level
64     4 s_lastcheck
68     4 s_checkinterval
72     4 s_creator_os
76     4 s_rev_level
80     2 s_def_resuid
82     2 s_def_resgid
-- EXT2_DYNAMIC_REV Specific --
84     4 s_first_ino
88     2 s_inode_size
90     2 s_block_group_nr
92     4 s_feature_compat
96     4 s_feature_incompat
100    4 s_feature_ro_compat
104    16 s_uuid
120    16 s_volume_name
136    64 s_last_mounted
200    4 s_algo_bitmap
-- Performance Hints --
204    1 s_prealloc_blocks
205    1 s_prealloc_dir_blocks
206    2 - (alignment)
-- Journaling Support --
208    16 s_journal_uuid
224    4 s_journal_inum
228    4 s_journal_dev
232    4 s_last_orphan
-- Unused --
236    788 - (padding)

```

1.1.1. s_inodes_count

32bit value indicating the total number of inodes, both used and free, in the file system.

1.1.2. **s_blocks_count**

32bit value indicating the total number of blocks, both used and free, in the file system.

1.1.3. **s_r_blocks_count**

32bit value indicating the total number of blocks reserved for the usage of the super user. This is most useful if for some reason a user, maliciously or not, fill the file system to capacity; the super user will have this specified amount of free blocks at his disposal so he can edit and save configuration files.

1.1.4. **s_free_blocks_count**

32bit value indicating the total number of free blocks, including the number of reserved blocks (see `s_r_blocks_count`). This is a sum of all free blocks of all the block groups.

1.1.5. **s_free_inodes_count**

32bit value indicating the total number of free inodes. This is a sum of all free inodes of all the block groups.

1.1.6. **s_first_data_block**

32bit value identifying the first data block, in other word the id of the block containing the superblock structure.

Note that this value is always 0 for file systems with a block size larger than 1KB, and always 1 for file systems with a block size of 1KB. The superblock is *always* starting at the 1024th byte of the disk, which normally happens to be the first byte of the 3rd sector.

1.1.7. **s_log_block_size**

The block size is computed using this 32bit value as the number of bits to shift left the value 1024. This value may only be positive.

```
block size = 1024 << s_log_block_size;
```

1.1.8. s_log_frag_size

The fragment size is computed using this 32bit value as the number of bits to shift left the value 1024. Note that a negative value would shift the bit right rather than left.

```
if( positive )
    fragmnet size = 1024 << s_log_frag_size;
else
    framgnet size = 1024 >> -s_log_frag_size;
```

1.1.9. s_blocks_per_group

32bit value indicating the total number of blocks per group. This value in combination with s_first_data_block can be used to determine the block groups boundaries.

1.1.10. s_frags_per_group

32bit value indicating the total number of fragments per group. It is also used to determine the size of the block bitmap of each block group.

1.1.11. s_inodes_per_group

32bit value indicating the total number of inodes per group. This is also used to determine the size of the inode bitmap of each block group.

1.1.12. s_mtime

Unix time, as defined by POSIX, of the last time the file system was mounted.

1.1.13. s_wtime

Unix time, as defined by POSIX, of the last write access to the file system.

1.1.14. s_mnt_count

32bit value indicating how many time the file system was mounted since the last time it was fully verified.

1.1.15. s_max_mnt_count

32bit value indicating the maximum number of times that the file system may be mounted before a full check is performed.

1.1.16. s_magic

16bit value identifying the file system as Ext2. The value is currently fixed to 0xEF53.

1.1.17. s_state

16bit value indicating the file system state. When the file system is mounted, this state is set to EXT2_ERROR_FS. When the file system is not yet mounted, this value may be either EXT2_VALID_FS or EXT2_ERROR_FS in the event the file system was not cleanly unmounted.

1.1.18. s_errors

16bit value indicating what the file system driver should do when an error is detected. The following values have been defined:

Table 1-1. EXT2_ERRORS values

EXT2_ERRORS_CONTINUE	1	continue as if nothing happened
EXT2_ERRORS_RO	2	remount read-only
EXT2_ERRORS_PANIC	3	cause a kernel panic
EXT2_ERRORS_DEFAULT	varies	as of revision 0.5, this is the same as EXT2_ERRORS_CONTINUE

1.1.19. s_minor_rev_level

16bit value identifying the minor revision level within its revision level.

1.1.20. s_lastcheck

Unix time, as defined by POSIX, of the last file system check.

1.1.21. s_checkinterval

Maximum Unix time interval, as defined by POSIX, allowed between file system checks.

1.1.22. s_creator_os

32bit identifier of the os that created the file system. Defined values are:

Table 1-2. EXT2_OS values

EXT2_OS_LINUX	0	Linux
EXT2_OS_HURD	1	Hurd
EXT2_OS_MASIX	2	MASIX
EXT2_OS_FREEBSD	3	FreeBSD
EXT2_OS_LITES4	4	Lites

1.1.23. s_rev_level

32bit revision level value. There are currently only 2 values defined:

Table 1-3. EXT2 revisions

EXT2_GOOD_OLD_REV	0	original format
EXT2_DYNAMIC_REV	1	V2 format with dynamic inode sizes

1.1.24. s_def_resuid

16bit value used as the default user id for reserved blocks.

1.1.25. s_def_resgid

16bit value used as the default group id for reserved blocks.

1.1.26. **s_first_ino**

32bit value used as index to the first inode useable for standard files. In the non-dynamic file system revisions, the first non-reserved inode was fixed to 11. With the introduction the dynamic revision of the file system it is now possible to modify this value.

1.1.27. **s_inode_size**

16bit value indicating the size of the inode structure. In non-dynamic file system revisions this value is assumed to be 128.

1.1.28. **s_block_group_nr**

16bit value used to indicate the block group number hosting this superblock structure. This can be used to rebuild the file system from any superblock backup.

1.1.29. **s_feature_compat**

32bit bitmask of compatible features. The file system implementation is free to support them or not without risk of damaging the meta-data. (more information will be added soon)

1.1.30. **s_feature_incompat**

32bit bitmask of incompatible features. The file system implementation should refuse to mount the file system if any of the indicated feature is unsupported. (more information will be added soon)

1.1.31. **s_feature_ro_compat**

32bit bitmask of “read-only” features. The file system implementation should mount as read-only if any of the indicated feature is unsupported. (more information will be added soon)

1.1.32. **s_uuid**

128bit value used as the volume id. This should, as much as possible, be unique for each file system formatted.

1.1.33. s_volume_name

16 bytes volume name, mostly unused. A valid volume name would consist of only ISO-Latin-1 characters and be 0 terminated.

1.1.34. s_last_mounted

64 bytes directory path where the file system was last mounted. While not normally used, it could serve for auto-finding the mountpoint when not indicated on the command line. Again the path should be zero terminated for compatibility reasons. Valid path is constructed from ISO-Latin-1 characters.

1.1.35. s_algo_bitmap

32bit value used by compression algorithms to determine the methods used. (I do not have any more detail about this field, if you do please do send me all the information you have, thanks).

1.2. Group Descriptor

The group descriptors is an array of the group_desc structure, each describing a “block group”, giving the location of its inode table, blocks and inodes bitmaps, and some other useful informations.

The group descriptors are located on the first block following the block containing the superblock structure. Here’s what one of the group descriptor looks like:

Figure 1-4. group_desc structure

offset	size	description
0	4	bg_block_bitmap
4	4	bg_inode_bitmap
8	4	bg_inode_table
12	2	bg_free_blocks_count
14	2	bg_free_inodes_count
16	2	bg_used_dirs_count
18	2	bg_pad
20	12	bg_reserved

For each group in the file system, such a group_desc is created. Each represent a single “block group” within the file system and the information within any one of them is pertinent only to the group it is describing. Every “Group Descriptor Table” contains all the information about all the groups.

All indicated “block id” are absolute.

1.2.1. bg_block_bitmap

32bit block id of the first block of the “block bitmap” for the group represented.

1.2.2. bg_inode_bitmap

32bit block id of the first block of the “inode bitmap” for the group represented.

1.2.3. bg_inode_table

32bit block id of the first block of the “inode table” for the group represented.

1.2.4. bg_free_blocks_count

16bit value indicating the total number of free blocks for the represented group.

1.2.5. bg_free_inodes_count

16bit value indicating the total number of free inodes for the represented group.

1.2.6. bg_used_dirs_count

16bit value indicating the number of inodes allocated to directories for the represented group.

1.2.7. bg_pad

16bit value used for padding the structure on a 32bit boundary.

1.2.8. bg_reserved

3 successive 32bit values reserved for future implementations.

1.3. Block Bitmap

The “Block Bitmap” is normally located at the first block, or second block if a superblock backup is present, of the block group. Its official location can be determined by reading the “bg_block_bitmap” in its associated group descriptor.

Each bit represent the current state of a block within that group, where 1 means “used” and 0 “free/available”. The first block of this block group is represented by bit 0 of byte 0, the second by bit 1 of byte 0. The 8th block is represented by bit 7 (most significant bit) of byte 0 while the 9th block is represented by bit 0 (least significant bit) of byte 1.

1.4. Inode Bitmap

The “Inode Bitmap” works in a similar way as the “Block Bitmap”, difference being in each bit representing an inode in the “Inode Table” rather than a block.

There is one inode bitmap per group and its location may be determined by reading the “bg_inode_bitmap” in its associated group descriptor.

When the inode table is created, all the reserved inodes are marked as used. For the “Good Old Revision” this means the first 11 bits of the inode bitmap.

1.5. Inode Table

The “Inode Table” is used to keep track of every file; their location, size, type and access rights are all stored in inodes. The filename is not stored in there though, within the inode tables all files are referenced by their inode number.

There is one inode table per group and it can be located by reading the “bg_inode_table” in its associated group descriptor. There are `s_inodes_per_group` inodes per table.

Each inode contain the information about a single physical file on the system. A file can be a directory, a socket, a buffer, character or block device, symbolic link or a regular file. So an inode can be seen as a block of information related to an entity, describing its location on disk, its size and its owner. An inode looks like this:

Figure 1-5. inode structure

offset	size	description
-----	-----	-----
0	2	i_mode

2	2	i_uid
4	4	i_size
8	4	i_atime
12	4	i_ctime
16	4	i_mtime
20	4	i_dtime
24	2	i_gid
26	2	i_links_count
28	4	i_blocks
32	4	i_flags
36	4	i_osd1
40	15 x 4	i_block
100	4	i_generation
104	4	i_file_acl
108	4	i_dir_acl
112	4	i_faddr
116	12	i_osd2

The first few entries of the inode tables are reserved. In the EXT2_GOOD_OLD_REV there are 11 entries reserved while in the newer EXT2_DYNAMIC_REV the number of reserved inodes entries is specified in the `s_first_ino` of the superblock structure. Here's a listing of the known reserved inode entries:

Table 1-4. EXT2_*_INO values

EXT2_BAD_INO	0x01	bad blocks inode
EXT2_ROOT_INO	0x02	root directory inode
EXT2_ACL_IDX_INO	0x03	ACL index inode (deprecated?)
EXT2_ACL_DATA_INO	0x04	ACL data inode (deprecated?)
EXT2_BOOT_LOADER_INO	0x05	boot loader inode
EXT2_UNDEL_DIR_INO	0x06	undelete directory inode

1.5.1. i_mode

16bit value used to indicate the format of the described file and the access rights. Here are the possible values, which can be combined in various ways:

Table 1-5. EXT2_S_I values

		-- file format --
EXT2_S_IFMT	0xF000	format mask
EXT2_S_IFSOCK	0xC000	socket
EXT2_S_IFLNK	0xA000	symbolic link
EXT2_S_IFREG	0x8000	regular file
EXT2_S_IFBLK	0x6000	block device

EXT2_S_IFDIR	0x4000	directory
EXT2_S_IFCHR	0x2000	character device
EXT2_S_IFIFO	0x1000	fifo
-- access rights --		
EXT2_S_ISUID	0x0800	SUID
EXT2_S_ISGID	0x0400	SGID
EXT2_S_ISVTX	0x0200	sticky bit
EXT2_S_IRWXU	0x01C0	user access rights mask
EXT2_S_IRUSR	0x0100	read
EXT2_S_IWUSR	0x0080	write
EXT2_S_IXUSR	0x0040	execute
EXT2_S_IRWXG	0x0038	group access rights mask
EXT2_S_IRGRP	0x0020	read
EXT2_S_IWGRP	0x0010	write
EXT2_S_IXGRP	0x0008	execute
EXT2_S_IRWXO	0x0007	others access rights mask
EXT2_S_IROTH	0x0004	read
EXT2_S_IWOTH	0x0002	write
EXT2_S_IXOTH	0x0001	execute

1.5.2. i_uid

16bit user id associated with the file.

1.5.3. i_size

32bit value indicating the size of the file in bytes.

1.5.4. i_atime

32bit value representing the number of seconds since january 1st 1970 of the last time this file was accessed.

1.5.5. i_ctime

32bit value representing the number of seconds since january 1st 1970 when the file was created.

1.5.6. i_mtime

32bit value representing the number of seconds since january 1st 1970 of the last time this file was modified.

1.5.7. i_dtime

32bit value representing the number of seconds since january 1st 1970 when the file was deleted. It is important that unless the file is deleted that this value is always 0.

1.5.8. i_gid

16bit value of the group having access to this file.

1.5.9. i_links_count

16bit value indicating how many times this particular inode is linked (referred to).

1.5.10. i_blocks

32bit value indicating the amount of blocks reserved for the associated file data. This includes both currently in used and currently reserved blocks in case the file grows in size.

A point worth of note is that this value indicate the number of 512 bytes block and not the number of blocks of the size indicated in the superblock. So if a file uses only 1 file system block and is 1024 bytes big, its .i_blocks value will be 2.

1.5.11. i_flags

32bit value indicating how the ext2 implementation should behave when accessing the data for this inode. (See the Behaviour flags section.)

1.5.12. i_osd1

32bit OS dependant value.

1.5.12.1. Hurd

32bit value labeled as “translator”.

1.5.12.2. Linux

32bit value currently reserved.

1.5.12.3. Masix

32bit value currently reserved.

1.5.13. i_block

Array used to locate the blocks the particular file is stored on. Each entry is a 32bit block number. The first 12 entries in this array are block numbers, which can be used to fetch the first 12 blocks associated with the file.

The 13th entry is an indirect block number. Which means that at the specified data block, you will find an array of direct block numbers.

The 14th entry is an bi-indirect block number. Which means that at the specified data block, you will find an array of indirect block number, which in turn contains an array of block numbers that can be accessed directly.

The 15th entry is an tri-indirect block number. It is a block number which contains an array of bi-indirect block number, etc.

Each indirect/bi-indirect/tri-indirect block array contains as many entries of 32bit block numbers as possible (to fill one entire block).

1.5.14. i_generation

32bit value used to indicate the file version (used by NFS).

1.5.15. i_file_acl

32bit value indicating the block number containing the extended attributes. In previous revisions this value was always 0.

A general description of ACL for Digital UNIX can be found at this url for the moment:
http://www.tru64unix.compaq.com/docs/base_doc/DOCUMENTATION/HTML/AA-Q0R2D-TET1_html/sec.c27.html

1.5.16. i_dir_acl

32bit value used to indicate the “high size” of the file. In previous revisions this value was always 0.

1.5.17. i_faddr

32bit value indicating the location of the last file fragment.

1.5.18. i_osd2

96bit OS dependant structure.

1.5.18.1. Hurd

Figure 1-6. inode osd2 structure: Hurd

offset	size	description
0	1	h_i_frag
1	1	h_i_fsize
2	2	h_i_mode_high
4	2	h_i_uid_high
6	2	h_i_gid_high
8	4	h_i_author

1.5.18.1.1. h_i_frag

8bit fragment number.

1.5.18.1.2. h_i_fsize

8bit fragment size.

1.5.18.1.3. h_i_mode_high**1.5.18.1.4. h_i_uid_high**

High 16bit of user id.

1.5.18.1.5. h_i_gid_high

High 16bit of group id.

1.5.18.1.6. h_i_author**1.5.18.2. Linux****Figure 1-7. inode osd2 structure: Linux**

offset	size	description
0	1	l_i_frag
1	1	l_i_fsize
2	2	reserved
4	2	l_i_uid_high
6	2	l_i_gid_high
8	4	reserved

1.5.18.2.1. l_i_frag

8bit fragment number.

1.5.18.2.2. l_i_fsize

8bit fragment size.

1.5.18.2.3. l_i_uid_high

High 16bit of user id.

1.5.18.2.4. l_i_gid_high

High 16bit of group id.

1.5.18.3. Masix

Figure 1-8. inode osd2 structure: Masix

offset	size	description
-----	-----	-----
0	1	m_i_frag
1	1	m_i_fsize
2	10	reserved

1.5.18.3.1. m_i_frag

8bit fragment number.

1.5.18.3.2. m_i_fsize

8bit fragment size.

1.6. Data Blocks

Data blocks are used to store the various files' content, including directory listing, extended attributes, symbolic links, etc.

Chapter 2. Directory Structure

Directories are stored as files and can be identified as such by looking up the *ext2_inode.i_mode* file format bits for the EXT2_S_IFDIR value.

The root directory is always the second entry of the inode table (EXT2_ROOT_INO is of value 2). Any subdirectory from there can be located by looking at the content of the root directory file.

2.1. Directory File Format

Figure 2-1. directory entry

offset	size	description
-----	-----	-----
0	4	inode
4	2	rec_len
6	1	name_len
7	1	file_type
8	...	name

Earlier implementations of Ext2 used a 16bit *name_len*, but since this value is stored in Intel (little-endian) byte order and most implementation restricted filenames to maximum 255 characters, allowing a byte to be recycled.

2.1.1. inode

32bit inode number of the file entry. A value of 0 indicate that the entry is not used.

2.1.2. rec_len

16bit unsigned displacement to the next directory entry from the start of the current directory entry.

2.1.3. name_len

8bit unsigned value indicating how many characters are contained in the name.

2.1.4. file_type

8bit unsigned value used to indicate file type. As noted, this value may be 0 in earlier implementations. Currently defined values are:

Table 2-1. EXT2_FT values

EXT2_FT_UNKNOWN	0
EXT2_FT_REG_FILE	1
EXT2_FT_DIR	2
EXT2_FT_CHRDEV	3
EXT2_FT_BLKDEV	4
EXT2_FT_FIFO	5
EXT2_FT_SOCKET	6
EXT2_FT_SYMLINK	7
EXT2_FT_MAX	8

2.1.5. name

Name of the entry. The allowed character set is the ISO-Latin-1.

2.2. Sample Directory

Here's a sample of the home directory of one user on my system:

```
$ ls -la /home/eks
.
..
.bash_profile
.bashrc
mbox
public_html
tmp
```

For which the following data representation can be found on the storage device:

Figure 2-2. Sample Directory Data Layout

offset	size	description
0	4	inode number (783362)

```

4      2 record length (9)
6      1 name length (1)
7      1 file type (EXT2_FT_DIR)
8      1 name (.)

9      4 inode number (1109761)
13     2 record length (10)
15     1 name length (2)
16     1 file type (EXT2_FT_DIR)
17     2 name (..)

19     4 inode number (783364)
23     2 record length (21)
25     1 name length (13)
26     1 file type (EXT2_FT_REG_FILE)
27    13 name (.bash_profile)

40     4 inode number (783363)
44     2 record length (15)
46     1 name length (7)
47     1 file type (EXT2_FT_REG_FILE)
48     7 name (.bashrc)

55     4 inode number (783377)
59     2 record length (12)
61     1 name length (4)
62     1 file type (EXT2_FT_REG_FILE)
63     4 name (mbox)

67     4 inode number (783545)
71     2 record length (19)
73     1 name length (11)
74     1 file type (EXT2_FT_DIR)
75    11 name (public_html)

86     4 inode number (669354)
90     2 record length (11)
92     1 name length (3)
93     1 file type (EXT2_FT_DIR)
94     3 name (tmp)

97     4 inode number (0)
101    2 record length (3999)
103    1 name length (0)
104    1 file type (EXT2_FT_UNKNOWN)
105    0 name ( )

```

It should be noted that some implementation will pad directory entries to have better performances on the host processor, it is thus important to use the *record length* and not the *name length* to find the next record.

2.3. Indexed Directory Format

Using the standard linked list directory format can become very slow once the number of files starts growing. To improve performances in such a system, a hashed index was created, which allow to quickly locate the particular file searched.

Bit EXT2_INDEX_FL in the behaviour control flags is set if the indexed directory format is used.

2.3.1. Index Structure

The root of the index tree is in the 0th block of the file. Space is reserved for a second level of the index tree in blocks 1 through 511 (for 4K filesystem blocks). Directory leaf blocks are appended starting at block 512, thus the tail of the directory file looks like a normal Ext2 directory and can be processed directly by `ext2_readdir`. For directories with less than about 90K files there is a hole running from block 1 to block 511, so an empty directory has just two blocks in it, though its size appears to be about 2 Meg in a directory listing.

So a directory file looks like:

```
0: Root index block
1: Index block/0
2: Index block/0
...
511: Index block/0
512: Dirent block
513: Dirent block
...
```

Each index block consists of 512 index entries of the form:

```
hash, block
```

where hash is a 32 bit hash with a collision flag in its least significant bit, and block is the logical block number of an index of leaf block, depending on the tree level.

The hash value of the 0th index entry isn't needed because it can always be obtained from the level about, so it is used to record the count of index entries in an index block. This gives a nice round branching factor of 512, the evenness being a nicety that mainly satisfies my need to seek regularity, rather than winning any real performance. (On the other hand, the largeness of the branching factor matters a great deal.)

The root index block has the same format as the other index blocks, with its first 8 bytes reserved for a small header:

```
1 byte header length (default: 8)
1 byte index type (default: 0)
1 byte hash version (default:0)
1 byte tree depth (default: 1)
```

The treatment of the header differs slightly in the attached patch. In particular, only a single level of the index tree (the root) is implemented here. This turns out to be sufficient to handle more than 90,000 entries, so it is enough for today. When a second level is added to the tree, capacity will increase to somewhere around 50 million entries, and there is nothing preventing the use of n levels, should there ever be a reason. It's doubtful that a third level will ever be required, but if it is, the design provides for it.

2.3.2. Lookup Algorithm

Lookup is straightforward:

- Compute a hash of the name
- Read the index root
- Use binary search (linear in the current code) to find the first index or leaf block that could contain the target hash (in tree order)
- Repeat the above until the lowest tree level is reached
- Read the leaf directory entry block and do a normal Ext2 directory block search in it.
- If the name is found, return its directory entry and buffer
- Otherwise, if the collision bit of the next directory entry is set, continue searching in the successor block

Normally, two logical blocks of the file will need to be accessed, and one or two metadata index blocks. The effect of the metadata index blocks can largely be ignored in terms of disk access time since these blocks are unlikely to be evicted from cache. There is some small CPU cost that can be addressed by moving the whole directory into the page cache.

2.3.3. Insert Algorithm

Insertion of new entries into the directory is considerably more complex than lookup, due to the need to split leaf blocks when they become full, and to satisfy the conditions that allow hash key collisions to be handled reliably and efficiently. I'll just summarize here:

- Probe the index as for lookup
- If the target leaf block is full, split it and note the block that will receive the new entry

- Insert the new entry in the leaf block using the normal Ext2 directory entry insertion code.

The details of splitting and hash collision handling are somewhat messy, but I will be happy to dwell on them at length if anyone is interested.

2.3.4. Splitting

In brief, when a leaf node fills up and we want to put a new entry into it the leaf has to be split, and its share of the hash space has to be partitioned. The most straightforward way to do this is to sort the entries by hash value and split somewhere in the middle of the sorted list. This operation is $\log(\text{number_of_entries_in_leaf})$ and is not a great cost so long as an efficient sorter is used. I used Combsort for this, although Quicksort would have been just as good in this case since average case performance is more important than worst case.

An alternative approach would be just to guess a median value for the hash key, and the partition could be done in linear time, but the resulting poorer partitioning of hash key space outweighs the small advantage of the linear partition algorithm. In any event, the number of entries needing sorting is bounded by the number that fit in a leaf.

2.3.5. Key Collisions

Some complexity is introduced by the need to handle sequences of hash key collisions. It is desirable to avoid splitting such sequences between blocks, so the split point of a block is adjusted with this in mind. But the possibility still remains that if the block fills up with identically-hashed entries, the sequence may still have to be split. This situation is flagged by placing a 1 in the low bit of the index entry that points at the successor block, which is naturally interpreted by the index probe as an intermediate value without any special coding. Thus, handling the collision problem imposes no real processing overhead, just some extra code and a slight reduction in the hash key space. The hash key space remains sufficient for any conceivable number of directory entries, up into the billions.

2.3.6. Hash Function

The exact properties of the hash function critically affect the performance of this indexing strategy, as I learned by trying a number of poor hash functions, at times intentionally. A poor hash function will result in many collisions or poor partitioning of the hash space. To illustrate why the latter is a problem, consider what happens when a block is split such that it covers just a few distinct hash values. The probability of later index entries hashing into the same, small hash space is very small. In practice, once a block is split, if its hash space is too small it tends to stay half full forever, an effect I observed in practice.

After some experimentation I came up with a hash function that gives reasonably good dispersal of hash keys across the entire 31 bit key space. This improved the average fullness of leaf blocks considerably, getting much closer to the theoretical average of 3/4 full.

But the current hash function is just a place holder, waiting for an better version based on some solid theory. I currently favor the idea of using crc32 as the default hash function, but I welcome suggestions.

Inevitably, no matter how good a hash function I come up with, somebody will come up with a better one later. For this reason the design allows for additional hash functiones to be added, with backward compatibility. This is accomplished simply, by including a hash function number in the index root. If a new, improved hash function is added, all the previous versions remain available, and previously created indexes remain readable.

Of course, the best strategy is to have a good hash function right from the beginning. The initial, quick hack has produced results that certainly have not been disappointing.

2.3.7. Performance

OK, if you have read this far then this is no doubt the part you've been waiting for. In short, the performance improvement over normal Ext2 has been stunning. With very small directories performance is similar to standard Ext2, but as directory size increases standard Ext2 quickly blows up quadratically, while htree-enhanced Ext2 continues to scale linearly.

Uli Luckas ran benchmarks for file creation in various sizes of directories ranging from 10,000 to 90,000 files. The results are pleasing: total file creation time stays very close to linear, versus quadratic increase with normal Ext2.

Time to create:

Figure 2-3. Performance of Indexed Directories

	Indexed	Normal
	=====	=====
10000 Files:	0m1.350s	0m23.670s
20000 Files:	0m2.720s	1m20.470s
30000 Files:	0m4.330s	3m9.320s
40000 Files:	0m5.890s	5m48.750s
50000 Files:	0m7.040s	9m31.270s
60000 Files:	0m8.610s	13m52.250s
70000 Files:	0m9.980s	19m24.070s
80000 Files:	0m12.060s	25m36.730s
90000 Files:	0m13.400s	33m18.550s

A graph is posted at: <http://www.innominate.org/~phillips/htree/performance.png>

All of these tests are CPU-bound, which may come as a surprise. The directories fit easily in cache, and the limiting factor in the case of standard Ext2 is the looking up of directory blocks in buffer cache, and the low level scan of directory entries. In the case of htree indexing there are a number of costs to be considered, all of them pretty well bounded. Notwithstanding, there are a few obvious optimizations to be done:

- Use binary search instead of linear search in the interior index nodes.
- If there is only one leaf block in a directory, bypass the index probe, go straight to the block.
- Map the directory into the page cache instead of the buffer cache.

Each of these optimizations will produce a noticeable improvement in performance, but naturally it will never be anything like the big jump going from N^2 to $\log_{512}(N)$, $\sim N$. In time the optimizations will be applied and we can expect to see another doubling or so in performance.

There will be a very slight performance hit when the directory gets big enough to need a second level. Because of caching this will be very small. Traversing the directories metadata index blocks will be a bigger cost, and once again, this cost can be reduced by moving the directory blocks into the page cache.

Typically, we will traverse 3 blocks to read or write a directory entry, and that number increases to 4-5 with really huge directories. But this is really nothing compared to normal Ext2, which traverses several hundred blocks in the same situation.

Chapter 3. Inodes, file identifiers

Every file, directory, symlink, special device, or anything else really stored in a ext2 file system, is identified by an inode. If you know the inode number of the file you want to read, even if you don't know the path to the file or even the file name, you can still locate the file on disk and read it.

3.1. Inode Number

The “inode number” is an index in the inode table to an inode structure. The size of the inode table is fixed at format time, it is built to hold a maximum number of entries. Due to the normally sufficiently large amount of entries reserved, the table is quite big and thus, it was split equally among all the “block groups” (see Chapter 1 for more information).

3.2. Locating the Inode structure

The `s_inodes_per_group` field in the superblock structure tells us how many inodes are defined per group. Knowing that inode 1 is the first inode defined in the inode table, one can use the following formulae:

```
group = (inode - 1) / s_inodes_per_group
```

to locate which blocks group holds the part of the inode table containing the searched inode entry, and:

```
index = (inode - 1) % s_inodes_per_group
```

to get the index within this partial inode table to the searched inode entry. Here are a couple of sample values that could be used to test your implementation:

Figure 3-1. Sample inode computations

```
s_inodes_per_group = 1712

inode number computation
-----
1      group = (1 - 1) / 1712 = 0
      index = (1 - 1) % 1712 = 0

2      group = (2 - 1) / 1712 = 0
      index = (2 - 1) % 1712 = 1

963    group = (963 - 1) / 1712 = 0
      index = (963 - 1) % 1712 = 962
```

```

1712    group = (1712 - 1) / 1712 = 0
        index = (1712 - 1) % 1712 = 1711

1713    group = (1713 - 1) / 1712 = 1
        index = (1713 - 1) % 1712 = 0

3424    group = (3424 - 1) / 1712 = 1
        index = (3424 - 1) % 1712 = 1711

3425    group = (3425 - 1) / 1712 = 2
        index = (3425 - 1) % 1712 = 0

```

As many of you are most likely already familiar with, an index of 0 means the first entry. The reason behind using 0 rather than 1 is that it can more easily be multiplied by the structure size to find the final offset of its location in memory or on disk.

3.3. Locating the Inode Table

As introduced in Section 3.1, the inode table is split equally among all group. If a file system was created to allow a thousand inodes, split between 5 groups, there would be 200 inodes per partial inode table. Figure 3-1 illustrates such similar distribution.

Each partial inode table can be located using the `bg_inode_table` field of the `group_descriptor` structure of its associated blocks group.

Chapter 4. File Attributes

Most of the file (also directory, symlink, device...) attributes are located in the inode associated with the file. Some other attributes are only available as extended attributes.

4.1. Standard Attributes

4.1.1. SUID, SGID and -rwxrwxrwx

There isn't much to say about those, they are located with the SGID and SUID bits in `ext2_inode.i_mode`.

4.1.2. File Size

The size of a file can be determined by looking at the `ext2_inode.i_size` field.

4.1.3. Owner and Group

Under most implementations, the owner and group are 16bit values, but on some recent Linux and Hurd implementations the owner and group id are 32bit. When 16bit values are used, only the "low" part should be used as valid, while when using 32bit value, both the "low" and "high" part should be used, the high part being shifted left 16 places then added to the low part.

The low part of owner and group are located in `ext2_inode.i_uid` and `ext2_inode.i_gid` respectively.

The high part of owner and group are located in `ext2_inode.osd2.hurd.h_i_uid_high` and `ext2_inode.osd2.hurd.h_i_gid_high`, respectively, for Hurd and located in `ext2_inode.osd2.linux.l_i_uid_high` and `ext2_inode.osd2.linux.l_i_gid_high`, respectively, for Linux.

4.2. Extended Attributes

Extended attributes are name:value pairs associated permanently with files and directories, similar to the environment strings associated with a process. An attribute may be defined or undefined. If it is defined, its value may be empty or non-empty.

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system. They are often used to provide additional functionality to a filesystem - for example, additional security features such as Access Control Lists (ACLs) may be implemented using extended attributes.

Extended attributes are accessed as atomic objects. Reading retrieves the whole value of an attribute and stores it in a buffer. Writing replaces any previous value with the new value.

In each ext2 inode, we have the `i_file_acl` field, reserved for Access Control Lists. This field is used for storing the block number on which the extended attributes of an inode are stored instead (ACLs are stored as extended attributes).

Extended attributes are stored on 'plain' disk blocks, which are not part of any files. The disk block layout is similar to the layout used for directories. After the attribute block header, entry headers follow. The size of entry headers varies with the length of the attribute name.

The attribute values are on the same block as their attribute entry descriptions, aligned to the end of the attribute block. This allows for additional attributes to be added more easily.

A list of attribute names associated with a file can be retrieved. The filesystem handler returns a string of names separated by null characters, terminated by two null characters at the end of the list.

4.2.1. Attribute Block Header

Figure 4-1. ext2_xattr_header structure

offset	size	description
-----	-----	-----
0	4	<code>h_magic</code>
4	4	<code>h_refcount</code>
8	4	<code>h_blocks</code>
12	4	<code>h_hash</code>
16	16	<code>reserved</code>

4.2.1.1. `h_magic`

32bit magic number of identification (`EXT2_XATTR_MAGIC = 0xEA020000`).

4.2.1.2. `h_refcount`

32bit value used as reference count. This value is incremented everytime a link is created to this attribute block and decremented when a link is destroyed. Whenever this value reaches 0 the attribute block can

be freed.

4.2.1.3. h_blocks

32bit value indicating how many blocks are currently used by the extended attributes.

4.2.1.4. h_hash

32bit hash value of all attributes.

4.2.2. Attribute Entry Header

Figure 4-2. ext2_xattr_header structure

offset	size	description
-----	-----	-----
0	1	e_name_len
1	1	e_name_index
2	2	e_value_offs
4	4	e_value_block
8	4	e_value_size
12	4	e_hash
16	...	e_name

The total size of an attribute entry is always rounded to the next 4-bytes boundary.

4.2.2.1. e_name_len

8bit unsigned value indicating the length of the name.

4.2.2.2. e_name_index

8bit unsigned value used as attribute name index.

4.2.2.3. e_value_offs

16bit unsigned offset to the value within the value block.

4.2.2.4. e_value_block

32bit id of the block holding the value.

4.2.2.5. e_value_size

32bit unsigned value indicating the size of the attribute value.

4.2.2.6. e_hash

32bit hash of attribute name and value.

4.2.2.7. e_name

Attribute name.

4.3. Behaviour Control Flags

The `i_flags` value in the inode structure allows to specify how the file system should behave in regard to the file. The following bits are currently defined:

Table 4-1. Behaviour Control Flags

EXT2_SECRM_FL	0x00000001	secure deletion
EXT2_UNRM_FL	0x00000002	record for undelete
EXT2_COMPR_FL	0x00000004	compressed file
EXT2_SYNC_FL	0x00000008	synchronous updates
EXT2_IMMUTABLE_FL	0x00000010	immutable file
EXT2_APPEND_FL	0x00000020	append only
EXT2_NODUMP_FL	0x00000040	do not dump/delete file
EXT2_NOATIME_FL	0x00000080	do not update <code>.i_atime</code>
EXT2_DIRTY_FL	0x00000100	dirty (file is in use?)
EXT2_COMPRBLK_FL	0x00000200	compressed blocks
EXT2_NOCOMPR_FL	0x00000400	access raw compressed data
EXT2_ECOMPR_FL	0x00000800	compression error
EXT2_BTREE_FL	0x00010000	b-tree format directory
EXT2_INDEX_FL	0x00010000	Hash indexed directory

EXT2_IMAGIC_FL	0x00020000	?
EXT3_JOURNAL_DATA_FL	0x00040000	journal file data
EXT2_RESERVED_FL	0x80000000	reserved for ext2 implementation

4.3.1. EXT2_SECRM_FL - Secure Deletion

Enabling this bit will cause random data to be written over the file's content several times before the blocks are unlinked. Note that this is highly implementation dependant and as such, it should not be assumed to be 100% secure. Make sure to study the implementation notes before relying on this option.

4.3.2. EXT2_UNRM_FL - Record for Undelete

When supported by the implementation, setting this bit will cause the deleted data to be moved to a temporary location, where the user can restore the original file without any risk of data loss. This is most useful when using ext2 on a desktop or workstation.

4.3.3. EXT2_COMPR_FL - Compressed File

The file's content is compressed. There is no note about the particular algorithm used other than maybe the `s_algo_bitmap` field of the superblock structure.

4.3.4. EXT2_SYNC_FL - Synchronous Updates

The file's content in memory will be constantly synchronized with the content on disk. This is mostly used for very sensitive boot files or encryption keys that you do not want to lose in case of a crash.

4.3.5. EXT2_IMMUTABLE_FL - Immutable File

The blocks associated with the file will not be exchanged. If for any reason a file system defragmentation is launched, such files will not be moved. Mostly used for stage2 and stage1.5 boot loaders.

4.3.6. EXT2_APPEND_FL - Append Only

Writing can only be used to append content at the end of the file and not modify the current content. Example of such use could be mailboxes, where anybody could send a message to a user but not modify any already present.

4.3.7. EXT2_NODUMP_FL - Do No Dump/Delete

Setting this bit will protect the file from deletion. As long as this bit is set, even if the `i_links_count` is 0, the file will not be removed.

4.3.8. EXT2_NOATIME_FL - Do Not Update `i_atime`

The `i_atime` field of the inode structure will not be modified when the file is accessed if this bit is set. The only good use I can think of that are related to security.

4.3.9. EXT2_DIRTY_FL - Dirty

I do not have information at this moment about the use of this bit.

4.3.10. EXT2_COMPRBLK_FL - Compressed Blocks

This flag is set if one or more blocks are compressed. You can have more information about compression on ext2 at <http://www.netspace.net.au/~reiter/e2compr/> Note that the project has not been updated since 1999.

4.3.11. EXT2_NOCOMPR_FL - Access Raw Compressed Data

When this flag is set, the file system implementation will not uncompress the data before forwarding it to the application but will rather give it as is.

4.3.12. EXT2_ECOMPR_FL - Compression Error

This flag is set if an error was detected when trying to uncompress the file.

4.3.13. EXT2_BTREE_FL - B-Tree Format Directory

4.3.14. EXT2_INDEX_FL - Hash Indexed Directory

When this bit is set, the format of the directory file is hash indexed. This is covered in details in Section 2.3.

4.3.15. EXT2_IMAGIC_FL -

4.3.16. EXT2_JOURNAL_DATA_FL - Journal File Data

4.3.17. EXT2_RESERVED_FL - Reserved

Appendix A. Credits

I would like to personally thank everybody who contributed to this document, you are numerous and in many cases I haven't kept track of all of you. Be sure that if you are not in this list, it's a mistake and do not hesitate to contact me, it will be a pleasure to add your name to the list.

Andreas Gruenbacher (a.gruenbacher@bestbits.at)
Section 4.2

Daniel Phillips (phillips@innominate.de)
Section 2.3.1
Section 2.3.2
Section 2.3.3
Section 2.3.4
Section 2.3.5
Section 2.3.6
Section 2.3.7

Jeremy Stanley of Access Data Inc.
Pointed out the inversed values for EXT2_S_IFSOCK and EXT2_S_IFLNK