

The EXT2FS Library

The EXT2FS Library
Version 1.38
June 2005

by Theodore Ts'o

Copyright © 1997, 1998, 1999, 2000, 2001, 2002, 2003, 2004, 2005 Theodore Ts'o

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

1 Introduction to the EXT2FS Library

The EXT2FS library is designed to allow user-level programs to manipulate an ext2 filesystem.

2 EXT2FS Library Functions

2.1 Filesystem-level functions

The following functions operate on a filesystem handle. Most EXT2FS Library functions require a filesystem handle as their first argument. There are two functions which create a filesystem handle, `ext2fs_open` and `ext2fs_initialize`.

The filesystem can also be closed using `ext2fs_close`, and any changes to the superblock and group descripts can be written out to disk using `ext2fs_flush`.

2.1.1 Opening an ext2 filesystem

Most `libext2fs` functions take a filesystem handle of type `ext2_filsys`. A filesystem handle is created either by opening an existing function using `ext2fs_open`, or by initializing a new filesystem using `ext2fs_initialize`.

`errcode_t ext2fs_open (const char *name, int flags, int [Function]
superblock, int block_size, io_manager manager, ext2_filsys *ret_fs)`

Opens a filesystem named *name*, using the *io_manager manager* to define the input/output routines needed to read and write the filesystem. In the case of the `unix_io` *io_manager*, *name* is interpreted as the Unix filename of the filesystem image. This is often a device file, such as `‘/dev/hda1’`.

The *superblock* parameter specifies the block number of the superblock which should be used when opening the filesystem. If *superblock* is zero, `ext2fs_open` will use the primary superblock located at offset 1024 bytes from the start of the filesystem image.

The *block_size* parameter specifies the block size used by the filesystem. Normally this is determined automatically from the filesystem superblock. If *block_size* is non-zero, it must match the block size found in the superblock, or the error `EXT2_ET_UNEXPECTED_BLOCK_SIZE` will be returned. The *block_size* parameter is also used to help fund the superblock when *superblock* is non-zero.

The *flags* argument contains a bitmask of flags which control how the filesystem open should be handled.

`EXT2_FLAG_RW`

Open the filesystem for reading and writing. Without this flag, the filesystem is opened for reading only.

`EXT2_FLAG_FORCE`

Open the filesystem regardless of the feature sets listed in the superblock.

2.1.2 Closing and flushing out changes

`errcode_t ext2fs_flush (ext2_filsys fs) [Function]`

Write any changes to the high-level filesystem data structures in the *fs* filesystem. The following data structures will be written out:

- The filesystem superblock
- The filesystem group descriptors
- The filesystem bitmaps, if read in via `ext2fs_read_bitmaps`.

`void ext2fs_free (ext2_filsys fs)` [Function]
 Close the `io_manager` abstraction for `fs` and release all memory associated with the filesystem handle.

`errcode_t ext2fs_close (ext2_filsys fs)` [Function]
 Flush out any changes to the high-level filesystem data structures using `ext2fs_flush` if the filesystem is marked dirty; then close and free the filesystem using `ext2fs_free`.

2.1.3 Initializing a filesystem

An ext2 filesystem is initializing by the `mke2fs` program. The two functions described here, `ext2fs_initialize` and `ext2fs_allocate_tables` do much of the initial work for setting up a filesystem. However, they don't do the whole job. `mke2fs` calls `ext2fs_initialize` to set up the filesystem superblock, and calls `ext2fs_allocate_tables` to allocate space for the inode table, and the inode and block bitmaps. In addition, `mke2fs` must also initialize the inode tables by clearing them with zeros, create the root and lost+found directories, and reserve the reserved inodes.

`errcode_t ext2fs_initialize (const char *name, int flags, struct ext2_super_block *param, io_manager manager, ext2_filsys *ret_fs)` [Function]

This function is used by the `mke2fs` program to initialize a filesystem. The `ext2fs_initialize` function creates a filesystem handle which is returned in `ret_fs` that has been properly setup for a filesystem to be located in `name`, using the `io_manager manager`. The prototype superblock in `param` is used to supply parameters such as the number of blocks in the filesystem, the block size, etc.

The `ext2fs_initialize` function does not actually do any I/O; that will be done when the application program calls `ext2fs_close` or `ext2fs_flush`. Also, this function only initializes the superblock and group descriptor structures. It does not create the inode table or the root directory. This must be done by the calling application, such as `mke2fs`.

The following values may be set in the `param` prototype superblock; if a value of 0 is found in a field, `ext2fs_initialize` will use a default value. The calling application should zero out the prototype entire superblock, and then fill in any appropriate values.

`s_blocks_count`

The number of blocks in the filesystem. This parameter is mandatory and must be set by the calling application.

`s_inodes_count`

The number of inodes in the filesystem. The default value is determined by calculating the size of the filesystem, and creating one inode for every 4096 bytes.

s_r_blocks_count

The number of blocks which should be reserved for the superuser. The default value is zero blocks.

s_log_block_size

The blocksize of the filesystem. Valid values are 0 (1024 bytes), 1 (2048 bytes), or 2 (4096 bytes). The default blocksize is 1024 bytes.

s_log_frag_size

The size of fragments. The ext2 filesystem does not support fragments (and may never support fragments). Currently this field must be the same as **s_log_block_size**.

s_first_data_block

The first data block for the filesystem. For filesystem with a blocksize of 1024 bytes, this value must be at least 1, since the superblock is located in block number 1. For filesystems with larger blocksizes, the superblock is still located at an offset of 1024 bytes, so the superblock is located in block number 0. By default, this value is set to 1 for filesystems with a block size of 1024 bytes, or 0 for filesystems with larger blocksizes.

s_max_mnt_count

This field defines the number of times that the filesystem can be mounted before it should be checked using **e2fsck**. When **e2fsck** is run without the **-f** option, **e2fsck** will skip the filesystem check if the number of times that the filesystem has been mounted is less than **s_max_mnt_count** and if the interval between the last time a filesystem check was performed and the current time is less than **s_checkinterval** (see below). The default value of **s_max_mnt_count** is 20.

s_checkinterval

This field defines the minimal interval between filesystem checks. See the previous entry for a discussion of how this field is used by **e2fsck**. The default value of this field is 180 days (six months).

s_errors This field defines the behavior which should be used by the kernel of errors are detected in the filesystem. Possible values include:

'EXT2_ERRORS_CONTINUE'

Continue execution when errors are detected.

'EXT2_ERRORS_RO'

Remount the filesystem read-only.

'EXT2_ERRORS_PANIC'

Panic.

The default behavior is **'EXT2_ERRORS_CONTINUE'**.

errcode_t ext2fs_allocate_tables (*ext2_filsys fs*) [Function]

Allocate space for the inode table and the block and inode bitmaps. The inode tables and block and inode bitmaps aren't actually initialized; this function just allocates the space for them.

2.1.4 Filesystem flag functions

The filesystem handle has a number of flags which can be manipulated using the following function. Some of these flags affect how the libext2fs filesystem behaves; others are provided solely for the application's convenience.

`void ext2fs_mark_changed (ext2_filsys fs)` [Function]

`int ext2fs_test_changed (ext2_filsys fs)` [Function]

This flag indicates whether or not the filesystem has been changed. It is not used by the ext2fs library.

`void ext2fs_mark_super_dirty (ext2_filsys fs)` [Function]

Mark the filesystem *fs* as being dirty; this will cause the superblock information to be flushed out when `ext2fs_close` is called. `ext2fs_mark_super_dirty` will also set the filesystem changed flag. The dirty flag is automatically cleared by `ext2fs_flush` when the superblock is written to disk.

`void ext2fs_mark_valid (ext2_filsys fs)` [Function]

`void ext2fs_unmark_valid (ext2_filsys fs)` [Function]

`int ext2fs_test_valid (ext2_filsys fs)` [Function]

This flag indicates whether or not the filesystem is free of errors. It is not used by libext2fs, and is solely for the application's convenience.

`void ext2fs_mark_ib_dirty (ext2_filsys fs)` [Function]

`void ext2fs_mark_bb_dirty (ext2_filsys fs)` [Function]

`int ext2fs_test_ib_dirty (ext2_filsys fs)` [Function]

`int ext2fs_test_bb_dirty (ext2_filsys fs)` [Function]

These flags indicate whether or not the inode or block bitmaps have been modified. If the flag is set, it will cause the appropriate bitmap to be written when the filesystem is closed or flushed.

2.2 Inode Functions

2.2.1 Reading and writing inodes

`errcode_t ext2fs_read_inode (ext2_filsys fs, ext2_ino_t ino, struct ext2_inode *inode)` [Function]

Read the inode number *ino* into *inode*.

`errcode_t ext2fs_write_inode (ext2_filsys fs, ext2_ino_t ino, struct ext2_inode *inode)` [Function]

Write *inode* to inode *ino*.

2.2.2 Iterating over inodes in a filesystem

The `inode_scan` abstraction is useful for iterating over all the inodes in a filesystem.

`errcode_t ext2fs_open_inode_scan` (`ext2_filsys fs`, `int` `buffer_blocks`, `ext2_inode_scan *scan`) [Function]

Initialize the iteration variable `scan`. This variable is used by `ext2fs_get_next_inode`. The `buffer_blocks` parameter controls how many blocks of the inode table are read in at a time. A large number of blocks requires more memory, but reduces the overhead in seeking and reading from the disk. If `buffer_blocks` is zero, a suitable default value will be used.

`void ext2fs_close_inode_scan` (`ext2_inode_scan scan`) [Function]

Release the memory associated with `scan` and invalidate it.

`errcode_t ext2fs_get_next_inode` (`ext2_inode_scan scan`, `ext2_ino_t` `*ino`, `struct ext2_inode *inode`) [Function]

This function returns the next inode from the filesystem; the inode number of the inode is stored in `ino`, and the inode is stored in `inode`.

If the inode is located in a block that has been marked as bad, `ext2fs_get_next_inode` will return the error `EXT2_ET_BAD_BLOCK_IN_INODE_TABLE`.

`errcode_t ext2fs_inode_scan_goto_blockgroup` (`ext2_inode_scan scan`, `int group`) [Function]

Start the inode scan at a particular ext2 blockgroup, `group`. This function may be safely called at any time while `scan` is valid.

`void ext2fs_set_inode_callback` (`ext2_inode_scan scan`, `errcode_t` `(*done_group)`(`ext2_filsys fs`, `ext2_inode_scan scan`, `dgrp_t group`, `void *private`), `void *done_group_data`) [Function]

Register a callback function which will be called by `ext2_get_next_inode` when all of the inodes in a block group have been processed.

`int ext2fs_inode_scan_flags` (`ext2_inode_scan scan`, `int set_flags`, `int clear_flags`) [Function]

Set the scan flags `set_flags` and clear the scan flags `clear_flags`. The following flags can be set using this interface:

‘`EXT2_SF_SKIP_MISSING_ITABLE`’

When a block group is missing an inode table, skip it. If this flag is not set `ext2fs_get_next_inode` will return the error `EXT2_ET_MISSING_INODE_TABLE`.

2.2.3 Iterating over blocks in an inode

`errcode_t ext2fs_block_iterate` (`ext2_filsys fs`, `ext2_ino_t ino`, `int flags`, `char *block_buf`, `int (*func)`(`ext2_filsys fs`, `blk_t *blocknr`, `int blockcnt`, `void *private`), `void *private`) [Function]

Iterate over all of the blocks in inode number `ino` in filesystem `fs`, by calling the function `func` for each block in the inode. The `block_buf` parameter should either be `NULL`, or if the `ext2fs_block_iterate` function is called repeatedly, the overhead

of allocating and freeing scratch memory can be avoided by passing a pointer to a scratch buffer which must be at least as big as three times the filesystem's blocksize.

The *flags* parameter controls how the iterator will function:

'BLOCK_FLAG_HOLE'

This flag indicates that the iterator function should be called on blocks where the block number is zero (also known as "holes".) It is also known as BLOCK_FLAG_APPEND, since it is also used by functions such as `ext2fs_expand_dir()` to add a new block to an inode.

'BLOCK_FLAG_TRAVERSE'

This flag indicates that the iterator function for the indirect, doubly indirect, etc. blocks should be called after all of the blocks contained in the indirect blocks are processed. This is useful if you are going to be deallocating blocks from an inode.

'BLOCK_FLAG_DATA_ONLY'

This flag indicates that the iterator function should be called for data blocks only.

The callback function *func* is called with a number of parameters; the *fs* and *private* parameters are self-explanatory, and their values are taken from the parameters to `ext2fs_block_iterate`. (The *private* data structure is generally used by callers to `ext2fs_block_iterate` so that some private data structure can be passed to the callback function. The *blockcnt* parameter, if non-negative, indicates the logical block number of a data block in the inode. If *blockcnt* is less than zero, then *func* was called on a metadata block, and *blockcnt* will be one of the following values: BLOCK_COUNT_IND, BLOCK_COUNT_DIND, BLOCK_COUNT_TIND, or BLOCK_COUNT_TRANSLATOR. The *blocknr* is a pointer to the inode or indirect block entry listing physical block number. The callback function may modify the physical block number, if it returns the *BLOCK_CHANGED* flag.

The callback function *func* returns a result code which is composed of the logical OR of the following flags:

'BLOCK_CHANGED'

This flag indicates that callback function has modified the physical block number pointed to by *blocknr*.

'BLOCK_ABORT'

This flag requests that `ext2fs_block_iterate` to stop immediately and return to the caller.

```
errcode_t ext2fs_block_iterate2 (ext2_filsys fs, ext2_ino_t ino, int [Function]
    flags, char *block_buf, int (*func)(ext2_filsys fs, blk_t *blocknr,
    e2_blkcnt_t blockcnt, blk_t ref_blk, int ref_offset, void *private), void
    *private)
```

This function is much like `ext2fs_block_iterate2`, except that the *blockcnt* type is a 64-bit signed quantity, to support larger files, and the addition of the *ref_blk* and

ref_offset arguments passed to the callback function, which identify the location of the physical block pointed to by pointer *blocknr*. If *ref_blk* is zero, then *ref_offset* contains the offset into the *i_blocks* array. If *ref_blk* is non-zero, then the physical block location is contained inside an indirect block group, and *ref_offset* contains the offset into the indirect block.

2.2.4 Convenience functions for Inodes

`errcode_t ext2fs_get_blocks (ext2_filsys fs, ext2_ino_t ino, blk_t [Function]
*blocks)`

Returns an array of blocks corresponding to the direct, indirect, doubly indirect, and triply indirect blocks as stored in the inode structure.

`errcode_t ext2fs_check_directory (ext2_filsys fs, ext2_ino_t ino) [Function]`

Returns 0 if *ino* is a directory, and ENOTDIR if it is not.

`int ext2_inode_has_valid_blocks (struct ext2_inode *inode) [Function]`

Returns 1 if the inode's block entries actually valid block entries, and 0 if not. Inodes which represent devices and fast symbolic links do not contain valid block entries.

2.3 Directory functions

2.3.1 Directory block functions

`errcode_t ext2fs_read_dir_block (ext2_filsys fs, blk_t block, void [Function]
*buf)`

This function reads a directory block, performing any necessary byte swapping if necessary.

`errcode_t ext2fs_write_dir_block (ext2_filsys fs, blk_t block, [Function]
void *buf)`

This function writes a directory block, performing any necessary byte swapping if necessary.

`errcode_t ext2fs_new_dir_block (ext2_filsys fs, [Function]
ext2_ino_t dir_ino, ext2_ino_t parent_ino, char **block)`

This function creates a new directory block in *block*. If *dir_ino* is non-zero, then *dir_ino* and *parent_ino* is used to initialize directory entries for '.' and '..', respectively.

2.3.2 Iterating over a directory

`errcode_t ext2fs_dir_iterate (ext2_filsys fs, ext2_ino_t dir, int [Function]
flags, char *block_buf, int (*func)(struct ext2_dir_entry *dirent, int
offset, int blocksize, char *buf, void *private), void *private)`

This function iterates over all of the directory entries in the directory *dir*, calling the callback function *func* for each directory entry in the directory. The *block_buf* parameter should either be NULL, or if the `ext2fs_dir_iterate` function is called

repeatedly, the overhead of allocating and freeing scratch memory can be avoided by passing a pointer to a scratch buffer which must be at least as big as the filesystem's blocksize.

The *flags* parameter controls how the iterator will function:

`'DIRENT_FLAG_INCLUDE_EMPTY'`

This flag indicates that the callback function should be called even for deleted or empty directory entries.

2.3.3 Creating and expanding directories

`errcode_t ext2fs_mkdir (ext2_filsys fs, ext2_ino_t parent, ext2_ino_t inum, const char *name)` [Function]

This function creates a new directory. If *inum* is zero, then a new inode will be allocated; otherwise, the directory will be created in the inode specified by *inum*. If *name* specifies the name of the new directory; if it is non-NULL, then the new directory will be linked into the parent directory *parent*.

`errcode_t ext2fs_expand_dir (ext2_filsys fs, ext2_ino_t dir)` [Function]

This function adds a new empty directory block and appends it to the directory *dir*. This allows functions such as `ext2fs_link` to add new directory entries to a directory which is full.

2.3.4 Creating and removing directory entries

`errcode_t ext2fs_link (ext2_filsys fs, ext2_ino_t dir, const char *name, ext2_ino_t ino, int flags)` [Function]

This function adds a new directory entry to the directory *dir*, with *name* and *ino* specifying the name and inode number in the directory entry, respectively.

The low 3 bits of the flags field is used to specify the file type of inode: (No other flags are currently defined.)

`'EXT2_FT_UNKNOWN'`

The file type is unknown.

`'EXT2_FT_REG_FILE'`

The file type is a normal file.

`'EXT2_FT_DIR'`

The file type is a directory.

`'EXT2_FT_CHRDEV'`

The file type is a character device.

`'EXT2_FT_BLKDEV'`

The file type is a block device.

`'EXT2_FT_FIFO'`

The file type is a named pipe.

‘EXT2_FT_SOCK’

The file type is a unix domain socket.

‘EXT2_FT_SYMLINK’

The file type is a symbolic link.

`errcode_t ext2fs_unlink (ext2_filsys fs, ext2_ino_t dir, const char [Function]
*name, ext2_ino_t ino, int flags)`

This function removes a directory entry from *dir*. The directory entry to be removed is the first one which is matched by *name* and *ino*. If *name* is non-NULL, the directory entry’s name must match *name*. If *ino* is non-zero, the directory entry’s inode number must match *ino*. No flags are currently defined for `ext2fs_unlink`; callers should pass in zero to this parameter.

2.3.5 Looking up filenames

`errcode_t ext2fs_lookup (ext2_filsys fs, ext2_ino_t dir, const char [Function]
*name, int namelen, char *buf, ext2_ino_t *inode)`

`errcode_t ext2fs_namei (ext2_filsys fs, ext2_ino_t root, ext2_ino_t [Function]
cwd, const char *name, ext2_ino_t *inode)`

`errcode_t ext2fs_namei_follow (ext2_filsys fs, ext2_ino_t root, [Function]
ext2_ino_t cwd, const char *name, ext2_ino_t *inode)`

`errcode_t ext2fs_follow_link (ext2_filsys fs, ext2_ino_t root, [Function]
ext2_ino_t cwd, ext2_ino_t inode, ext2_ino_t *res_inode)`

2.3.6 Translating inode numbers to filenames

`errcode_t ext2fs_get_pathname (ext2_filsys fs, ext2_ino_t dir, [Function]
ext2_ino_t ino, char **name)`

2.4 Bitmap Functions

2.4.1 Reading and Writing Bitmaps

`errcode_t ext2fs_write_inode_bitmap (ext2_filsys fs) [Function]`

`errcode_t ext2fs_write_block_bitmap (ext2_filsys fs) [Function]`

`errcode_t ext2fs_read_inode_bitmap (ext2_filsys fs) [Function]`

`errcode_t ext2fs_read_block_bitmap (ext2_filsys fs) [Function]`

`errcode_t ext2fs_read_bitmaps (ext2_filsys fs) [Function]`

`errcode_t ext2fs_write_bitmaps (ext2_filsys fs) [Function]`

2.4.2 Allocating Bitmaps

`errcode_t ext2fs_allocate_generic_bitmap (--u32 start, --u32 [Function]
end, _u32 real_end, const char *descr, ext2fs_generic_bitmap *ret)`

`errcode_t ext2fs_allocate_block_bitmap (ext2_filsys fs, const [Function]
char *descr, ext2fs_block_bitmap *ret)`

`errcode_t ext2fs_allocate_inode_bitmap (ext2_filsys fs, const [Function]
char *descr, ext2fs_inode_bitmap *ret)`

2.4.3 Freeing bitmaps

void ext2fs_free_generic_bitmap (ext2fs_inode_bitmap bitmap) [Function]

void ext2fs_free_block_bitmap (ext2fs_block_bitmap bitmap) [Function]

void ext2fs_free_inode_bitmap (ext2fs_inode_bitmap bitmap) [Function]

2.4.4 Bitmap Operations

void ext2fs_mark_block_bitmap (ext2fs_block_bitmap bitmap, blk_t block) [Function]

void ext2fs_unmark_block_bitmap (ext2fs_block_bitmap bitmap, blk_t block) [Function]

int ext2fs_test_block_bitmap (ext2fs_block_bitmap bitmap, blk_t block) [Function]

These functions set, clear, and test bits in a block bitmap *bitmap*.

void ext2fs_mark_inode_bitmap (ext2fs_inode_bitmap bitmap, ext2_ino_t inode) [Function]

void ext2fs_unmark_inode_bitmap (ext2fs_inode_bitmap bitmap, ext2_ino_t inode) [Function]

int ext2fs_test_inode_bitmap (ext2fs_inode_bitmap bitmap, ext2_ino_t inode) [Function]

These functions set, clear, and test bits in an inode bitmap *bitmap*.

void ext2fs_fast_mark_block_bitmap (ext2fs_block_bitmap bitmap, blk_t block) [Function]

void ext2fs_fast_unmark_block_bitmap (ext2fs_block_bitmap bitmap, blk_t block) [Function]

int ext2fs_fast_test_block_bitmap (ext2fs_block_bitmap bitmap, blk_t block) [Function]

void ext2fs_fast_mark_inode_bitmap (ext2fs_inode_bitmap bitmap, ext2_ino_t inode) [Function]

void ext2fs_fast_unmark_inode_bitmap (ext2fs_inode_bitmap bitmap, ext2_ino_t inode) [Function]

int ext2fs_fast_test_inode_bitmap (ext2fs_inode_bitmap bitmap, ext2_ino_t inode) [Function]

These “fast” functions are like their normal counterparts; however, they are implemented as inline functions and do not perform bounds checks on the inode number or block number; they are assumed to be correct. They should only be used in speed-critical applications, where the inode or block number has already been validated by other means.

blk_t ext2fs_get_block_bitmap_start (ext2fs_block_bitmap bitmap) [Function]

ext2_ino_t ext2fs_get_inode_bitmap_start (ext2fs_inode_bitmap bitmap) [Function]

Return the first inode or block which is stored in the bitmap.

`blk_t ext2fs_get_block_bitmap_end (ext2fs_block_bitmap bitmap)` [Function]
`ext2_ino_t ext2fs_get_inode_bitmap_end (ext2fs_inode_bitmap bitmap)` [Function]

Return the last inode or block which is stored in the bitmap.

2.4.5 Comparing bitmaps

`errcode_t ext2fs_compare_block_bitmap (ext2fs_block_bitmap bm1, ext2fs_block_bitmap bm2)` [Function]

`errcode_t ext2fs_compare_inode_bitmap (ext2fs_inode_bitmap bm1, ext2fs_inode_bitmap bm2)` [Function]

2.4.6 Modifying Bitmaps

`errcode_t ext2fs_fudge_inode_bitmap_end (ext2fs_inode_bitmap bitmap, ext2_ino_t end, ext2_ino_t *oend)` [Function]

`errcode_t ext2fs_fudge_block_bitmap_end (ext2fs_block_bitmap bitmap, blk_t end, blk_t *oend)` [Function]

2.4.7 Resizing Bitmaps

`errcode_t ext2fs_resize_generic_bitmap (--u32 new_end, --u32 new_real_end, ext2fs_generic_bitmap bmap)` [Function]

`errcode_t ext2fs_resize_inode_bitmap (--u32 new_end, --u32 new_real_end, ext2fs_inode_bitmap bmap)` [Function]

`errcode_t ext2fs_resize_block_bitmap (--u32 new_end, --u32 new_real_end, ext2fs_block_bitmap bmap)` [Function]

2.4.8 Clearing Bitmaps

`void ext2fs_clear_inode_bitmap (ext2fs_inode_bitmap bitmap)` [Function]
 This function sets all of the bits in the inode bitmap *bitmap* to be zero.

`void ext2fs_clear_block_bitmap (ext2fs_block_bitmap bitmap)` [Function]
 This function sets all of the bits in the block bitmap *bitmap* to be zero.

2.5 EXT2 data abstractions

The ext2 library has a number of abstractions which are useful for ext2 utility programs.

2.5.1 Badblocks list management

`errcode_t ext2fs_badblocks_list_create (ext2_badblocks_list *ret, int size)` [Function]

`void ext2fs_badblocks_list_free (ext2_badblocks_list bb)` [Function]

`errcode_t ext2fs_badblocks_list_add (ext2_badblocks_list bb, blk_t blk)` [Function]

`int ext2fs_badblocks_list_test (ext2_badblocks_list bb, blk_t blk)` [Function]

```

errcode_t ext2fs_badblocks_list_iterate_begin (Function]
    (ext2_badblocks_list bb, ext2_badblocks_iterate *ret)
int ext2fs_badblocks_list_iterate (Function]
    (ext2_badblocks_iterate iter,
    blk_t *blk)
void ext2fs_badblocks_list_iterate_end (Function]
    (ext2_badblocks_iterate
    iter)
errcode_t ext2fs_update_bb_inode (Function]
    (ext2_filsys fs,
    ext2_badblocks_list bb_list)
errcode_t ext2fs_read_bb_inode (Function]
    (ext2_filsys fs, ext2_badblocks_list
    *bb_list)
errcode_t ext2fs_read_bb_FILE (Function]
    (ext2_filsys fs, FILE *f,
    ext2_badblocks_list *bb_list, void (*invalid)(ext2_filsys fs, blk_t blk))

```

2.5.2 Directory-block list management

The dblink abstraction stores a list of blocks belonging to directories. This list can be useful when a program needs to iterate over all directory entries in a filesystem; `e2fsck` does this in pass 2 of its operations, and `debugfs` needs to do this when it is trying to turn an inode number into a pathname.

```

errcode_t ext2fs_init_dblink (Function]
    (ext2_filsys fs, ext2_dblink
    *ret_dblink)
    Creates a dblink data structure and return it in ret_dblink.

void ext2fs_free_dblink (Function]
    (ext2_dblink dblink)
    Free a dblink data structure.

errcode_t ext2fs_add_dir_block (Function]
    (ext2_dblink dblink, ext2_ino_t
    ino, blk_t blk, int blockcnt)
    Add an entry to the dblink data structure. This call records the fact that block number
    blockcnt of directory inode ino is stored in block blk.

errcode_t ext2fs_set_dir_block (Function]
    (ext2_dblink dblink, ext2_ino_t
    ino, blk_t blk, int blockcnt)
    Change an entry in the dblink data structure; this changes the location of block number
    blockcnt of directory indoe ino to be block blk.

errcode_t ext2fs_dblink_iterate (Function]
    (ext2_dblink dblink, int
    (*func)(ext2_filsys fs, struct ext2_db_entry *db_info, void *private), void
    *private)
    This iterator calls func for every entry in the dblink data structure.

errcode_t ext2fs_dblink_dir_iterate (Function]
    (ext2_dblink dblink, int
    flags, char *block_buf, int (*func)(ext2_ino_t dir, int entry, struct
    ext2_dir_entry *dirent, int offset, int blocksize, char *buf, void
    *private), void *private)

```

This iterator takes reads in the directory block indicated in each dblink entry, and calls *func* for each directory entry in each directory block. If *dblist* contains all the

directory blocks in a filesystem, this function provides a convenient way to iterate over all directory entries for that filesystem.

2.5.3 Inode count functions

The `icount` abstraction is a specialized data type used by `e2fsck` to store how many times a particular inode is referenced by the filesystem. This is used twice; once to store the actual number of times that the inode is reference; and once to store the claimed number of times the inode is referenced according to the inode structure.

This abstraction is designed to be extremely efficient for storing this sort of information, by taking advantage of the following properties of inode counts, namely (1) inode counts are very often zero (because the inode is currently not in use), and (2) many files have a inode count of 1 (because they are a file which has no additional hard links).

`errcode_t ext2fs_create_icount2 (ext2_filsys fs, int flags, int size, ext2_icount_t hint, ext2_icount_t *ret)` [Function]

Creates an `icount` structure for a filesystem `fs`, with initial space for `size` inodes whose count is greater than 1. The `flags` parameter is either 0 or `EXT2_ICOUNT_OPT_INCREMENT`, which indicates that `icount` structure should be able to increment inode counts quickly. The `icount` structure is returned in `ret`. The returned `icount` structure initially has a count of zero for all inodes.

The `hint` parameter allows the caller to optionally pass in another `icount` structure which is used to initialize the array of inodes whose count is greater than 1. It is used purely as a speed optimization so that the `icount` structure can determine in advance which inodes are likely to contain a count greater than 1.

`void ext2fs_free_icount (ext2_icount_t icount)` [Function]

Frees an `icount` structure.

`errcode_t ext2fs_icount_fetch (ext2_icount_t icount, ext2_ino_t ino, __u16 *ret)` [Function]

Returns in `ret` fetches the count for a particular inode `ino`.

`errcode_t ext2fs_icount_increment (ext2_icount_t icount, ext2_ino_t ino, __u16 *ret)` [Function]

Increments the ref count for inode `ino`.

`errcode_t ext2fs_icount_decrement (ext2_icount_t icount, ext2_ino_t ino, __u16 *ret)` [Function]

Decrements the ref count for inode `ino`.

`errcode_t ext2fs_icount_store (ext2_icount_t icount, ext2_ino_t ino, __u16 count)` [Function]

Sets the reference count for inode `ino` to be `count`.

`ext2_ino_t ext2fs_get_icount_size (ext2_icount_t icount)` [Function]

Returns the current number of inodes in `icount` which has a count greater than 1.


```
errcode_t ext2fs_icount_validate (ext2_icount_t icount, FILE *f) [Function]
```

Validates the internal rep invariant of *icount*; if there are any problems, print out debugging information to *f*. This function is intended for debugging and testing use only.

2.6 Byte-swapping functions

```
void ext2fs_swap_super (struct ext2_super_block *super) [Function]
```

```
void ext2fs_swap_group_desc (struct ext2_group_desc *gdp) [Function]
```

```
void ext2fs_swap_inode (ext2_filsys fs, struct ext2_inode *to, struct ext2_inode *from, int hostorder) [Function]
```

```
int ext2fs_native_flag (void) [Function]
```

2.7 Other functions

```
/* alloc.c */
```

```
errcode_t ext2fs_new_inode (ext2_filsys fs, ext2_ino_t dir, int mode, ext2fs_inode_bitmap map, ext2_ino_t *ret) [Function]
```

```
errcode_t ext2fs_new_block (ext2_filsys fs, blk_t goal, ext2fs_block_bitmap map, blk_t *ret) [Function]
```

```
errcode_t ext2fs_get_free_blocks (ext2_filsys fs, blk_t start, blk_t finish, int num, ext2fs_block_bitmap map, blk_t *ret) [Function]
```

```
/* check_desc.c */
```

```
errcode_t ext2fs_check_desc (ext2_filsys fs) [Function]
```

```
errcode_t ext2_get_num_dirs (ext2_filsys fs, ext2_ino_t *ret_num_dirs) [Function]
```

```
/* getsize.c */
```

```
errcode_t ext2fs_get_device_size (const char *file, int blocksize, blk_t *retblocks) [Function]
```

```
/* ismounted.c */
```

```
errcode_t ext2fs_check_if_mounted (const char *file, int *mount_flags) [Function]
```

```
/* version.c */
```

```
int ext2fs_get_library_version (const char **ver_string, const char **date_string) [Function]
```

This function returns the current version of the ext2 library. The return value contains an integer version code, which consists of the major version number of the library multiplied by 100, plus the minor version number of the library. Hence, if the library version is 1.08, the returned value will be 108.

If *ver_string* and/or *date_string* are non-NULL, they will be set to point at a constant string containing the library version and/or release date, respectively.

```
int ext2fs_parse_version_string (const char *ver_string) [Function]
```

This function takes a version string which may included in an application and returns a version code using the same algorithm used by `ext2fs_get_library_version`. It can be used by programs included in the `e2fsprogs` distribution to assure that they are using an up-to-date ext2 shared library.

```
/* inline functions */
```

```
int ext2fs_group_of_blk (ext2_filsys fs, blk_t blk) [Function]
```

This function returns the block group which contains the block `blk`.

```
int ext2fs_group_of_ino (ext2_filsys fs, ext2_ino_t ino) [Function]
```

This function returns the block group which contains the inode `ino`.

Concept Index

(Index is nonexistent)

Function and Type Index

(Index is nonexistent)

Table of Contents

1	Introduction to the EXT2FS Library	1
2	EXT2FS Library Functions	2
2.1	Filesystem-level functions	2
2.1.1	Opening an ext2 filesystem	2
2.1.2	Closing and flushing out changes	2
2.1.3	Initializing a filesystem	3
2.1.4	Filesystem flag functions	5
2.2	Inode Functions	5
2.2.1	Reading and writing inodes	5
2.2.2	Iterating over inodes in a filesystem	5
2.2.3	Iterating over blocks in an inode	6
2.2.4	Convenience functions for Inodes	8
2.3	Directory functions	8
2.3.1	Directory block functions	8
2.3.2	Iterating over a directory	8
2.3.3	Creating and expanding directories	9
2.3.4	Creating and removing directory entries	9
2.3.5	Looking up filenames	10
2.3.6	Translating inode numbers to filenames	10
2.4	Bitmap Functions	10
2.4.1	Reading and Writing Bitmaps	10
2.4.2	Allocating Bitmaps	10
2.4.3	Freeing bitmaps	11
2.4.4	Bitmap Operations	11
2.4.5	Comparing bitmaps	12
2.4.6	Modifying Bitmaps	12
2.4.7	Resizing Bitmaps	12
2.4.8	Clearing Bitmaps	12
2.5	EXT2 data abstractions	12
2.5.1	Badblocks list management	12
2.5.2	Directory-block list management	13
2.5.3	Inode count functions	14
2.6	Byte-swapping functions	15
2.7	Other functions	15
	Concept Index	17
	Function and Type Index	18

