# SneakyFS

1.0

Jonathan Gillett
Joseph Heron
Daniel Smullen

Document Revision 1.1

# Contents

# Chapter 1

# Deprecated List

**Global UNIT_TESTING**

This definition is no longer used since the unit test suite was no longer used and deprecated once the high-level file system functions were implemented.

# Chapter 2

# Data Structure Index

## 2.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 3

# File Index

## 3.1 File List

Here is a list of all files with brief descriptions:

# Chapter 4

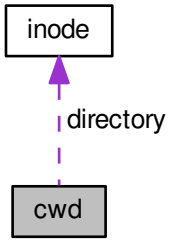
# Data Structure Documentation

## 4.1 cwd Struct Reference

Holds the current working directory data.

`#include <I_node.h>`

Collaboration diagram for cwd:




### Data Fields

- inode directory
- uint32_t cur_index

### 4.1.1 Detailed Description

Holds the current working directory data.

This struct holds the current working directory in memory. It is used in a pseudo-object oriented fashion in order to encapsulate the data pertaining to the contents of the current working directory and the last directory element that was retrieved from reading the directory contents.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

### 4.1.2 Field Documentation

#### 4.1.2.1 uint32_t cur_index

This contains the inode for the current working directory.

#### 4.1.2.2 inode directory

The documentation for this struct was generated from the following file:

- l_node.h

## 4.2 data_index Struct Reference

`#include <index_block.h>`

### Data Fields

- uint32_t index_location
- locations data_locations

### 4.2.1 Field Documentation

#### 4.2.1.1 locations data_locations

#### 4.2.1.2 uint32_t index_location

The documentation for this struct was generated from the following file:

- index_block.h

## 4.3 free_block_list Struct Reference

The free_block_list is an array containing the free blocks on disk. Elements are marked as free (false), or used (true). The super block always points to the first index block of the free_block_list.

`#include <free_block_list.h>`

### Data Fields

- bool free_blocks [NUMBLKS]

### 4.3.1 Detailed Description

The free_block_list is an array containing the free blocks on disk. Elements are marked as free (false), or used (true). The super block always points to the first index block of the free_block_list.

### 4.3.2 Field Documentation

#### 4.3.2.1 bool free_blocks[NUMBLKS]

The documentation for this struct was generated from the following file:

- free_block_list.h

## 4.4 inode Struct Reference

Contains the data stored in inodes.

`#include <I_node.h>`

### Data Fields

- char name [MAX_NAME_LEN]
- bool type
- bool read
- bool write
- bool execute
- time_t date_of_create
- time_t date_last_accessed
- time_t date_last_modified
- uint32_t file_owner
- uint32_t last_user_modified
- uint32_t file_size
- uint32_t location
- bool encrypted
- uint32_t check_sum
- uuid_t uuid

### 4.4.1 Detailed Description

Contains the data stored in inodes.

This struct defines the data which will be stored inside blocks designated as inodes on the disk. In particular, inodes store data pertaining to the attributes of they files they represent, as well as linkages to the index block structure that allows the file system to track their associated data blocks. Since files can be either a directory or a data file, inodes provide all the data required to track both types of data structures.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

### 4.4.2 Field Documentation

#### 4.4.2.1 uint32_t check_sum

This flags whether the file's data contents are encrypted.

#### 4.4.2.2 time_t date_last_accessed

This stores what the date of creation is for the file in POSIX time format.

#### 4.4.2.3 time_t date_last_modified

This stores the date that the file was last accessed by the file system in POSIX time format.

#### 4.4.2.4 time_t date_of_create

This flags whether the file is marked as executable or not.

#### 4.4.2.5 bool encrypted

This stores the location of the file's first index block on disk.

#### 4.4.2.6 bool execute

This flags whether the file is marked as writable or not.

#### 4.4.2.7 uint32_t file_owner

This stores the date that the file was last modified by the file system in POSIX time format.

#### 4.4.2.8 uint32_t file_size

This stores the name of the last user who modified the file.

#### 4.4.2.9 uint32_t last_user_modified

This stores the name of the owner of the file.

#### 4.4.2.10 uint32_t location

This stores the size of the file in bytes.

#### 4.4.2.11 char name[MAX_NAME_LEN]

#### 4.4.2.12 bool read

The type of file. Files can be either data files, or directories.

#### 4.4.2.13 bool type

The human-readable name of the file.

#### 4.4.2.14 uuid_t uuid

This stores the encryption checksum for the file, to determine whether it has been encrypted or unencrypted correctly.

#### 4.4.2.15 bool write

This flags whether the file is marked as readable or not.

The documentation for this struct was generated from the following file:

- l_node.h

## 4.5 Superblock Struct Reference

The superblock contains information pertainent to the disk.

`#include <super_block.h>`

### 4.5.1 Detailed Description

The superblock contains information pertainent to the disk.

This struct defines the data which will be stored inside the super block. In particular, the super block contains information pertaining to disk attributes, and links to critical file system data structures.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

The documentation for this struct was generated from the following file:

- super_block.h

## 4.6 superblock Struct Reference

`#include <super_block.h>`

### Data Fields

- uint32_t size_of_disk
- uint32_t block_size
- uint32_t free_block_list
- uint32_t root_dir
- uint32_t device_id
- uuid_t uuid

### 4.6.1 Field Documentation

#### 4.6.1.1 uint32_t block_size

Total number of blocks on the disk.

#### 4.6.1.2 uint32_t device_id

Location of the root directory's inode on disk.

#### 4.6.1.3 uint32_t free_block_list

Block size of the disk, in bytes per block.

#### 4.6.1.4 uint32_t root_dir

Location of the free block list data structure on disk.

#### 4.6.1.5 uint32_t size_of_disk

#### 4.6.1.6 uuid_t uuid

Device identifier for the disk. No longer required since UUID was implemented.

The documentation for this struct was generated from the following file:

- super_block.h

## 4.7 swoft Struct Reference

Pseudo-object containing the array of open file descriptors.

`#include <system_open_file_table.h>`

### Data Fields

- uint32_t fd [NUMOFL]
- bool taken [NUMOFL]

### 4.7.1 Detailed Description

Pseudo-object containing the array of open file descriptors.

The system wide open file table contains an array of file descriptors, which are locations of inodes that have been read from the disk into memory. These are used as handles to the files which exist on disk, and are synchronized with the data on disk upon modification. One instance needs to exist per file.

#### Author

Daniel Smullen
Jonathan Gillett
Joseph Heron

#### Copyright

GNU General Public License V3

### 4.7.2 Field Documentation

#### 4.7.2.1 uint32_t fd[NUMOFL]

#### 4.7.2.2 bool taken[NUMOFL]

The documentation for this struct was generated from the following file:

- system_open_file_table.h

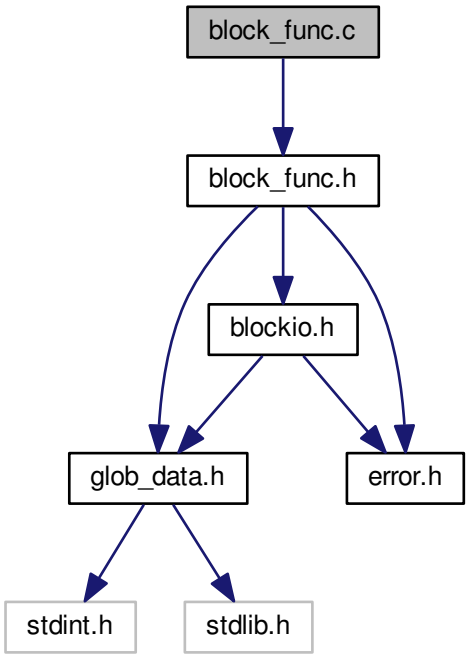
# Chapter 5

# File Documentation

## 5.1 block_func.c File Reference

```
#include "block_func.h"
```

Include dependency graph for block_func.c:



## Functions

- int write_block (uint32_t location, byte *buf)

  *A wrapper for the put_block function.*
- int read_block (uint32_t location, byte *buf)

  *A wrapper for the get_block function.*
- byte * get_data (locations location)

  *Get the data blocks and concatenate them into a large byte buffer.*

### 5.1.1 Function Documentation

#### 5.1.1.1 byte* get_data ( locations *location* )

Get the data blocks and concatenate them into a large byte buffer.

**Parameters**

| | |
|---|---|
| *location* | The locations of the data blocks on disk. |

**Returns**

Returns the data_buffer containing all of the bytes of data.

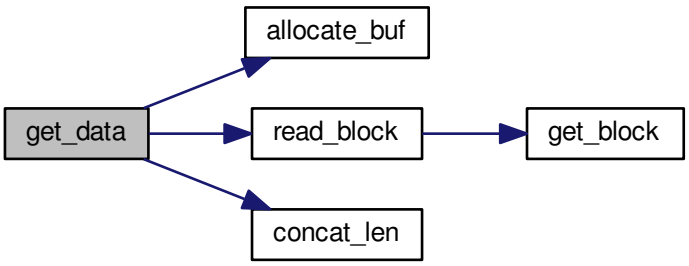
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

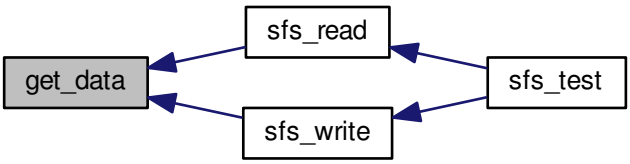
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.1.1.2 int read_block ( uint32_t *location,* byte * *buf* )

A wrapper for the get_block function.

**Parameters**

| | |
|---|---|
| *location* | The location on disk for the buffer to be written to. |
| *The* | buffer to store on disk. |

**Returns**

Returns an integer value. If the value >= 0 the function was successful. Otherwise, the function was unsuccessful.

**Author**

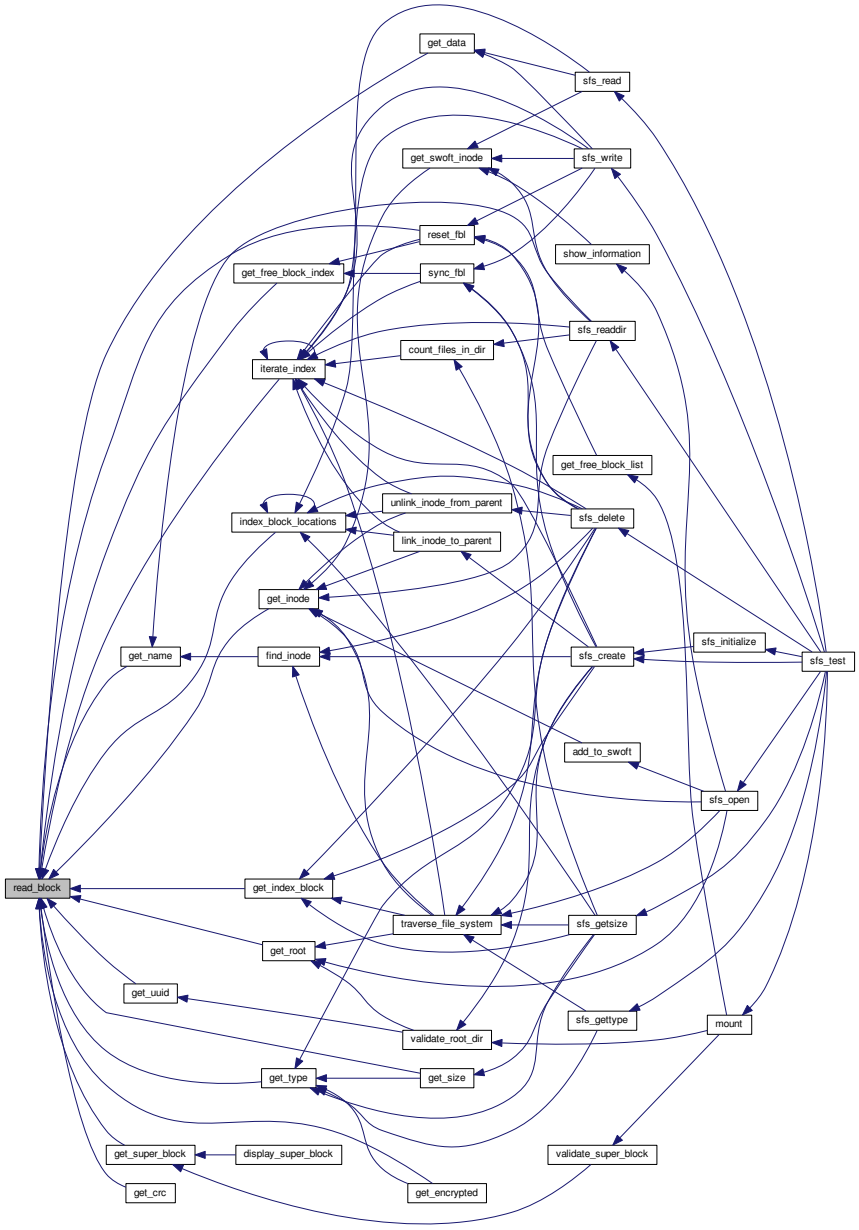
Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

Here is the caller graph for this function:




#### 5.1.1.3 int write_block ( uint32_t *location*, byte * *buf* )

A wrapper for the put_block function.

**Parameters**

| | |
|---|---|
| *location* | The location on disk for the buffer to be written to. |
| *The* | buffer to store on disk. |

### Returns

Returns an integer value. If the value >= 0, the function was successful. Otherwise, the function was unsuccessful.

### Author

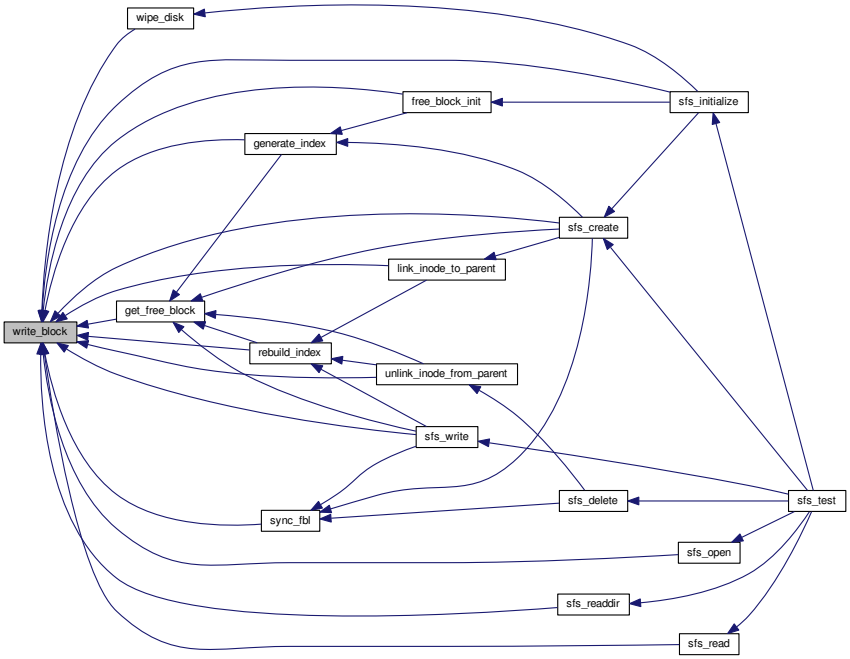
Daniel Smullen
Jonathan Gillett
Joseph Heron

### Copyright

GNU General Public License V3

Here is the call graph for this function:

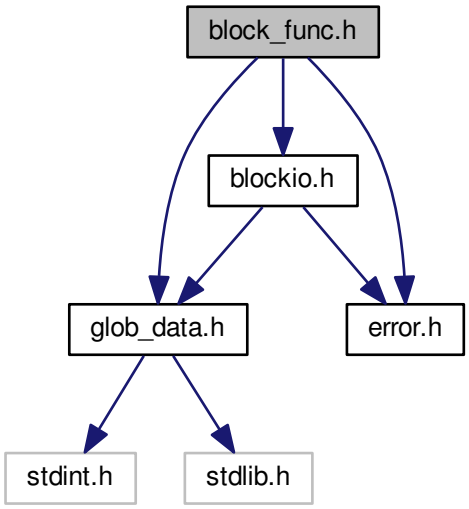

Here is the caller graph for this function:



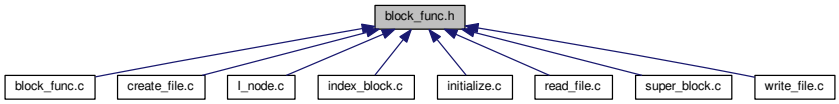
## 5.2 block_func.h File Reference

```
#include "blockio.h"
#include "glob_data.h"
#include "error.h"
```

Include dependency graph for block_func.h:




This graph shows which files directly or indirectly include this file:




### Functions

- int write_block (uint32_t location, byte *buf)

  *A wrapper for the put_block function.*

- int read_block (uint32_t location, byte *buf)

  *A wrapper for the get_block function.*

- byte * get_data (locations location)

  *Get the data blocks and concatenate them into a large byte buffer.*

### 5.2.1 Function Documentation

#### 5.2.1.1 byte* get_data ( locations *location* )

Get the data blocks and concatenate them into a large byte buffer.

**Parameters**

| | |
|---|---|
| *location* | The locations of the data blocks on disk. |

**Returns**

Returns the data_buffer containing all of the bytes of data.

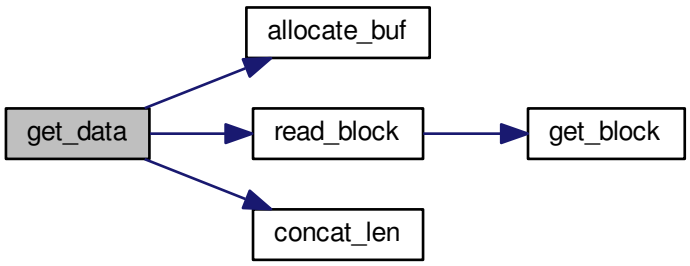
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

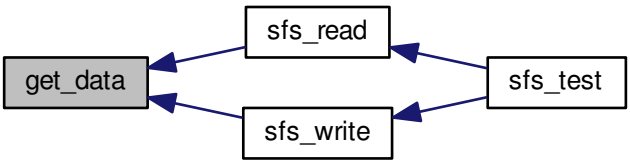
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.2.1.2 int read_block ( uint32_t *location,* byte * *buf* )

A wrapper for the get_block function.

**Parameters**

| | |
|---|---|
| *location* | The location on disk for the buffer to be written to. |
| *The* | buffer to store on disk. |

**Returns**

Returns an integer value. If the value >= 0 the function was successful. Otherwise, the function was unsuccessful.

**Author**

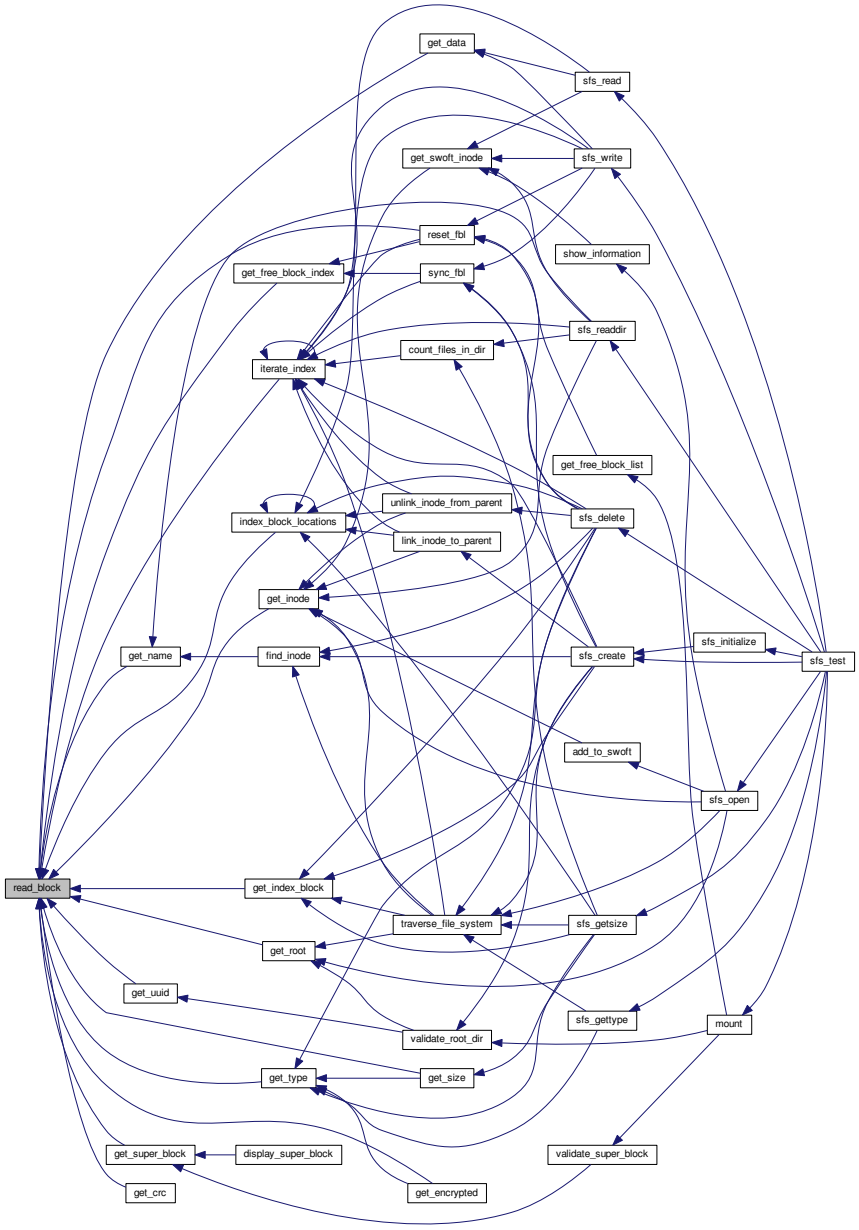
Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

Here is the caller graph for this function:




#### 5.2.1.3 int write_block ( uint32_t *location*, byte * *buf* )

A wrapper for the put_block function.

**Parameters**

| | |
|---|---|
| *location* | The location on disk for the buffer to be written to. |
| *The* | buffer to store on disk. |

### Returns

Returns an integer value. If the value >= 0, the function was successful. Otherwise, the function was unsuccessful.

### Author

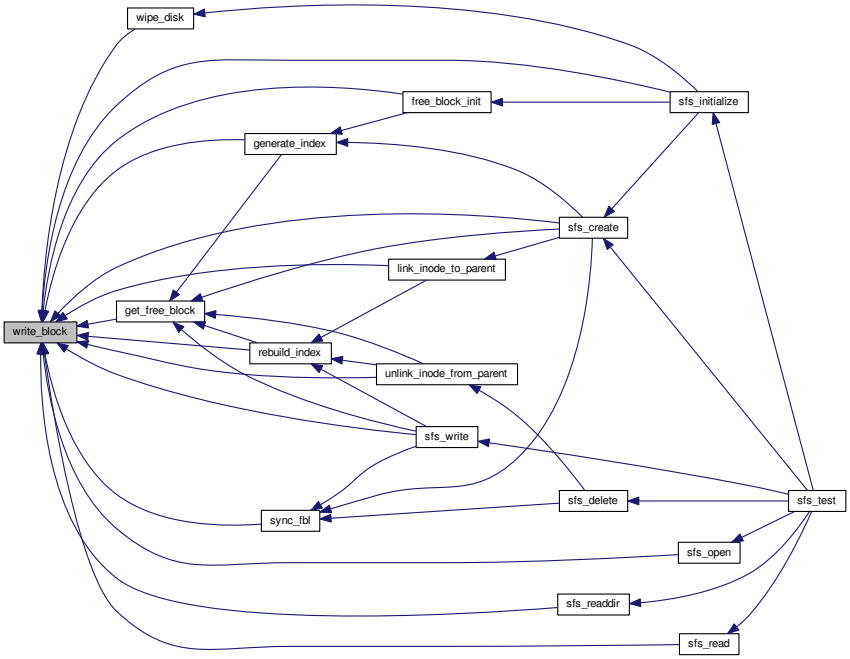
Daniel Smullen
Jonathan Gillett
Joseph Heron

### Copyright

GNU General Public License V3

Here is the call graph for this function:

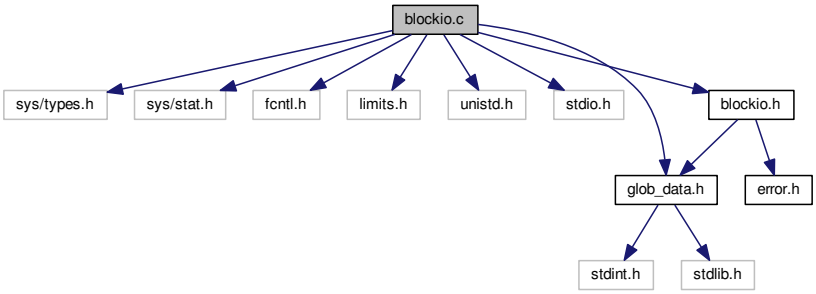

Here is the caller graph for this function:



## 5.3 blockio.c File Reference

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <limits.h>
#include <unistd.h>
#include <stdio.h>
#include "blockio.h"
#include "glob_data.h"
```

Include dependency graph for blockio.c:




### Macros

- #define DISKFILE "simdisk.data"
- #define DISKFILEMODE S_IRUSR|S_IWUSR|S_IRWXG

### Functions

- int get_block (uint32_t blknum, byte *buf)

  *Retrieves a block from disk.*
- int put_block (uint32_t blknum, byte *buf)

  *Writes a block to disk.*

### 5.3.1 Macro Definition Documentation

#### 5.3.1.1 #define DISKFILE "simdisk.data"

#### 5.3.1.2 #define DISKFILEMODE S_IRUSR|S_IWUSR|S_IRWXG

### 5.3.2 Function Documentation

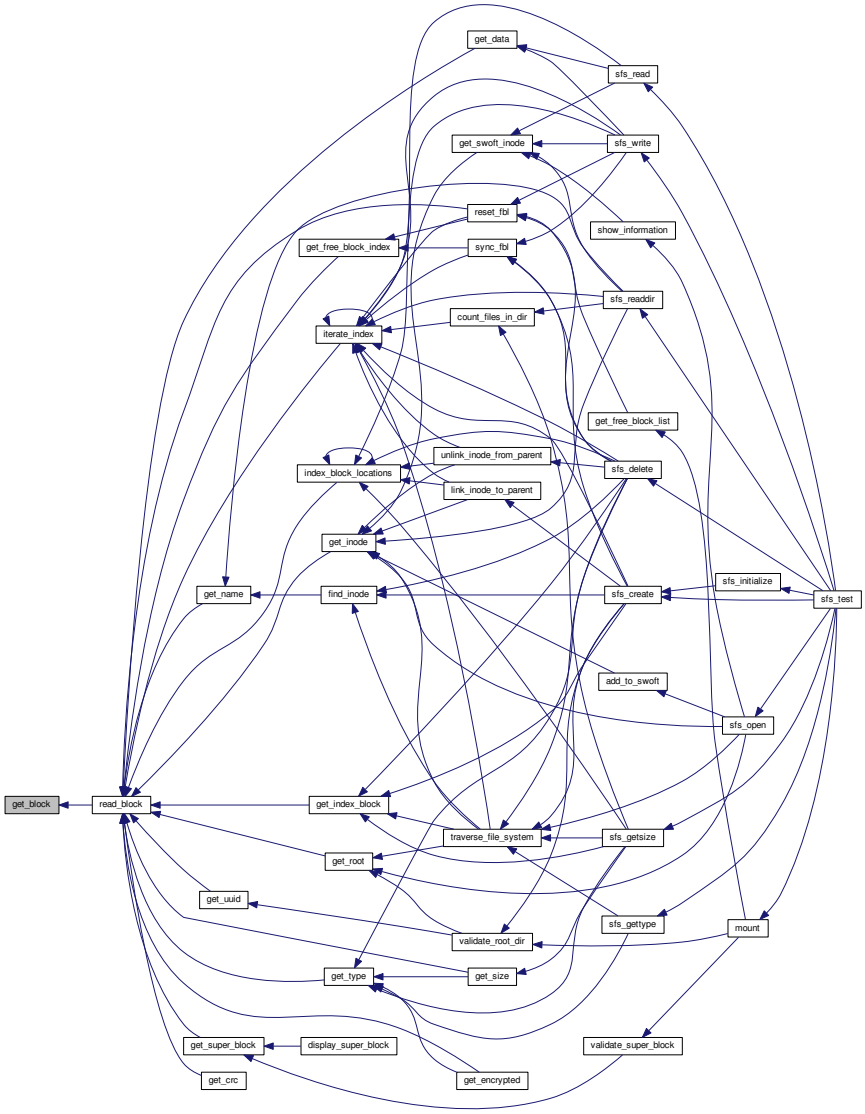
#### 5.3.2.1 int get_block ( uint32_t *blknum*, byte * *buf* )

Retrieves a block from disk.

**Parameters**

| | |
|---|---|
| *blknum* | Which disk block to retrieve. |
| *buf* | Where in memory to put the retrieved data. |

Here is the caller graph for this function:



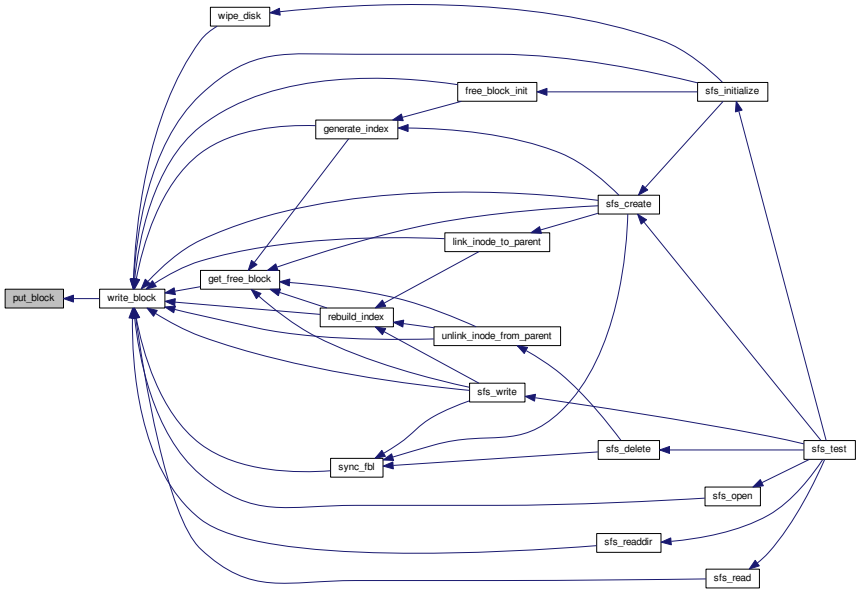

#### 5.3.2.2 int put_block ( uint32_t *blknum*, byte * *buf* )

Writes a block to disk.

**Parameters**

| | |
|---|---|
| *blknum* | Which disk block to update. |
| *buf* | Where in memory to get new disk block contents. |

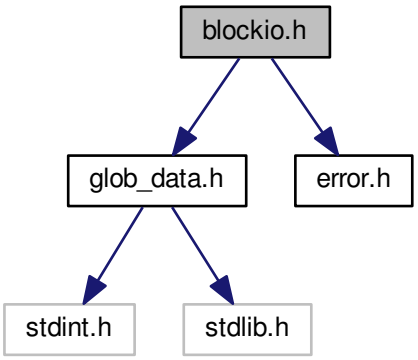
Here is the caller graph for this function:



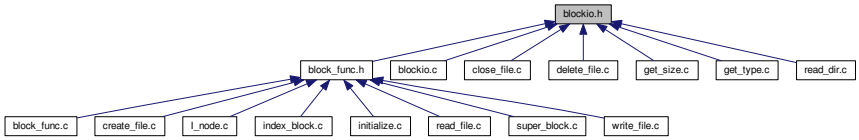

## 5.4 blockio.h File Reference

```
#include "glob_data.h"
#include "error.h"
```

Include dependency graph for blockio.h:



This graph shows which files directly or indirectly include this file:




## Functions

- int get_block (uint32_t blknum, byte *buf)

  *Retrieves a block from disk.*
- int put_block (uint32_t blknum, byte *buf)

  *Writes a block to disk.*

### 5.4.1 Function Documentation

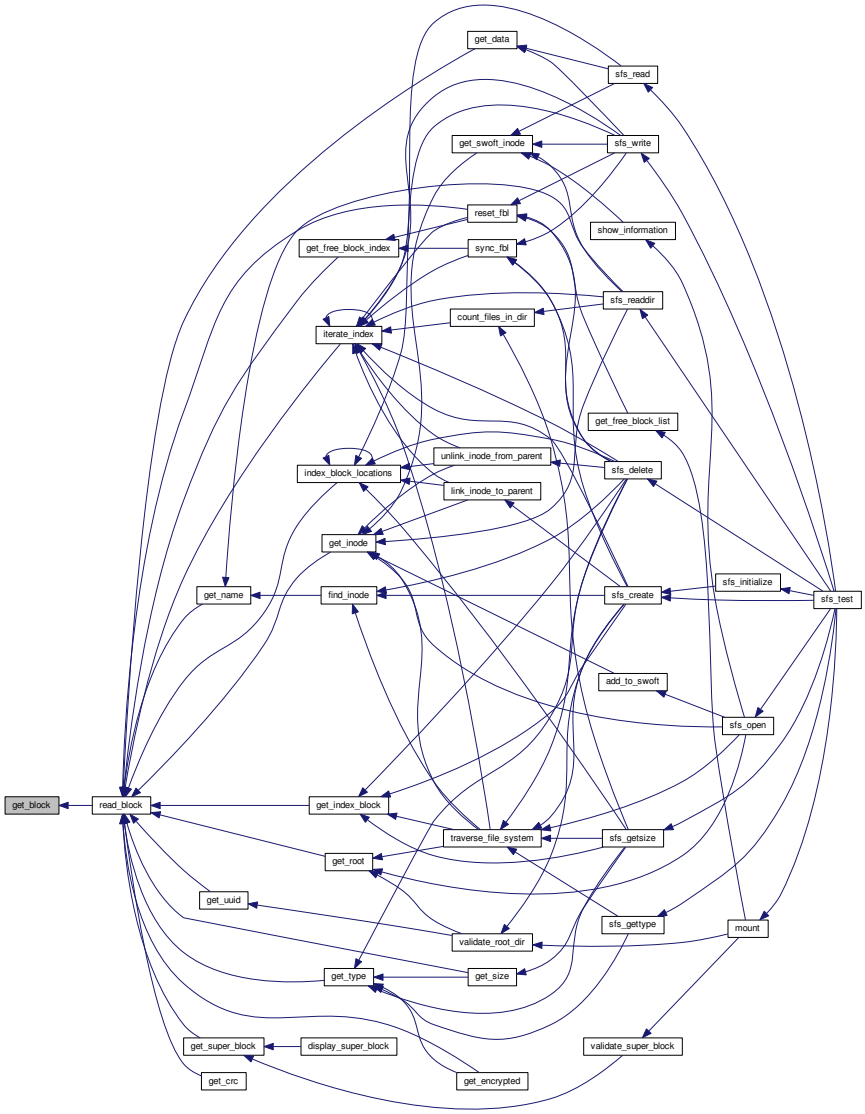
#### 5.4.1.1 int get_block ( uint32_t *blknum*, byte * *buf* )

Retrieves a block from disk.

**Parameters**

| | |
|---|---|
| *blknum* | Which disk block to retrieve. |
| *buf* | Where in memory to put the retrieved data. |

Here is the caller graph for this function:



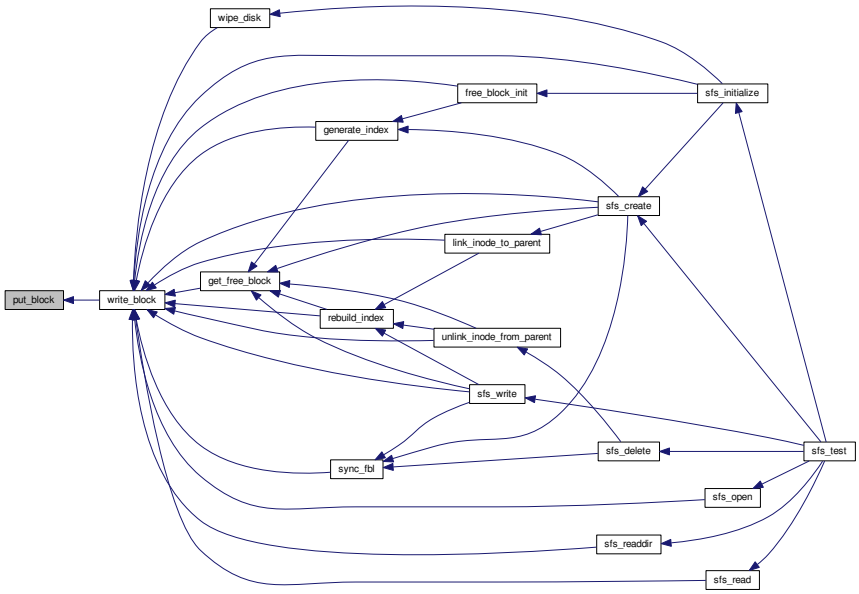

#### 5.4.1.2 int put_block ( uint32_t *blknum*, byte * *buf* )

Writes a block to disk.

**Parameters**

| | |
|---|---|
| *blknum* | Which disk block to update. |
| *buf* | Where in memory to get new disk block contents. |

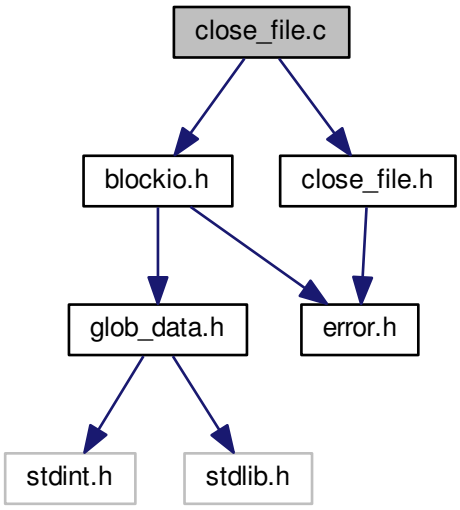
Here is the caller graph for this function:




## 5.5 close_file.c File Reference

```
#include "blockio.h"
#include "close_file.h"
```

Include dependency graph for close_file.c:




## Functions

- int sfs_close (int fd)

  *Closes the file, which indicates that the file is no longer needed.*

### 5.5.1 Function Documentation

#### 5.5.1.1 int sfs_close ( int *fd* )

Closes the file, which indicates that the file is no longer needed.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to close. |

**Returns**

Returns an integer value. If the value is $>$ 0 the file close was a success, if the value $<=$ 0 the file close was unsuccessful.

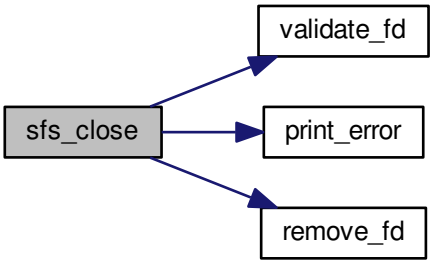
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

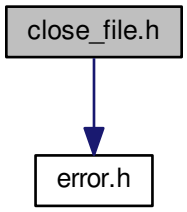



Here is the caller graph for this function:

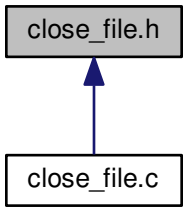
## 5.6 close_file.h File Reference

```
#include "error.h"
```

Include dependency graph for close_file.h:




This graph shows which files directly or indirectly include this file:




### Functions

- int sfs_close (int fd)

  *Closes the file, which indicates that the file is no longer needed.*

### 5.6.1 Function Documentation

#### 5.6.1.1 int sfs_close ( int *fd* )

Closes the file, which indicates that the file is no longer needed.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to close. |

**Returns**

Returns an integer value. If the value is $>$ 0 the file close was a success, if the value $<=$ 0 the file close was unsuccessful.

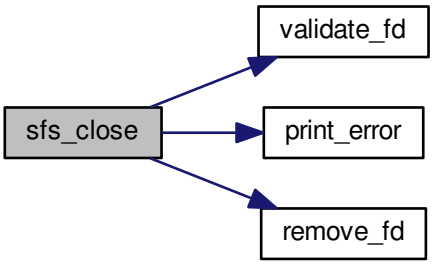
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

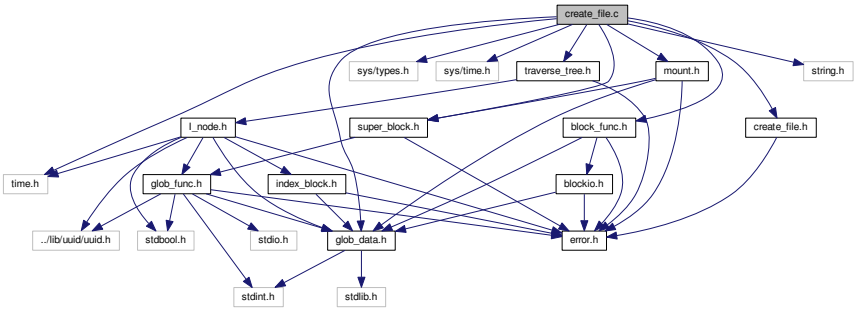



Here is the caller graph for this function:

## 5.7 create_file.c File Reference

```
#include "glob_data.h"
#include "super_block.h"
#include "traverse_tree.h"
#include "block_func.h"
#include "create_file.h"
#include "mount.h"
#include <string.h>
```

Include dependency graph for create_file.c:




### Functions

- int sfs_create (char *pathname, int type)

  *Creates a new file or directory at the path specified.*

### 5.7.1 Function Documentation

#### 5.7.1.1 int sfs_create ( char * *pathname,* int *type* )

Creates a new file or directory at the path specified.

Create a file with the pathname specified. If there is not already a file with the same pathname, the pathname must contain the full directory path. The parameter type specifies whether a regular file or a directory file should be created.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of file to create, it must be a full directory path. |
| *type* | The type of file to create, zero for a regular file, one for a directory. |

**Returns**

Returns an integer value. If the value > 0 the file creation was a success, and if the value <= 0 the file createion was unsuccessful.

**Exceptions**

| | |
|---|---|
| *INVALID_FILE_NAME* | If the specified file name is already in use. |

| | |
|---|---|
| *INVALID_FILE_TYPE* | If the file type specified is not a regular file (0) or a directory file (1). |
| *INVALID_PATH* | If the pathname specified does not already exist. |
| *INSUFFICIENT_DISK_SPACE* | If the length of the blocks to be written is greater than the amount of available blocks on disk. |

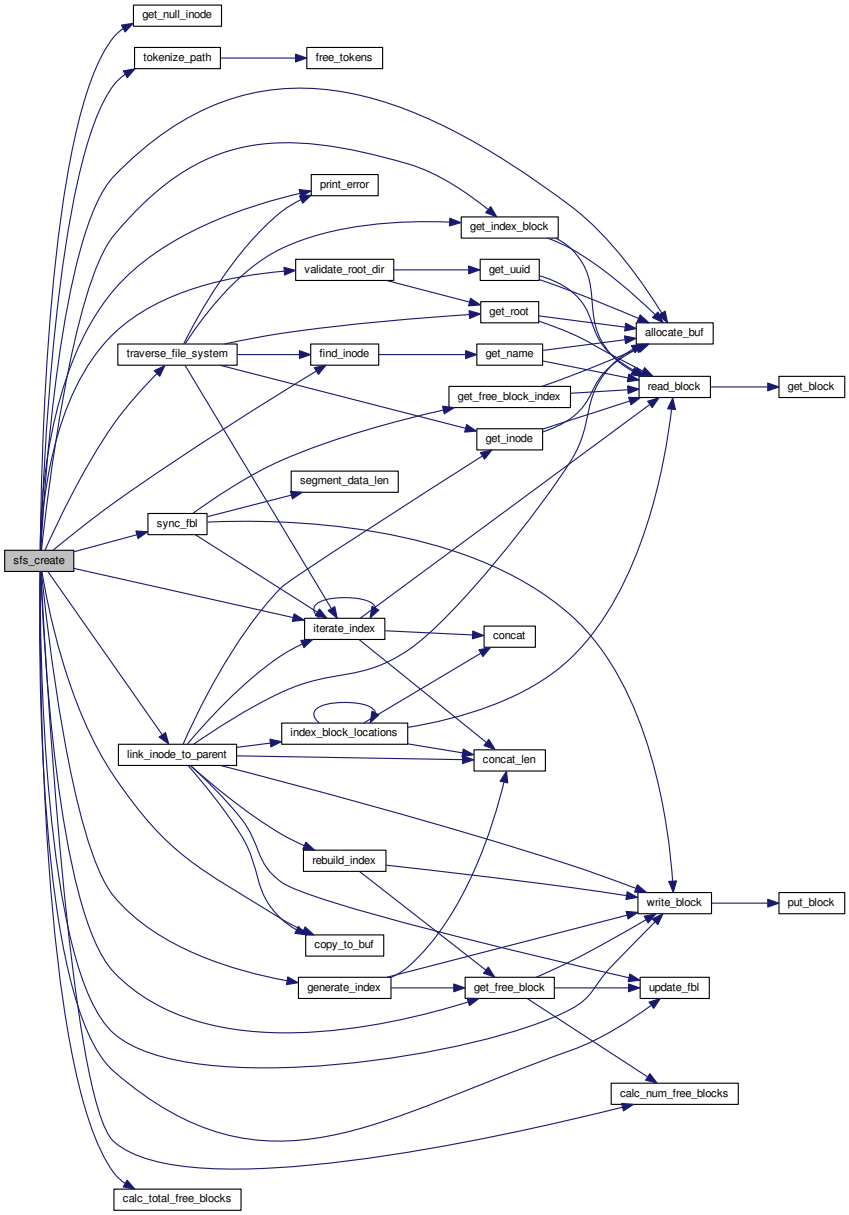
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

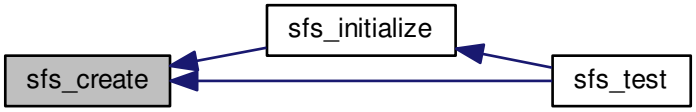
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



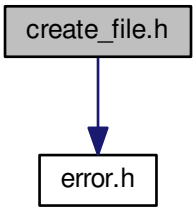
Here is the caller graph for this function:



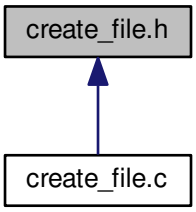

## 5.8 create_file.h File Reference

```
#include "error.h"
```

Include dependency graph for create_file.h:




This graph shows which files directly or indirectly include this file:



## Macros

- #define CREATE_SIZE 2

## Functions

- int sfs_create (char *pathname, int type)

  *Creates a new file or directory at the path specified.*

### 5.8.1 Macro Definition Documentation

#### 5.8.1.1 #define CREATE_SIZE 2

### 5.8.2 Function Documentation

#### 5.8.2.1 int sfs_create ( char * *pathname*, int *type* )

Creates a new file or directory at the path specified.

Create a file with the pathname specified. If there is not already a file with the same pathname, the pathname must contain the full directory path. The parameter type specifies whether a regular file or a directory file should be created.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of file to create, it must be a full directory path. |
| *type* | The type of file to create, zero for a regular file, one for a directory. |

**Returns**

Returns an integer value. If the value > 0 the file creation was a success, and if the value <= 0 the file createion was unsuccessful.

**Exceptions**

| | |
|---|---|
| *INVALID_FILE_NAME* | If the specified file name is already in use. |
| *INVALID_FILE_TYPE* | If the file type specified is not a regular file (0) or a directory file (1). |
| *INVALID_PATH* | If the pathname specified does not already exist. |
| *INSUFFICIENT_DISK_SPACE* | If the length of the blocks to be written is greater than the amount of available blocks on disk. |

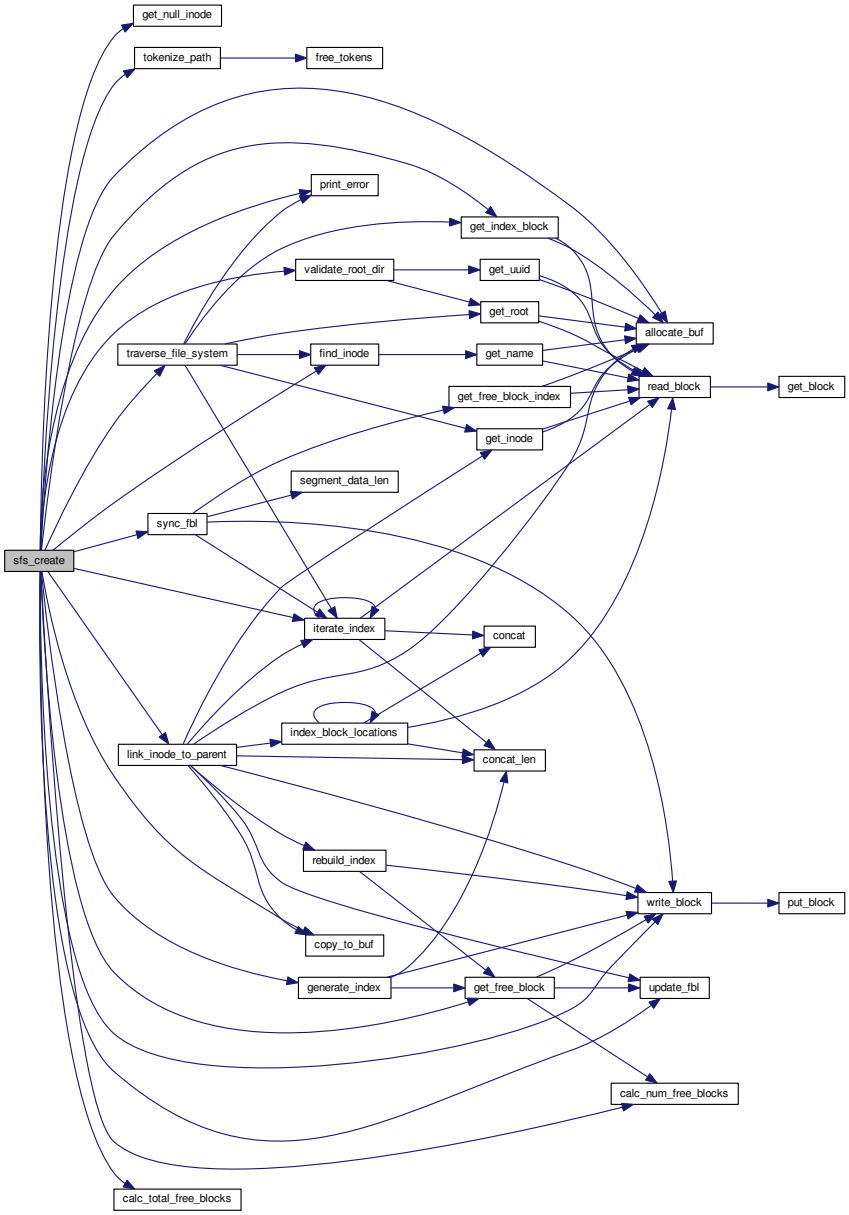
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

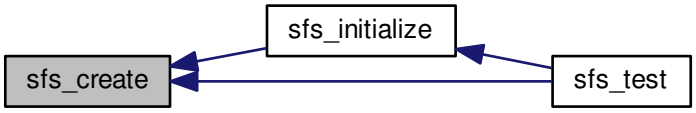
### Copyright

GNU General Public License V3

Here is the call graph for this function:



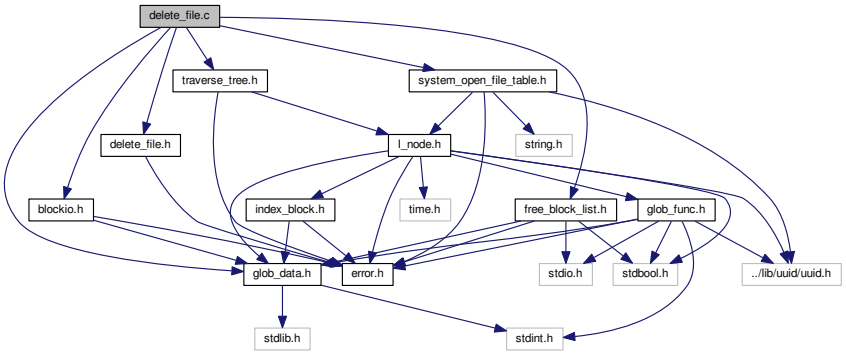
Here is the caller graph for this function:




## 5.9 delete_file.c File Reference

```
#include "glob_data.h"
#include "blockio.h"
#include "traverse_tree.h"
#include "free_block_list.h"
#include "system_open_file_table.h"
#include "delete_file.h"
```

Include dependency graph for delete_file.c:




### Functions

- int sfs_delete (char *pathname)

  *Delete a file with the pathname specified.*

### 5.9.1 Function Documentation

#### 5.9.1.1 int sfs_delete ( char * *pathname* )

Delete a file with the pathname specified.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of file to create, must be full directory path |

**Returns**

Returns an integer value, if the value is $>$ 0 then the file was deleted successfully. If the value is $<=$ 0 then the file failed to delete.

**Exceptions**

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the specified path does not already exist. |

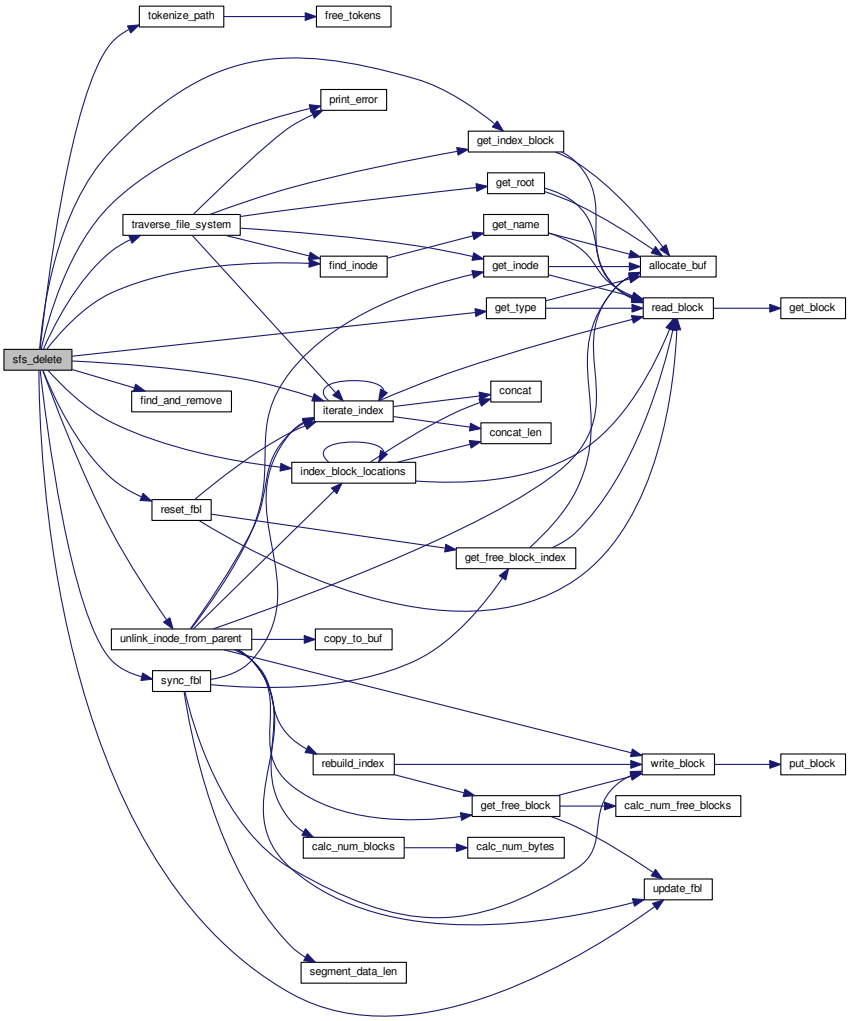
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

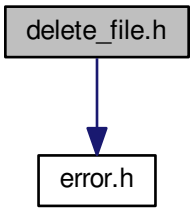



Here is the caller graph for this function:

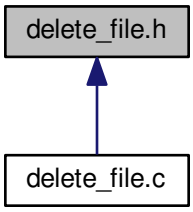
## 5.10 delete_file.h File Reference

```
#include "error.h"
```

Include dependency graph for delete_file.h:




This graph shows which files directly or indirectly include this file:




### Functions

- int sfs_delete (char *pathname)

  *Delete a file with the pathname specified.*

### 5.10.1 Function Documentation

#### 5.10.1.1 int sfs_delete ( char * *pathname* )

Delete a file with the pathname specified.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of file to create, must be full directory path |

**Returns**

Returns an integer value, if the value is $>$ 0 then the file was deleted successfully. If the value is $<=$ 0 then the file failed to delete.

**Exceptions**

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the specified path does not already exist. |

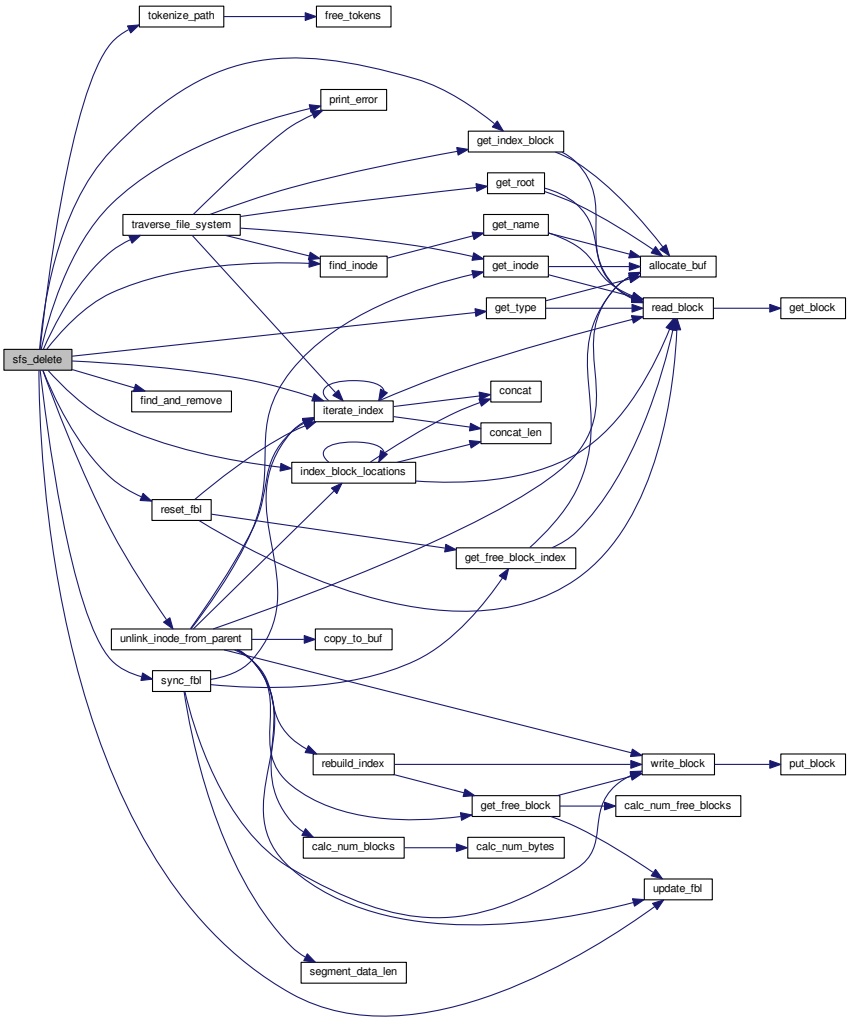
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

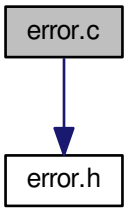



Here is the caller graph for this function:

## 5.11 error.c File Reference

```
#include "error.h"
```

Include dependency graph for error.c:




### Functions

- void print_error (error_code errorno)

  *Prints an error code.*

### 5.11.1 Function Documentation

#### 5.11.1.1 void print_error ( error_code *errorno* )

Prints an error code.

Outputs the specified error code to the console. Fatal errors will cause the application to terminate immediately to prevent file system or disk corruption.

**Parameters**

| | |
|---|---|
| *The* | specified error to output. |

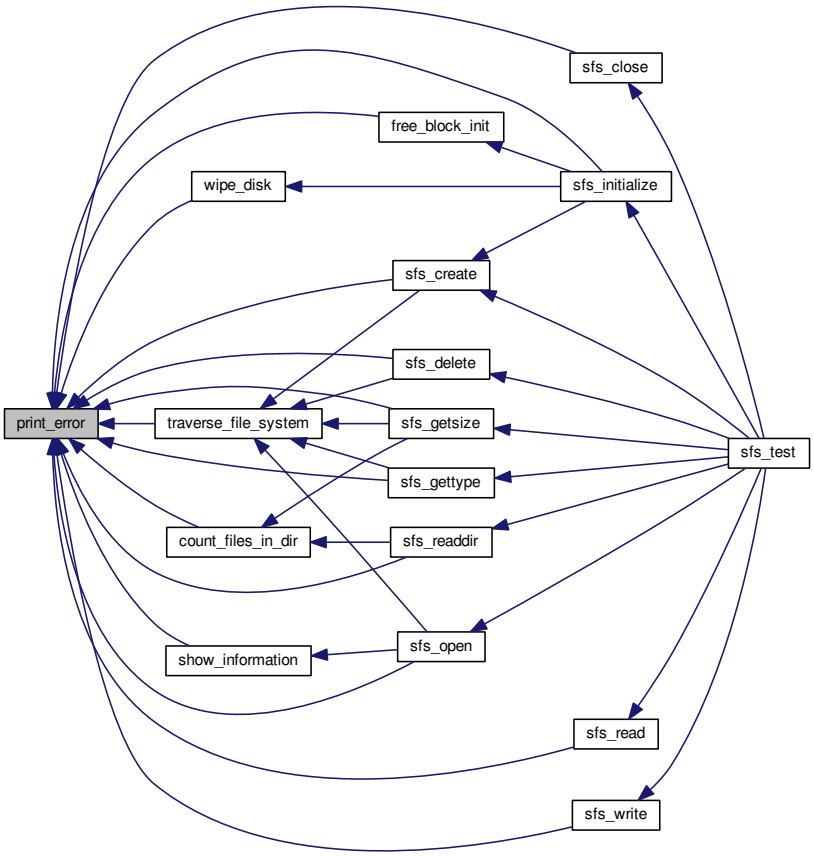
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




## 5.12 error.h File Reference

This graph shows which files directly or indirectly include this file:

## Enumerations

- enum error_code {
  SUCCESS, INVALID_PARAMETER, DISK_READ_ERROR, DISK_WRITE_ERROR,
  FILE_NOT_FOUND, INVALID_FILE_TYPE, INVALID_FILE_NAME, INVALID_FILE_DESCRIPTOR,
  INVALID_PATH, INVALID_PATH_LENGTH, INSUFFICIENT_DISK_SPACE, ERROR_UPDATING_SB,
  ERROR_UPDATING_FBL, ERROR_UPDATING_SWOFT, INDEX_ALLOCATION_ERROR, ERROR_BLOCK_-
  LINKAGE,
  ERROR_BUFFER_SEGMENTATION, PARENT_NOT_FOUND, DIRECTORY_HAS_CHILDREN, DIRECTORY-
  _TRAVERSED,
  DIRECTORY_EMPTY, FILE_EMPTY, FILE_PAST_EOF, UNKNOWN }

  *List of error codes with name definitions.*

## Functions

- void print_error (error_code errorno)

  *Prints an error code.*

### 5.12.1 Enumeration Type Documentation

#### 5.12.1.1 enum error_code

List of error codes with name definitions.

Contains a list of constant integers used as error codes with a human-readable name definitions.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

**Enumerator:**

***SUCCESS*** No error occurred, execution was successful.

***INVALID_PARAMETER*** An invalid parameter was given to the function.

***DISK_READ_ERROR*** A block of data could not be read from disk.

***DISK_WRITE_ERROR*** A block of data could not be written to disk.

***FILE_NOT_FOUND*** The specified file could not be found on disk.

***INVALID_FILE_TYPE*** The file type specified is invalid.

***INVALID_FILE_NAME*** The file name specified is invalid.

***INVALID_FILE_DESCRIPTOR*** The file descriptor specified is invalid or does not exist.

***INVALID_PATH*** The path specified is invalid.

***INVALID_PATH_LENGTH*** The path length given is too long, path tokens cannot be longer than 6 characters.

***INSUFFICIENT_DISK_SPACE*** The requested operation does not have sufficient blocks available on disk to complete execution.

***ERROR_UPDATING_SB*** Fatal error: The super block could not be updated. The file system or disk are corrupted.

***ERROR_UPDATING_FBL*** Fatal error: The free block list could not be updated. The file system or disk may be corrupted.

***ERROR_UPDATING_SWOFT*** The system-wide open file table cannot be updated. This may be indicative of memory corruption.

***INDEX_ALLOCATION_ERROR*** The given index block data structure is corrupted, invalid, or empty.

***ERROR_BLOCK_LINKAGE*** An error occurred concerning the linkage of an inode to an index block structure.

***ERROR_BUFFER_SEGMENTATION*** An error occurred segmenting a buffer into block-sized chunks.

***PARENT_NOT_FOUND*** An inode has become loose, and its parent cannot be found.

***DIRECTORY_HAS_CHILDREN*** An attempt was made to delete a directory which has children. Directories must be empty to be deleted.

***DIRECTORY_TRAVERSED*** A directory was fully traversed. Further attempts to read the contents of the directory will begin from the top of the structure.

***DIRECTORY_EMPTY*** An attempt was made to read the contents of an empty directory.

***FILE_EMPTY*** An attempt was made to perform operations on a file which contains no data.

***FILE_PAST_EOF*** An attempt was made to write to a file past the current length of the file, without appending. Appending is the only allowed way to increase the length of a file.

***UNKNOWN*** An unknown error occurred.

### 5.12.2 Function Documentation

#### 5.12.2.1 void print_error ( error_code *errorno* )

Prints an error code.

Outputs the specified error code to the console. Fatal errors will cause the application to terminate immediately to prevent file system or disk corruption.

**Parameters**

| | |
|---|---|
| *The* | specified error to output. |

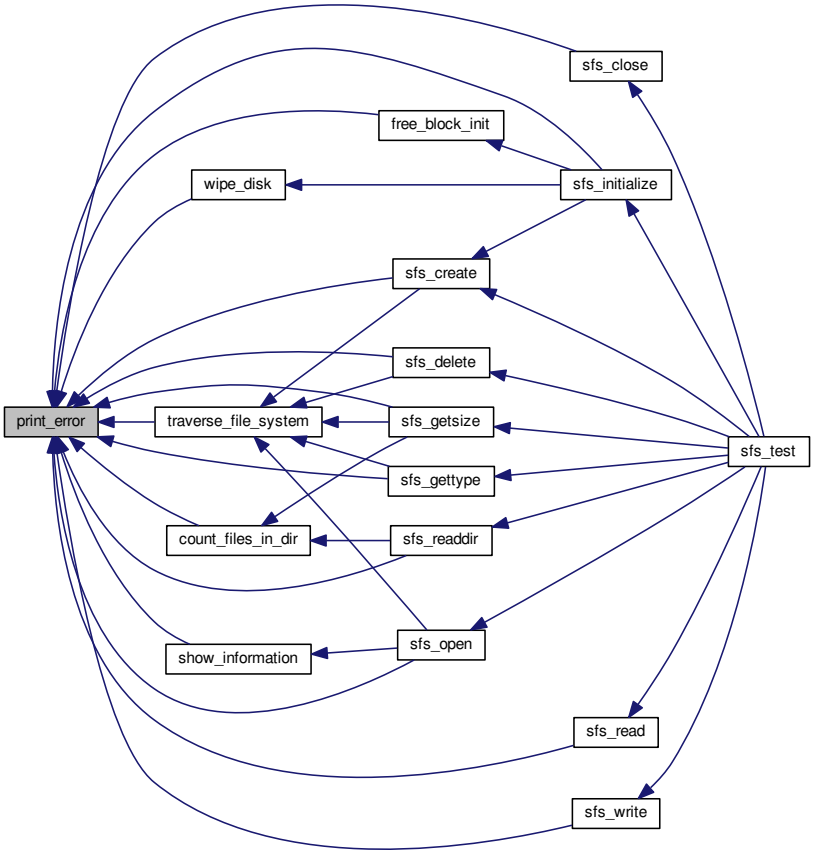
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

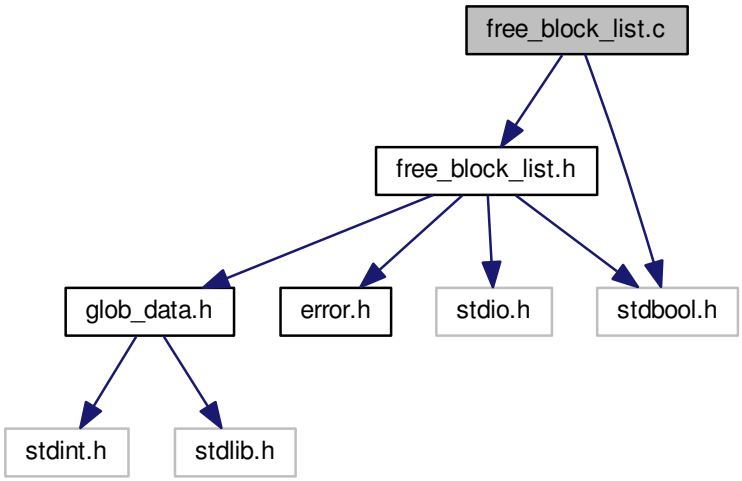
Here is the caller graph for this function:




## 5.13 free_block_list.c File Reference

```
#include "free_block_list.h"
#include <stdbool.h>
```

Include dependency graph for free_block_list.c:




## Functions

- free_block_list * get_free_block_list (void)

  *Obtains a handle to the free block list static instance, regardless of whether it is available in memory or on disk.*
- locations calc_total_free_blocks (void)

  *Used to calculate the total amount of available disk space.*
- locations calc_num_free_blocks (uint32_t num_blocks)

  *Used to calculate whether there is enough disk space before starting to create a file on disk.*
- uint32_t get_free_block (void)

  *Finds the next free block in the static free block list, marks it as used and returns the location of the block.*
- free_block_list * update_fbl (locations used, locations free)

  *Accessor method for the free block list static instance in memory.*
- free_block_list * sync_fbl (void)

  *Synchronizes the free block static instance in memory by writing it to disk.*
- free_block_list * reset_fbl (void)

  *Overwrites the free block list in memory with the most recent version on disk.*
- free_block_list * wipe_fbl (void)

  *Reinitializes the free block list in memory.*

### 5.13.1 Function Documentation

#### 5.13.1.1 locations calc_num_free_blocks ( uint32_t *num_blocks* )

Used to calculate whether there is enough disk space before starting to create a file on disk.

Finds the specified number of free blocks in the static instance of the free block list in memory. It returns NULL if there are not enough free blocks, otherwise it returns an array of the free locations.

**Parameters**

| | |
|---|---|
| *num_blocks* | The number of free blocks needed. |

**Returns**

Returns an array of the free locations, or NULL if there are not enough free blocks.

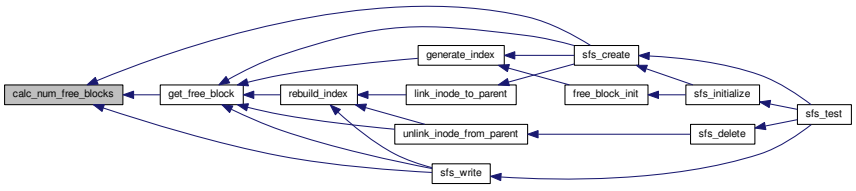
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.13.1.2 locations calc_total_free_blocks ( void )

Used to calculate the total amount of available disk space.

Finds all of the free blocks that exist in the static instance of the free block list in memory and returns them as a locations object.

**Returns**

Returns an array containing all of the free locations.

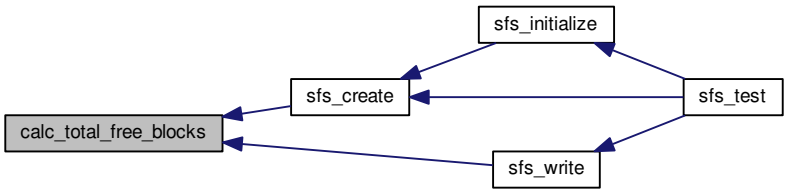
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.13.1.3 uint32_t get_free_block ( void )

Finds the next free block in the static free block list, marks it as used and returns the location of the block.

This function finds a single free block and then marks that block as used in the free block list, it ensures that the block on disk is empty (all NULL) before returning the block location. It essentially operates as a facade for the calc_free_blocks function and the update_fbl function when you only want to get one block at a time. If there are no free blocks available, it will return 0 (the superblock's location), which is always used by the superblock.

**Returns**

The location of a free block. Return 0 if no free blocks available.

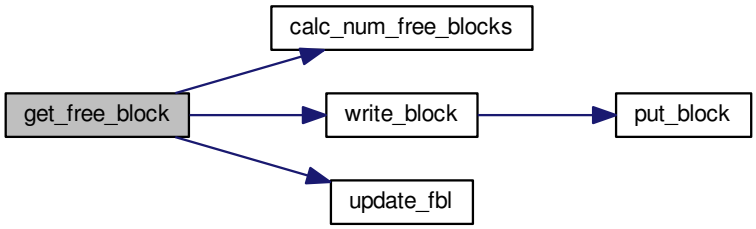
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

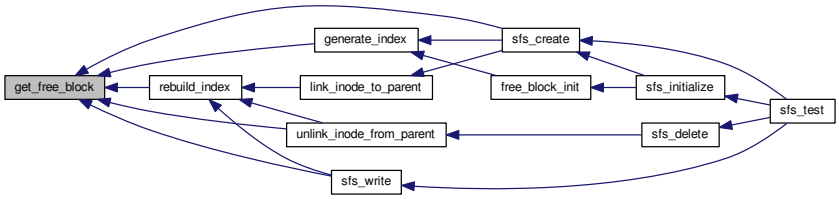
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.13.1.4 free_block_list* get_free_block_list ( void )

Obtains a handle to the free block list static instance, regardless of whether it is available in memory or on disk.

Gets the free block list independently from where it is located. If the free block list does not already exist in memory (file system has just started) it will read the free block list from disk using the index provided by the super block and set the static instance of the free block list in memory. Then, return the pointer to the static instance in memory. If an error occurs, a NULL pointer is returned.

**Returns**

A pointer to the static instance of free block list in memory, NULL if an error occurred.

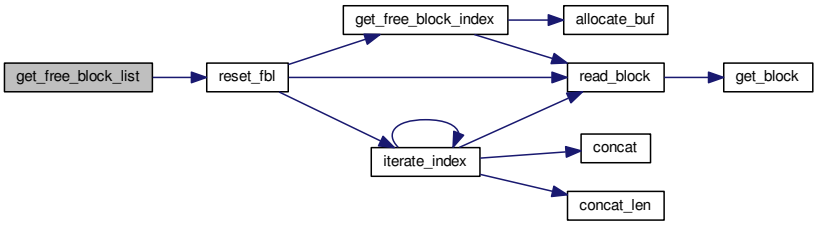
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

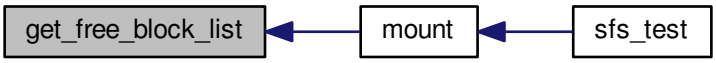
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.13.1.5 free_block_list* reset_fbl ( void )

Overwrites the free block list in memory with the most recent version on disk.

Resets the free block list in memory by loading the free blocks list from disk and overwriting the static instance of the free block list in memory. This function is used to refresh the state of the free block list in memory if an error occurs.

**Returns**

Returns a pointer to the static instance of free block list in memory, NULL if an error occurred.

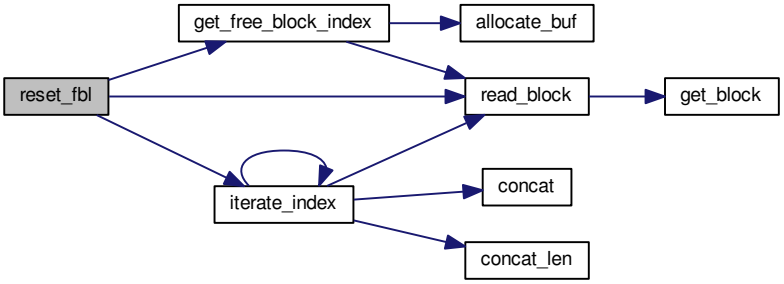
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

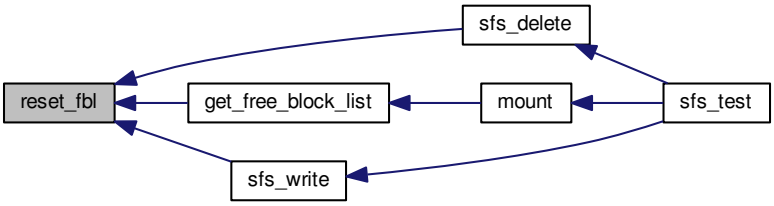
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.13.1.6 free_block_list* sync_fbl ( void )

Synchronizes the free block static instance in memory by writing it to disk.

Writes the static free block list in memory to disk. This synchronizes the static free block list in memory with the contents on disk, making the current state of the free blocks on disk permanent.

**Returns**

Returns a pointer to the static instance of free block list in memory, NULL if an error occurred synchronizing the free block list to disk.

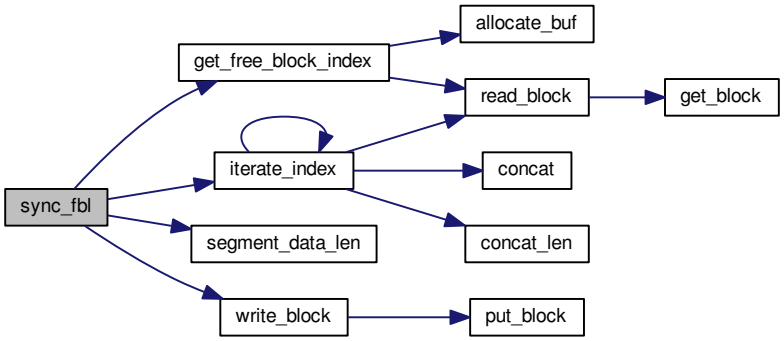
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

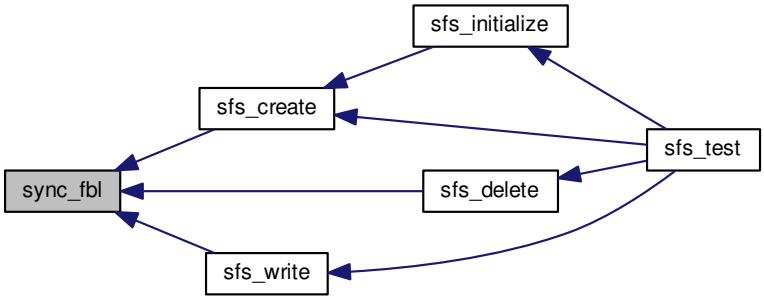
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.13.1.7 free_block_list* update_fbl ( locations *used*, locations *free* )

Accessor method for the free block list static instance in memory.

The update FBL method updates the static FBL entry in memory. It will take an array of all the locations to mark as used as the first argument, and an array of all the locations to mark as unused as the second argument. If the arguments used or free are NULL then no blocks are marked for that type.

**Precondition**

Parameters used and free MUST be NULL terminated arrays of locations.

**Parameters**

| | |
|---|---|
| *used* | A NULL terminated array of locations to mark as used in the free block list, if it is NULL then no locations are marked as used. |
| *free* | A NULL terminated array of locations to mark as free in the free block list, if it is NULL then no locations are marked as free. |

**Returns**

Returns a pointer to the static instance of free block list in memory, NULL if an error occurred.

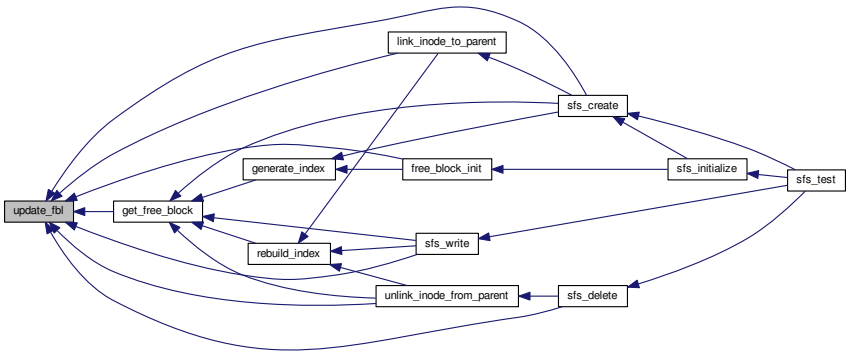
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.13.1.8 free_block_list* wipe_fbl ( void )

Reinitializes the free block list in memory.

Wipes the contents of the free block list in memory, which sets all of the locations in the free block list in memory as free. This function is used to reset the free block list in memory after initializing a new disk.

**Returns**

Returns a pointer to the free block list, NULL if an error occurred.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

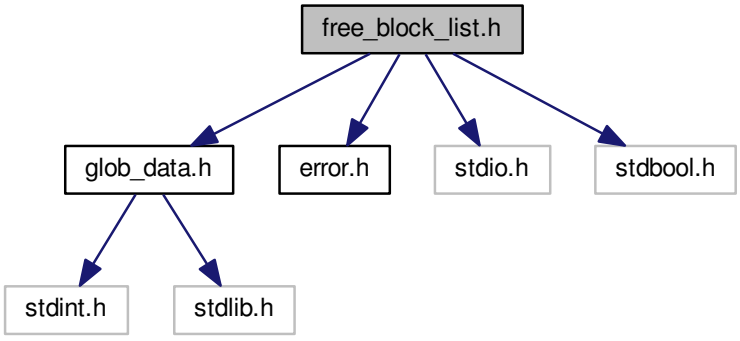
GNU General Public License V3

Here is the caller graph for this function:

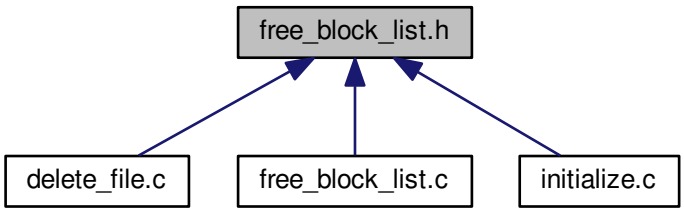

## 5.14 free_block_list.h File Reference

```
#include "glob_data.h"
#include "error.h"
#include <stdio.h>
#include <stdbool.h>
```

Include dependency graph for free_block_list.h:



This graph shows which files directly or indirectly include this file:




## Data Structures

- struct free_block_list

  *The free_block_list is an array containing the free blocks on disk. Elements are marked as free (false), or used (true). The super block always points to the first index block of the free_block_list.*

## Functions

- free_block_list * get_free_block_list (void)

  *Obtains a handle to the free block list static instance, regardless of whether it is available in memory or on disk.*
- locations calc_total_free_blocks (void)

  *Used to calculate the total amount of available disk space.*
- locations calc_num_free_blocks (uint32_t num_blocks)

  *Used to calculate whether there is enough disk space before starting to create a file on disk.*
- uint32_t get_free_block (void)

  *Finds the next free block in the static free block list, marks it as used and returns the location of the block.*
- free_block_list * update_fbl (locations used, locations free)

  *Accessor method for the free block list static instance in memory.*
- free_block_list * sync_fbl (void)

  *Synchronizes the free block static instance in memory by writing it to disk.*
- free_block_list * reset_fbl (void)

  *Overwrites the free block list in memory with the most recent version on disk.*
- free_block_list * wipe_fbl (void)

  *Reinitializes the free block list in memory.*

### 5.14.1 Function Documentation

#### 5.14.1.1 locations calc_num_free_blocks ( uint32_t *num_blocks* )

Used to calculate whether there is enough disk space before starting to create a file on disk.

Finds the specified number of free blocks in the static instance of the free block list in memory. It returns NULL if there are not enough free blocks, otherwise it returns an array of the free locations.

**Parameters**

| | |
|---|---|
| *num_blocks* | The number of free blocks needed. |

**Returns**

Returns an array of the free locations, or NULL if there are not enough free blocks.

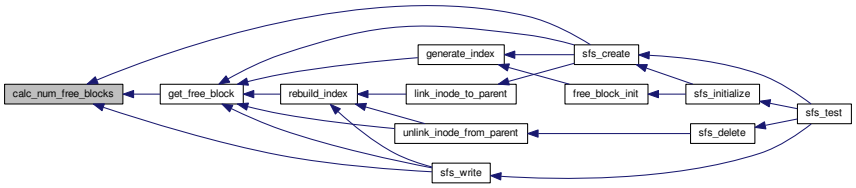
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.14.1.2 locations calc_total_free_blocks ( void )

Used to calculate the total amount of available disk space.

Finds all of the free blocks that exist in the static instance of the free block list in memory and returns them as a locations object.

**Returns**

Returns an array containing all of the free locations.

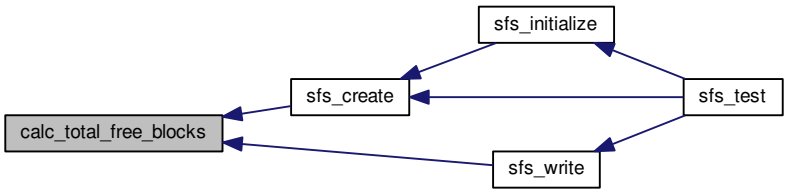
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.14.1.3 uint32_t get_free_block ( void )

Finds the next free block in the static free block list, marks it as used and returns the location of the block.

This function finds a single free block and then marks that block as used in the free block list, it ensures that the block on disk is empty (all NULL) before returning the block location. It essentially operates as a facade for the calc_free_blocks function and the update_fbl function when you only want to get one block at a time. If there are no free blocks available, it will return 0 (the superblock's location), which is always used by the superblock.

**Returns**

The location of a free block. Return 0 if no free blocks available.

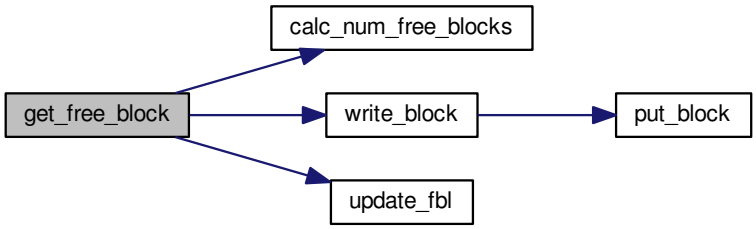
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

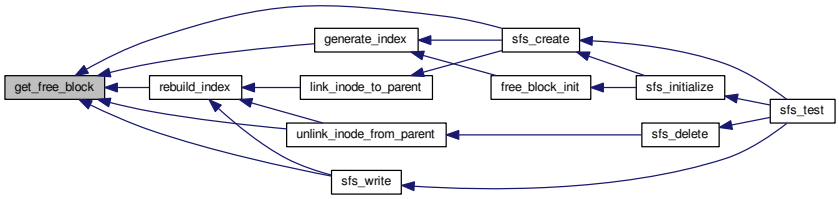
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.14.1.4 free_block_list* get_free_block_list ( void )

Obtains a handle to the free block list static instance, regardless of whether it is available in memory or on disk.

Gets the free block list independently from where it is located. If the free block list does not already exist in memory (file system has just started) it will read the free block list from disk using the index provided by the super block and set the static instance of the free block list in memory. Then, return the pointer to the static instance in memory. If an error occurs, a NULL pointer is returned.

**Returns**

A pointer to the static instance of free block list in memory, NULL if an error occurred.

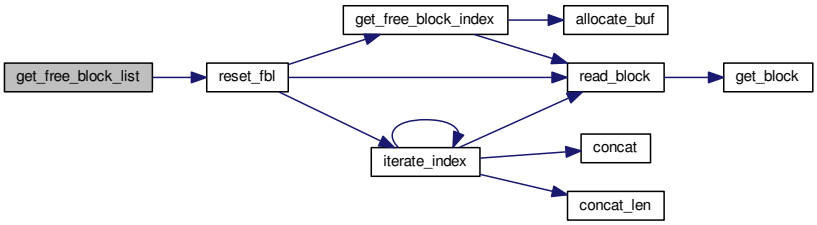
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

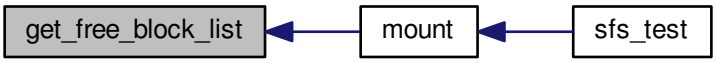
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.14.1.5 free_block_list* reset_fbl ( void )

Overwrites the free block list in memory with the most recent version on disk.

Resets the free block list in memory by loading the free blocks list from disk and overwriting the static instance of the free block list in memory. This function is used to refresh the state of the free block list in memory if an error occurs.

**Returns**

Returns a pointer to the static instance of free block list in memory, NULL if an error occurred.

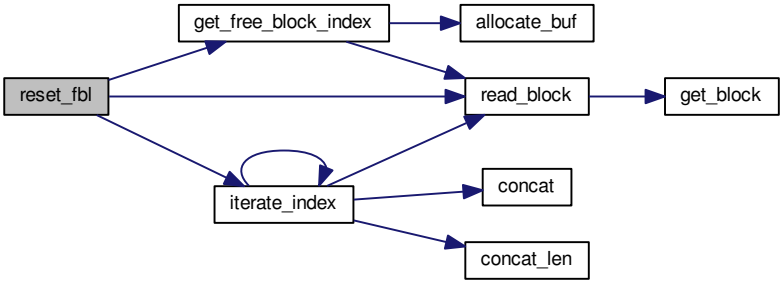
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

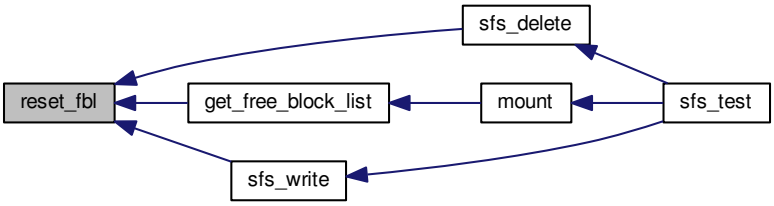
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.14.1.6 free_block_list* sync_fbl ( void )

Synchronizes the free block static instance in memory by writing it to disk.

Writes the static free block list in memory to disk. This synchronizes the static free block list in memory with the contents on disk, making the current state of the free blocks on disk permanent.

**Returns**

Returns a pointer to the static instance of free block list in memory, NULL if an error occurred synchronizing the free block list to disk.

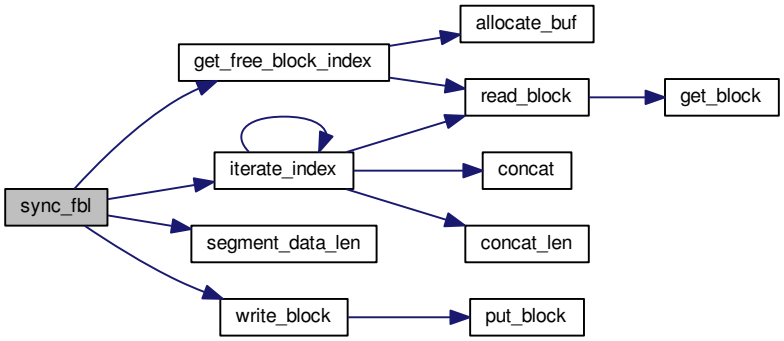
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

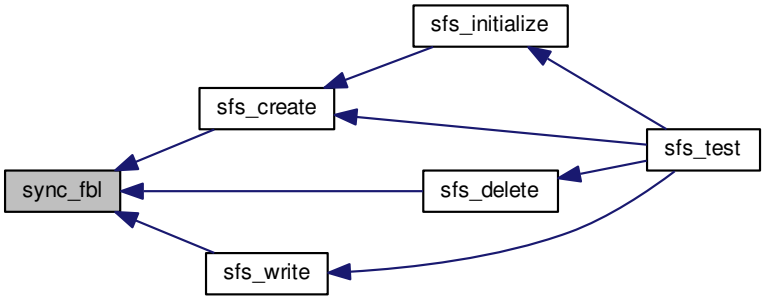
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.14.1.7 free_block_list* update_fbl ( locations *used*, locations *free* )

Accessor method for the free block list static instance in memory.

The update FBL method updates the static FBL entry in memory. It will take an array of all the locations to mark as used as the first argument, and an array of all the locations to mark as unused as the second argument. If the arguments used or free are NULL then no blocks are marked for that type.

**Precondition**

Parameters used and free MUST be NULL terminated arrays of locations.

**Parameters**

| | |
|---|---|
| *used* | A NULL terminated array of locations to mark as used in the free block list, if it is NULL then no locations are marked as used. |
| *free* | A NULL terminated array of locations to mark as free in the free block list, if it is NULL then no locations are marked as free. |

**Returns**

Returns a pointer to the static instance of free block list in memory, NULL if an error occurred.

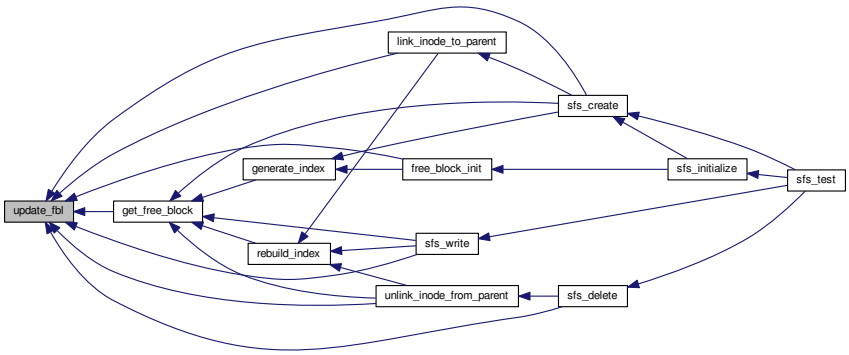
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.14.1.8 free_block_list* wipe_fbl ( void )

Reinitializes the free block list in memory.

Wipes the contents of the free block list in memory, which sets all of the locations in the free block list in memory as free. This function is used to reset the free block list in memory after initializing a new disk.

**Returns**

Returns a pointer to the free block list, NULL if an error occurred.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

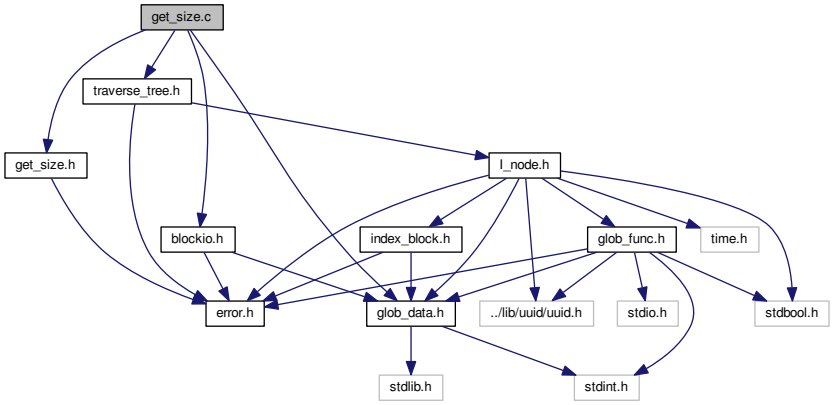
GNU General Public License V3

Here is the caller graph for this function:


## 5.15 get_size.c File Reference

```
#include "glob_data.h"
#include "blockio.h"
#include "traverse_tree.h"
#include "get_size.h"
```

Include dependency graph for get_size.c:




### Functions

- int sfs_getsize (char *pathname)

  *Get the size (in blocks) of the file with the pathname specified.*

### 5.15.1 Function Documentation

#### 5.15.1.1 int sfs_getsize ( char * *pathname* )

Get the size (in blocks) of the file with the pathname specified.

Traverses the index block structure of a given file, reads all the locations inside, and the number of blocks is returned. If the file is a data file, the size of the file is returned. If the file is a directory, the number of elements within the directory will be returned.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of the file or directory, must be full directory path. Relative paths are not accepted. |

**Returns**

Returns the size of the given file. If the value $< 0$ then an error occurred.

**Exceptions**

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the given pathname does not exist. |
| *INVALID_PATH* | If if the path specified is invalid, incomplete, or does not exist. |

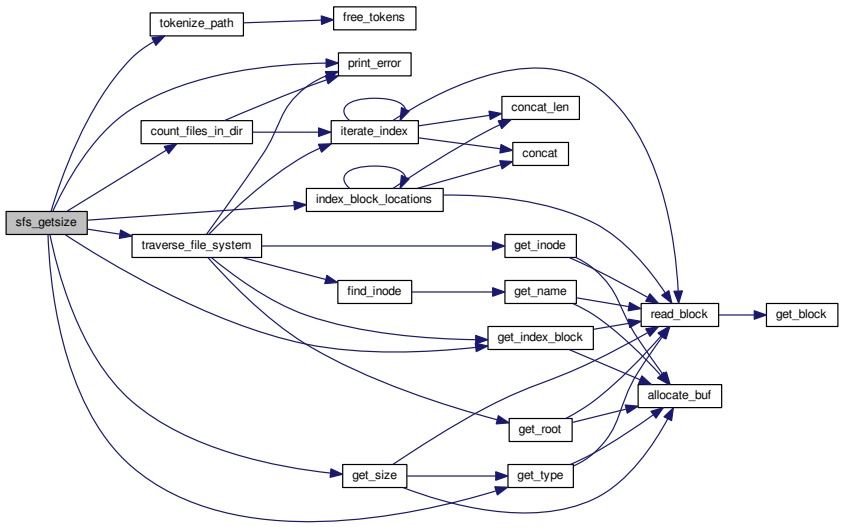
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

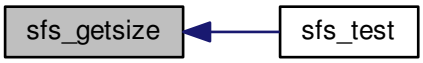
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



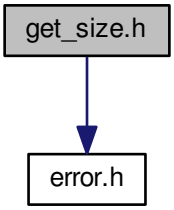
Here is the caller graph for this function:



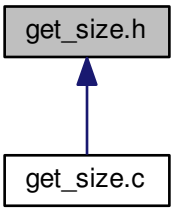

## 5.16 get_size.h File Reference

```
#include "error.h"
```

Include dependency graph for get_size.h:




This graph shows which files directly or indirectly include this file:



## Functions

- int sfs_getsize (char *pathname)

  *Get the size (in blocks) of the file with the pathname specified.*

### 5.16.1 Function Documentation

#### 5.16.1.1 int sfs_getsize ( char * *pathname* )

Get the size (in blocks) of the file with the pathname specified.

Traverses the index block structure of a given file, reads all the locations inside, and the number of blocks is returned. If the file is a data file, the size of the file is returned. If the file is a directory, the number of elements within the directory will be returned.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of the file or directory, must be full directory path. Relative paths are not accepted. |

**Returns**

Returns the size of the given file. If the value $< 0$ then an error occurred.

**Exceptions**

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the given pathname does not exist. |
| *INVALID_PATH* | If if the path specified is invalid, incomplete, or does not exist. |

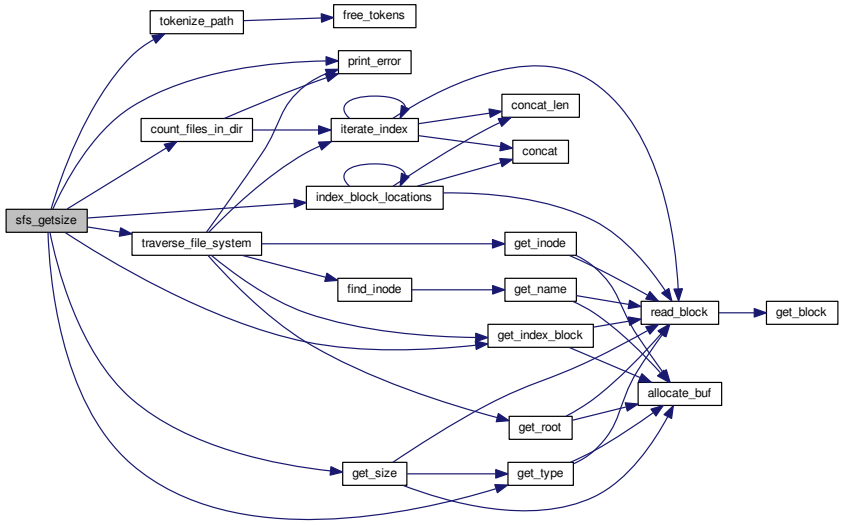
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

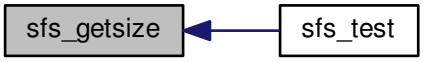
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



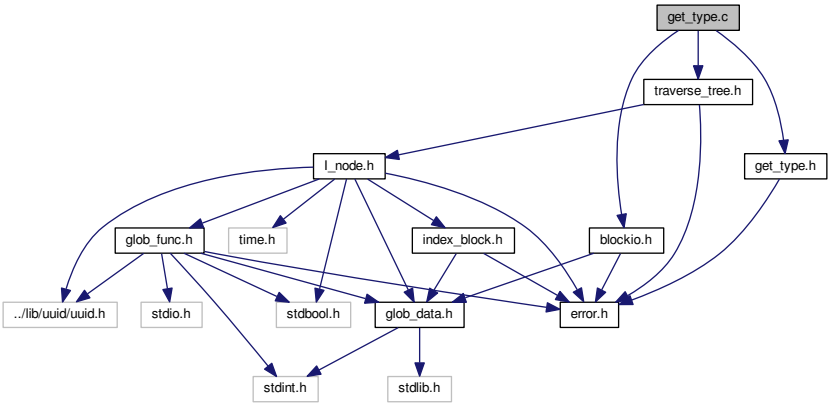

Here is the caller graph for this function:




## 5.17 get_type.c File Reference

```
#include "blockio.h"
#include "traverse_tree.h"
#include "get_type.h"
```

Include dependency graph for get_type.c:




## Functions

- int sfs_gettype (char *pathname)

  *Get the type of the file with the pathname specified.*

### 5.17.1 Function Documentation

#### 5.17.1.1 int sfs_gettype ( char * *pathname* )

Get the type of the file with the pathname specified.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of file to create, must be full directory path. |

**Returns**

Returns the file type of the given file. If the file type >= 0 then the file type retrieval was a success.If the file type is 0, or 1 then it is a file, or directory respectively. If the file type > 1 then the file type is not known. If the file type < 0 then the file type retrieval failed.

**Exceptions**

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the specified path does not exist. |
| *INVALID_PATH* | If the specified path is invalid. |

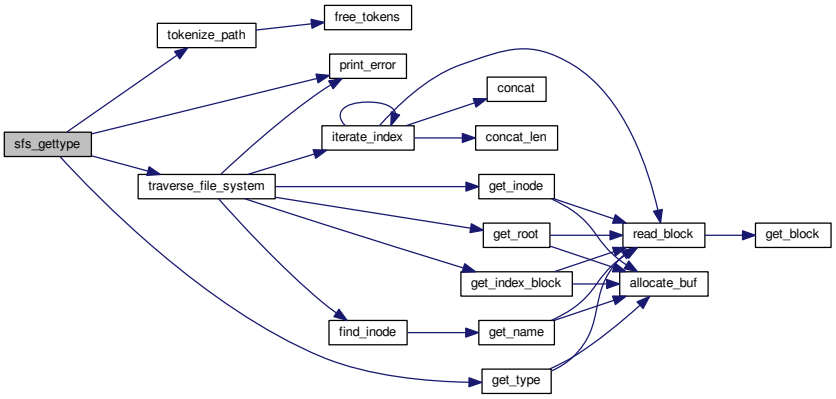
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

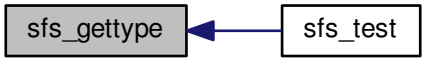
### Copyright

GNU General Public License V3

Here is the call graph for this function:



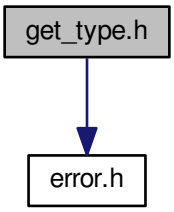

Here is the caller graph for this function:



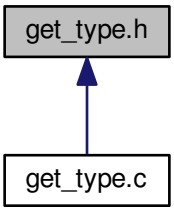
## 5.18 get_type.h File Reference

```
#include "error.h"
```

Include dependency graph for get_type.h:




This graph shows which files directly or indirectly include this file:




### Functions

- int sfs_gettype (char *pathname)

  *Get the type of the file with the pathname specified.*

### 5.18.1 Function Documentation

#### 5.18.1.1 int sfs_gettype ( char * *pathname* )

Get the type of the file with the pathname specified.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname of file to create, must be full directory path. |

### Returns

Returns the file type of the given file. If the file type >= 0 then the file type retrieval was a success.If the file type is 0, or 1 then it is a file, or directory respectively. If the file type > 1 then the file type is not known. If the file type < 0 then the file type retrieval failed.

### Exceptions

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the specified path does not exist. |
| *INVALID_PATH* | If the specified path is invalid. |

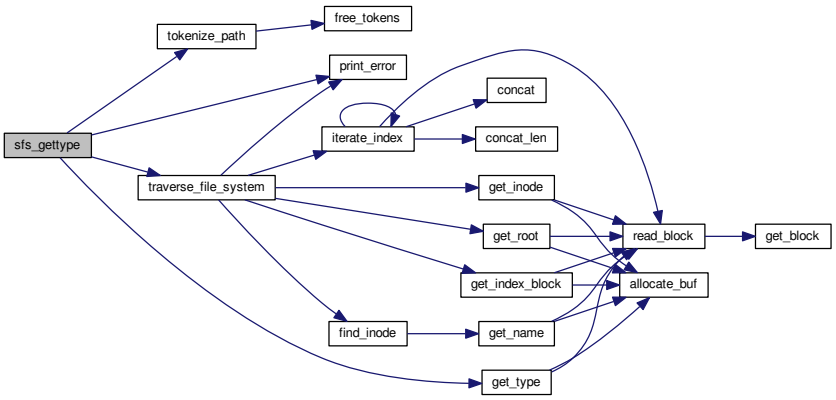
### Author

Daniel Smullen
Jonathan Gillett
Joseph Heron

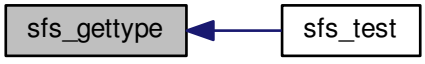
### Copyright

GNU General Public License V3

Here is the call graph for this function:



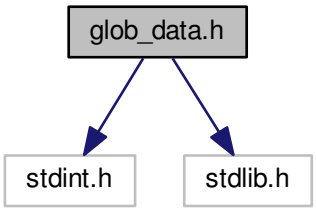

Here is the caller graph for this function:



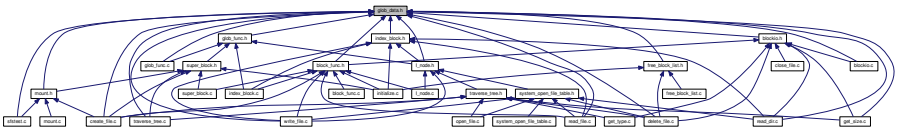
## 5.19 glob_data.h File Reference

```
#include <stdint.h>
#include <stdlib.h>
```

Include dependency graph for glob_data.h:




This graph shows which files directly or indirectly include this file:



### Macros

- #define MAX_IO_LENGTH 1024

  *Defines the maximum length of any input or output buffer in the test interface for the file system.*
- #define IO_BUF_FORMAT "%512s"

  *Defines the format of the input/output buffer.*
- #define MAX_INPUT_LENGTH 512

  *This is the maximum length of input strings (e.g., file names) read from the standard input. This should be large enough for most purposes. the format definition is necessary because macro substitutions do not take place within quoted strings.*
- #define INPUT_BUF_FORMAT "%1024s"

  *Defines the format of the input buffer.*
- #define UNIT_TESTING

  *Define if we are doing unit testing instead of executing sfs_test.*
- #define BLKSIZE 128

  *The size of blocks on the simulated disk.*
- #define NUMBLKS 512

  *The number of blocks on simulated disk.*
- #define NUMOFL 32

  *The number of files that can be open at once. Determines the size of the system-wide open file table.*
- #define MAX_NAME_LEN 7

*The maximum length of a component in a pathname including the NULL terminator.*

- #define SUPER_BLOCK 0

  *Determines the default location of the super block on disk.*

- #define FBL_INDEX 1

  *Determines the default location of the free block list's index data structure.*

### Typedefs

- typedef uint8_t byte

  *A byte is defined as an 8-bit unsigned integer.*

- typedef byte block [BLKSIZE]

  *A block is defined as an array of bytes, with each element specified as the size of one block.*

- typedef uint32_t * locations

  *A NULL terminated array of locations for blocks on disk. Locations' indices are represented by 32-bit unsigned integers.*

### Variables

- uint32_t FBL_DATA_SIZE

  *The size of the free block list data blocks, not including the overhead of the size of each index block needed to index the free block list data blocks.*

- uint32_t FBL_TOTAL_SIZE

  *The total size of the free block list data blocks, including the overhead of the size of each index block needed to index the free block list data blocks.*

- uint32_t ROOT

  *The default location for the root directory's inode.*

## 5.19.1 Macro Definition Documentation

#### 5.19.1.1 #define BLKSIZE 128

The size of blocks on the simulated disk.

BLKSIZE

#### 5.19.1.2 #define FBL_INDEX 1

Determines the default location of the free block list's index data structure.

FBL_INDEX

#### 5.19.1.3 #define INPUT_BUF_FORMAT "%1024s"

Defines the format of the input buffer.

INPUT_BUF_FORMAT

#### 5.19.1.4 #define IO_BUF_FORMAT "%512s"

Defines the format of the input/output buffer.

IO_BUF_FORMAT

#### 5.19.1.5 #define MAX_INPUT_LENGTH 512

This is the maximum length of input strings (e.g., file names) read from the standard input. This should be large enough for most purposes. the format definition is necessary because macro substitutions do not take place within quoted strings.

MAX_INPUT_LENGTH

#### 5.19.1.6 #define MAX_IO_LENGTH 1024

Defines the maximum length of any input or output buffer in the test interface for the file system.

MAX_IO_LENGTH

This is the maximum number of bytes that can be read from or written to a file with a single file system call using this program. Since files are limited to 512 bytes length, this should be sufficient. The format definition is necessary because macro substitutions do not take place within quoted strings. The maximum length is defined as 1024 bytes.

#### 5.19.1.7 #define MAX_NAME_LEN 7

The maximum length of a component in a pathname including the NULL terminator.

MAX_NAME_LEN

#### 5.19.1.8 #define NUMBLKS 512

The number of blocks on simulated disk.

NUMBLKS

#### 5.19.1.9 #define NUMOFL 32

The number of files that can be open at once. Determines the size of the system-wide open file table.

NUMOFL

#### 5.19.1.10 #define SUPER_BLOCK 0

Determines the default location of the super block on disk.

SUPER_BLOCK

#### 5.19.1.11 #define UNIT_TESTING

Define if we are doing unit testing instead of executing sfs_test.

UNIT_TESTING

**Deprecated** This definition is no longer used since the unit test suite was no longer used and deprecated once the high-level file system functions were implemented.

### 5.19.2 Typedef Documentation

#### 5.19.2.1 block

A block is defined as an array of bytes, with each element specified as the size of one block.

**See Also**

BLKSIZE

#### 5.19.2.2 byte

A byte is defined as an 8-bit unsigned integer.

#### 5.19.2.3 locations

A NULL terminated array of locations for blocks on disk. Locations' indices are represented by 32-bit unsigned integers.

### 5.19.3 Variable Documentation

#### 5.19.3.1 FBL_DATA_SIZE

The size of the free block list data blocks, not including the overhead of the size of each index block needed to index the free block list data blocks.

#### 5.19.3.2 FBL_TOTAL_SIZE

The total size of the free block list data blocks, including the overhead of the size of each index block needed to index the free block list data blocks.

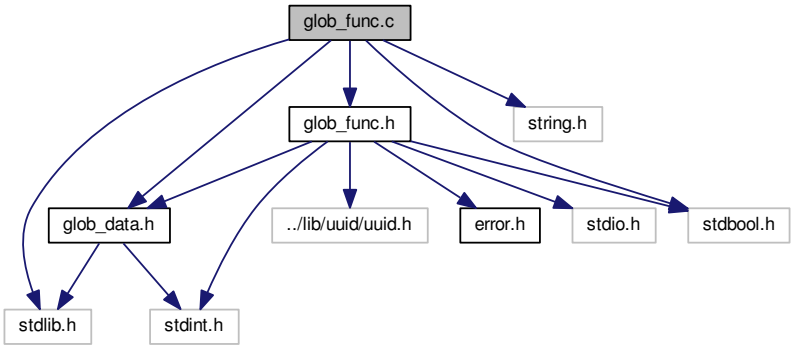
#### 5.19.3.3 ROOT

The default location for the root directory's inode.

## 5.20 glob_func.c File Reference

```
#include "glob_data.h"
#include "glob_func.h"
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
```

Include dependency graph for glob_func.c:




## Functions

- byte * allocate_buf (byte *buf, uint32_t size)

  *Allocate a buffer to write to memory of a given size.*
- byte * copy_to_buf (byte *buf1, byte *buf2, uint32_t size1, uint32_t size2)

  *Copy the data from one buffer to another, given the size of both.*
- uint32_t calc_num_bytes (byte *buf)

  *Calculate the number of bytes in a byte buffer.*
- uint32_t calc_num_blocks (byte *buf)

  *Calculates the number of blocks needed to store data.*
- void * concat (void *src_1, void *src_2, uint32_t size)

  *Concatenate two arrays.*
- void * concat_len (void *src_1, void *src_2, uint32_t size, uint32_t len)

  *Concatenate two byte arrays.*
- char ** tokenize_path (char *pathname)

  *Tokenize a path into an array of tokens.*
- bool free_tokens (char **tokens)

  *Frees the memory allocated for the tokens.*

### 5.20.1 Function Documentation

#### 5.20.1.1 byte* allocate_buf ( byte * *buf*, uint32_t *size* )

Allocate a buffer to write to memory of a given size.

**Parameters**

| | |
|---|---|
| *buf* | The buffer to allocate the space for. |
| *size* | The size for the new buffer's elements. |

**Returns**

Returns a pointer to the new buffer.

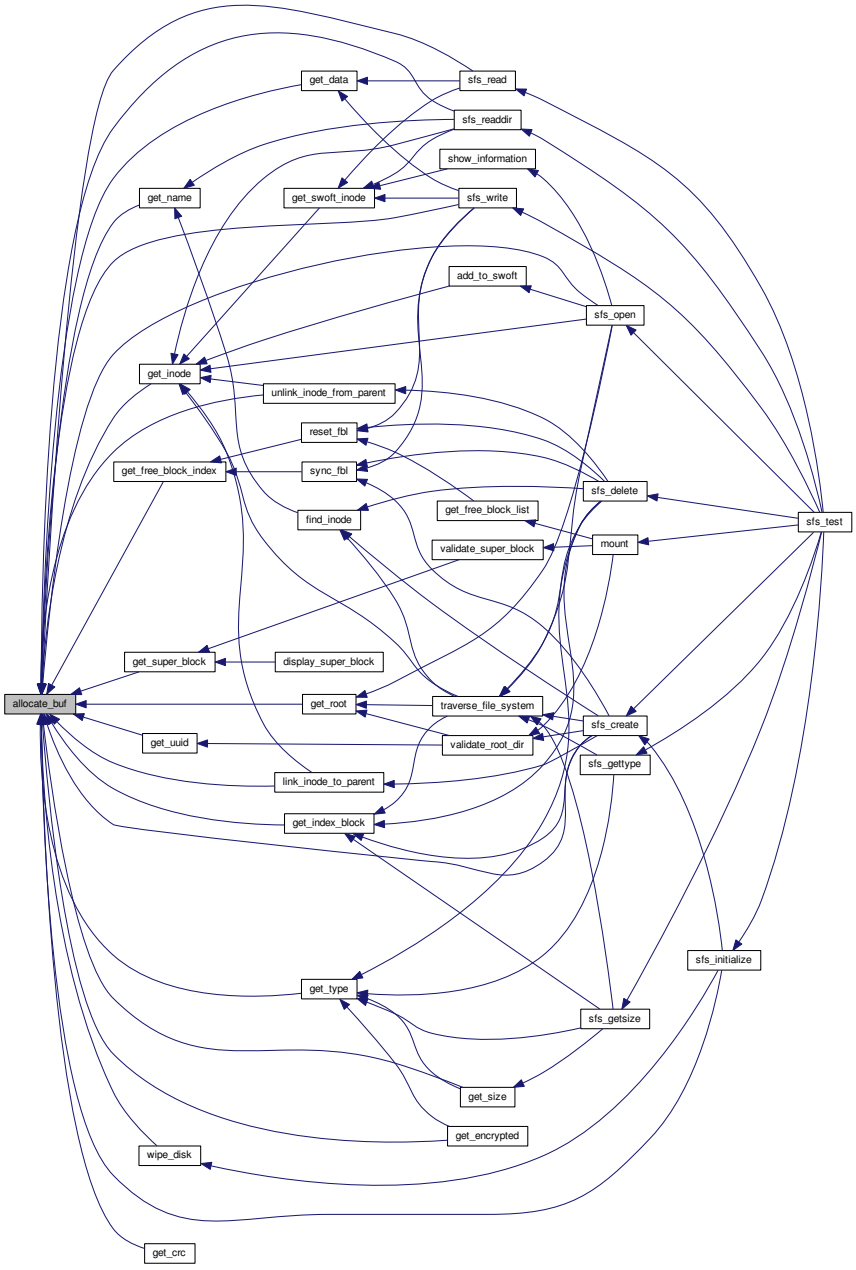
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.20.1.2 uint32_t calc_num_blocks ( byte * *buf* )

Calculates the number of blocks needed to store data.

Helper function which calculates the number of blocks needed to store the data, if an error occurs 0 is returned. Reads up to MAX_IO_LENGTH + 1, if the buffer is still not terminated then the IO buffer is larger than the maximum specified by the file system, an error occurs and 0 is returned.

**Precondition**

The buf parameter must be NULL terminated.

**Parameters**

| | |
|---|---|
| *buf* | A NULL terminated buffer. |

**Returns**

Returns the number of blocks needed to store the given data.

**Author**

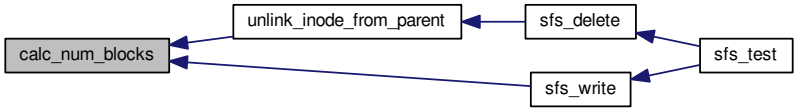
Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:


Here is the caller graph for this function:




#### 5.20.1.3 uint32_t calc_num_bytes ( byte * *buf* )

Calculate the number of bytes in a byte buffer.

**Precondition**

The buf parameter must be NULL terminated.

**Parameters**

| | |
|---|---|
| *buf* | A NULL terminated buffer. |

**Returns**

Returns the number of bytes inside the buffer, up to the NULL terminator.

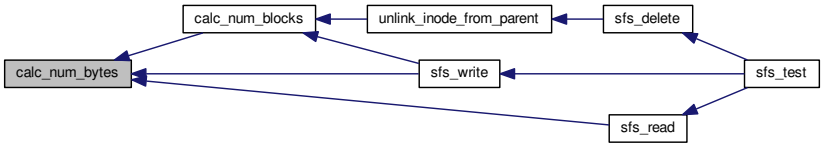
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.20.1.4 void* concat ( void * *src_1*, void * *src_2*, uint32_t *size* )

Concatenate two arrays.

Concatenates two NULL termianted arrays, a src_1 and a src_2 array into a single NULL terminated array containing the contents of both arrays. The terminating NULL character in src_1 is overwritten by the first character of src_2, and a NULL character is included at the end of the new array. A pointer to the resulting array containing the contents of both arrays is then returned.

**Precondition**

The arrays src_1 and src_2 MUST BE NULL TERMINATED.
The arrays to concatenate MUST contain the same data type for the concatenation to function properly.

**Parameters**

| | |
|---|---|
| *src_1* | A pointer to a NULL terminated array. |
| *src_2* | A pointer to a NULL terminated array. |
| *size* | The size in bytes of each item in the array. |

**Returns**

Returns a pointer to the resulting array containing the concatenated results.

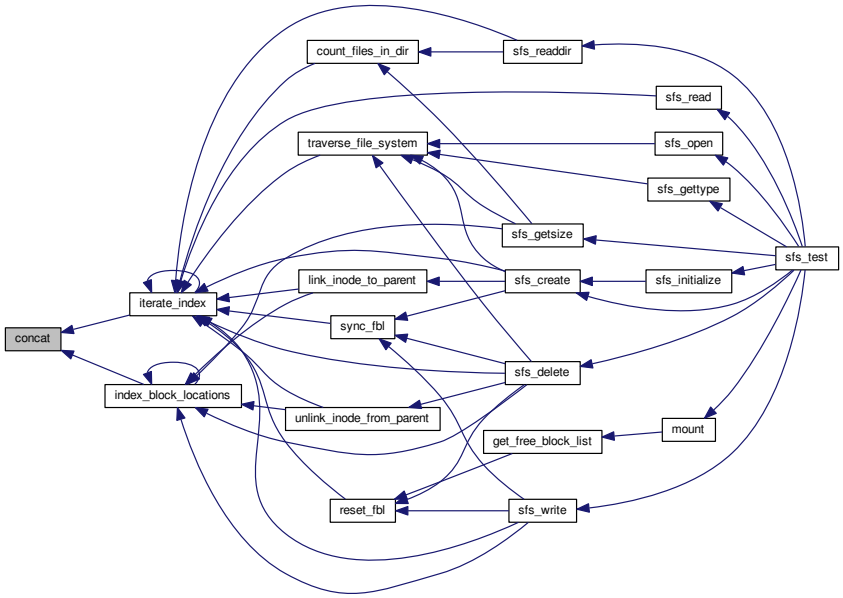
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.20.1.5 void* concat_len ( void * *src_1,* void * *src_2,* uint32_t *size,* uint32_t *len* )

Concatenate two byte arrays.

Concatenates a specified number of bytes from src_2 to src_1, into a single NULL terminated array containing the contents of both arrays. The difference is that when performing concatenation it concatenates a specified number of bytes from src_2 to src_1. The terminating NULL character in src_1 is overwritten by the first character of src_2, and a NULL character is included at the end of the new array. A pointer to the resulting array containing the contents of both arrays is then returned.

**Precondition**

The array src_1 MUST BE NULL TERMINATED.
The arrays to concatenate MUST contain the same data type for the concatenation to function properly.
The specified length, must be <= the length of src_2.

**Parameters**

| | |
|---|---|
| *src_1* | A pointer to a NULL terminated array. |
| *src_2* | A pointer to an array. |
| *size* | The size in bytes of each item in the array. |
| *len* | The length in BYTES of the data to concatenate from src_2 to src_1. |

**Returns**

Returns a pointer to the resulting array containing the concatenated results.

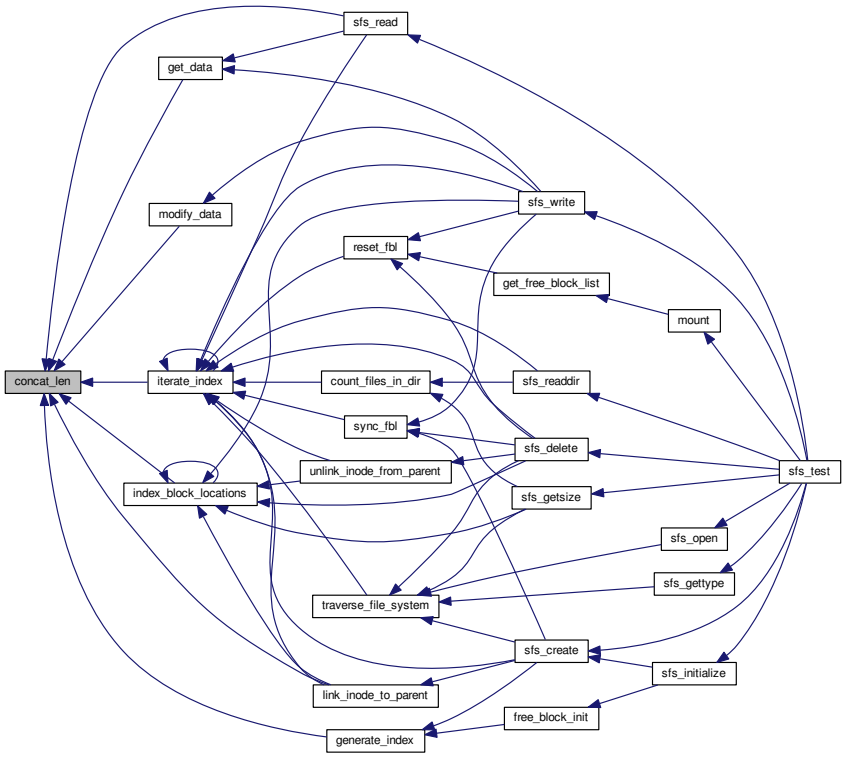
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.20.1.6 byte* copy_to_buf ( byte * *buf1*, byte * *buf2*, uint32_t *size1*, uint32_t *size2* )

Copy the data from one buffer to another, given the size of both.

**Parameters**

| | |
|---|---|
| *buf1* | The buffer to copy from. |
| *buf2* | The buffer to copy to. |
| *size1* | The size of the buffer to copy from. |
| *size2* | The size of the buffer to copy to. |

**Returns**

Returns the second buffer with the contents of the first buffer inside.

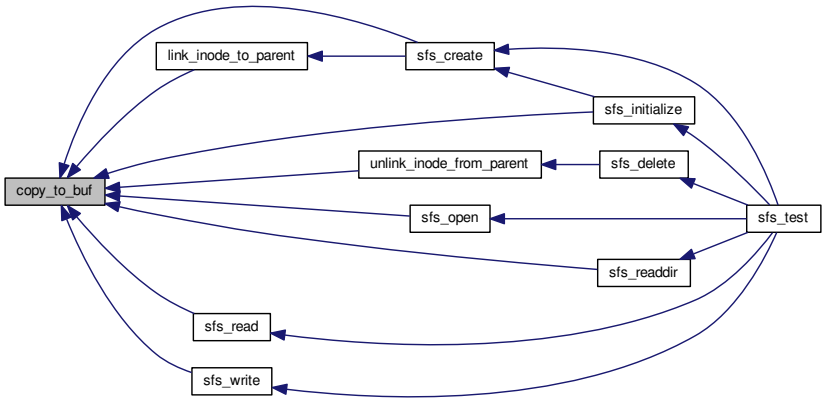
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.20.1.7 bool free_tokens ( char ** *tokens* )

Frees the memory allocated for the tokens.

This function is used to free memory used by dynamic memory allocation methods required for two dimensional string manipulation, preventing memory leaks.

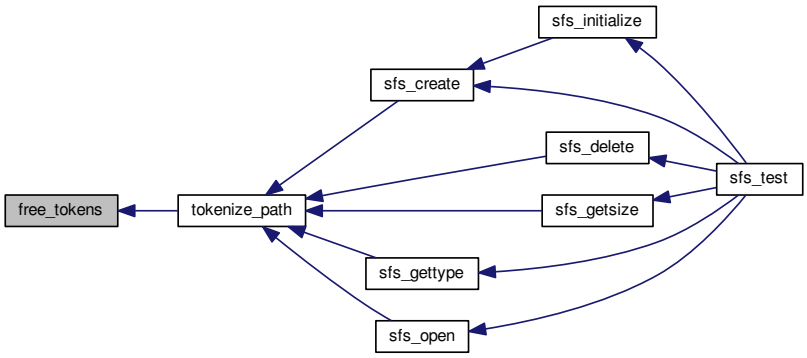
**Parameters**

| | |
|---|---|
| *tokens* | The NULL terminated 2D array filled with tokens to be freed. |

**Returns**

Returns true if the memory was freed, false if an error occurred.

Here is the caller graph for this function:




#### 5.20.1.8 char** tokenize_path ( char * *pathname* )

Tokenize a path into an array of tokens.

Tokenizes the path provided into an array of tokens for each component in the path and returns an array to a null terminated array of tokens. For example using pathname = "/foo/bar" the resulting tokens array would be: * [0] => "foo" [1] => "bar" [2] => NULL If an error occurs a NULL pointer will be returned. Tokens are each a string, therefore the resultant pointer will point to a two-dimensional array of characters.

**Precondition**

Each component in the pathname must be at most (including the NULL termination) MAX_NAME_LEN otherwise an error occurs, and a NULL pointer returned.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname to tokenize. |

**Returns**

Returns a pointer to the 2D tokens array.

**Author**

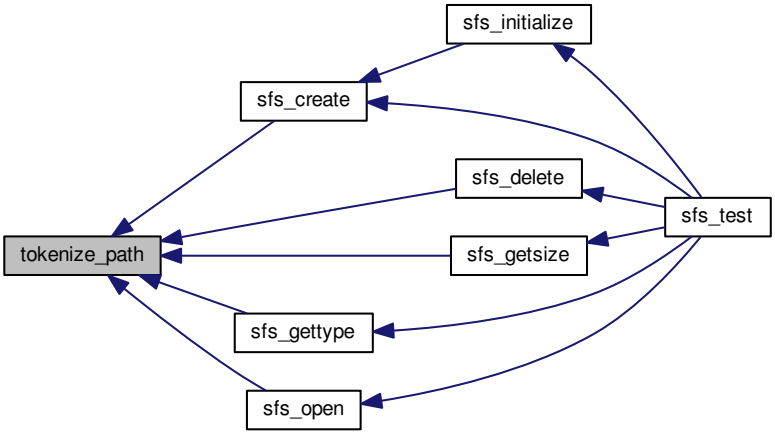
Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

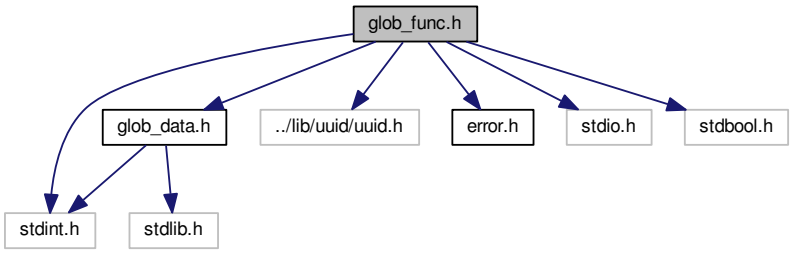

Here is the caller graph for this function:



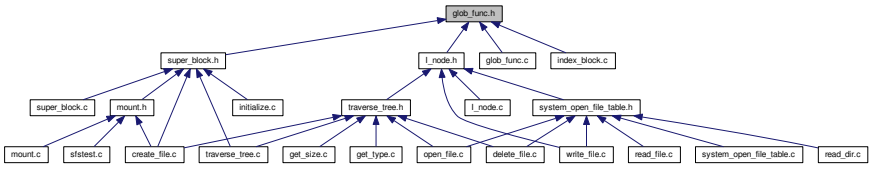

## 5.21 glob_func.h File Reference

```
#include "glob_data.h"
#include "../lib/uuid/uuid.h"
#include "error.h"
#include <stdint.h>
#include <stdio.h>
#include <stdbool.h>
```

Include dependency graph for glob_func.h:




This graph shows which files directly or indirectly include this file:




## Functions

- byte * allocate_buf (byte *buf, uint32_t size)

  *Allocate a buffer to write to memory of a given size.*

- byte * copy_to_buf (byte *buf1, byte *buf2, uint32_t size1, uint32_t size2)

  *Copy the data from one buffer to another, given the size of both.*

- uint32_t calc_num_bytes (byte *buf)

  *Calculate the number of bytes in a byte buffer.*

- uint32_t calc_num_blocks (byte *buf)

  *Calculates the number of blocks needed to store data.*

- void * concat (void *src_1, void *src_2, uint32_t size)

  *Concatenate two arrays.*

- void * concat_len (void *src_1, void *src_2, uint32_t size, uint32_t len)

  *Concatenate two byte arrays.*

- char ** tokenize_path (char *pathname)

  *Tokenize a path into an array of tokens.*

- bool free_tokens (char **tokens)

  *Frees the memory allocated for the tokens.*

### 5.21.1 Function Documentation

#### 5.21.1.1 byte* allocate_buf ( byte * *buf*, uint32_t *size* )

Allocate a buffer to write to memory of a given size.

**Parameters**

| | |
|---|---|
| *buf* | The buffer to allocate the space for. |
| *size* | The size for the new buffer's elements. |

**Returns**

Returns a pointer to the new buffer.

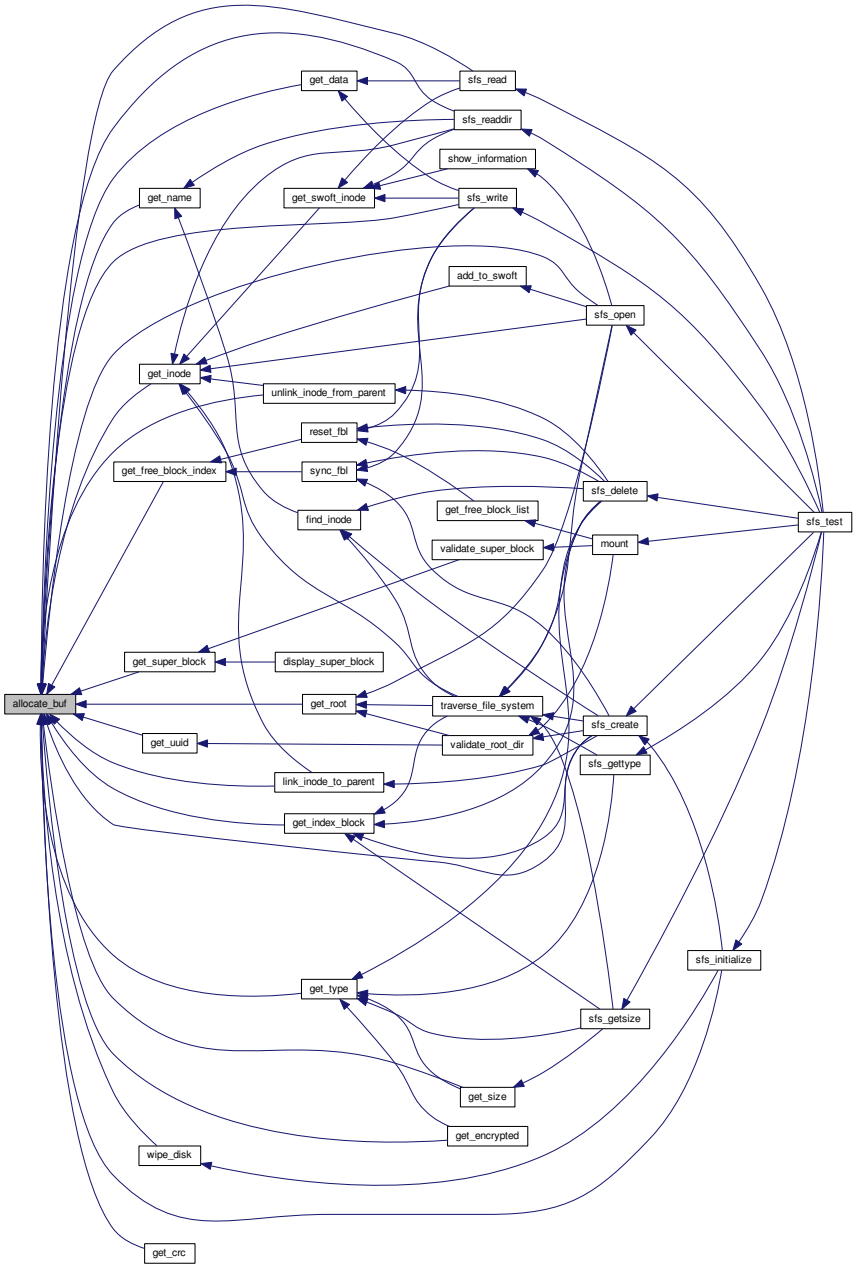
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.21.1.2 uint32_t calc_num_blocks ( byte * *buf* )

Calculates the number of blocks needed to store data.

Helper function which calculates the number of blocks needed to store the data, if an error occurs 0 is returned. Reads up to MAX_IO_LENGTH + 1, if the buffer is still not terminated then the IO buffer is larger than the maximum specified by the file system, an error occurs and 0 is returned.

**Precondition**

The buf parameter must be NULL terminated.

**Parameters**

| | |
|---|---|
| *buf* | A NULL terminated buffer. |

**Returns**

Returns the number of blocks needed to store the given data.

**Author**

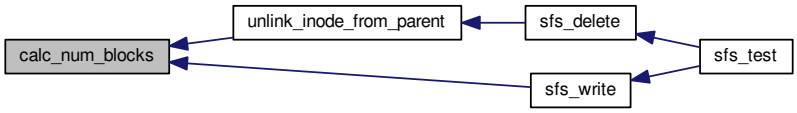
Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:


Here is the caller graph for this function:




#### 5.21.1.3 uint32_t calc_num_bytes ( byte * *buf* )

Calculate the number of bytes in a byte buffer.

**Precondition**

The buf parameter must be NULL terminated.

**Parameters**

| | |
|---|---|
| *buf* | A NULL terminated buffer. |

**Returns**

Returns the number of bytes inside the buffer, up to the NULL terminator.

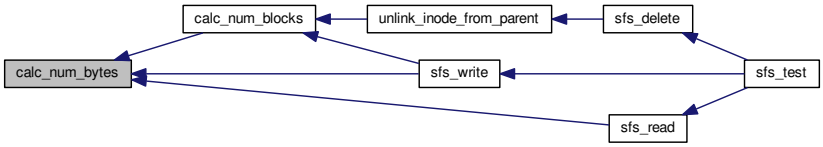
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.21.1.4 void* concat ( void * *src_1*, void * *src_2*, uint32_t *size* )

Concatenate two arrays.

Concatenates two NULL termianted arrays, a src_1 and a src_2 array into a single NULL terminated array containing the contents of both arrays. The terminating NULL character in src_1 is overwritten by the first character of src_2, and a NULL character is included at the end of the new array. A pointer to the resulting array containing the contents of both arrays is then returned.

**Precondition**

The arrays src_1 and src_2 MUST BE NULL TERMINATED.
The arrays to concatenate MUST contain the same data type for the concatenation to function properly.

**Parameters**

| | |
|---|---|
| *src_1* | A pointer to a NULL terminated array. |
| *src_2* | A pointer to a NULL terminated array. |
| *size* | The size in bytes of each item in the array. |

**Returns**

Returns a pointer to the resulting array containing the concatenated results.

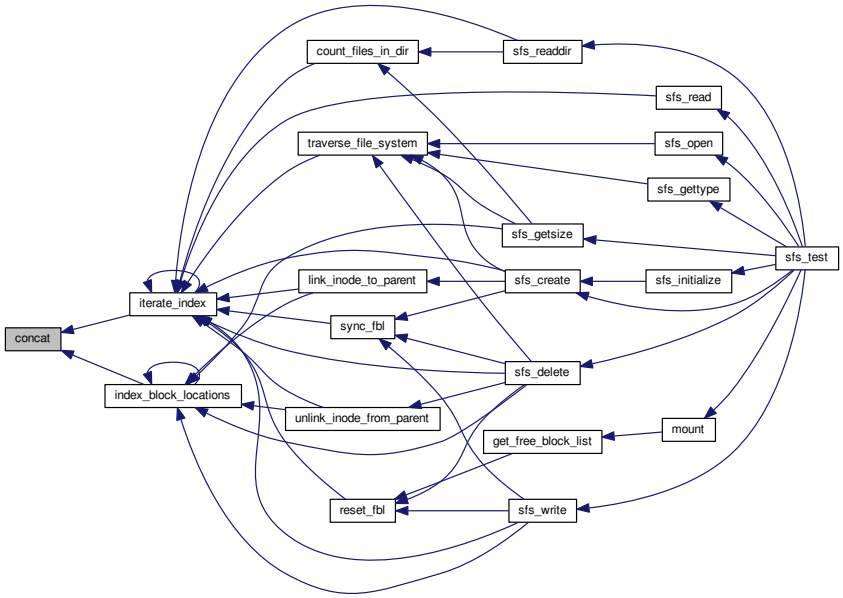
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




### 5.21.1.5 void* concat_len ( void * *src_1,* void * *src_2,* uint32_t *size,* uint32_t *len* )

Concatenate two byte arrays.

Concatenates a specified number of bytes from src_2 to src_1, into a single NULL terminated array containing the contents of both arrays. The difference is that when performing concatenation it concatenates a specified number of bytes from src_2 to src_1. The terminating NULL character in src_1 is overwritten by the first character of src_2, and a NULL character is included at the end of the new array. A pointer to the resulting array containing the contents of both arrays is then returned.

**Precondition**

The array src_1 MUST BE NULL TERMINATED.
The arrays to concatenate MUST contain the same data type for the concatenation to function properly.
The specified length, must be <= the length of src_2.

**Parameters**

| | |
|---|---|
| *src_1* | A pointer to a NULL terminated array. |
| *src_2* | A pointer to an array. |
| *size* | The size in bytes of each item in the array. |
| *len* | The length in BYTES of the data to concatenate from src_2 to src_1. |

**Returns**

Returns a pointer to the resulting array containing the concatenated results.

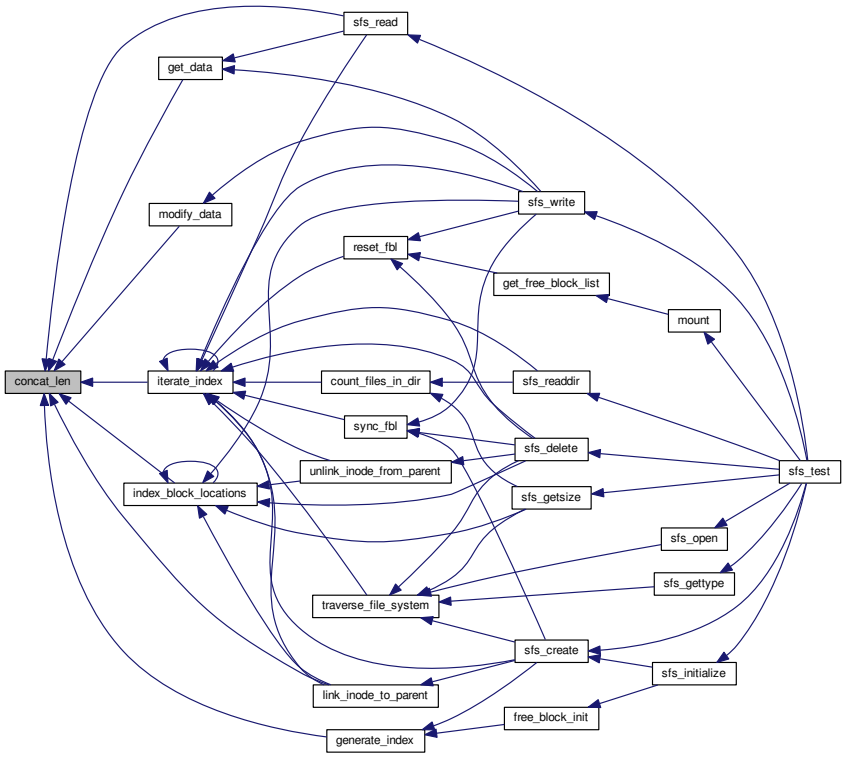
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.21.1.6 byte* copy_to_buf ( byte * *buf1*, byte * *buf2*, uint32_t *size1*, uint32_t *size2* )

Copy the data from one buffer to another, given the size of both.

**Parameters**

| | |
|---|---|
| *buf1* | The buffer to copy from. |
| *buf2* | The buffer to copy to. |
| *size1* | The size of the buffer to copy from. |
| *size2* | The size of the buffer to copy to. |

**Returns**

Returns the second buffer with the contents of the first buffer inside.

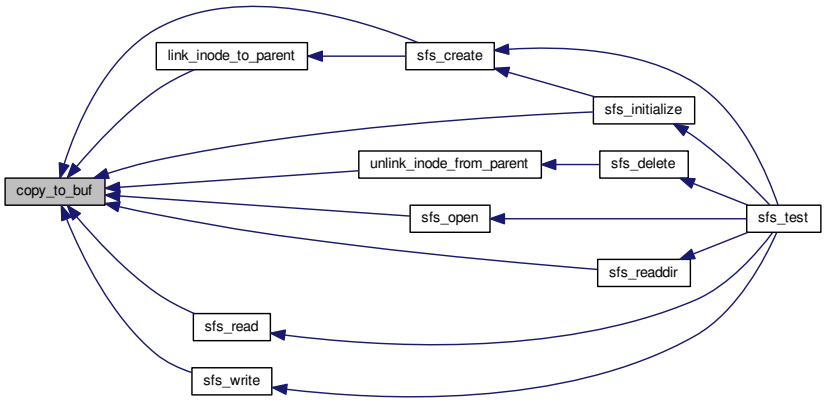
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.21.1.7 bool free_tokens ( char ** *tokens* )

Frees the memory allocated for the tokens.

This function is used to free memory used by dynamic memory allocation methods required for two dimensional string manipulation, preventing memory leaks.

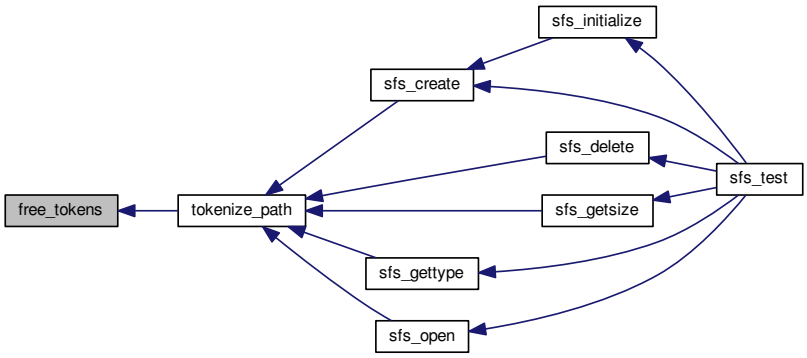
**Parameters**

| | |
|---|---|
| *tokens* | The NULL terminated 2D array filled with tokens to be freed. |

**Returns**

Returns true if the memory was freed, false if an error occurred.

Here is the caller graph for this function:




#### 5.21.1.8 char** tokenize_path ( char * *pathname* )

Tokenize a path into an array of tokens.

Tokenizes the path provided into an array of tokens for each component in the path and returns an array to a null terminated array of tokens. For example using pathname = "/foo/bar" the resulting tokens array would be: * [0] => "foo" [1] => "bar" [2] => NULL If an error occurs a NULL pointer will be returned. Tokens are each a string, therefore the resultant pointer will point to a two-dimensional array of characters.

**Precondition**

Each component in the pathname must be at most (including the NULL termination) MAX_NAME_LEN otherwise an error occurs, and a NULL pointer returned.

**Parameters**

| | |
|---|---|
| *pathname* | The pathname to tokenize. |

**Returns**

Returns a pointer to the 2D tokens array.

**Author**

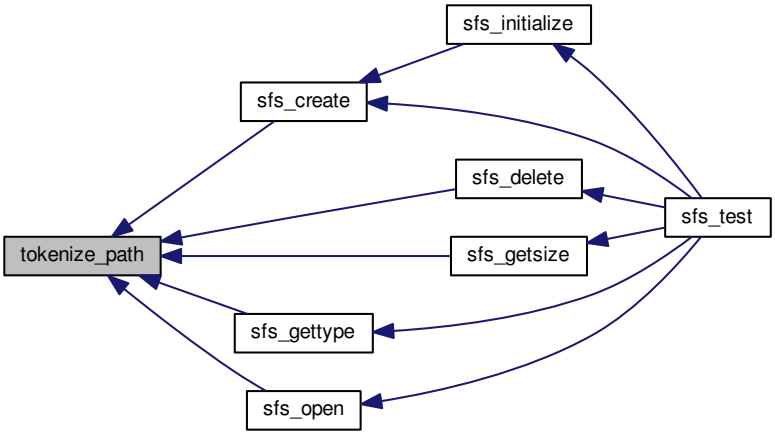
Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

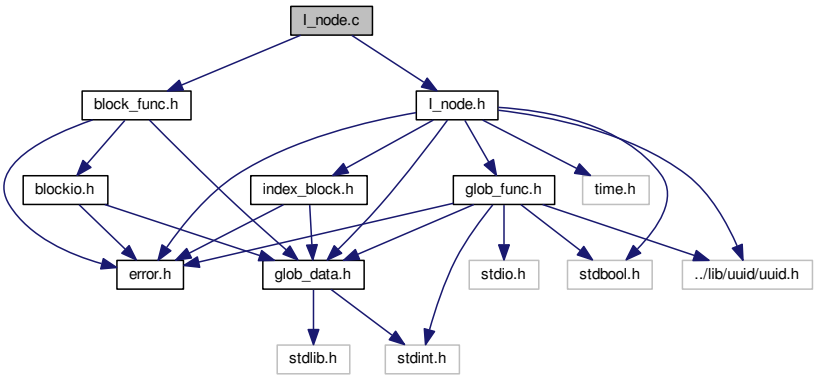

Here is the caller graph for this function:




## 5.22 I_node.c File Reference

```
#include "I_node.h"
#include "block_func.h"
```

Include dependency graph for l_node.c:




## Functions

- inode get_null_inode ()

  *Get an inode set to null state.*
- inode * get_inode (uint32_t block_num)

  *Get the contents of an inode given its location on disk.*
- unsigned char * get_uuid (uint32_t block_num)

  *Get an inode's UUID given its location on disk.*
- uint32_t get_index_block (uint32_t block_num)

  *Get the index block which an inode links to given the inode's location on disk.*
- int get_type (uint32_t block_num)

  *Get the type of file from an inode, whether it is a directory or a data file.*
- uint32_t get_size (uint32_t block_num)

  *Get size of a file given its inode's location on disk.*
- int get_encrypted (uint32_t block_num)

  *Check if the specified file is encrypted given its inode's location on disk.*
- char * get_name (uint32_t block_num)

  *Get the name of the file given its inode's location on disk.*
- uint32_t get_crc (uint32_t block_num)

  *Get the encryption checksum of a file given its inode's location on disk.*
- uint32_t find_inode (locations index_blocks, char *name)
- int link_inode_to_parent (uint32_t parent_location, uint32_t inode_location)

  *Links the inode to a parent's index structure.*
- int unlink_inode_from_parent (uint32_t parent_location, uint32_t inode_location)

  *Unlinks a child's inode from the parent's index structure.*
- int get_index_entry (inode directory)

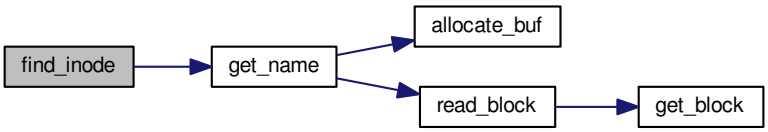
  *Gets the next entry for the current directory and increments the index of the last accessed item within a directory.*
- void reset_index_entry ()

  *Resets the index entry count. Attempting to get the next directory item after this function has executed will get the first item in the directory.*

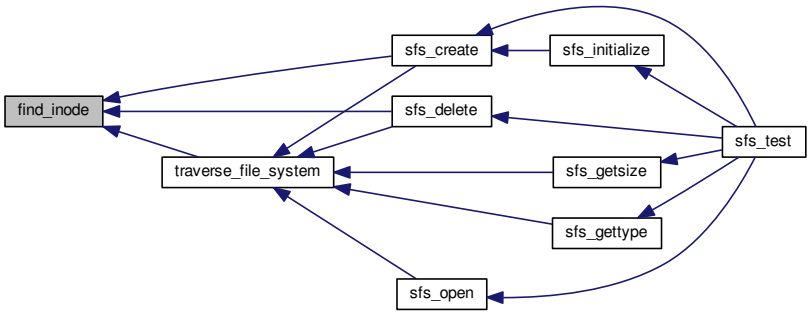
### 5.22.1 Function Documentation

#### 5.22.1.1 uint32_t find_inode ( locations *index_blocks*, char * *name* )

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.22.1.2 uint32_t get_crc ( uint32_t *block_num* )

Get the encryption checksum of a file given its inode's location on disk.

This function retrieves the encryption checksum for a file's data blocks, so that it can be determined whether the encryption and decryption process was completed successfully. This is done by retrieving the inode and analyzing the contents based on the defined inode data structure.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the checksum, which is a string. If the function returns NULL, then the function was unsuccessful.

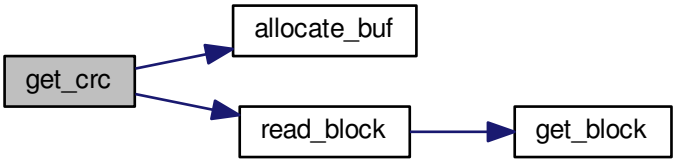
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:




#### 5.22.1.3 int get_encrypted ( uint32_t *block_num* )

Check if the specified file is encrypted given its inode's location on disk.

This function checks whether a file's data blocks are encrypted by retrieving the inode and analyzing the contents based on the defined inode data structure. This allows for the file system to determine whether to decrypt the data blocks associated with the file before reading, or whether they are aren't encrypted and can be read normally.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns whether the file is encrypted. If the returned value is 0 then the file is not encrypted and if it is 1 then the file is encrypted. Otherwise, the function was unsuccessful.

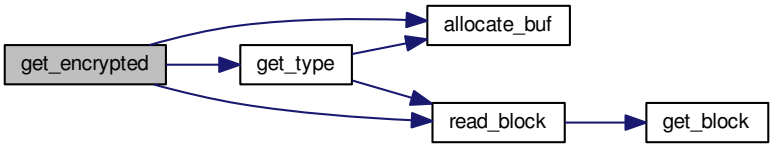
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:




#### 5.22.1.4 uint32_t get_index_block ( uint32_t *block_num* )

Get the index block which an inode links to given the inode's location on disk.

This function retrieves the first index block location from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. This allows the inode to be associated with its index block data structure, which in turn allows for the retrieval of all the data blocks associated with the file.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns the location of the index block on disk. If index >= 0 then the function was successful, if index < 0 then the function was unsuccessful.

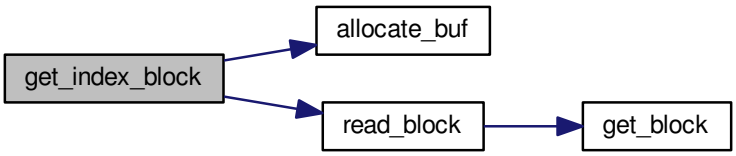
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

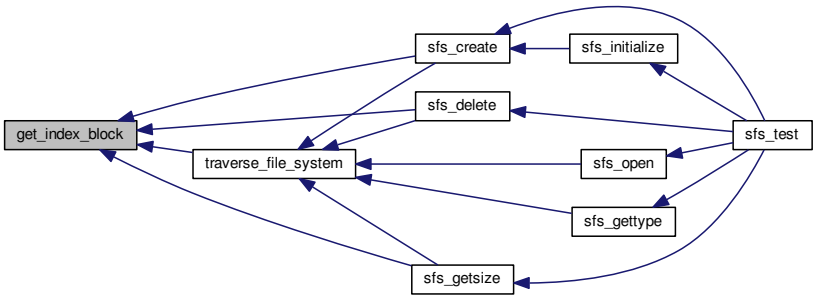
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.22.1.5 int get_index_entry ( inode *directory* )

Gets the next entry for the current directory and increments the index of the last accessed item within a directory.

This function is used when reading directories. Directory elements are retrieved from an index data structure that corresponds with a directory's inode. Each of these elements must be iterated through in order to fully traverse a directory's contents. After the entire directory contents are traversed, the function returns a zero value. If the contents are traversed again, the function restarts from the beginning.

**Parameters**

| | |
|---|---|
| *directory* | The inode of the current directory. |

**Returns**

Returns the directory element.

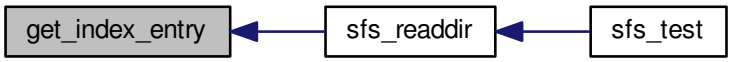
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.22.1.6 inode* get_inode ( uint32_t *block_num* )

Get the contents of an inode given its location on disk.

This function retrieves an inode given its location by retrieving the block and analyzing the contents based on the defined inode data structure.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the inode at the given location. If the returned value is NULL, an inode was not found at the specified location.

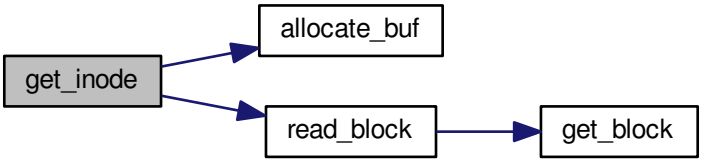
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

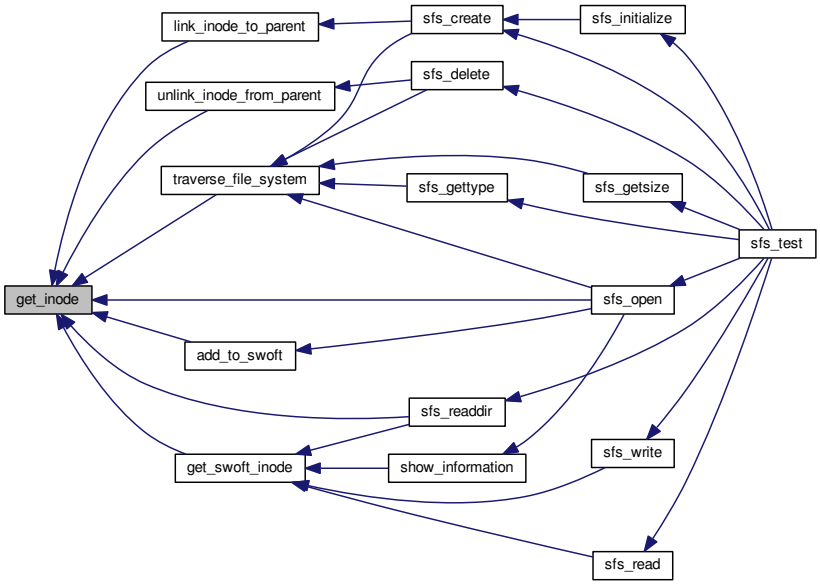
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.22.1.7 char* get_name ( uint32_t *block_num* )

Get the name of the file given its inode's location on disk.

This function determines the human-readable name of a file by retrieving the inode and analyzing the contents based on the defined inode data structure. The name is a human-readable string.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the file's name, which is a string. If the function returns NULL, then the function was unsuccessful.

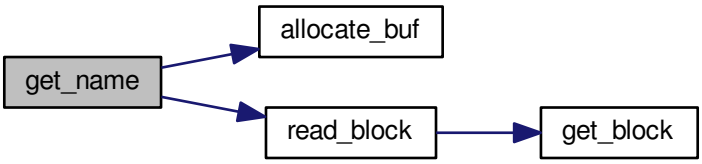
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

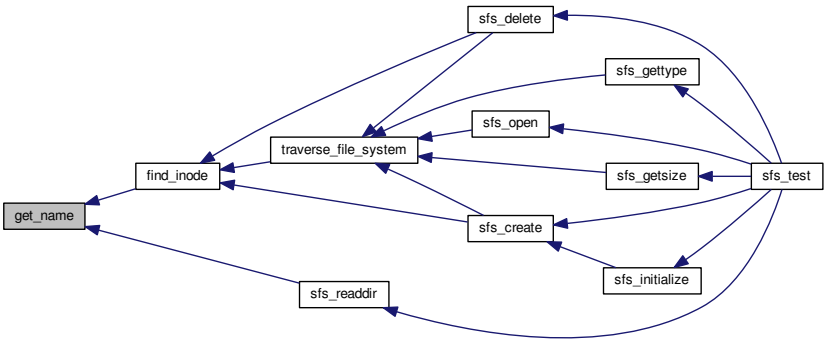
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.22.1.8 inode get_null_inode ( )

Get an inode set to null state.

This function generates a single block of null-initialized data using the predefined BLKSIZE constant. It then outputs it as type inode.

**Returns**

Returns a null-initialized inode data structure.

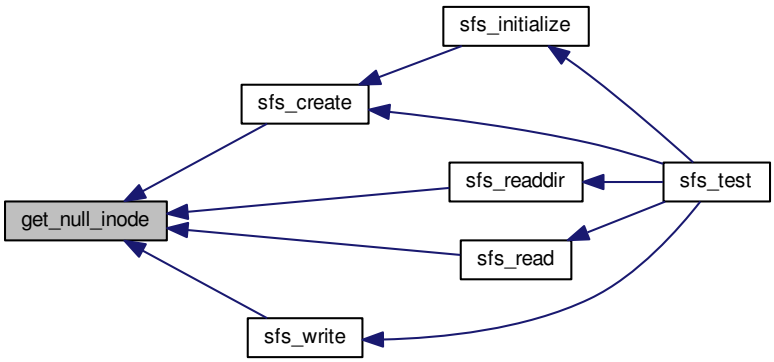
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.22.1.9 uint32_t get_size ( uint32_t *block_num* )

Get size of a file given its inode's location on disk.

This function retrieves the file size attribute from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. This allows for the file system to quickly retrieve the size of the data (in bytes) stored within a file's data blocks.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns the size of the data blocks' contents for the file in bytes. If the returned size >= 0, then the function was successful, if the returned size < 0, then the function was unsuccessful.

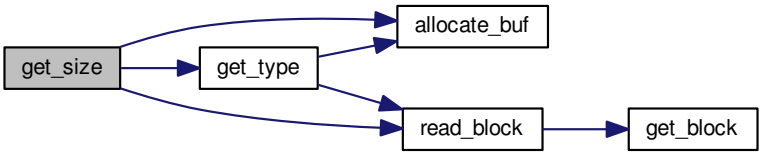
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

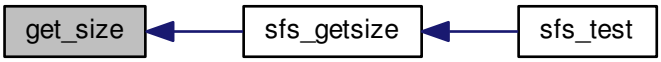
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.22.1.10 int get_type ( uint32_t *block_num* )

Get the type of file from an inode, whether it is a directory or a data file.

This function retrieves the file type attribute from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. This allows for the file system to determine whether an inode is for a data file or a directory.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns the type of the file. If the function returns 1, then the inode is for a directory, if the function returns 0, then the inode is for a data file, Otherwise retrieving the type was unsuccessful.

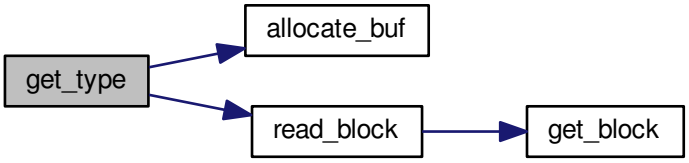
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

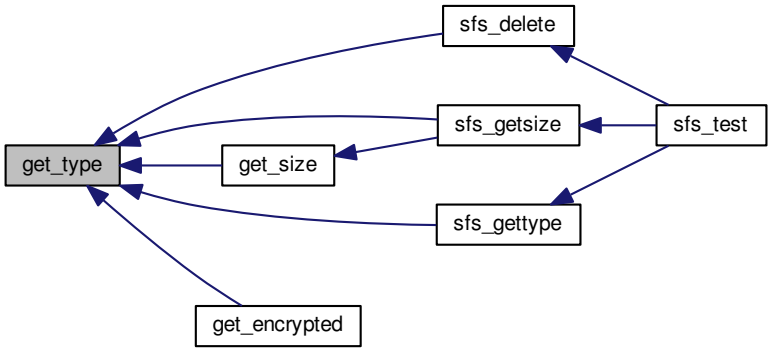
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.22.1.11 unsigned char* get_uuid ( uint32_t *block_num* )

Get an inode's UUID given its location on disk.

This function retrieves a UUID from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. UUIDs are human-readable strings, therefore the return type for this operation is an array of characters.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the UUID from within the inode at the given location.

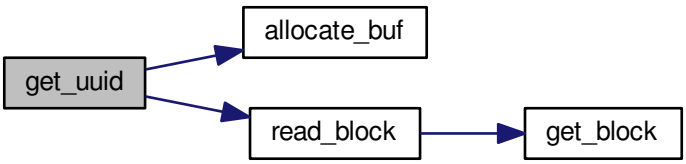
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

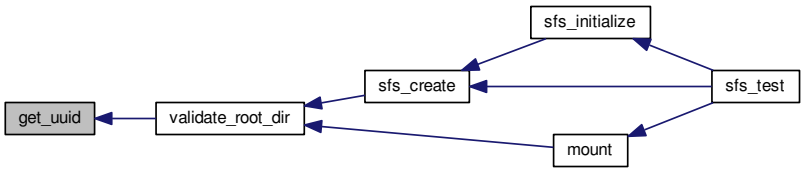
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.22.1.12 int link_inode_to_parent ( uint32_t *parent_location*, uint32_t *inode_location* )

Links the inode to a parent's index structure.

This function works by accessing the specified parent inode, and accessing the index data structure that it links to within its defined inode data structure. After accessing this index structure, the specified child inode is appended to the index. The index is rebuilt and written to disk, and the newly created index is linked to within the parent's inode. This inode is then rewritten at the original location, overwriting the old one. Upon completion of this operation the old index blocks are freed. If a failure occurs during this operation, the changes are automatically undone. The internal implementation of this function utilizes copy-on-write journalling in order to prevent any damage to the file system from failed operations.

**Parameters**

| | |
|---|---|
| *parent_location* | The parent's location on disk. |
| *inode_location* | The child inode's location on disk. |

**Returns**

Returns a value to signify success or failure. If the value is >= 0 the function was successful. If value is < 0, the function was unsuccessful.

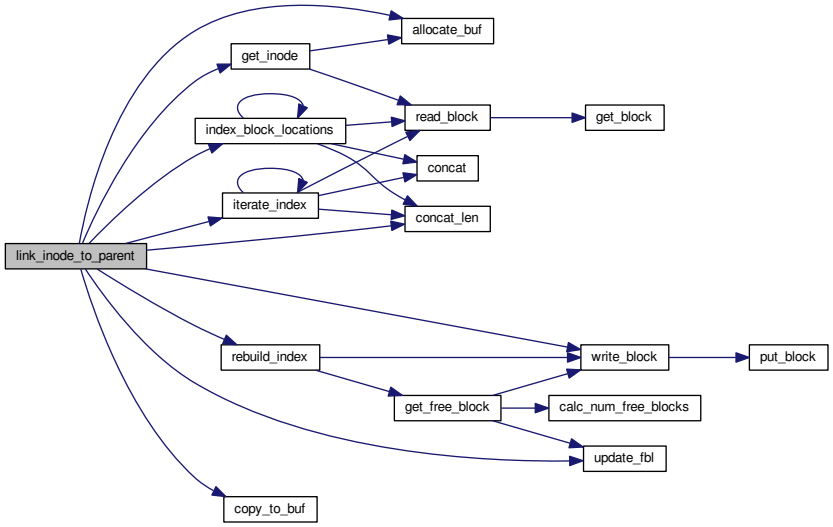
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

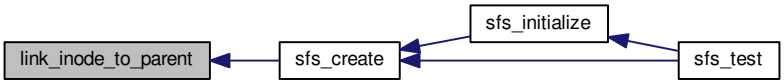
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.22.1.13 void reset_index_entry ( )

Resets the index entry count. Attempting to get the next directory item after this function has executed will get the first item in the directory.

This function is used to reset the iterator for traversing the contents of a directory structure. After this function is called, any further attempts at traversing the contents of a directory will start over again from the first element.

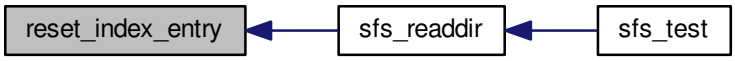
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.22.1.14 int unlink_inode_from_parent ( uint32_t *parent_location,* uint32_t *inode_location* )

Unlinks a child's inode from the parent's index structure.

This function works by accessing the specified parent inode, and accessing the index data structure that it links to within its defined inode data structure. After accessing this index structure, the block at each indexed location is loaded and the name attribute within the inode at that block is checked against the name within the specified child inode. Once a match is found, the corresponding indexed block is removed from the index structure. The index is rebuilt and written to disk, and the newly created index is linked to within the parent's inode. This inode is then rewritten at the original location, overwriting the old one. Upon completion of this operation the old index blocks are freed. If a failure occurs during this operation, the changes are automatically undone. The internal implementation of this function utilizes copy-on-write journalling in order to prevent any damage to the file system from failed operations.

**Parameters**

| | |
|---|---|
| *parent_location* | The parent's location on disk. |
| *inode_location* | The child inode's location on disk. |

**Returns**

Returns a value to signify success or failure. If the value is >= 0 the function was successful. If value is < 0, the function was unsuccessful.

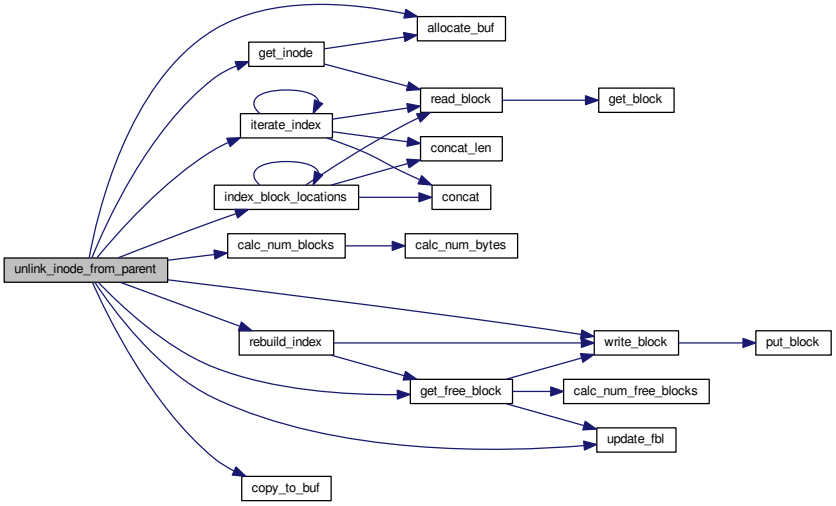
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

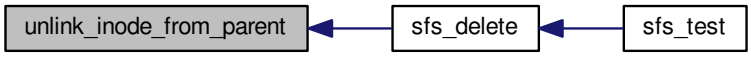
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



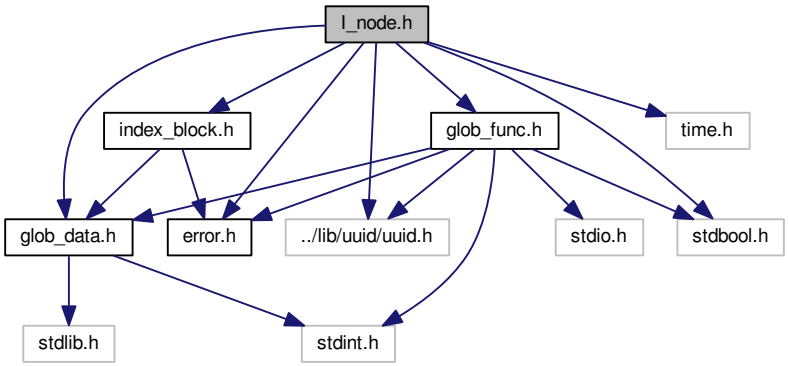

Here is the caller graph for this function:



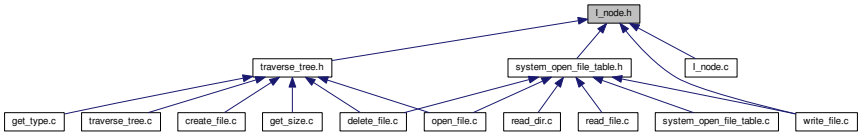
## 5.23 l_node.h File Reference

```
#include "glob_data.h"
#include "glob_func.h"
#include "index_block.h"
#include "error.h"
#include "../lib/uuid/uuid.h"
#include <stdbool.h>
#include <time.h>
```

Include dependency graph for l_node.h:




This graph shows which files directly or indirectly include this file:




### Data Structures

- struct inode

  *Contains the data stored in inodes.*
- struct cwd

  *Holds the current working directory data.*

### Functions

- inode get_null_inode ()

  *Get an inode set to null state.*

- inode * get_inode (uint32_t block_num)

  *Get the contents of an inode given its location on disk.*
- unsigned char * get_uuid (uint32_t block_num)

  *Get an inode's UUID given its location on disk.*
- uint32_t get_index_block (uint32_t block_num)

  *Get the index block which an inode links to given the inode's location on disk.*
- int get_type (uint32_t block_num)

  *Get the type of file from an inode, whether it is a directory or a data file.*
- uint32_t get_size (uint32_t block_num)

  *Get size of a file given its inode's location on disk.*
- int get_encrypted (uint32_t block_num)

  *Check if the specified file is encrypted given its inode's location on disk.*
- char * get_name (uint32_t block_num)

  *Get the name of the file given its inode's location on disk.*
- uint32_t get_crc (uint32_t block_num)

  *Get the encryption checksum of a file given its inode's location on disk.*
- uint32_t find_inode (locations index_blocks, char *name)
- int link_inode_to_parent (uint32_t parent_location, uint32_t inode_location)

  *Links the inode to a parent's index structure.*
- int unlink_inode_from_parent (uint32_t parent_location, uint32_t inode_location)

  *Unlinks a child's inode from the parent's index structure.*
- int get_index_entry (inode directory)

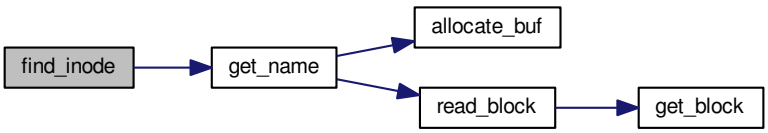
  *Gets the next entry for the current directory and increments the index of the last accessed item within a directory.*
- void reset_index_entry ()

  *Resets the index entry count. Attempting to get the next directory item after this function has executed will get the first item in the directory.*

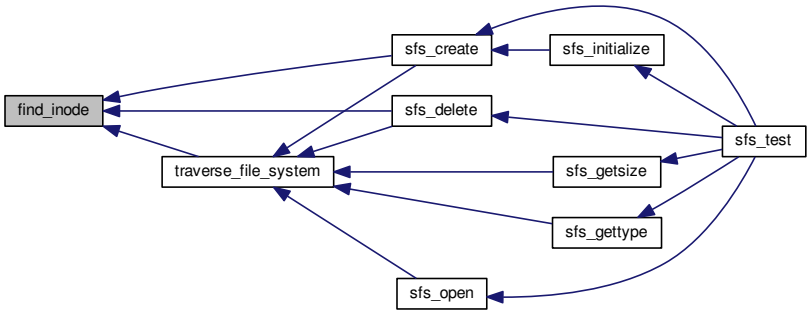
### 5.23.1 Function Documentation

#### 5.23.1.1 uint32_t find_inode ( locations *index_blocks*, char * *name* )

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.23.1.2 uint32_t get_crc ( uint32_t *block_num* )

Get the encryption checksum of a file given its inode's location on disk.

This function retrieves the encryption checksum for a file's data blocks, so that it can be determined whether the encryption and decryption process was completed successfully. This is done by retrieving the inode and analyzing the contents based on the defined inode data structure.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the checksum, which is a string. If the function returns NULL, then the function was unsuccessful.

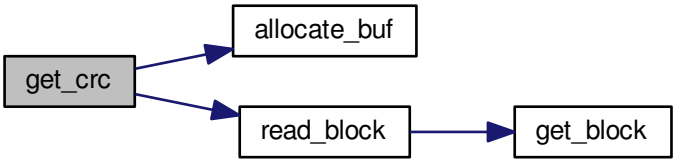
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:




#### 5.23.1.3 int get_encrypted ( uint32_t *block_num* )

Check if the specified file is encrypted given its inode's location on disk.

This function checks whether a file's data blocks are encrypted by retrieving the inode and analyzing the contents based on the defined inode data structure. This allows for the file system to determine whether to decrypt the data blocks associated with the file before reading, or whether they are aren't encrypted and can be read normally.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns whether the file is encrypted. If the returned value is 0 then the file is not encrypted and if it is 1 then the file is encrypted. Otherwise, the function was unsuccessful.

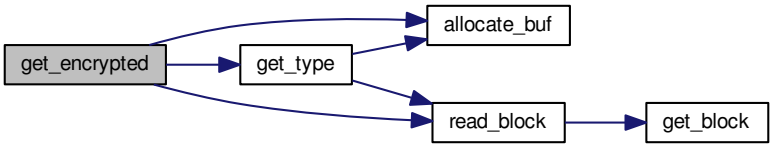
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:




#### 5.23.1.4 uint32_t get_index_block ( uint32_t *block_num* )

Get the index block which an inode links to given the inode's location on disk.

This function retrieves the first index block location from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. This allows the inode to be associated with its index block data structure, which in turn allows for the retrieval of all the data blocks associated with the file.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns the location of the index block on disk. If index >= 0 then the function was successful, if index < 0 then the function was unsuccessful.

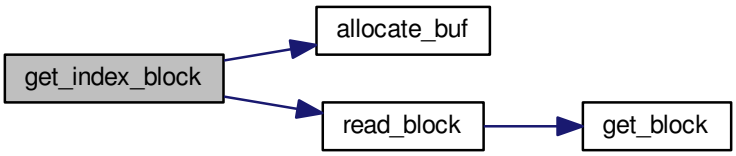
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

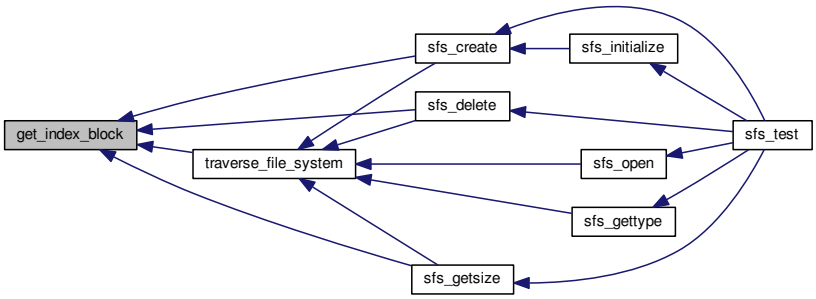
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.23.1.5 int get_index_entry ( inode *directory* )

Gets the next entry for the current directory and increments the index of the last accessed item within a directory.

This function is used when reading directories. Directory elements are retrieved from an index data structure that corresponds with a directory's inode. Each of these elements must be iterated through in order to fully traverse a directory's contents. After the entire directory contents are traversed, the function returns a zero value. If the contents are traversed again, the function restarts from the beginning.

**Parameters**

| | |
|---|---|
| *directory* | The inode of the current directory. |

**Returns**

Returns the directory element.

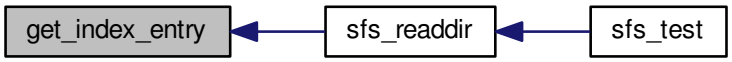
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.23.1.6 inode* get_inode ( uint32_t *block_num* )

Get the contents of an inode given its location on disk.

This function retrieves an inode given its location by retrieving the block and analyzing the contents based on the defined inode data structure.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the inode at the given location. If the returned value is NULL, an inode was not found at the specified location.

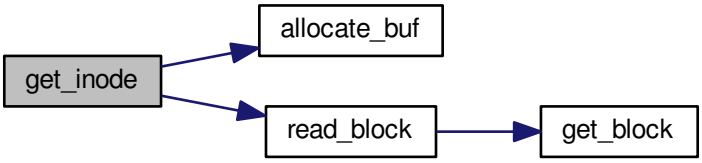
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

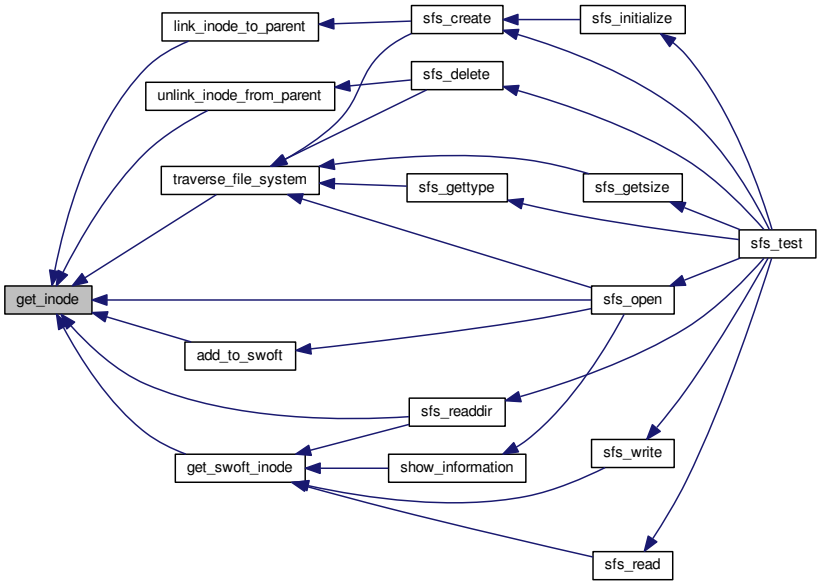
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.23.1.7 char* get_name ( uint32_t *block_num* )

Get the name of the file given its inode's location on disk.

This function determines the human-readable name of a file by retrieving the inode and analyzing the contents based on the defined inode data structure. The name is a human-readable string.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the file's name, which is a string. If the function returns NULL, then the function was unsuccessful.

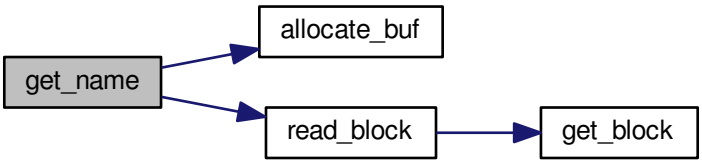
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

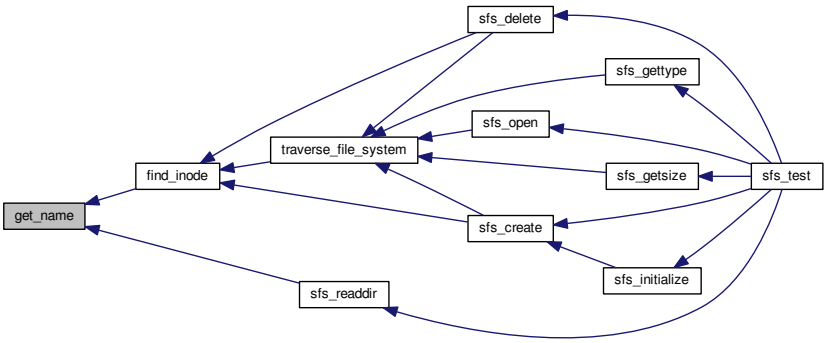
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.23.1.8 inode get_null_inode ( )

Get an inode set to null state.

This function generates a single block of null-initialized data using the predefined BLKSIZE constant. It then outputs it as type inode.

**Returns**

Returns a null-initialized inode data structure.

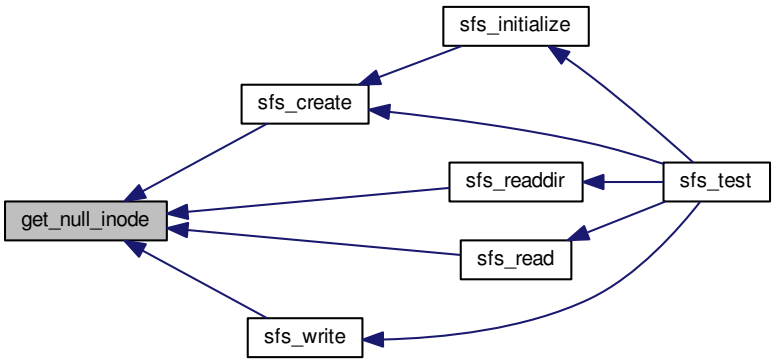
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.23.1.9 uint32_t get_size ( uint32_t *block_num* )

Get size of a file given its inode's location on disk.

This function retrieves the file size attribute from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. This allows for the file system to quickly retrieve the size of the data (in bytes) stored within a file's data blocks.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns the size of the data blocks' contents for the file in bytes. If the returned size >= 0, then the function was successful, if the returned size < 0, then the function was unsuccessful.

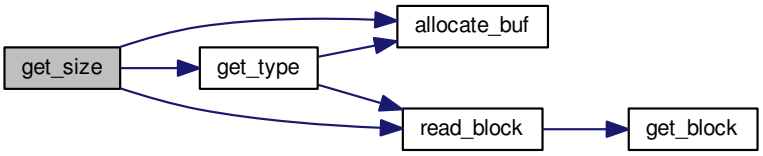
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

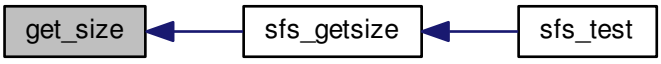
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.23.1.10 int get_type ( uint32_t *block_num* )

Get the type of file from an inode, whether it is a directory or a data file.

This function retrieves the file type attribute from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. This allows for the file system to determine whether an inode is for a data file or a directory.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns the type of the file. If the function returns 1, then the inode is for a directory, if the function returns 0, then the inode is for a data file, Otherwise retrieving the type was unsuccessful.

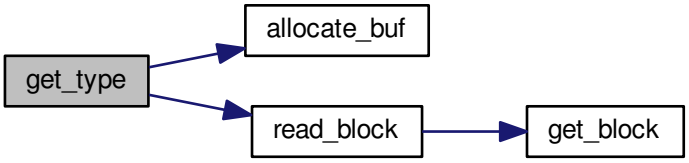
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

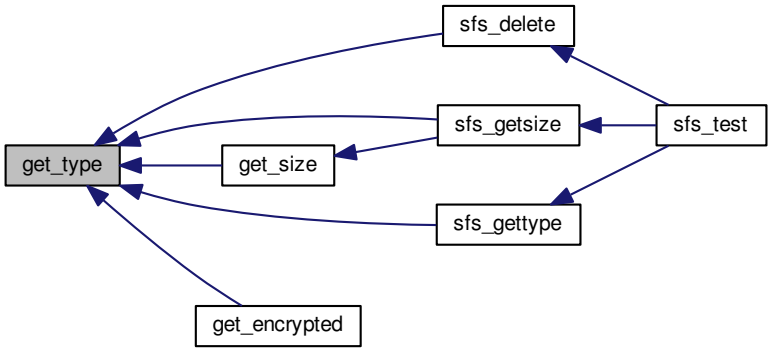
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.23.1.11 unsigned char* get_uuid ( uint32_t *block_num* )

Get an inode's UUID given its location on disk.

This function retrieves a UUID from an inode by retrieving the block and analyzing the contents based on the defined inode data structure. UUIDs are human-readable strings, therefore the return type for this operation is an array of characters.

**Parameters**

| | |
|---|---|
| *block_num* | The block location of the inode on disk. |

**Returns**

Returns a pointer to the UUID from within the inode at the given location.

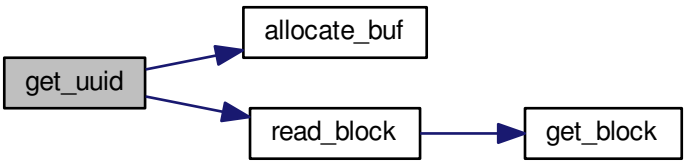
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

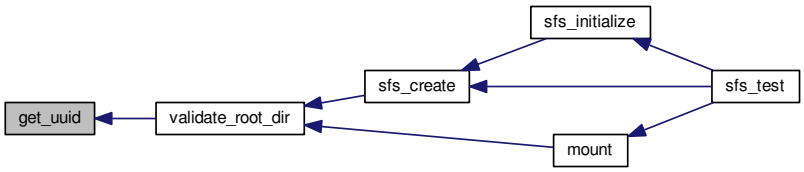
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.23.1.12 int link_inode_to_parent ( uint32_t *parent_location*, uint32_t *inode_location* )

Links the inode to a parent's index structure.

This function works by accessing the specified parent inode, and accessing the index data structure that it links to within its defined inode data structure. After accessing this index structure, the specified child inode is appended to the index. The index is rebuilt and written to disk, and the newly created index is linked to within the parent's inode. This inode is then rewritten at the original location, overwriting the old one. Upon completion of this operation the old index blocks are freed. If a failure occurs during this operation, the changes are automatically undone. The internal implementation of this function utilizes copy-on-write journalling in order to prevent any damage to the file system from failed operations.

**Parameters**

| | |
|---|---|
| *parent_location* | The parent's location on disk. |
| *inode_location* | The child inode's location on disk. |

**Returns**

Returns a value to signify success or failure. If the value is >= 0 the function was successful. If value is < 0, the function was unsuccessful.

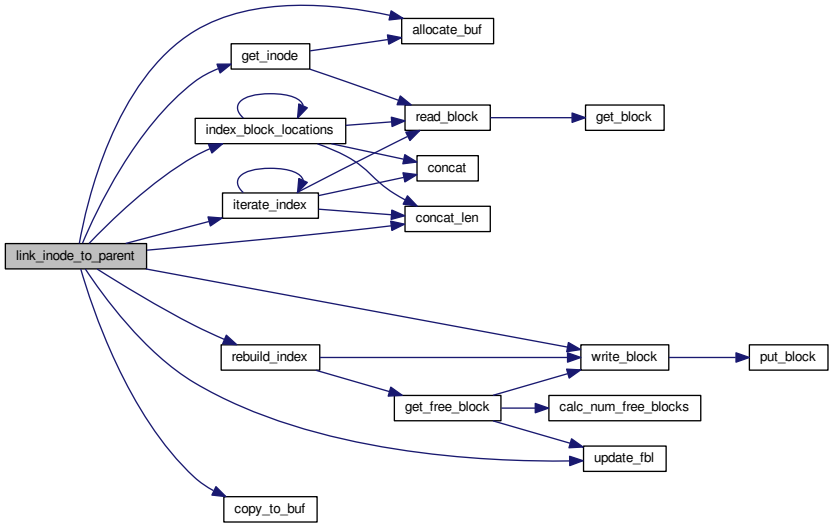
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

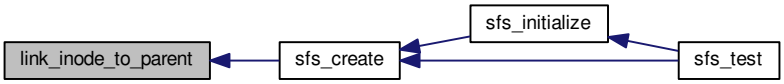
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.23.1.13 void reset_index_entry ( )

Resets the index entry count. Attempting to get the next directory item after this function has executed will get the first item in the directory.

This function is used to reset the iterator for traversing the contents of a directory structure. After this function is called, any further attempts at traversing the contents of a directory will start over again from the first element.

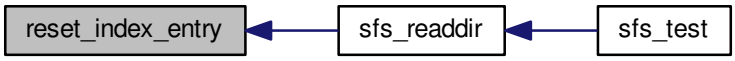
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.23.1.14 int unlink_inode_from_parent ( uint32_t *parent_location,* uint32_t *inode_location* )

Unlinks a child's inode from the parent's index structure.

This function works by accessing the specified parent inode, and accessing the index data structure that it links to within its defined inode data structure. After accessing this index structure, the block at each indexed location is loaded and the name attribute within the inode at that block is checked against the name within the specified child inode. Once a match is found, the corresponding indexed block is removed from the index structure. The index is rebuilt and written to disk, and the newly created index is linked to within the parent's inode. This inode is then rewritten at the original location, overwriting the old one. Upon completion of this operation the old index blocks are freed. If a failure occurs during this operation, the changes are automatically undone. The internal implementation of this function utilizes copy-on-write journalling in order to prevent any damage to the file system from failed operations.

**Parameters**

| | |
|---|---|
| *parent_location* | The parent's location on disk. |
| *inode_location* | The child inode's location on disk. |

**Returns**

Returns a value to signify success or failure. If the value is >= 0 the function was successful. If value is < 0, the function was unsuccessful.

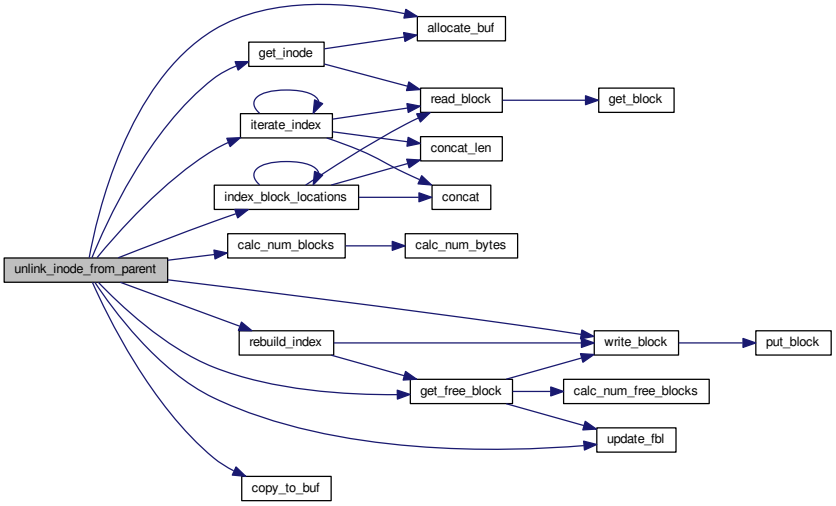
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

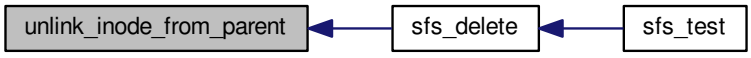
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



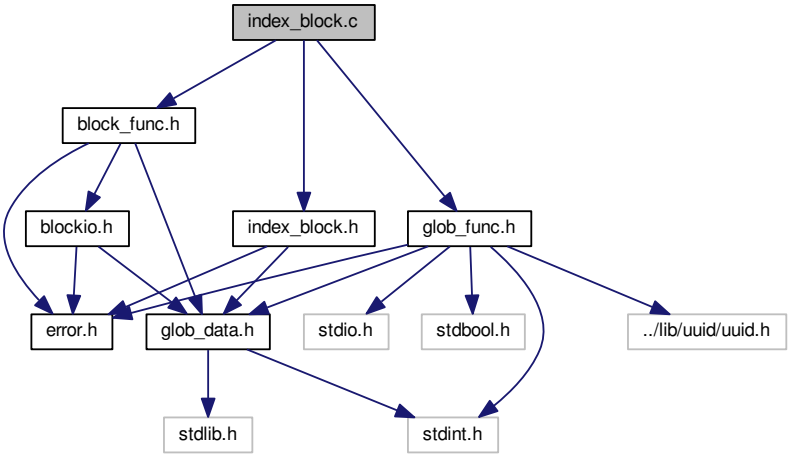

Here is the caller graph for this function:



## 5.24 index_block.c File Reference

```
#include "index_block.h"
#include "glob_func.h"
#include "block_func.h"
```

Include dependency graph for index_block.c:




### Functions

- data_index generate_index (uint32_t num_blocks)

  *Generates the indexes for the data blocks and then writes the indexes (which contain the locations of data blocks) to disk.*
- uint32_t rebuild_index (locations data_locations)

  *Takes a series of loose data block locations and builds a new index to link them with an inode.*
- locations iterate_index (uint32_t location, locations data_blocks)

  *This function takes an index block data structure and returns the locations of all data blocks which it links to.*
- uint32_t calc_index_blocks (uint32_t num_blocks)

  *Calculates the number of index blocks that are needed in order to write the number of data blocks specified to the file system.*
- int count_files_in_dir (uint32_t location)

  *Wrapper for iterate_index which simply gives you a count of the number of files in a directory.*
- locations index_block_locations (uint32_t location, locations index_blocks)

  *Provides the locations of the index blocks within an index block data structure.*

### 5.24.1 Function Documentation

#### 5.24.1.1 uint32_t calc_index_blocks ( uint32_t *num_blocks* )

Calculates the number of index blocks that are needed in order to write the number of data blocks specified to the file system.

This function uses a calculation based on the specified block size of the disk in order to determine how many locations can be indexed within one block, and then divides the number of specified blocks by this value. Since fractions of blocks are not allowed, the function then ceilings the resultant value in order to provide the full length of the required data structure.

**Parameters**

| | |
|---|---|
| *num_blocks* | The number of data blocks required to be indexed. |

**Returns**

Returns the number of index blocks needed to write the data blocks to the file system.

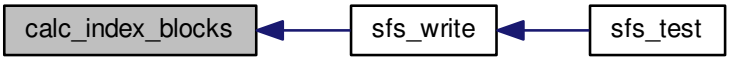
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.24.1.2 int count_files_in_dir ( uint32_t *location* )

Wrapper for iterate_index which simply gives you a count of the number of files in a directory.

**Parameters**

| | |
|---|---|
| *location* | The location of the index block on disk to iterate across. |

**Returns**

Returns the number of locations found in the index block. If return = 0 then the directory is empty. If return $>$ 0 then the function was successful, and if return = -1 then the function was unsuccessful.

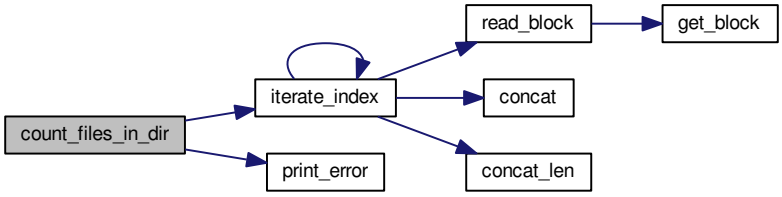
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

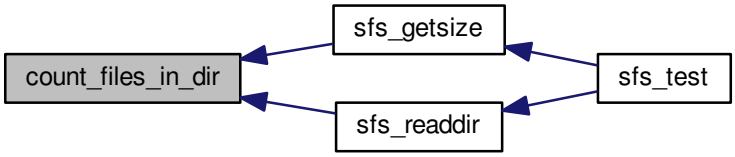
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.24.1.3 data_index generate_index ( uint32_t *num_blocks* )

Generates the indexes for the data blocks and then writes the indexes (which contain the locations of data blocks) to disk.

This function returns a data_index pseudo-object which contains the location of the first index block, and the locations of all the data blocks indexed by the all the indices (spanning the entire index data structure).

**Parameters**

| | |
|---|---|
| *num_blocks* | The number of blocks to generate indexes for. |

**Returns**

Returns a data_index struct which contains the location of the first index block, and the locations of all data blocks indexed

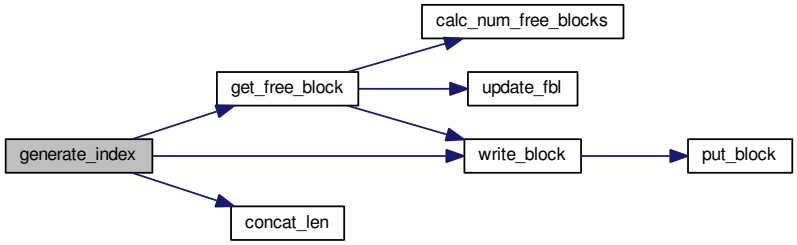
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

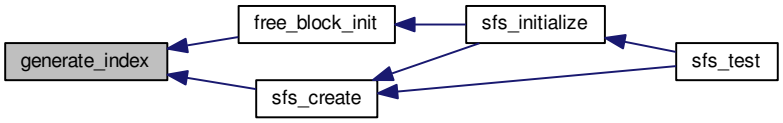
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.24.1.4 locations index_block_locations ( uint32_t *location*, locations *index_blocks* )

Provides the locations of the index blocks within an index block data structure.

This function iterates recursively through the index blocks in an index block data structure and returns a pointer to a NULL terminated array containing all of the index block locations on disk. If an error occurs then NULL is returned.

**Parameters**

| | |
|---|---|
| *location* | The location of the first index block. |
| *index_blocks* | An argument used by the function when it recursively calls itself, the argument should be NULL when calling index_block_locations for the first index location. |

**Returns**

Returns a list of index block locations. If there is an error the value returned will be NULL.

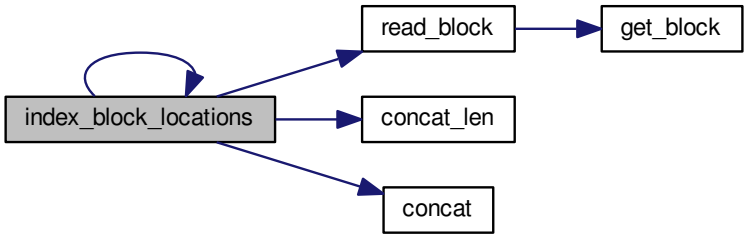
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

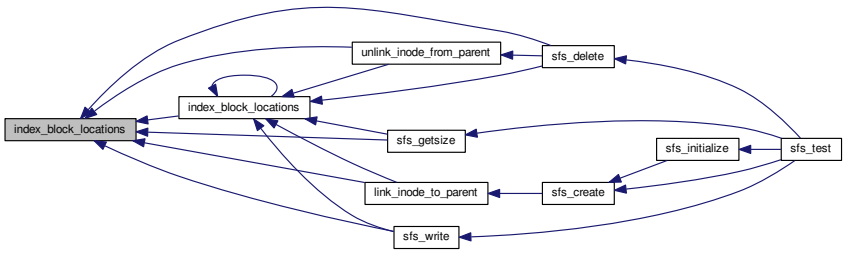
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.24.1.5 locations iterate_index ( uint32_t *location,* locations *data_blocks* )

This function takes an index block data structure and returns the locations of all data blocks which it links to.

This function iterates recursively through the index blocks and returns a pointer to a NULL terminated array containing all of the data block locations stored in the indices. In the case of large files that require multiple indexes, when we reach the value ceil(BLOCKSIZE/sizeof(uint32_t) which is the last entry in the index block, we recursively call iterate_index on the index block at this index. When we reach a NULL location inside the index block, we have reached the end of the indexes. The function returns a NULL pointer if an error occurred.

**Parameters**

| | |
|---|---|
| *location* | The location of the first index block to iterate through. |
| *data_blocks* | An argument used by the function when it recursively calls itself, the argument should be NULL when calling iterate_index for the first index location. |

**Returns**

Returns a pointer to a NULL terminated array containing all of the data block locations stored in the indices, the pointer is NULL if an error occurred.

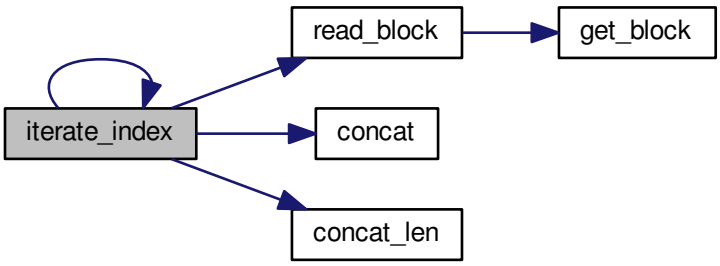
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

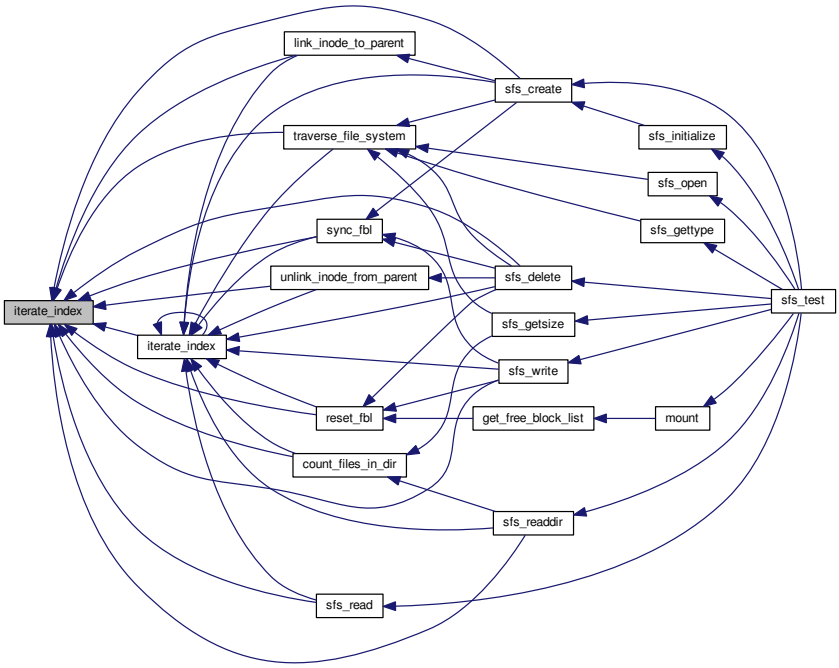
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.24.1.6 uint32_t rebuild_index ( locations *data_locations* )

Takes a series of loose data block locations and builds a new index to link them with an inode.

This function takes a NULL terminated array of data locations and turns it into a new index structure, then writes it on disk and returns the location of the first index block. The function operates similar to generate_index except you already have all the data locations. If an error occurs 0 is returned. If NO data_locations are specified an empty index is created and the location is returned.

**Parameters**

| | |
|---|---|
| *data_locations* | A NULL terminated array of data locations, if NO data locations are provided an empty index is created and the location is returned |

**Returns**

The location of the FIRST index block, if an error occurs 0 is returned

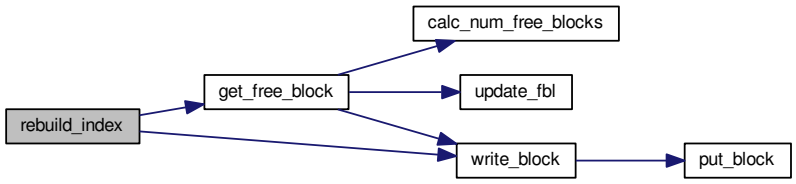
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

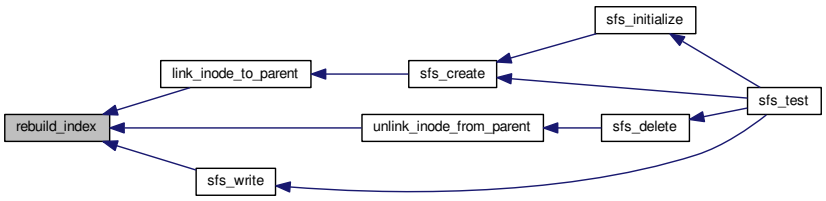
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



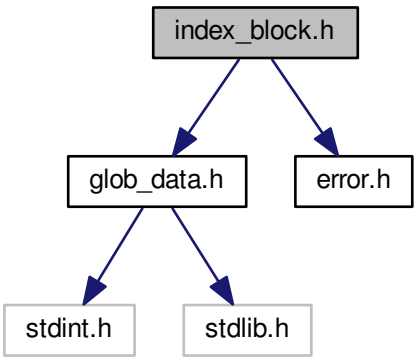

Here is the caller graph for this function:



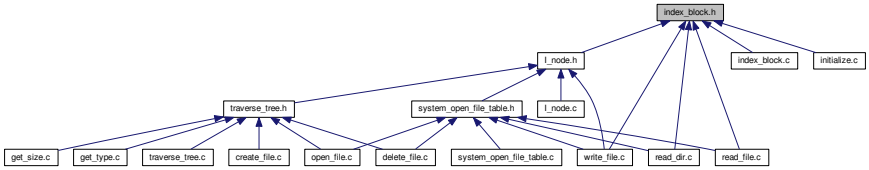

## 5.25 index_block.h File Reference

```
#include "glob_data.h"
#include "error.h"
```

Include dependency graph for index_block.h:




This graph shows which files directly or indirectly include this file:




## Data Structures

- struct data_index

## Typedefs

- typedef uint32_t * index

  *This type definition uses an array of 32-bit unsigned integers to represent index locations on disk.*

## Functions

- data_index generate_index (uint32_t num_blocks)

  *Generates the indexes for the data blocks and then writes the indexes (which contain the locations of data blocks) to disk.*
- uint32_t rebuild_index (locations data_locations)

  *Takes a series of loose data block locations and builds a new index to link them with an inode.*
- locations iterate_index (uint32_t location, locations data_blocks)

  *This function takes an index block data structure and returns the locations of all data blocks which it links to.*

- uint32_t calc_index_blocks (uint32_t num_blocks)

  *Calculates the number of index blocks that are needed in order to write the number of data blocks specified to the file system.*
- int count_files_in_dir (uint32_t location)

  *Wrapper for iterate_index which simply gives you a count of the number of files in a directory.*
- locations index_block_locations (uint32_t location, locations index_blocks)

  *Provides the locations of the index blocks within an index block data structure.*

### 5.25.1 Typedef Documentation

#### 5.25.1.1 index

This type definition uses an array of 32-bit unsigned integers to represent index locations on disk.

This type definition represents an index block data structure. It is supported by pseudo-object oriented methods in order to return arrays of variables to functions by way of pointer arithmetic based logical segmentations of data types in memory.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

### 5.25.2 Function Documentation

#### 5.25.2.1 uint32_t calc_index_blocks ( uint32_t *num_blocks* )

Calculates the number of index blocks that are needed in order to write the number of data blocks specified to the file system.

This function uses a calculation based on the specified block size of the disk in order to determine how many locations can be indexed within one block, and then divides the number of specified blocks by this value. Since fractions of blocks are not allowed, the function then ceilings the resultant value in order to provide the full length of the required data structure.

**Parameters**

| | |
|---|---|
| *num_blocks* | The number of data blocks required to be indexed. |

**Returns**

Returns the number of index blocks needed to write the data blocks to the file system.

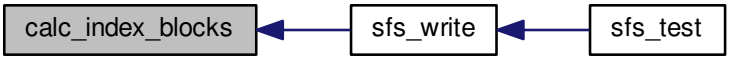
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.25.2.2 int count_files_in_dir ( uint32_t *location* )

Wrapper for iterate_index which simply gives you a count of the number of files in a directory.

**Parameters**

| | |
|---|---|
| *location* | The location of the index block on disk to iterate across. |

**Returns**

Returns the number of locations found in the index block. If return = 0 then the directory is empty. If return $>$ 0 then the function was successful, and if return = -1 then the function was unsuccessful.

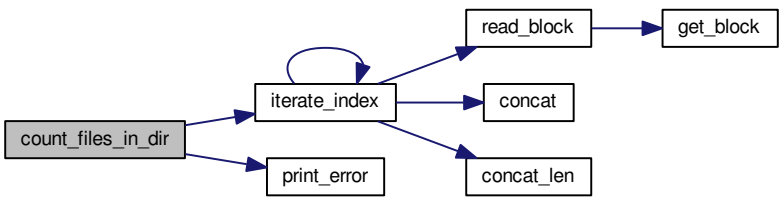
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

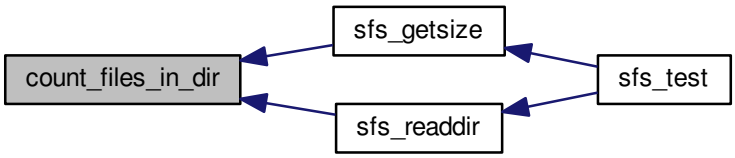
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.25.2.3 data_index generate_index ( uint32_t *num_blocks* )

Generates the indexes for the data blocks and then writes the indexes (which contain the locations of data blocks) to disk.

This function returns a data_index pseudo-object which contains the location of the first index block, and the locations of all the data blocks indexed by the all the indices (spanning the entire index data structure).

**Parameters**

| | |
|---|---|
| *num_blocks* | The number of blocks to generate indexes for. |

**Returns**

Returns a data_index struct which contains the location of the first index block, and the locations of all data blocks indexed

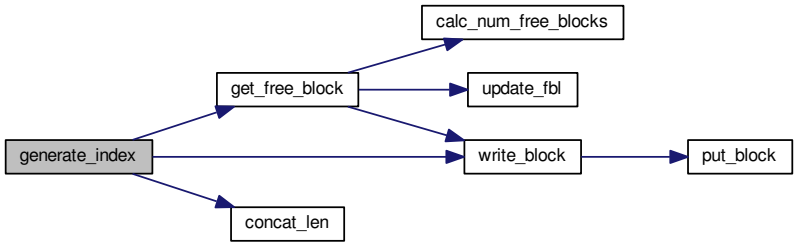
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

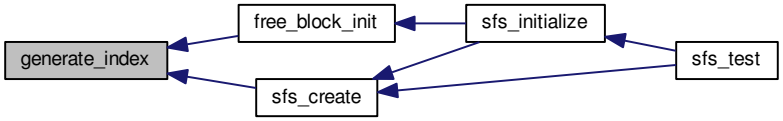
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.25.2.4 locations index_block_locations ( uint32_t *location*, locations *index_blocks* )

Provides the locations of the index blocks within an index block data structure.

This function iterates recursively through the index blocks in an index block data structure and returns a pointer to a NULL terminated array containing all of the index block locations on disk. If an error occurs then NULL is returned.

**Parameters**

| | |
|---|---|
| *location* | The location of the first index block. |
| *index_blocks* | An argument used by the function when it recursively calls itself, the argument should be NULL when calling index_block_locations for the first index location. |

**Returns**

Returns a list of index block locations. If there is an error the value returned will be NULL.

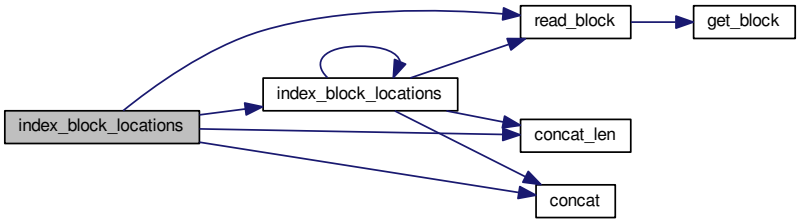
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

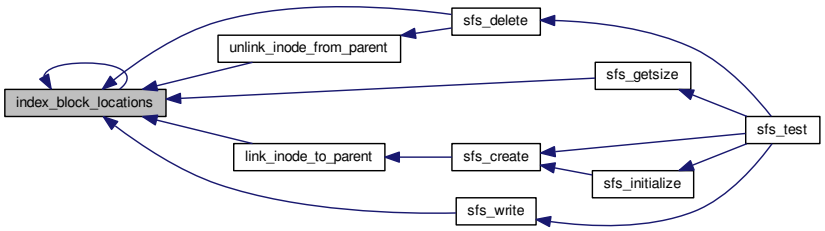
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.25.2.5 locations iterate_index ( uint32_t *location*, locations *data_blocks* )

This function takes an index block data structure and returns the locations of all data blocks which it links to.

This function iterates recursively through the index blocks and returns a pointer to a NULL terminated array containing all of the data block locations stored in the indices. In the case of large files that require multiple indexes, when we reach the value ceil(BLOCKSIZE/sizeof(uint32_t) which is the last entry in the index block, we recursively call iterate_index on the index block at this index. When we reach a NULL location inside the index block, we have reached the end of the indexes. The function returns a NULL pointer if an error occurred.

**Parameters**

| | |
|---|---|
| *location* | The location of the first index block to iterate through. |
| *data_blocks* | An argument used by the function when it recursively calls itself, the argument should be NULL when calling iterate_index for the first index location. |

**Returns**

Returns a pointer to a NULL terminated array containing all of the data block locations stored in the indices, the pointer is NULL if an error occurred.

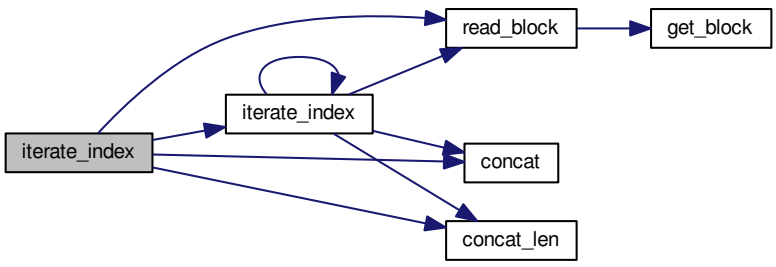
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

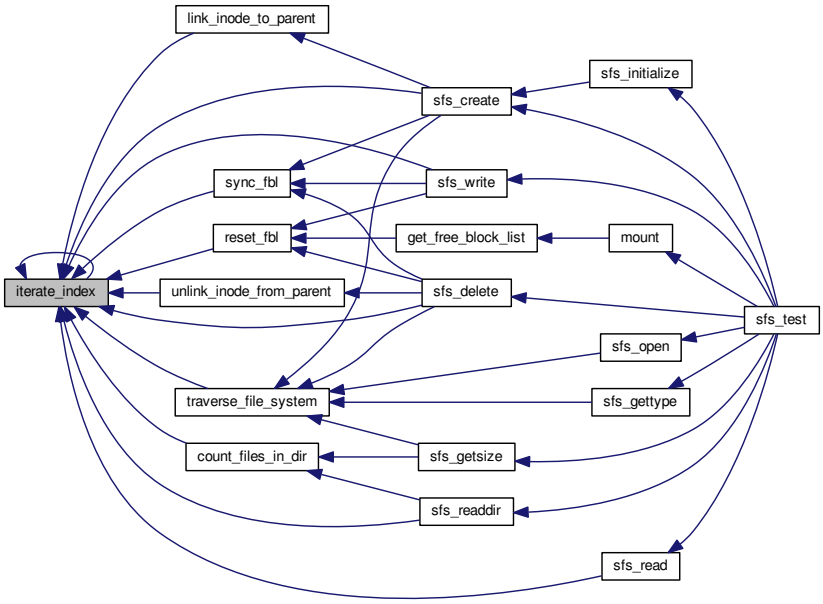
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.25.2.6 uint32_t rebuild_index ( locations *data_locations* )

Takes a series of loose data block locations and builds a new index to link them with an inode.

This function takes a NULL terminated array of data locations and turns it into a new index structure, then writes it on disk and returns the location of the first index block. The function operates similar to generate_index except you already have all the data locations. If an error occurs 0 is returned. If NO data_locations are specified an empty index is created and the location is returned.

**Parameters**

| | |
|---|---|
| *data_locations* | A NULL terminated array of data locations, if NO data locations are provided an empty index is created and the location is returned |

**Returns**

The location of the FIRST index block, if an error occurs 0 is returned

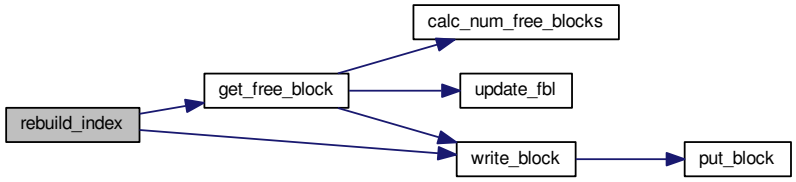
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

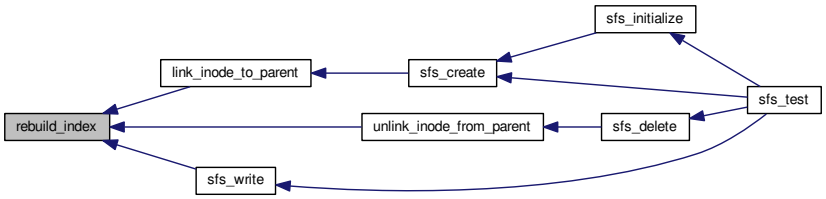
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



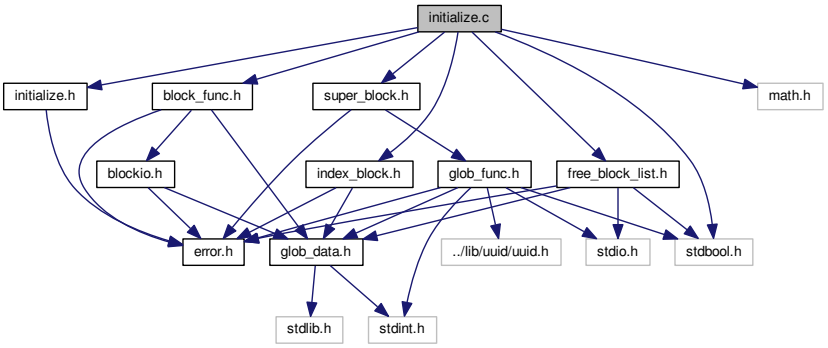

Here is the caller graph for this function:




## 5.26 initialize.c File Reference

```
#include "initialize.h"
#include "block_func.h"
#include "super_block.h"
#include "free_block_list.h"
#include "index_block.h"
#include <math.h>
#include <stdbool.h>
```

Include dependency graph for initialize.c:




## Functions

- int sfs_initialize (int erase)

  *Initialize the superblock for the file system.*
- int free_block_init (void)

  *Allocate all of the blocks in the free block list.*
- int wipe_disk (void)

  *Wipes the drive block by block.*

## Variables

- uint32_t FBL_DATA_SIZE

  *The size of the free block list data blocks, not including the overhead of the size of each index block needed to index the free block list data blocks.*
- uint32_t FBL_TOTAL_SIZE

  *The total size of the free block list data blocks, including the overhead of the size of each index block needed to index the free block list data blocks.*
- uint32_t ROOT

  *The default location for the root directory's inode.*

### 5.26.1 Function Documentation

#### 5.26.1.1 int free_block_init ( void )

Allocate all of the blocks in the free block list.

This function initializes the free block list data structure for the first time. This is used only when the disk is being reinitialized. In effect, writing all NULL values to this data structure (save for the superblock) marks all values on the disk free to be written on.

**Returns**

Returns an integer value, if the free block list initialization fails the value will be -1. Otherwise, it will be 0.

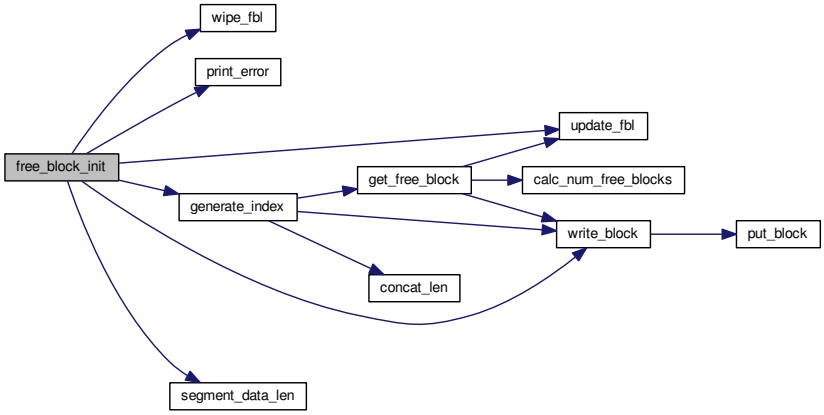
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

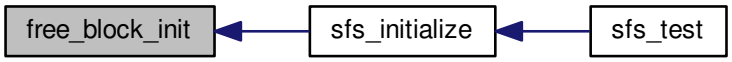
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.26.1.2 int sfs_initialize ( int *erase* )

Initialize the superblock for the file system.

**Parameters**

| | |
|---|---|
| *erase* | Determines whether or not to delete the contents of the file system. If this value is 1 then it will erase every block on the disk and then re-create the super block, free block list data structure and the root directory. If the value is 0 then it will re-create the super block, the free block list blocks and the root directory. |

**Returns**

Returns an integer value, if the value $>$ 0 then the initialization was successful. If the value $<=$ 0 then the initialization failed.

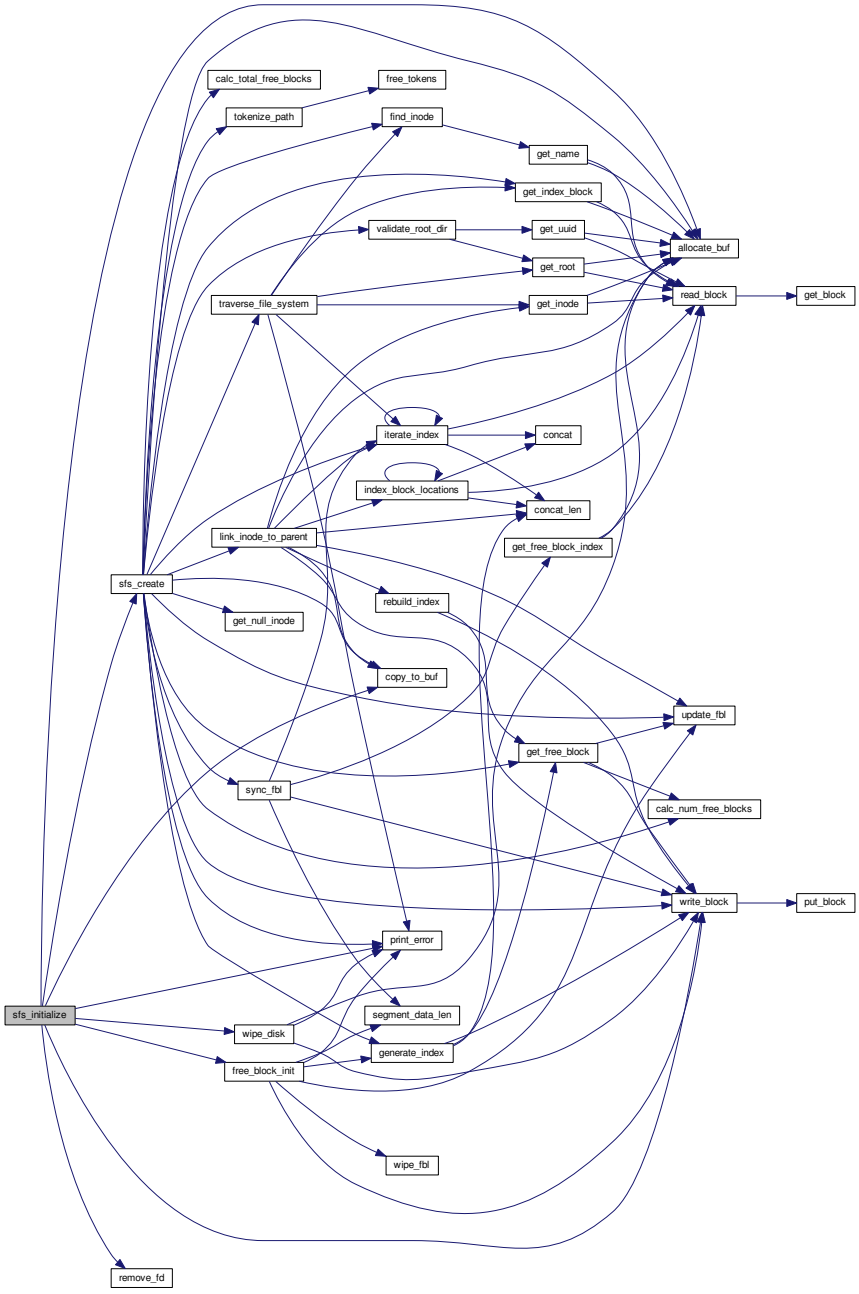
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

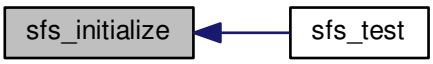
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.26.1.3 int wipe_disk ( void )

Wipes the drive block by block.

This function ensures that every block on the disk is initialized to zero. Failing to wipe the disk when the file system is reinitialized will result in having garbage data residing in all the blocks that have not yet been written to. These blocks are marked as free in the free block list structure regardless, so they are simple overwritten and the data inside is ignored. This is analogous to a "quick format" command.

**Returns**

Returns an integer value. If the wipe fails the value will be -1, otherwise it will be 0.

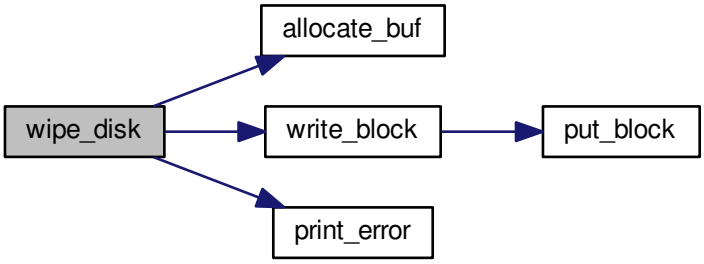
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

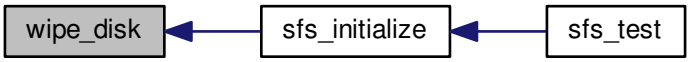
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




### 5.26.2 Variable Documentation

#### 5.26.2.1 uint32_t FBL_DATA_SIZE

The size of the free block list data blocks, not including the overhead of the size of each index block needed to index the free block list data blocks.

#### 5.26.2.2 uint32_t FBL_TOTAL_SIZE

The total size of the free block list data blocks, including the overhead of the size of each index block needed to index the free block list data blocks.

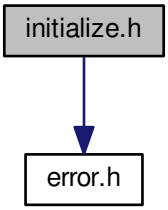
#### 5.26.2.3 uint32_t ROOT

The default location for the root directory's inode.

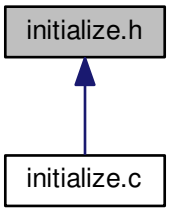
## 5.27 initialize.h File Reference

```
#include "error.h"
```

Include dependency graph for initialize.h:



This graph shows which files directly or indirectly include this file:




## Functions

- int sfs_initialize (int erase)

  *Initialize the superblock for the file system.*
- int free_block_init (void)

  *Allocate all of the blocks in the free block list.*
- int wipe_disk (void)

  *Wipes the drive block by block.*

### 5.27.1 Function Documentation

#### 5.27.1.1 int free_block_init ( void )

Allocate all of the blocks in the free block list.

This function initializes the free block list data structure for the first time. This is used only when the disk is being reinitialized. In effect, writing all NULL values to this data structure (save for the superblock) marks all values on the disk free to be written on.

**Returns**

Returns an integer value, if the free block list initialization fails the value will be -1. Otherwise, it will be 0.

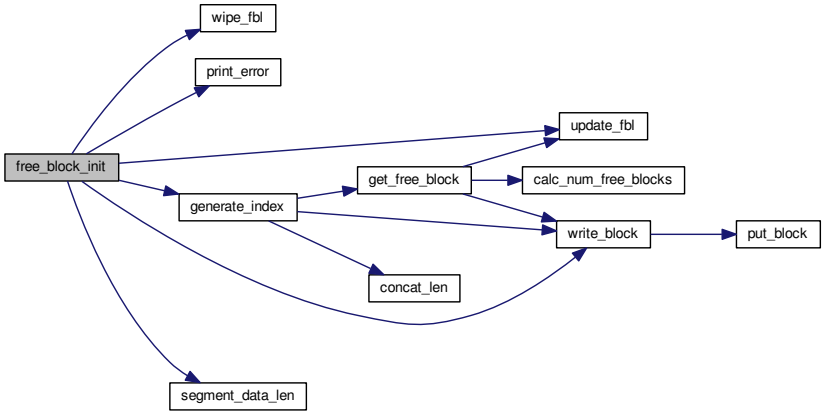
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

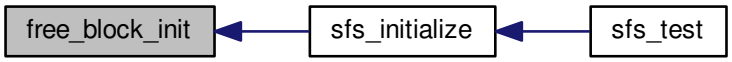
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.27.1.2 int sfs_initialize ( int *erase* )

Initialize the superblock for the file system.

**Parameters**

| | |
|---|---|
| *erase* | Determines whether or not to delete the contents of the file system. If this value is 1 then it will erase every block on the disk and then re-create the super block, free block list data structure and the root directory. If the value is 0 then it will re-create the super block, the free block list blocks and the root directory. |

**Returns**

Returns an integer value, if the value $>$ 0 then the initialization was successful. If the value $<=$ 0 then the initialization failed.

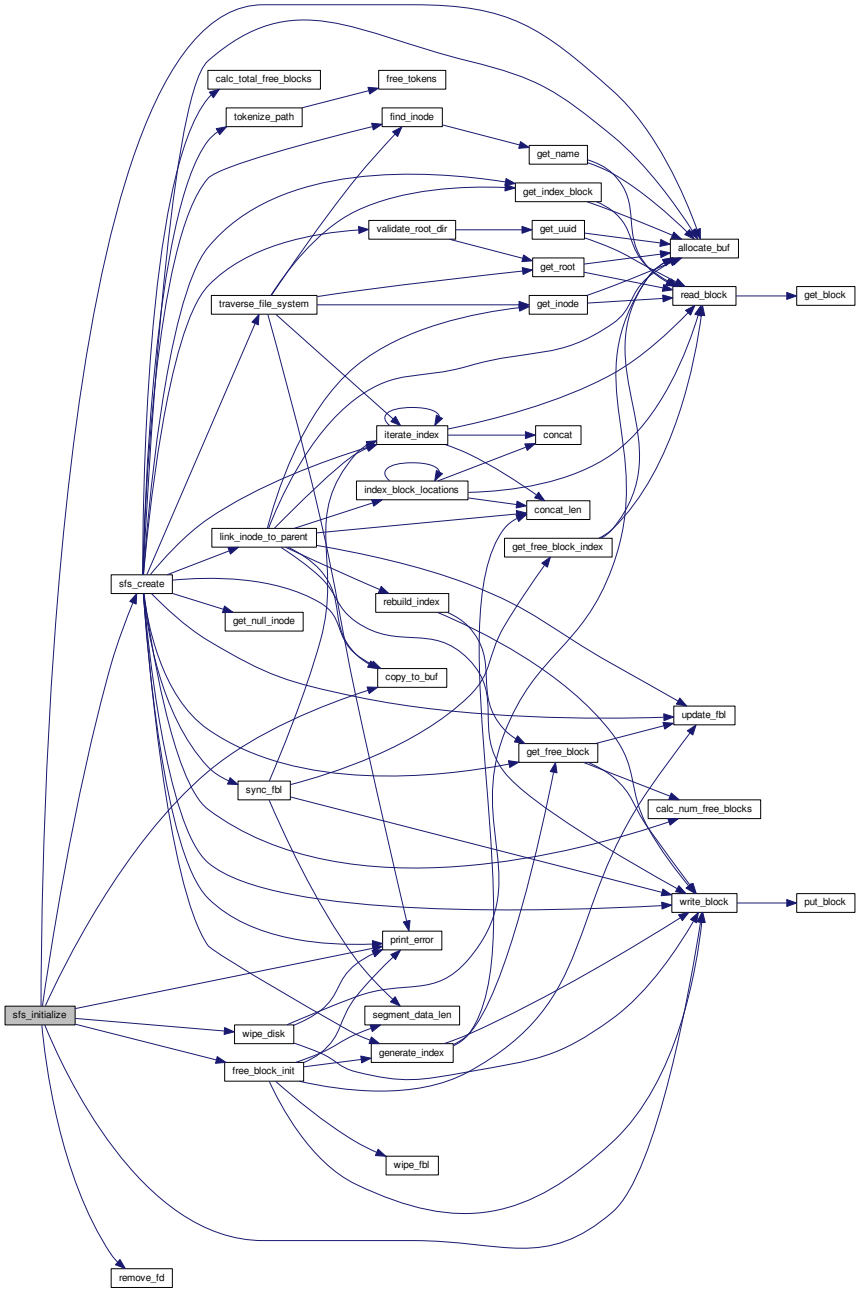
### Author

Daniel Smullen
Jonathan Gillett
Joseph Heron

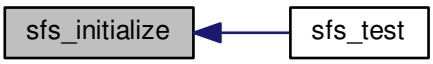
### Copyright

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.27.1.3 int wipe_disk ( void )

Wipes the drive block by block.

This function ensures that every block on the disk is initialized to zero. Failing to wipe the disk when the file system is reinitialized will result in having garbage data residing in all the blocks that have not yet been written to. These blocks are marked as free in the free block list structure regardless, so they are simple overwritten and the data inside is ignored. This is analogous to a "quick format" command.

**Returns**

Returns an integer value. If the wipe fails the value will be -1, otherwise it will be 0.

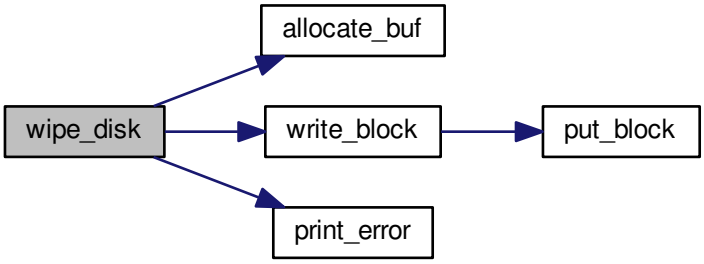
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

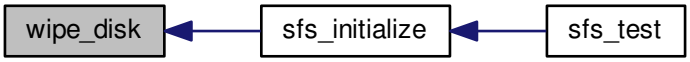
**Copyright**

GNU General Public License V3

Here is the call graph for this function:

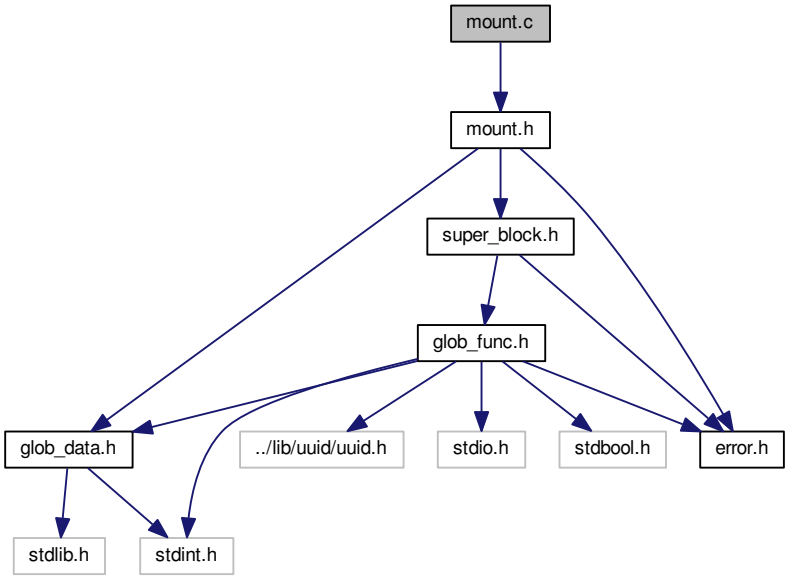


Here is the caller graph for this function:




## 5.28 mount.c File Reference

`#include "mount.h"`
Include dependency graph for mount.c:




### Functions

- int mount (void)

  *Attempt to mount the filesystem on disk.*
- int validate_super_block (void)

  *Validate the super block on the disk.*
- int validate_root_dir (void)

  *Validate that there is a root directory on the disk.*

## Variables

- char * INVALID_UUID = "Invalid UUID.\n"

### 5.28.1 Function Documentation

#### 5.28.1.1 int mount ( void )

Attempt to mount the filesystem on disk.

This funcion calls validation methods to ensure that the mount is possible, and file system isn't corrupted. It verifies that the unique identifiers and checksums associated with the various structures on disk are valid and in working order.

**Returns**

Returns an integer value, if value = 1 the file system has passed validation. If value = -1 the file system has failed to validate and should be reinitialized.

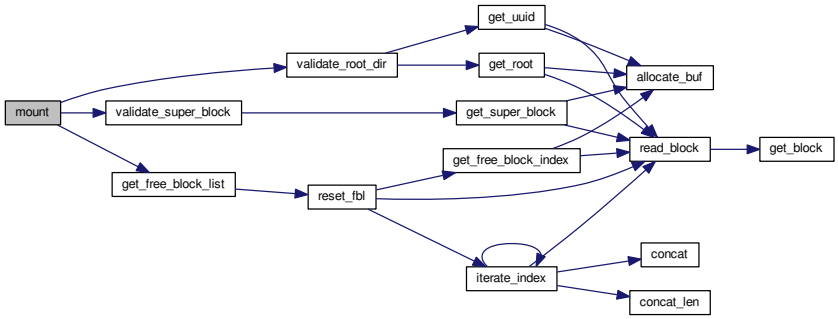
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:


#### 5.28.1.2 int validate_root_dir ( void )

Validate that there is a root directory on the disk.

This function checks the location for the root directory on disk specified by the super block, and determines if the file at that location is valid.

**Returns**

Returns an integer value, if value = 1 the root directory structure which exists on disk passed validation. If value = -1, the structure failed validation.

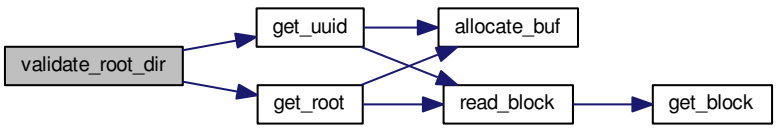
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

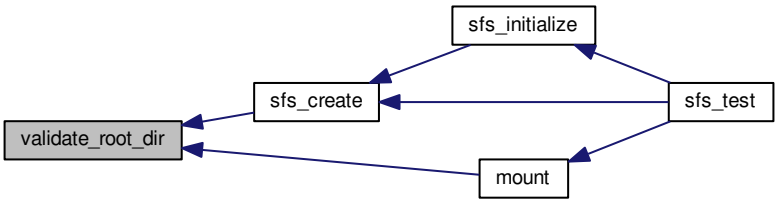
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.28.1.3 int validate_super_block ( void )

Validate the super block on the disk.

This function determines the integrity of the super block data by validating its unique identifier and checksum.

**Returns**

Returns an integer value. If value = 1 the super block has passed validation. If the value = -1, the super block has failed validation and should be reinitialized.

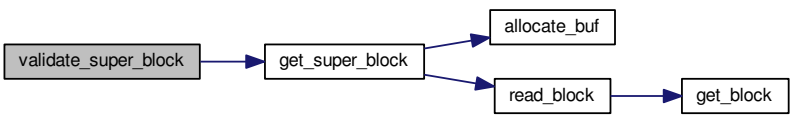
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

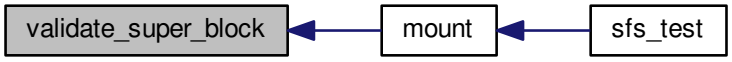
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:



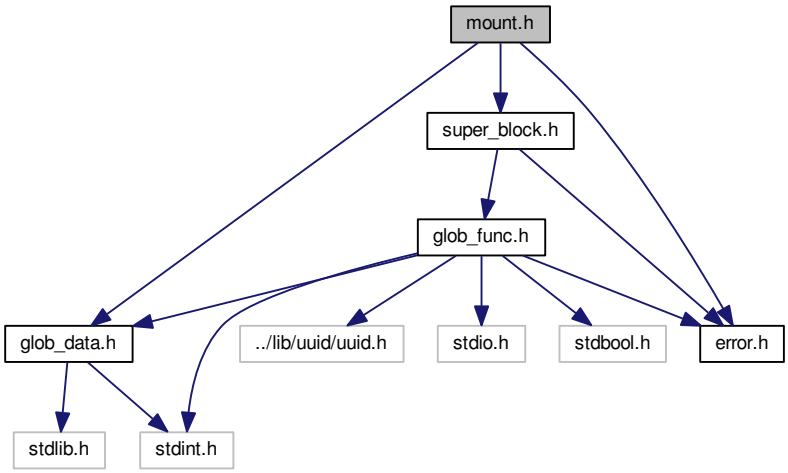

### 5.28.2 Variable Documentation

#### 5.28.2.1 char* INVALID_UUID = "Invalid UUID.\n"

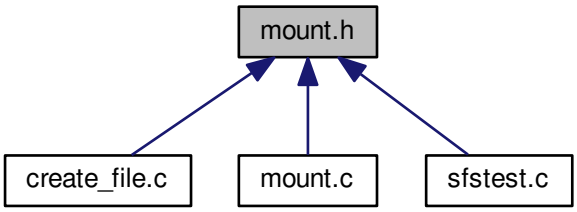
## 5.29 mount.h File Reference

```
#include "glob_data.h"
#include "super_block.h"
#include "error.h"
```

Include dependency graph for mount.h:



This graph shows which files directly or indirectly include this file:




## Functions

- int mount (void)

  *Attempt to mount the filesystem on disk.*
- int validate_super_block (void)

  *Validate the super block on the disk.*
- int validate_root_dir (void)

  *Validate that there is a root directory on the disk.*

### 5.29.1 Function Documentation

#### 5.29.1.1 int mount ( void )

Attempt to mount the filesystem on disk.

This funcion calls validation methods to ensure that the mount is possible, and file system isn't corrupted. It verifies that the unique identifiers and checksums associated with the various structures on disk are valid and in working order.

**Returns**

Returns an integer value, if value = 1 the file system has passed validation. If value = -1 the file system has failed to validate and should be reinitialized.

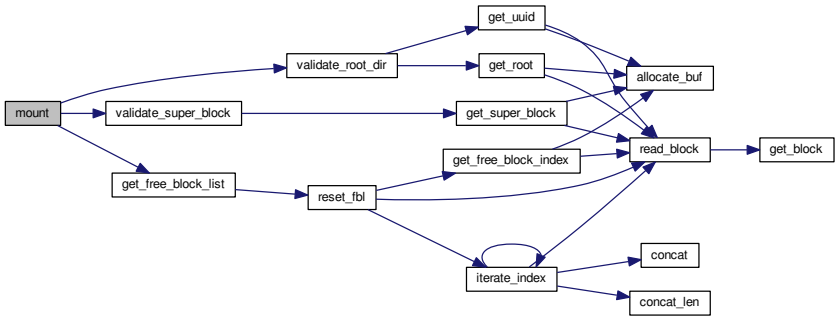
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:


#### 5.29.1.2 int validate_root_dir ( void )

Validate that there is a root directory on the disk.

This function checks the location for the root directory on disk specified by the super block, and determines if the file at that location is valid.

**Returns**

Returns an integer value, if value = 1 the root directory structure which exists on disk passed validation. If value = -1, the structure failed validation.

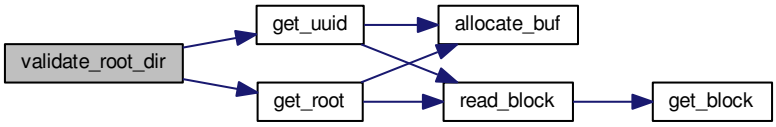
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

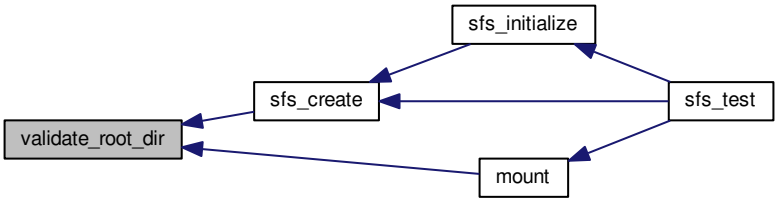
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.29.1.3 int validate_super_block ( void )

Validate the super block on the disk.

This function determines the integrity of the super block data by validating its unique identifier and checksum.

**Returns**

Returns an integer value. If value = 1 the super block has passed validation. If the value = -1, the super block has failed validation and should be reinitialized.

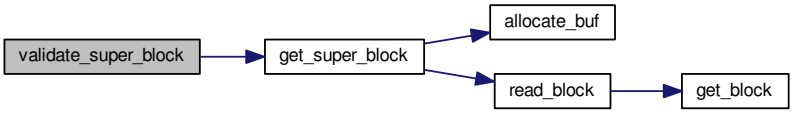
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

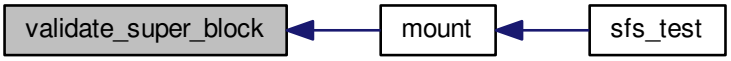
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



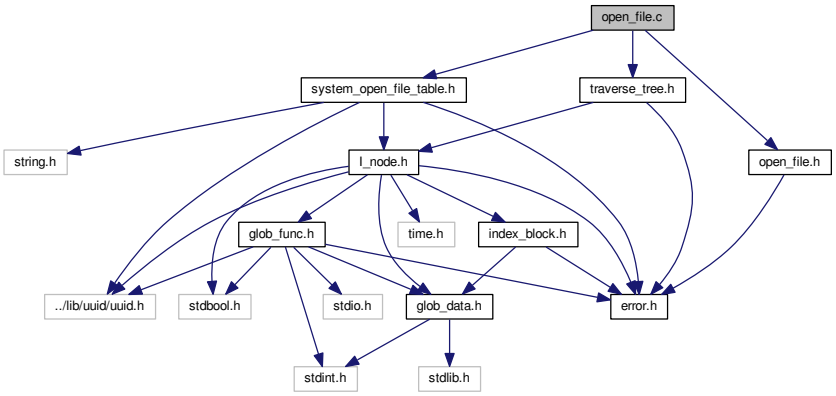

Here is the caller graph for this function:




## 5.30 open_file.c File Reference

```
#include "system_open_file_table.h"
#include "traverse_tree.h"
#include "open_file.h"
```

Include dependency graph for open_file.c:



## Functions

- int sfs_open (char *pathname)

  *Opens the file specified by the pathname, if the file is successfully opened a file descriptor is returned.*
- int show_information (int fd)

  *Show all of the information stored in the given file descriptor.*

### 5.30.1 Function Documentation

#### 5.30.1.1 int sfs_open ( char * *pathname* )

Opens the file specified by the pathname, if the file is successfully opened a file descriptor is returned.

Since file descriptors are simply copies of inodes, this function traverses the specified path and attempts to find the inode with a name that matches the one specified in the path. This is done by traversing the index blocks at each token in the path specified, and checking the name of the path token against the inodes at each location specified by the index. Once the inode with the name matching the final path token has been found, the inode pseudo-object returned is added to the system-wide open file table.

**Parameters**

| | |
|---|---|
| *pathname* | The full pathname of a file. |

**Returns**

Returns a file descriptor for the file opened. If the fd >= 0 then the file has successfully opened. If the fd < 0 then the file has failed to open.

**Exceptions**

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the specified path does not exist. |
| *INVALID_PATH* | If the specified path is invalid. |

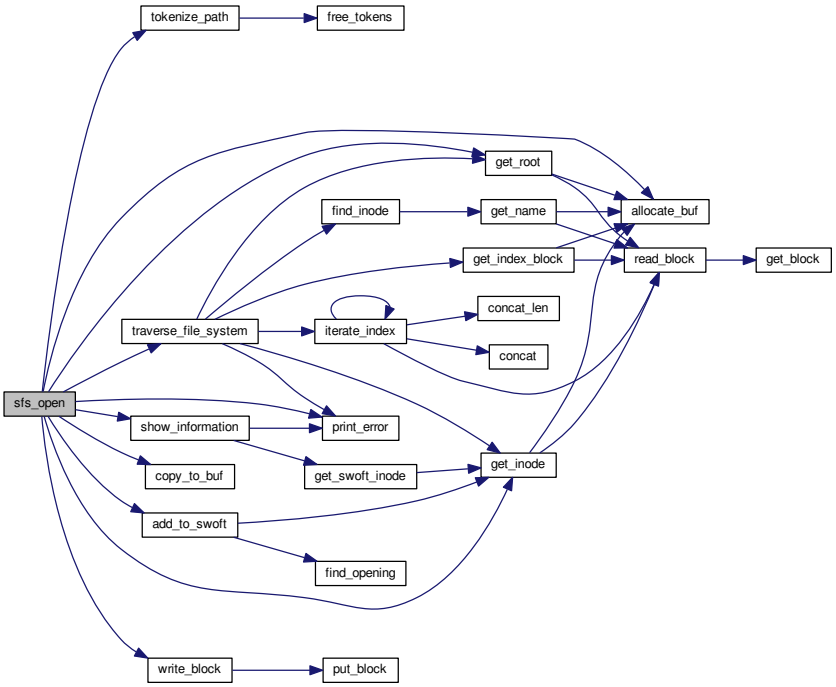
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:


#### 5.30.1.2 int show_information ( int *fd* )

Show all of the information stored in the given file descriptor.

Since file descriptors are simply inode pseudo-objects, this function simply prints the attributes associated with an inode's defined data structure.

**Parameters**

| | |
|---|---|
| *fd* | integer, the file descriptor. |

**Returns**

Returns an integer value. The function is adjusted to output the file descriptor if the file descriptor is retrieved successfully, to allow for it to be used on the return line. If the value returned is >= 0 then the function successfully retrieved a file descriptor. If the value < 0 then the function was unsuccessful.

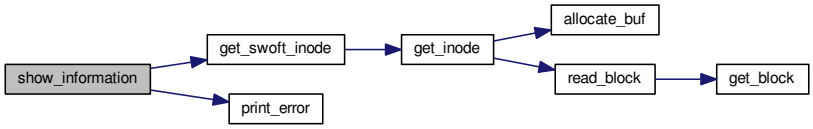
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

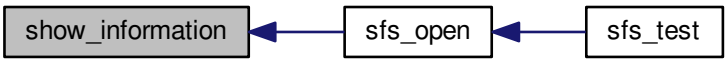
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



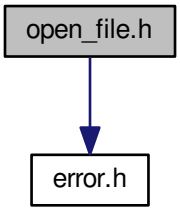

Here is the caller graph for this function:



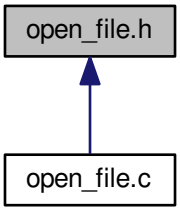

## 5.31 open_file.h File Reference

```
#include "error.h"
```

Include dependency graph for open_file.h:




This graph shows which files directly or indirectly include this file:




## Functions

- int sfs_open (char *pathname)

  *Opens the file specified by the pathname, if the file is successfully opened a file descriptor is returned.*
- int show_information (int fd)

  *Show all of the information stored in the given file descriptor.*

### 5.31.1 Function Documentation

#### 5.31.1.1 int sfs_open ( char * *pathname* )

Opens the file specified by the pathname, if the file is successfully opened a file descriptor is returned.

Since file descriptors are simply copies of inodes, this function traverses the specified path and attempts to find the inode with a name that matches the one specified in the path. This is done by traversing the index blocks at each token in the path specified, and checking the name of the path token against the inodes at each location specified by the index. Once the inode with the name matching the final path token has been found, the inode pseudo-object returned is added to the system-wide open file table.

### Parameters

| | |
|---|---|
| *pathname* | The full pathname of a file. |

### Returns

Returns a file descriptor for the file opened. If the fd >= 0 then the file has successfully opened. If the fd < 0 then the file has failed to open.

### Exceptions

| | |
|---|---|
| *FILE_NOT_FOUND* | If the file at the specified path does not exist. |
| *INVALID_PATH* | If the specified path is invalid. |

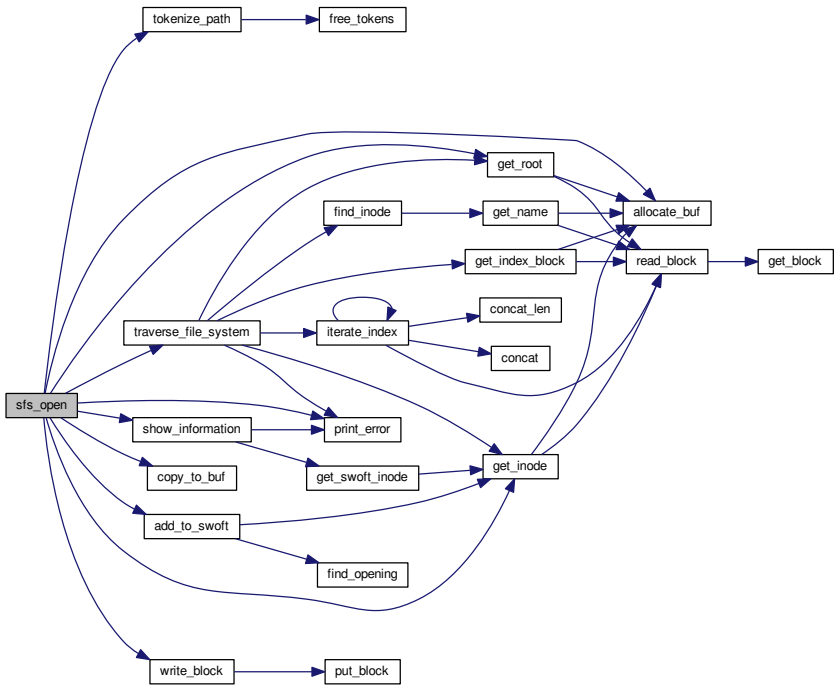
### Author

Daniel Smullen
Jonathan Gillett
Joseph Heron

### Copyright

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:


#### 5.31.1.2 int show_information ( int *fd* )

Show all of the information stored in the given file descriptor.

Since file descriptors are simply inode pseudo-objects, this function simply prints the attributes associated with an inode's defined data structure.

**Parameters**

| | |
|---|---|
| *fd* | integer, the file descriptor. |

**Returns**

Returns an integer value. The function is adjusted to output the file descriptor if the file descriptor is retrieved successfully, to allow for it to be used on the return line. If the value returned is >= 0 then the function successfully retrieved a file descriptor. If the value < 0 then the function was unsuccessful.

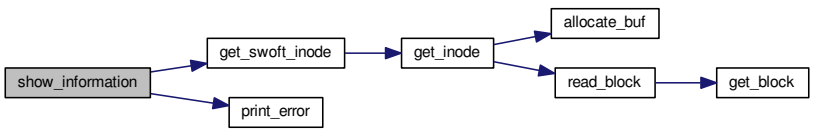
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

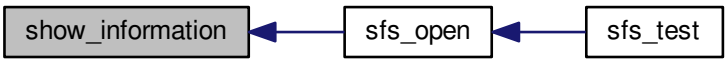
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



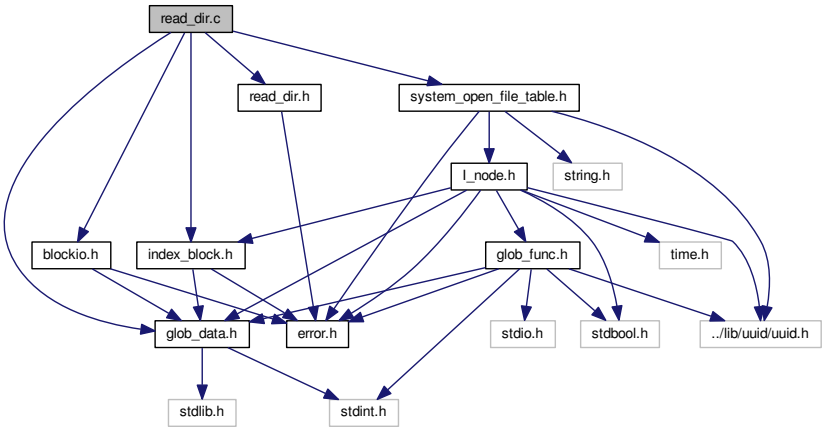
Here is the caller graph for this function:




## 5.32 read_dir.c File Reference

```
#include "glob_data.h"
#include "blockio.h"
#include "index_block.h"
#include "system_open_file_table.h"
#include "read_dir.h"
```

Include dependency graph for read_dir.c:




### Functions

- int sfs_readdir (int fd, char ∗mem_pointer)

  *List the contents of a directory.*

### 5.32.1 Function Documentation

#### 5.32.1.1 int sfs_readdir ( int *fd*, char ∗ *mem_pointer* )

List the contents of a directory.

Reads the file name components from a directory file. This is done by reading in all the locations from within a directory file's index structure and analyzing the inodes at each location. The first time sfs_readdir is called, the first file name component in the directory will be placed into memory at the location pointed to by mem_pointer. Each successive call to sfs_readdir will place the next name component from the directory into the mem_pointer buffer. When all names have been returned, sfs_readdir will place nothing in the buffer, and return a value of zero to indicate the directory has been completely scanned.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to read data from. |
| *mem_pointer* | The memory location to store the file name components. |

**Returns**

Returns an integer value. If the value $>$ 0 reading the directory was successful. If the value = 0, there is nothing contents in the directory. If the value $<$ 0 the directory read operation was unsuccessful.

**Exceptions**

| | |
|---|---|
| *INVALID_FILE_DESCRIPTOR* | If the file descriptor specified does not exist. |

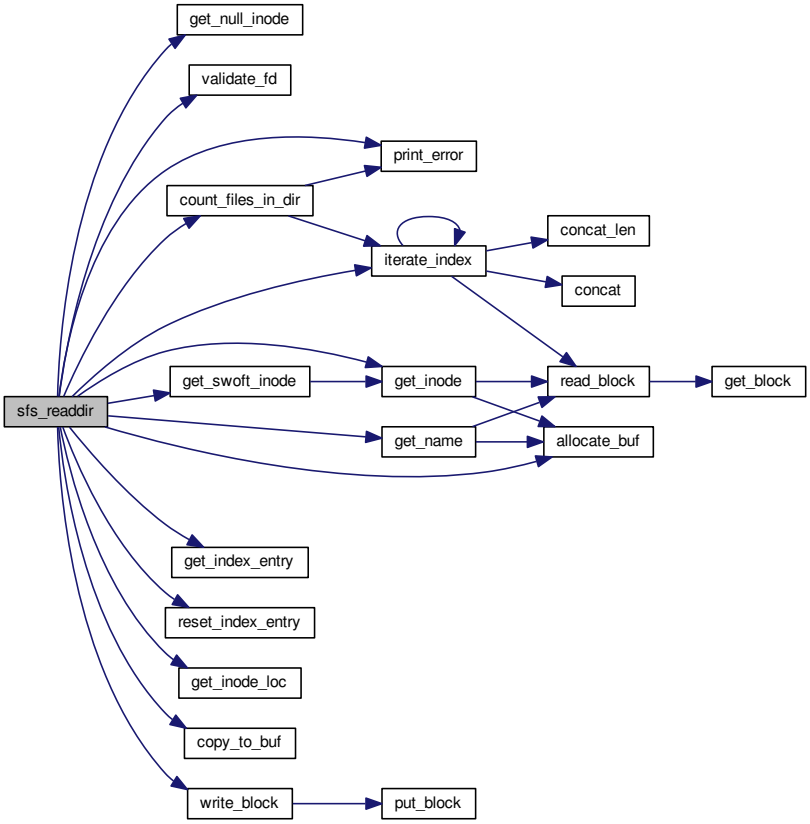
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

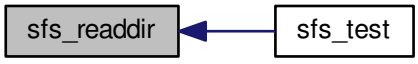
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



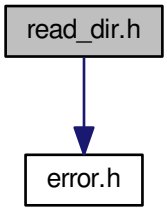

Here is the caller graph for this function:



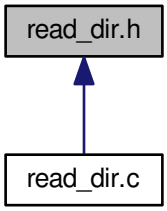

## 5.33 read_dir.h File Reference

```
#include "error.h"
```

Include dependency graph for read_dir.h:




This graph shows which files directly or indirectly include this file:




## Functions

- int sfs_readdir (int fd, char ∗mem_pointer)

  *List the contents of a directory.*

### 5.33.1 Function Documentation

#### 5.33.1.1 int sfs_readdir ( int *fd,* char ∗ *mem_pointer* )

List the contents of a directory.

Reads the file name components from a directory file. This is done by reading in all the locations from within a directory file's index structure and analyzing the inodes at each location. The first time sfs_readdir is called, the first file name component in the directory will be placed into memory at the location pointed to by mem_pointer. Each successive call to sfs_readdir will place the next name component from the directory into the mem_pointer buffer. When all names have been returned, sfs_readdir will place nothing in the buffer, and return a value of zero to indicate the directory has been completely scanned.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to read data from. |
| *mem_pointer* | The memory location to store the file name components. |

**Returns**

Returns an integer value. If the value $>$ 0 reading the directory was successful. If the value = 0, there is nothing contents in the directory. If the value $<$ 0 the directory read operation was unsuccessful.

**Exceptions**

| | |
|---|---|
| *INVALID_FILE_DESCRIPTOR* | If the file descriptor specified does not exist. |

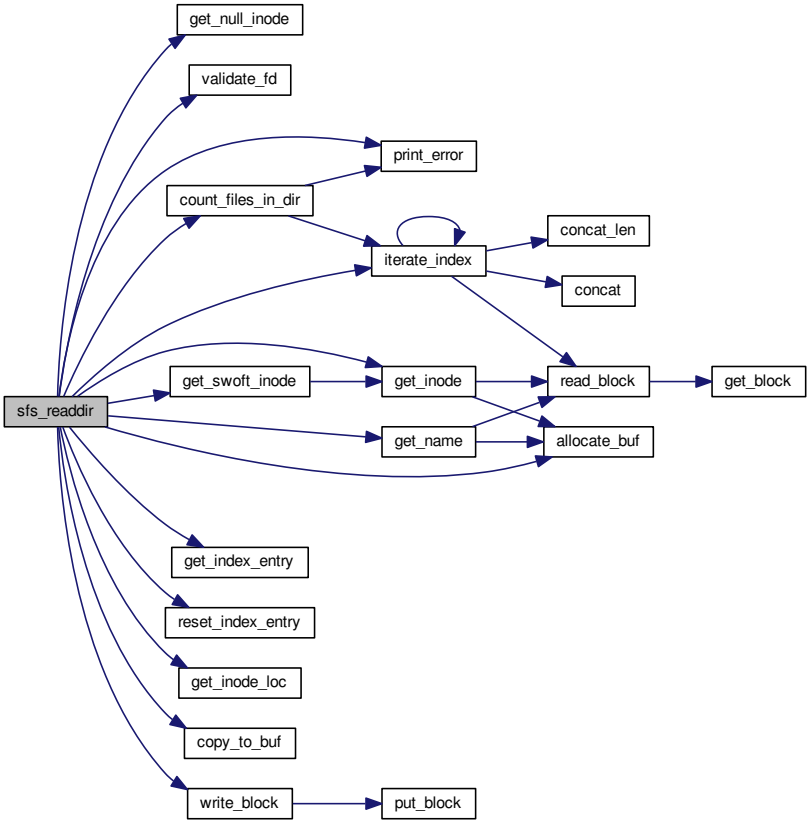
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

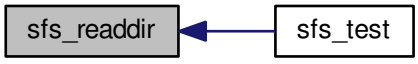
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:

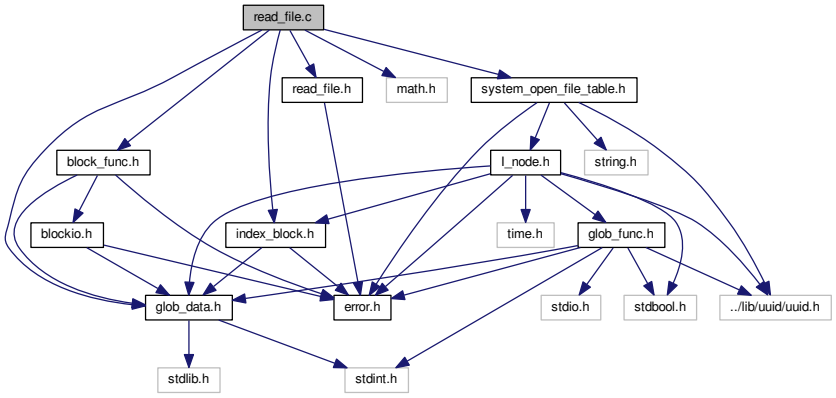



## 5.34 read_file.c File Reference

```
#include "glob_data.h"
```

```
#include "block_func.h"
#include "index_block.h"
#include "system_open_file_table.h"
#include <math.h>
#include "read_file.h"
```

Include dependency graph for read_file.c:




## Functions

- int sfs_read (int fd, int start, int length, byte *mem_pointer)

  *Read the contents of a file.*

### 5.34.1 Function Documentation

#### 5.34.1.1 int sfs_read ( int *fd*, int *start*, int *length*, byte * *mem_pointer* )

Read the contents of a file.

Copy the length of bytes of data specified from a regular file to memory location specified by mem_pointer. The parameter start gives the offset of the first byte in the file that should be copied.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to read data from. |
| *start* | The offset of the first byte in the file that should be copied. |
| *length* | The length of bytes of data to copy from the file. |
| *mem_pointer* | The memory location to store the data read from the file. |

**Returns**

Returns an integer value, if the value > 0 then the file was read successfully. If the value <= 0 then the file failed to be read.

### Exceptions

| | |
|---|---|
| *FILE_LENGTH_OVERRUN* | If the file cannot be read correctly because the length of bytes specified exceeds the length of the file, no data will be copied from the file if an overrun occurs. |
| *INVALID_FILE_DESCRIPTOR* | If the file descriptor specified does not exist. |

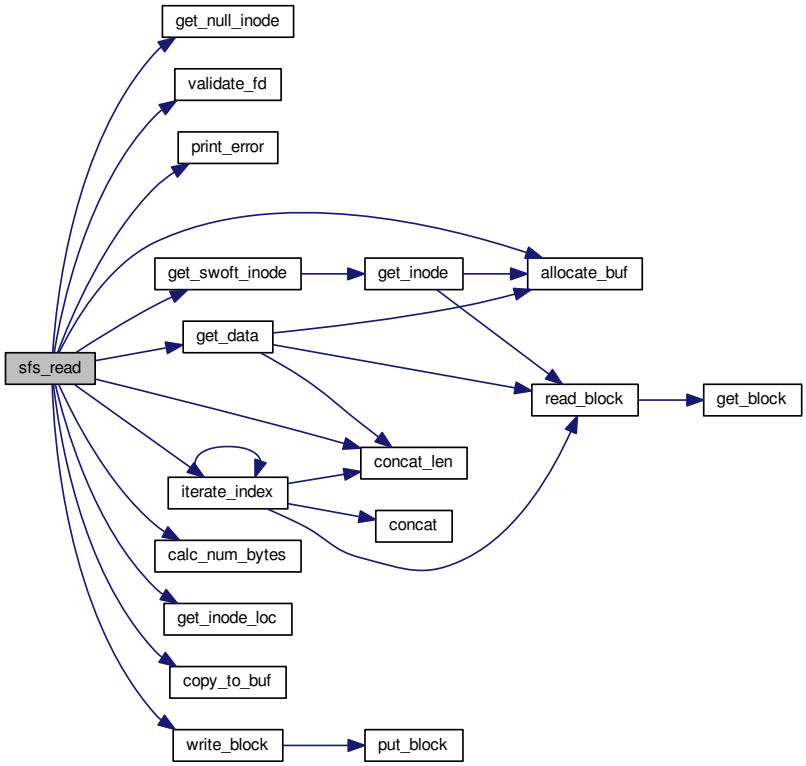
### Author

Daniel Smullen
Jonathan Gillett
Joseph Heron

### Copyright

GNU General Public License V3

Here is the call graph for this function:

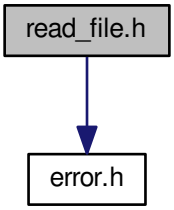


Here is the caller graph for this function:

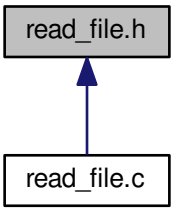

## 5.35 read_file.h File Reference

```
#include "error.h"
```

Include dependency graph for read_file.h:




This graph shows which files directly or indirectly include this file:



## Functions

- int sfs_read (int fd, int start, int length, byte *mem_pointer)

  *Read the contents of a file.*

### 5.35.1 Function Documentation

#### 5.35.1.1 int sfs_read ( int *fd*, int *start*, int *length*, byte * *mem_pointer* )

Read the contents of a file.

Copy the length of bytes of data specified from a regular file to memory location specified by mem_pointer. The parameter start gives the offset of the first byte in the file that should be copied.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to read data from. |
| *start* | The offset of the first byte in the file that should be copied. |
| *length* | The length of bytes of data to copy from the file. |
| *mem_pointer* | The memory location to store the data read from the file. |

**Returns**

Returns an integer value, if the value > 0 then the file was read successfully. If the value <= 0 then the file failed to be read.

**Exceptions**

| | |
|---|---|
| *FILE_LENGTH_OVERRUN* | If the file cannot be read correctly because the length of bytes specified exceeds the length of the file, no data will be copied from the file if an overrun occurs. |
| *INVALID_FILE_DESCRIPTOR* | If the file descriptor specified does not exist. |

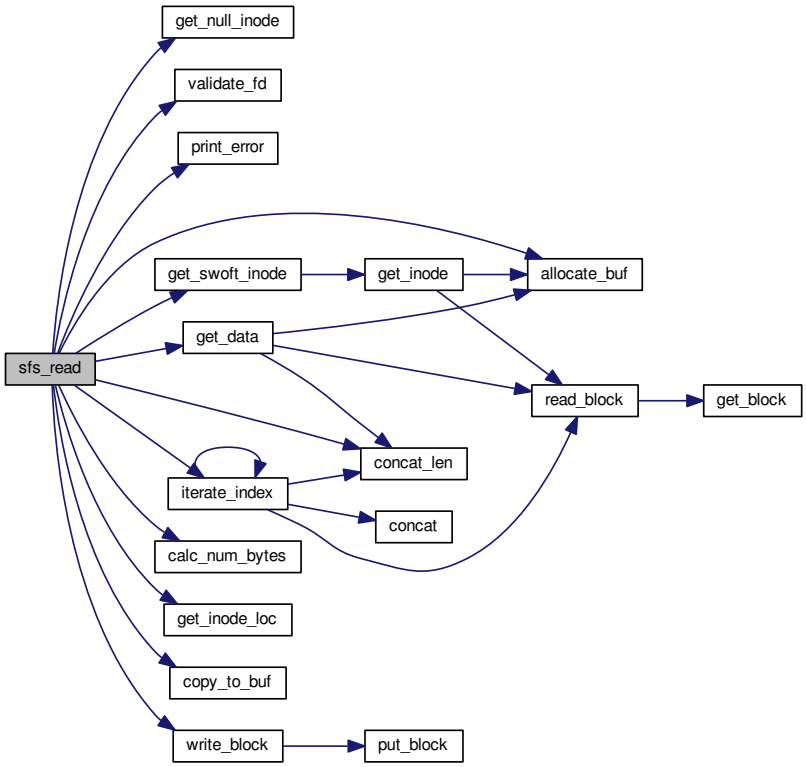
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the call graph for this function:

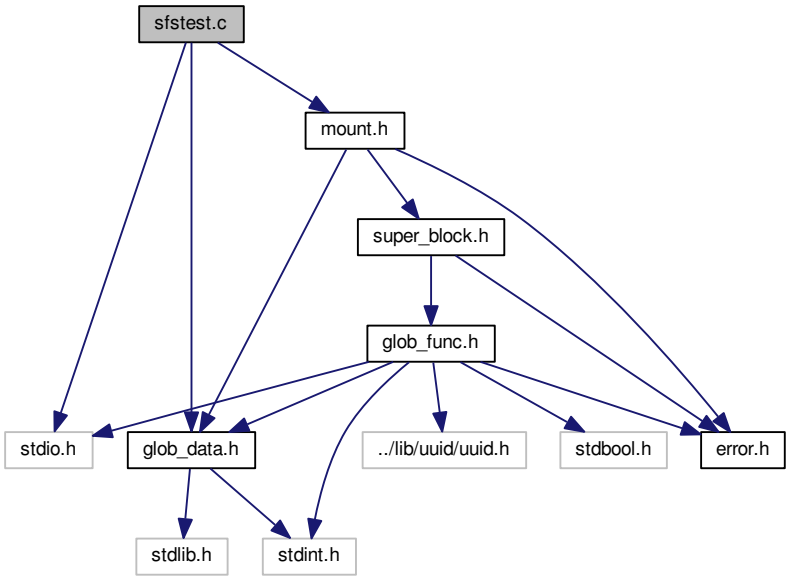



Here is the caller graph for this function:


## 5.36 sfstest.c File Reference

```
#include <stdio.h>
#include "glob_data.h"
#include "mount.h"
```

Include dependency graph for sfstest.c:




## Functions

- sfs_test ()

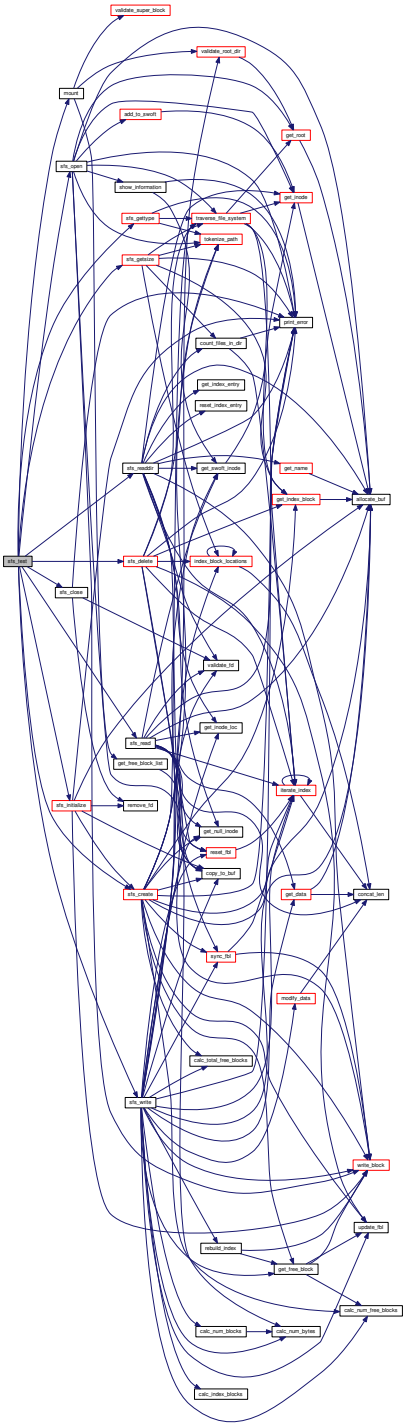
## Variables

- char command_buffer [MAX_INPUT_LENGTH+1] = {NULL}
- char io_buffer [MAX_IO_LENGTH+1] = {NULL}
- char data_buffer_1 [MAX_INPUT_LENGTH] = {NULL}
- int p1
- int p2
- int p3

### 5.36.1 Function Documentation

#### 5.36.1.1 sfs_test ( )

Here is the call graph for this function:




## 5.36.2 Variable Documentation

#### 5.36.2.1 char command_buffer[MAX_INPUT_LENGTH+1] = {NULL}

#### 5.36.2.2 char data_buffer_1[MAX_INPUT_LENGTH] = {NULL}

#### 5.36.2.3 char io_buffer[MAX_IO_LENGTH+1] = {NULL}

#### 5.36.2.4 int p1

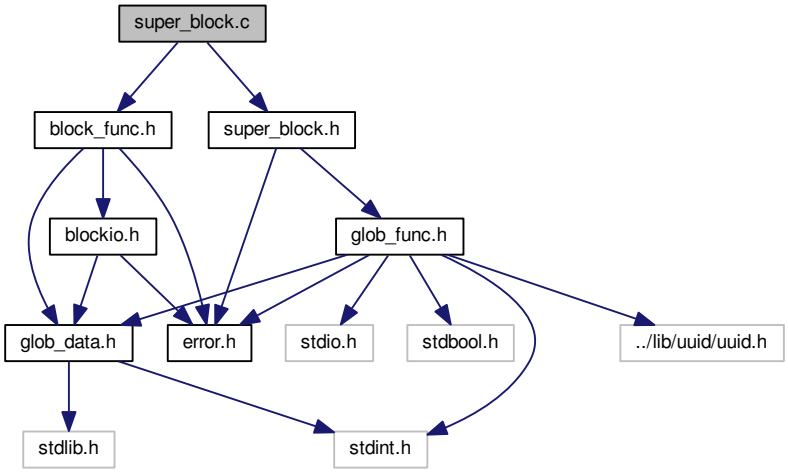
#### 5.36.2.5 int p2

#### 5.36.2.6 int p3

## 5.37 super_block.c File Reference

```
#include "super_block.h"
#include "block_func.h"
```

Include dependency graph for super_block.c:




### Functions

- superblock * get_super_block (void)

  *Get a handle to the super block.*

- uint32_t get_root (void)

  *Get the location of the root directory.*

- uint32_t get_free_block_index (void)

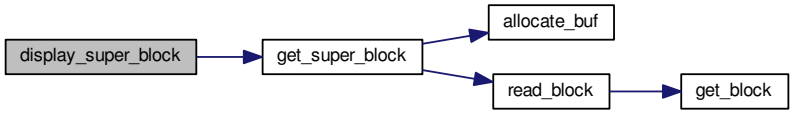
  *Get the location of the free block list's first index block.*

- int display_super_block (void)

### 5.37.1 Function Documentation

#### 5.37.1.1 int display_super_block ( void )

Here is the call graph for this function:




#### 5.37.1.2 uint32_t get_free_block_index ( void )

Get the location of the free block list's first index block.

This function is an accessor method for the pseudo-object attribute inside the super block which contains the location of the free block list structure's first index block on disk.

**Returns**

Returns the location of the free block list's index block.

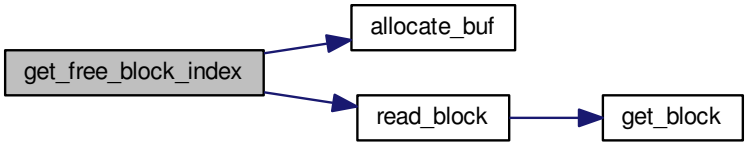
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

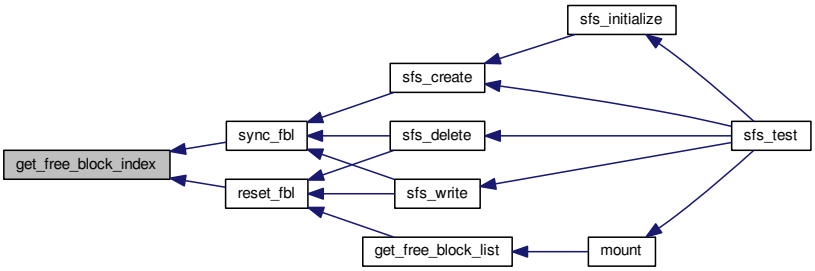
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.37.1.3 uint32_t get_root ( void )

Get the location of the root directory.

This function is an accessor method for the pseudo-object attribute inside the super block which contains the location of the root directory's inode on disk.

**Returns**

Returns the location of root directory on disk. If a failure occurs, return 0.

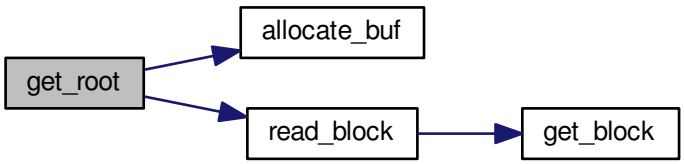
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

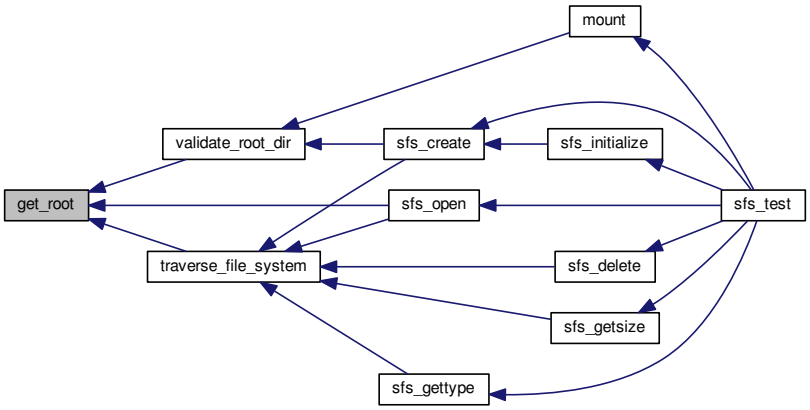
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.37.1.4 superblock* get_super_block ( void )

Get a handle to the super block.

This function implements a pseudo-object oriented method for accessing the super block data. It returns the super block as a pointer to a struct, whose individual members can be accessed.

**Returns**

Returns a pointer to the super block data.

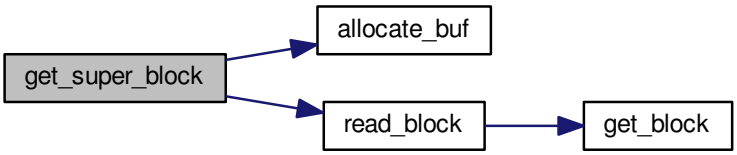
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

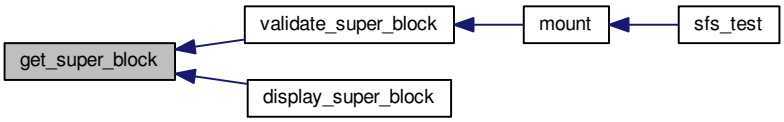
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



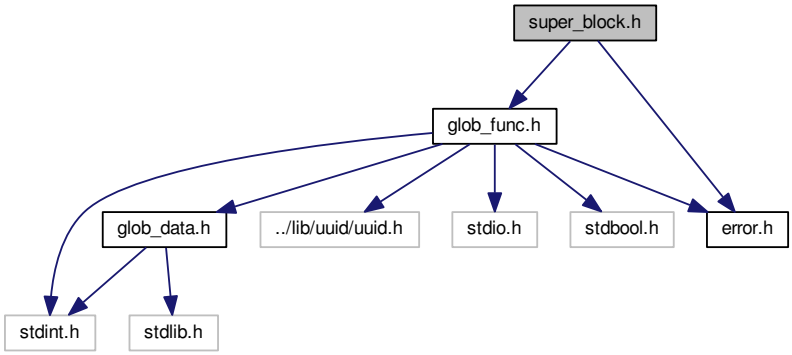

Here is the caller graph for this function:



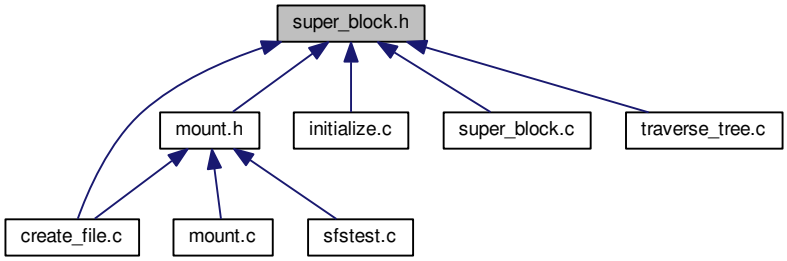

## 5.38 super_block.h File Reference

```
#include "glob_func.h"
#include "error.h"
```

Include dependency graph for super_block.h:




This graph shows which files directly or indirectly include this file:




## Data Structures

- struct superblock

## Functions

- superblock * get_super_block (void)

  *Get a handle to the super block.*
- uint32_t get_root (void)

  *Get the location of the root directory.*
- uint32_t get_free_block_index (void)

  *Get the location of the free block list's first index block.*

### 5.38.1 Function Documentation

#### 5.38.1.1 uint32_t get_free_block_index ( void )

Get the location of the free block list's first index block.

This function is an accessor method for the pseudo-object attribute inside the super block which contains the location of the free block list structure's first index block on disk.

**Returns**

Returns the location of the free block list's index block.

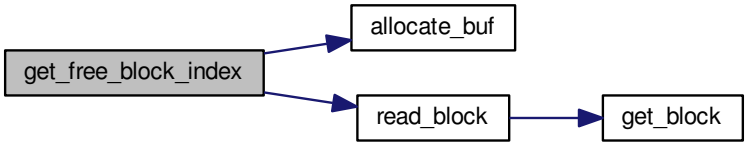
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

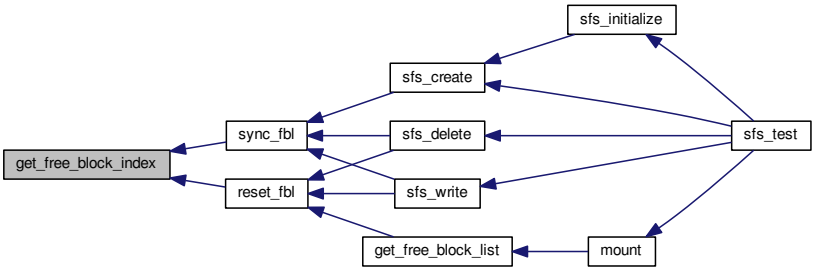
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.38.1.2 uint32_t get_root ( void )

Get the location of the root directory.

This function is an accessor method for the pseudo-object attribute inside the super block which contains the location of the root directory's inode on disk.

**Returns**

Returns the location of root directory on disk. If a failure occurs, return 0.

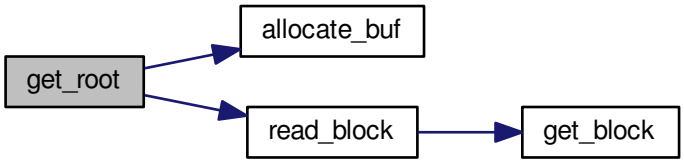
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

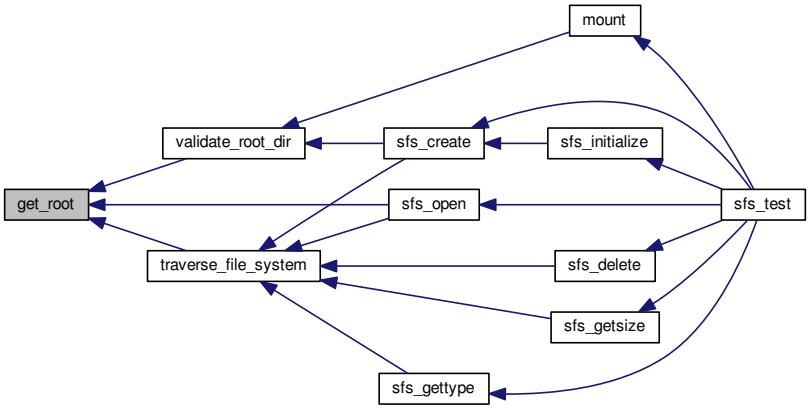
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.38.1.3 superblock* get_super_block ( void )

Get a handle to the super block.

This function implements a pseudo-object oriented method for accessing the super block data. It returns the super block as a pointer to a struct, whose individual members can be accessed.

**Returns**

Returns a pointer to the super block data.

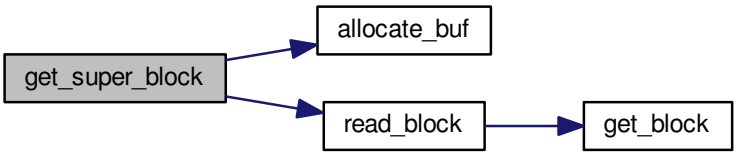
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

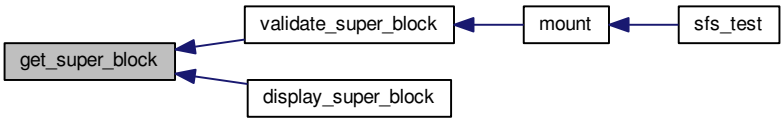
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



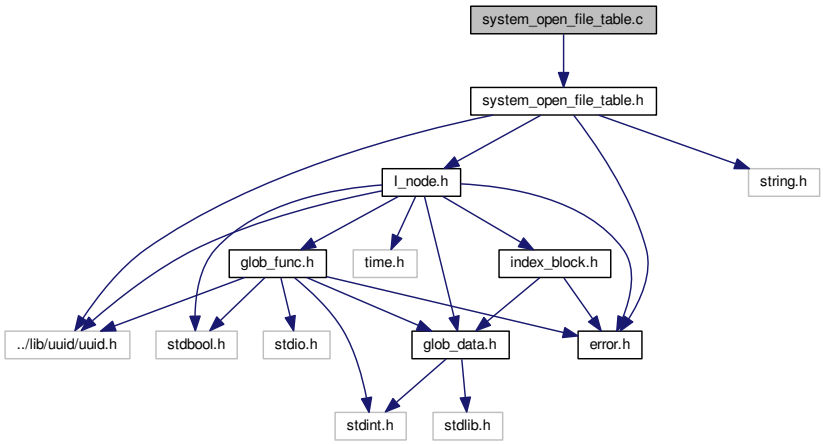

Here is the caller graph for this function:




## 5.39 system_open_file_table.c File Reference

```
#include "system_open_file_table.h"
```

Include dependency graph for system_open_file_table.c:




## Functions

- int add_to_swoft (uint32_t block_num)

  *Add the inode to the system wide open file table.*
- int find_opening (void)

  *Finds an open location in the swoft.*
- int validate_fd (int fd)

  *Validate whether the given file descriptor is a valid file descriptor in the system wide open file table.*
- inode get_swoft_inode (int fd)

  *Get an inode instance from the system wide open file table.*
- uint32_t get_inode_loc (int fd)

  *Get the location of the inode on disk which corresponds to a file descriptor.*
- void remove_fd (int fd)

  *Remove a file descriptor from the system wide open file table.*
- uint32_t find_and_remove (uint32_t inode_location)

  *Find all the system wide open file table indices that have the same data as the given location on disk, and remove them from the system wide open file table.*

## Variables

- swoft system_open_tb = {0}

  *The static instance of the system wide open file table.*

### 5.39.1 Function Documentation

#### 5.39.1.1 int add_to_swoft ( uint32_t *block_num* )

Add the inode to the system wide open file table.

This function takes an inode's block location on disk, reads that block, and assigns the returned inode location on disk into memory to a file descriptor - this is an index on the system wide open file table.

**Parameters**

| | |
|---|---|
| *block_num* | The inode's location on disk. |

**Returns**

Returns an integer value, if the value >= 0 then the inode was found and added to the system wide open file table. If the value = -1 then the table was full. If the value = -2 then the inode was not found. By default, any other return value signals that the function was unsuccessful.

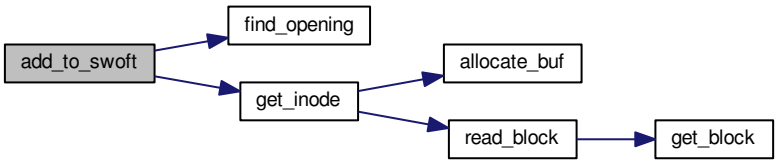
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

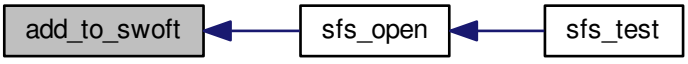
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.39.1.2 uint32_t find_and_remove ( uint32_t *inode_location* )

Find all the system wide open file table indices that have the same data as the given location on disk, and remove them from the system wide open file table.

This function is used when deleting files. In order to remove all open handles to a file when it is removed from disk, the system wide open file table must have all corresponding handles removed.

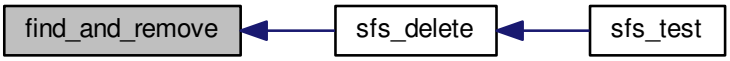
**Parameters**

| | |
|---|---|
| *inode_location* | The location of the inode on disk. |

**Returns**

Returns an integer value indicating the number of entries deleted from the system wide open file table.

Here is the caller graph for this function:




#### 5.39.1.3 int find_opening ( void )

Finds an open location in the swoft.

This function finds the next open location in the system wide open file table and returns the first location in the swoft that is available.

**Returns**

Returns the first location in the swoft that is available, if the value $>=$ 0 then it is the first available location found. Otherwise if the value $<$ 0 then the swoft is full.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:


#### 5.39.1.4 uint32_t get_inode_loc ( int *fd* )

Get the location of the inode on disk which corresponds to a file descriptor.

This function takes a file descriptor, and if there is a valid inode pseudo-object at that index on the system wide open file table, the location of that object on disk is returned from accessing it within the pseudo-object.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor. |

**Returns**

Returns the location of the inode on disk.

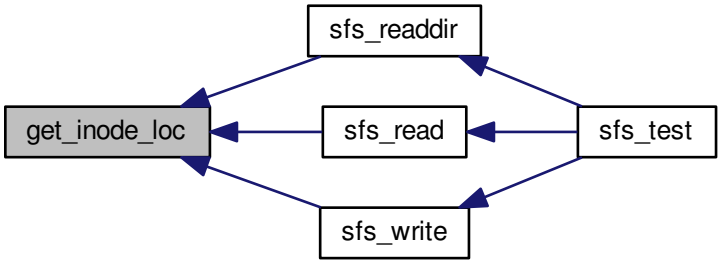
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.39.1.5 inode get_swoft_inode ( int *fd* )

Get an inode instance from the system wide open file table.

This function takes a file descriptor, which is an index on the system wide open file table. If there is an inode pseudo-object at the index specified, it is returned.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor. |

**Returns**

Returns the inode instance from the system wide open file table given the file descriptor.

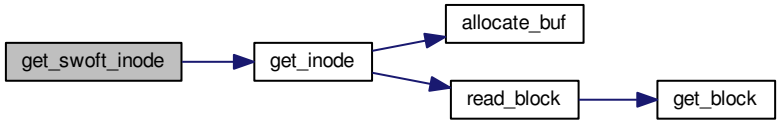
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

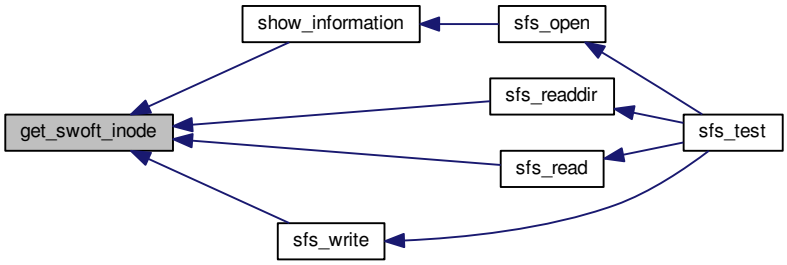
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.39.1.6 void remove_fd ( int *fd* )

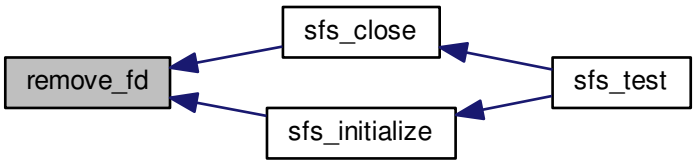
Remove a file descriptor from the system wide open file table.

This function accesses an index on the system wide open file table, and frees whatever data is in memory corresponding to that index.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor to remove. |

Here is the caller graph for this function:




#### 5.39.1.7 int validate_fd ( int *fd* )

Validate whether the given file descriptor is a valid file descriptor in the system wide open file table.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor. |

**Returns**

Returns an integer value, if the value >= 0 the file descriptor was valid, otherwise the file descriptor was invalid, or the function was unsuccessful.

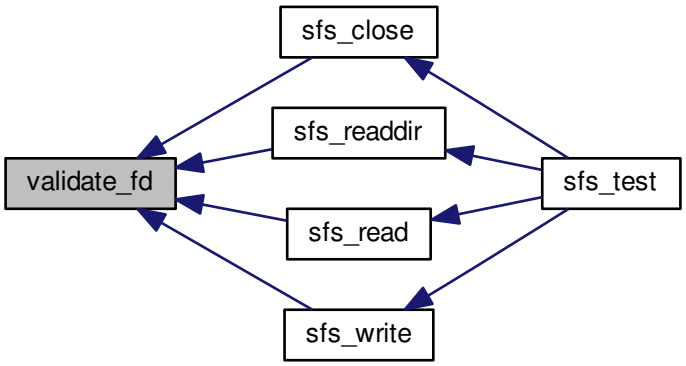
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




### 5.39.2 Variable Documentation

#### 5.39.2.1 system_open_tb = {0}

The static instance of the system wide open file table.

This variable holds the singleton instance of the system wide open file table. It represents an instance of a pseudo-object which can only be accessed and modified through the accessor and mutator methods provided.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

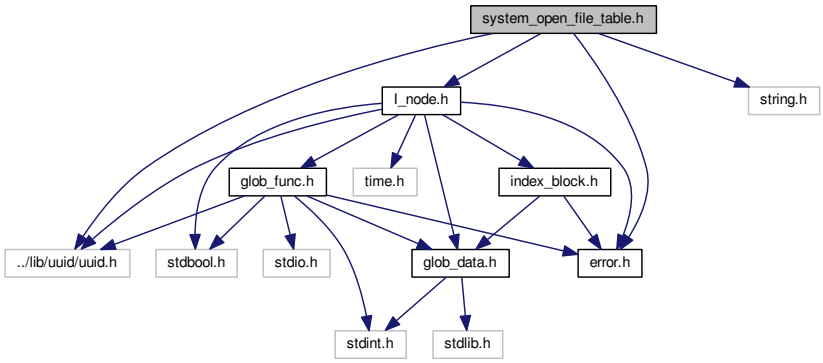
**Copyright**

GNU General Public License V3

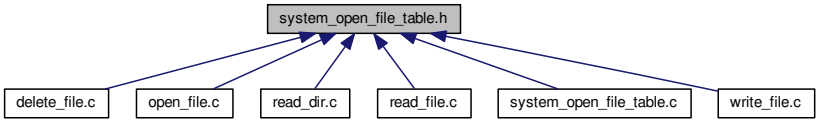
## 5.40 system_open_file_table.h File Reference

```
#include "I_node.h"
#include "../lib/uuid/uuid.h"
#include "error.h"
#include <string.h>
```

Include dependency graph for system_open_file_table.h:




This graph shows which files directly or indirectly include this file:




## Data Structures

- struct swoft

  *Pseudo-object containing the array of open file descriptors.*

## Functions

- int add_to_swoft (uint32_t block_num)

  *Add the inode to the system wide open file table.*
- int find_opening (void)

  *Finds an open location in the swoft.*
- int validate_fd (int fd)

  *Validate whether the given file descriptor is a valid file descriptor in the system wide open file table.*
- inode get_swoft_inode (int fd)

  *Get an inode instance from the system wide open file table.*
- uint32_t get_inode_loc (int fd)

  *Get the location of the inode on disk which corresponds to a file descriptor.*
- void remove_fd (int fd)

  *Remove a file descriptor from the system wide open file table.*
- uint32_t find_and_remove (uint32_t inode_location)

*Find all the system wide open file table indices that have the same data as the given location on disk, and remove them from the system wide open file table.*

## Variables

- swoft system_open_tb

  *The static instance of the system wide open file table.*

### 5.40.1 Function Documentation

#### 5.40.1.1 int add_to_swoft ( uint32_t *block_num* )

Add the inode to the system wide open file table.

This function takes an inode's block location on disk, reads that block, and assigns the returned inode location on disk into memory to a file descriptor - this is an index on the system wide open file table.

**Parameters**

| | |
|---|---|
| *block_num* | The inode's location on disk. |

**Returns**

Returns an integer value, if the value >= 0 then the inode was found and added to the system wide open file table. If the value = -1 then the table was full. If the value = -2 then the inode was not found. By default, any other return value signals that the function was unsuccessful.

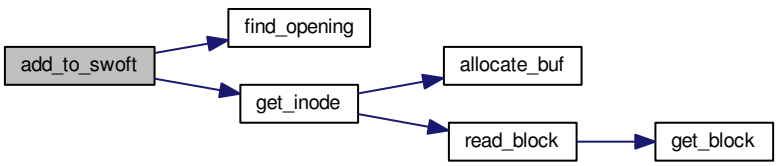
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

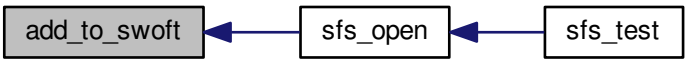
**Copyright**

GNU General Public License V3

Here is the call graph for this function:



Here is the caller graph for this function:




#### 5.40.1.2 uint32_t find_and_remove ( uint32_t *inode_location* )

Find all the system wide open file table indices that have the same data as the given location on disk, and remove them from the system wide open file table.

This function is used when deleting files. In order to remove all open handles to a file when it is removed from disk, the system wide open file table must have all corresponding handles removed.

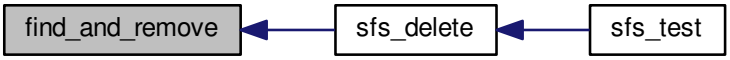
**Parameters**

| | |
|---|---|
| *inode_location* | The location of the inode on disk. |

**Returns**

Returns an integer value indicating the number of entries deleted from the system wide open file table.

Here is the caller graph for this function:




#### 5.40.1.3 int find_opening ( void )

Finds an open location in the swoft.

This function finds the next open location in the system wide open file table and returns the first location in the swoft that is available.

**Returns**

Returns the first location in the swoft that is available, if the value $>= 0$ then it is the first available location found. Otherwise if the value $< 0$ then the swoft is full.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:


#### 5.40.1.4 uint32_t get_inode_loc ( int *fd* )

Get the location of the inode on disk which corresponds to a file descriptor.

This function takes a file descriptor, and if there is a valid inode pseudo-object at that index on the system wide open file table, the location of that object on disk is returned from accessing it within the pseudo-object.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor. |

**Returns**

Returns the location of the inode on disk.

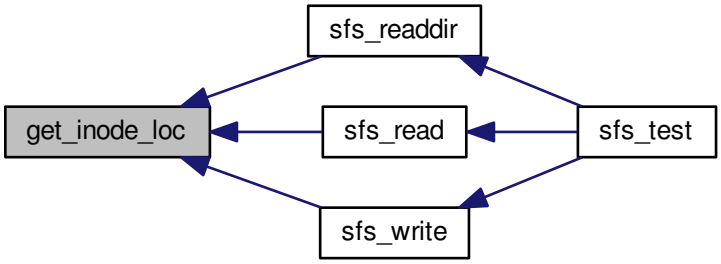
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




#### 5.40.1.5 inode get_swoft_inode ( int *fd* )

Get an inode instance from the system wide open file table.

This function takes a file descriptor, which is an index on the system wide open file table. If there is an inode pseudo-object at the index specified, it is returned.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor. |

**Returns**

Returns the inode instance from the system wide open file table given the file descriptor.

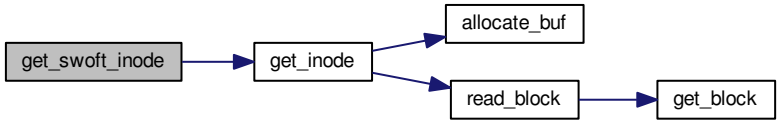
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

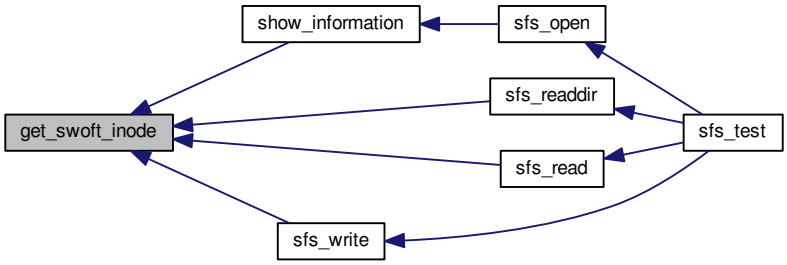
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.40.1.6 void remove_fd ( int *fd* )

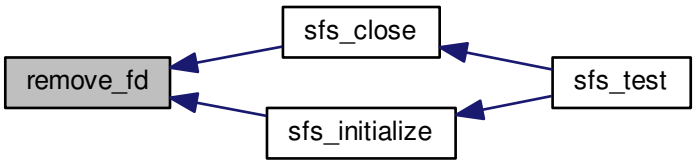
Remove a file descriptor from the system wide open file table.

This function accesses an index on the system wide open file table, and frees whatever data is in memory corresponding to that index.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor to remove. |

Here is the caller graph for this function:




#### 5.40.1.7 int validate_fd ( int *fd* )

Validate whether the given file descriptor is a valid file descriptor in the system wide open file table.

**Parameters**

| | |
|---|---|
| *fd* | The file descriptor. |

**Returns**

Returns an integer value, if the value >= 0 the file descriptor was valid, otherwise the file descriptor was invalid, or the function was unsuccessful.

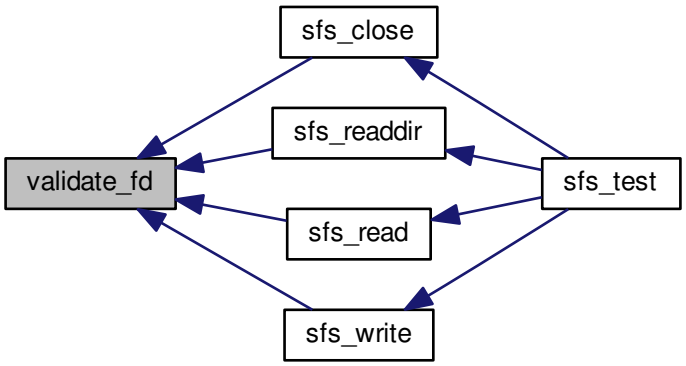
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:




### 5.40.2 Variable Documentation

#### 5.40.2.1 swoft system_open_tb

The static instance of the system wide open file table.

This variable holds the singleton instance of the system wide open file table. It represents an instance of a pseudo-object which can only be accessed and modified through the accessor and mutator methods provided.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

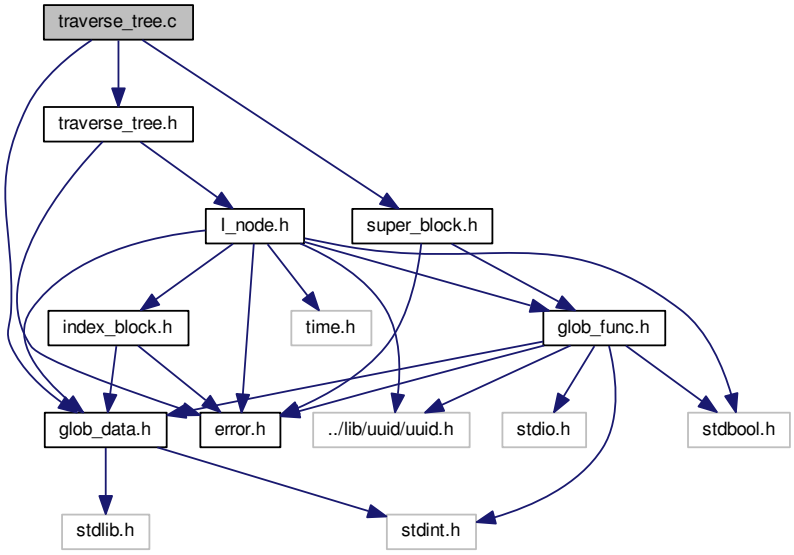
**Copyright**

GNU General Public License V3

## 5.41 traverse_tree.c File Reference

```
#include "glob_data.h"
#include "traverse_tree.h"
#include "super_block.h"
```

Include dependency graph for traverse_tree.c:




## Functions

- uint32_t * traverse_file_system (char **tokens, bool create)

  *Traverse the file system structure.*

### 5.41.1 Function Documentation

#### 5.41.1.1 uint32_t* traverse_file_system ( char ** *tokens,* bool *create* )

Traverse the file system structure.

Given the tokens for the path, traverse the file system to either the last token or the second last based on the value in the create flag.

**Parameters**

| | |
|---|---|
| *tokens* | The pre-parsed pathname tokens. |
| *create* | Flags whether to get the parent directory or the child entity if create is true, get the parent. Otherwise, get the child. |

**Returns**

Returns a location on disk. If create is true, the location of the parent is returned and the last token in the tokens array. If create is false, the location of the inode is returned and the second element will be null.

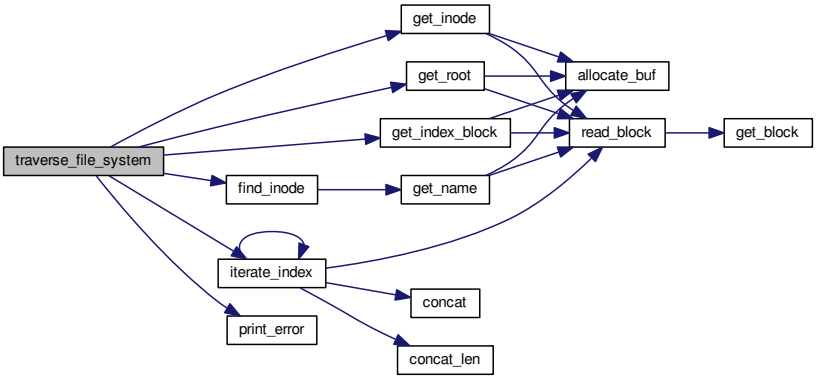
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

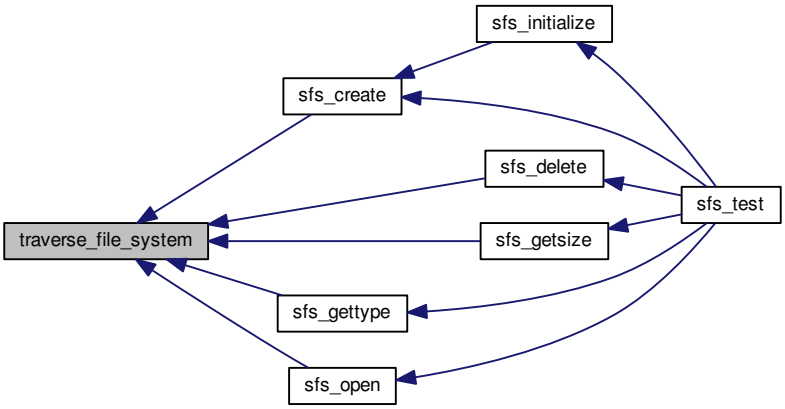
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:

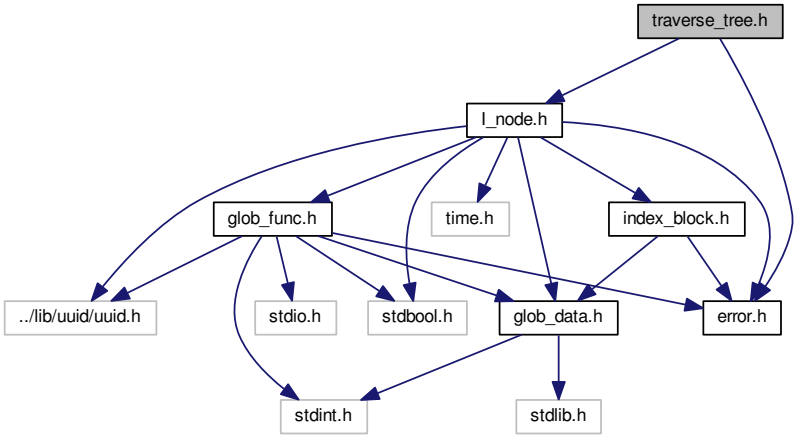



## 5.42 traverse_tree.h File Reference

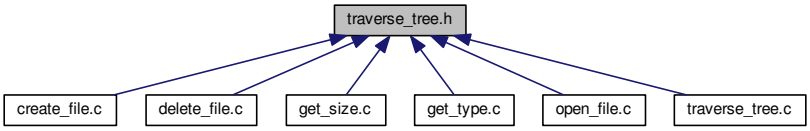
```
#include "I_node.h"
```

```
#include "error.h"
```

Include dependency graph for traverse_tree.h:




This graph shows which files directly or indirectly include this file:




## Functions

- uint32_t * traverse_file_system (char **tokens, bool create)

  *Traverse the file system structure.*

### 5.42.1 Function Documentation

#### 5.42.1.1 uint32_t* traverse_file_system ( char ** *tokens*, bool *create* )

Traverse the file system structure.

Given the tokens for the path, traverse the file system to either the last token or the second last based on the value in the create flag.

**Parameters**

| | |
|---|---|
| *tokens* | The pre-parsed pathname tokens. |
| *create* | Flags whether to get the parent directory or the child entity if create is true, get the parent. Otherwise, get the child. |

### Returns

Returns a location on disk. If create is true, the location of the parent is returned and the last token in the tokens array. If create is false, the location of the inode is returned and the second element will be null.

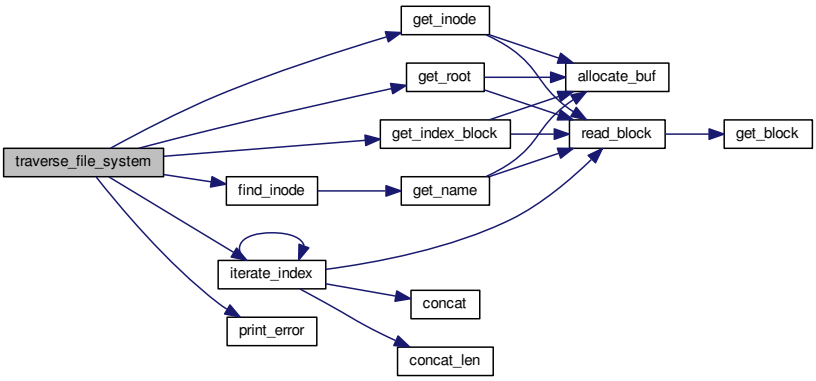
### Author

Daniel Smullen
Jonathan Gillett
Joseph Heron

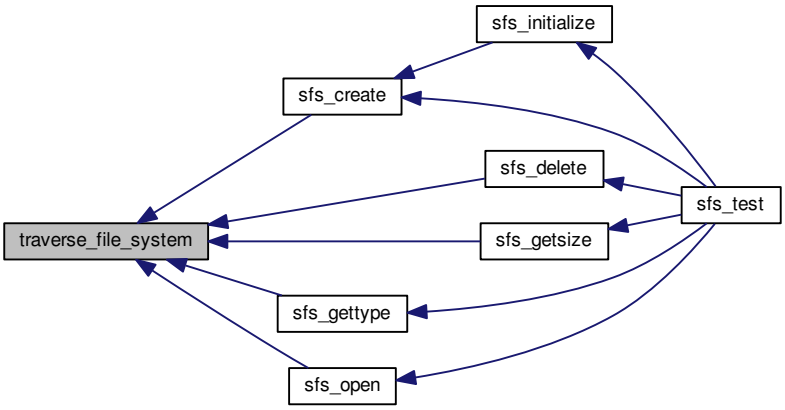
### Copyright

GNU General Public License V3

Here is the call graph for this function:



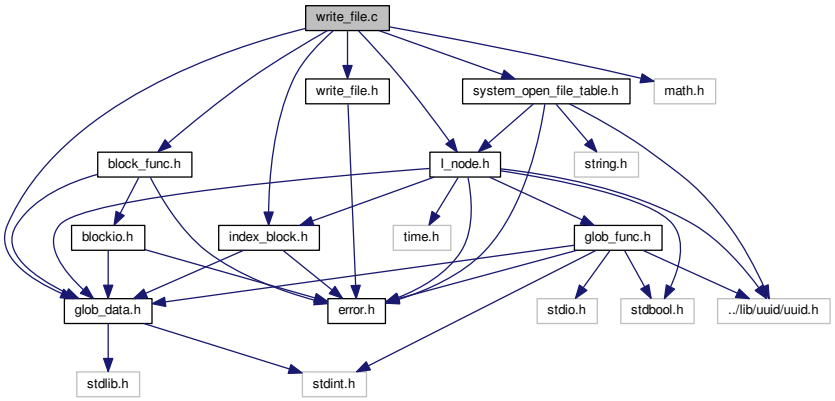

Here is the caller graph for this function:



## 5.43 write_file.c File Reference

```
#include "glob_data.h"
#include "block_func.h"
#include "I_node.h"
#include "system_open_file_table.h"
#include "index_block.h"
#include "write_file.h"
#include <math.h>
```

Include dependency graph for write_file.c:




### Functions

- block ∗ modify_data (int32_t start, uint32_t length, byte ∗data_buf, byte ∗actual_data)

  *Modifies or appends data in a buffer.*
- block ∗ segment_data (byte ∗data_buf)

  *Segments a buffer into block sized chunks of data.*
- block ∗ segment_data_len (byte ∗data_buf, uint32_t length)

  *Segments a buffer into block sized chunks of data.*
- int sfs_write (int fd, int start, int length, byte ∗mem_pointer)

  *Write data to a file.*

### 5.43.1 Function Documentation

#### 5.43.1.1 block∗ modify_data ( int32_t *start*, uint32_t *length*, byte ∗ *data_buf*, byte ∗ *actual_data* )

Modifies or appends data in a buffer.

Modifies the data in the buffer with the actual data provided at the starting point and length of data specified. If the starting value is -1 then the data is appended to the buffer for the length specified.

**Note**

In order to provide support for appending data using start = -1, the parameter type MUST be signed, which is int32_t.

**Parameters**

| | |
|---|---|
| *start* | The starting offset to alter the data at. If the value provided is -1, append data to the end of the buffer. If start >= 0 then actual data is inserted (overwrites) from start for the length of actual data. |
| *length* | The length of data to overwrite in the buffer, or to append to the buffer. |
| *data_buf* | A pointer to the data buffer to be altered or appended to. |
| *actual_data* | The data to overwrite the data buffer with or append to. |

**Returns**

Returns a null terminated array of blocks, if an error occurs then NULL is returned.

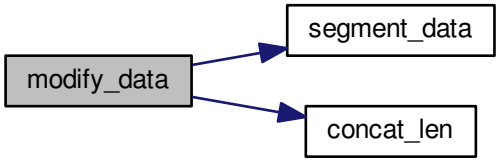
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

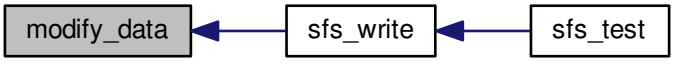
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:



#### 5.43.1.2 block* segment_data ( byte * *data_buf* )

Segments a buffer into block sized chunks of data.

Takes a NULL terminated data buffer and returns an array of blocks, it is similar to modify_data except that no data is actually being modified. Returns an array of blocks where the last index is an entire empty block containing NULL. If an error occurs NULL is returned.

**Parameters**

| | |
|---|---|
| *data_buf* | A NULL terminated buffer containing the data to be segmented into blocks. |

**Returns**

A null terminated array of blocks, where the last index is NULL, if an error occurs then NULL is returned.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:


#### 5.43.1.3 block* segment_data_len ( byte * *data_buf,* uint32_t *length* )

Segments a buffer into block sized chunks of data.

Takes a data buffer that in a multiple of BLKSIZE and returns an array of blocks, it is similar to modify_data except that no data is actually being modified. Returns a 2D array where each index in the array points a block, the last index points to NULL. If an error occurs NULL is returned.

**Precondition**

data_buf MUST be a buffer that is EXACTLY a multiple of BLKSIZE, it does not necessarily need to be NULL terminated.

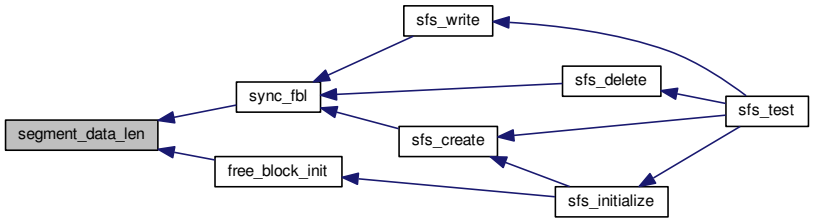
**Parameters**

| | |
|---|---|
| *data_buf* | A buffer containing the data to be segmented into blocks. |
| *length* | The length of the buffer in bytes to be segmented into blocks. |

**Returns**

Returns a null terminated array of blocks, where the last index is NULL, if an error occurs then NULL is returned.

Here is the caller graph for this function:




#### 5.43.1.4 int sfs_write ( int *fd,* int *start,* int *length,* byte * *mem_pointer* )

Write data to a file.

Write the length bytes of data specified from a memory location to the specified file. The parameter start gives the offset of the first byte in the file that should be copied to. Alternatively, the value of start may be set to -1, this indicates that the specified number of bytes should be appended to the file. Appending (setting start to -1) is the only allowable way to increase the length of a file.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to write data to. |
| *start* | The offset of the first byte in the file that should be written to, if start is -1 then the data will be appended to the file. |
| *length* | The length of bytes of data to be written to the file. |
| *mem_pointer* | The memory location containing the bytes to write to file. |

**Returns**

Returns an integer value, if the value $>$ 0 the file write was a success, and if the value $<=$ 0 the file write was unsuccessful.

**Exceptions**

| | |
|---|---|
| *INVALID_FILE_DESCRIPTOR* | If the file descriptor specified does not exist or is invalid. |
| *INSUFFICIENT_DISK_SPACE* | If the length of the blocks to be written plus the size of the index overhead is greater than the amount of free blocks on disk. |

**Author**

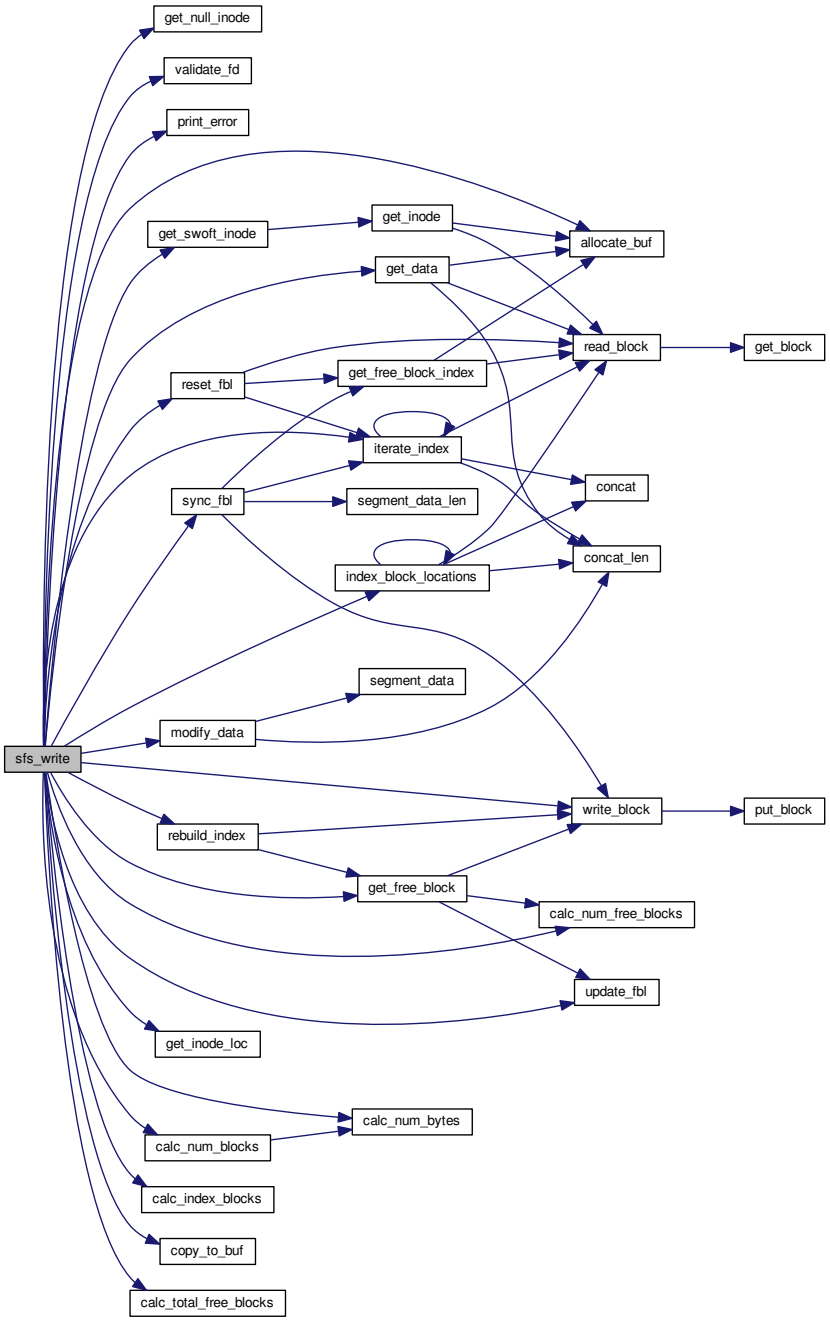
Daniel Smullen
Jonathan Gillett
Joseph Heron

### Copyright

GNU General Public License V3

Display the number of free blocks

Here is the call graph for this function:

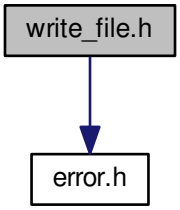


Here is the caller graph for this function:

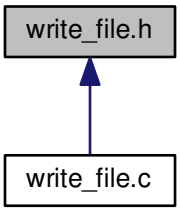

## 5.44 write_file.h File Reference

```
#include "error.h"
```

Include dependency graph for write_file.h:




This graph shows which files directly or indirectly include this file:



## Functions

- block ∗ modify_data (int32_t start, uint32_t length, byte ∗data_buf, byte ∗actual_data)

  *Modifies or appends data in a buffer.*
- block ∗ segment_data (byte ∗data_buf)

  *Segments a buffer into block sized chunks of data.*
- block ∗ segment_data_len (byte ∗data_buf, uint32_t length)

  *Segments a buffer into block sized chunks of data.*
- int sfs_write (int fd, int start, int length, byte ∗mem_pointer)

  *Write data to a file.*

### 5.44.1 Function Documentation

#### 5.44.1.1 block∗ modify_data ( int32_t *start*, uint32_t *length*, byte ∗ *data_buf*, byte ∗ *actual_data* )

Modifies or appends data in a buffer.

Modifies the data in the buffer with the actual data provided at the starting point and length of data specified. If the starting value is -1 then the data is appended to the buffer for the length specified.

**Note**

In order to provide support for appending data using start = -1, the parameter type MUST be signed, which is int32_t.

**Parameters**

| | |
|---|---|
| *start* | The starting offset to alter the data at. If the value provided is -1, append data to the end of the buffer. If start >= 0 then actual data is inserted (overwrites) from start for the length of actual data. |
| *length* | The length of data to overwrite in the buffer, or to append to the buffer. |
| *data_buf* | A pointer to the data buffer to be altered or appended to. |
| *actual_data* | The data to overwrite the data buffer with or append to. |

**Returns**

Returns a null terminated array of blocks, if an error occurs then NULL is returned.

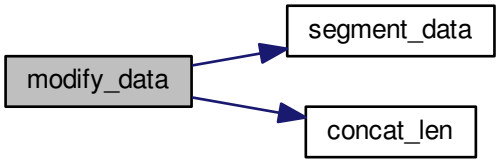
**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

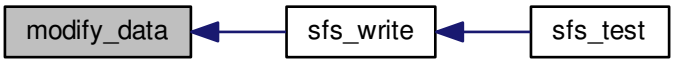
**Copyright**

GNU General Public License V3

Here is the call graph for this function:




Here is the caller graph for this function:




#### 5.44.1.2 block* segment_data ( byte * *data_buf* )

Segments a buffer into block sized chunks of data.

Takes a NULL terminated data buffer and returns an array of blocks, it is similar to modify_data except that no data is actually being modified. Returns an array of blocks where the last index is an entire empty block containing NULL. If an error occurs NULL is returned.

**Parameters**

| | |
|---|---|
| *data_buf* | A NULL terminated buffer containing the data to be segmented into blocks. |

**Returns**

A null terminated array of blocks, where the last index is NULL, if an error occurs then NULL is returned.

**Author**

Daniel Smullen
Jonathan Gillett
Joseph Heron

**Copyright**

GNU General Public License V3

Here is the caller graph for this function:


#### 5.44.1.3 block* segment_data_len ( byte * *data_buf*, uint32_t *length* )

Segments a buffer into block sized chunks of data.

Takes a data buffer that in a multiple of BLKSIZE and returns an array of blocks, it is similar to modify_data except that no data is actually being modified. Returns a 2D array where each index in the array points a block, the last index points to NULL. If an error occurs NULL is returned.

**Precondition**

data_buf MUST be a buffer that is EXACTLY a multiple of BLKSIZE, it does not necessarily need to be NULL terminated.

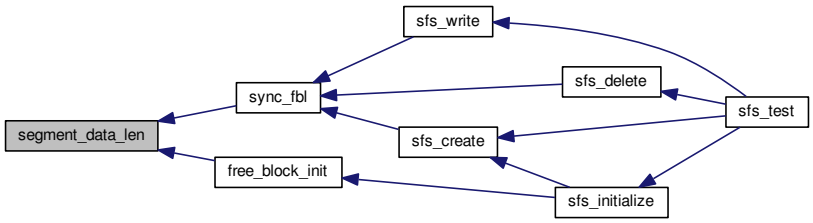
**Parameters**

| | |
|---|---|
| *data_buf* | A buffer containing the data to be segmented into blocks. |
| *length* | The length of the buffer in bytes to be segmented into blocks. |

**Returns**

Returns a null terminated array of blocks, where the last index is NULL, if an error occurs then NULL is returned.

Here is the caller graph for this function:



#### 5.44.1.4 int sfs_write ( int *fd,* int *start,* int *length,* byte * *mem_pointer* )

Write data to a file.

Write the length bytes of data specified from a memory location to the specified file. The parameter start gives the offset of the first byte in the file that should be copied to. Alternatively, the value of start may be set to -1, this indicates that the specified number of bytes should be appended to the file. Appending (setting start to -1) is the only allowable way to increase the length of a file.

**Parameters**

| | |
|---|---|
| *fd* | A file descriptor for the file to write data to. |
| *start* | The offset of the first byte in the file that should be written to, if start is -1 then the data will be appended to the file. |
| *length* | The length of bytes of data to be written to the file. |
| *mem_pointer* | The memory location containing the bytes to write to file. |

**Returns**

Returns an integer value, if the value $>$ 0 the file write was a success, and if the value $<=$ 0 the file write was unsuccessful.

**Exceptions**

| | |
|---|---|
| *INVALID_FILE_DESCRIPTOR* | If the file descriptor specified does not exist or is invalid. |
| *INSUFFICIENT_DISK_SPACE* | If the length of the blocks to be written plus the size of the index overhead is greater than the amount of free blocks on disk. |

**Author**

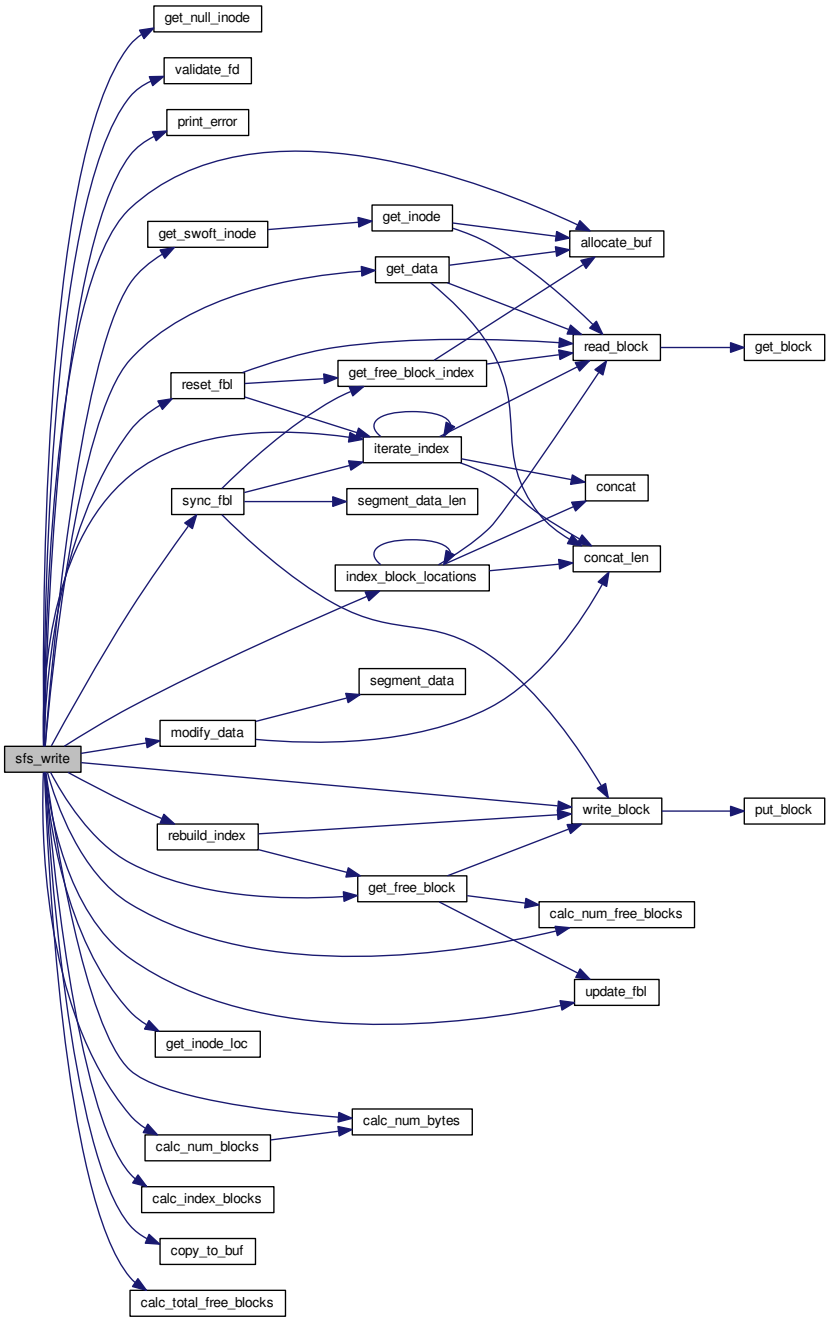
Daniel Smullen
Jonathan Gillett
Joseph Heron

### Copyright

GNU General Public License V3

Display the number of free blocks

Here is the call graph for this function:



Here is the caller graph for this function: