VKP FAT12 Shell

Generated by Doxygen 1.8.9.1

Contents

1 VKP-FAT12-FileSystem

2 Test List

globalScope> Global cat_main (int argc, char *argv[])

If given the name of a file path culminating in a file name with an extension, cat shall display file contents.

If given '.' or '..', cat shall print "Cannot use cat on ./...".

If number of arguments is greater or less than one, cat shall print "Invalid argument count; cat takes a file name to cat.".

If path given points to root, attributes indicate a long filename (should never occur), or a file is not a subdirectory, cat shall print "File %s is incompatible with cat.".

If path given does not lead to a file at all, cat shall print "Could not find file [filename] to cat!".

globalScope> Global cd_main (int argc, char *argv[])

If cd is provided with a valid path, the working directory shall be set to that path.

If there is any quantity of arguments other than one for a path, cd shall exit printing, "Invalid argument count; cd takes a file name to search for.".

If cd ends up in root, the string "In root!" shall be printed.

If cd cannot move to the path provided, it shall print "Could not find file/folder: [path]".

cd shall be able to support . and .. and relative as well as absolute paths.

globalScope> Global df_main (int argc, char *argv[])

If called with any number of arguments, df shall display the number of K-blocks as well as the FAT sector count, the used sector count, the free sector count, and the percentage of sectors used.

globalScope> Global echo_main (int argc, char *argv[])

Echo shall print the first argument provided to it to console.

If echo is provided with a number of arguments other than one, echo shall exit printing, "Invalid argument count; need something to echo.".

globalScope> Global ls_main (int argc, char *argv[])

If ls is called with no arguments, ls shall list the files and folders of the current working directory, providing their individual FLCs, sizes, dates, and names.

If ls is called with an argument that is a valid path to a directory, ls shall list the files and folders of the provided directory, displaying their individual FLCs, sizes, dates, and names.

If ls is called with an argument that is a valid path to a file, ls shall print the listing for that individual file, displaying its FLC, size, date, and name.

If ls is called with an argument that is an invalid path to a directory, ls shall exit printing, "Could not find path".

If the arguments provided to ls number more than one, ls shall exit printing, "Too many arguments!".

Any and all file/folder listings provided by ls shall be sorted in alphabetical order by file name and extension if applicable.

ls shall not print any file whose attributes are 0 or are 0x0f under any circumstances.

globalScope> Global mkdir_main (int argc, char *argv[])

If provided with a single argument containing a non-existent filename, mkdir shall create a folder with the given name within the current working directory.

If provided with a single argument containing a path and culminating in a non-existent filename, mkdir shall create a folder with the given filename within the provided directory.

If provided with anything other than one argument, mkdir shall exit printing, "Invalid argument count; mkdir takes the path of the directory to create.".

If provided with a single argument containing a valid path to an existing directory, mkdir shall print "File [file_name] already exists.".

If provided with ".", "..", mkdir shall exit printing, "[entry] is not allowed.".

If during the process of trying to create a new directory, mkdir cannot allocate a directory sector, mkdir shall exit printing, "Failed to allocate directory sector.".

If there is not enough room in a directory to add a new directory, a successful mkdir call shall expand the directory before attempting to create the new directory.

If during the process of trying to create a new directory, mkdir cannot allocate a directory header, mkdir shall exit printing, "Failed to allocate directory header.".

If successful in creating a directory, mkdir shall add a timestamp accurate to the closest two seconds to the newly created directory.

globalScope> Global mount_main (int argc, char *argv[])

If mount is provided with a single argument describing a valid OS path to a FAT12-formatted bootable floppy image, mount shall load the image and make it available for use as a file system.

If mount is provided with a single argument describing an invalid OS path, mount shall exit printing, "Could not mount image [image_path]".

If mount is provided with a number of arguments other than one, mount shall exit printing, "Invalid argument count. mount takes the path of the floppy to mount.".

if mount has successfully mounted an image in the past during execution, mount shall exit printing, "File system at [mounted_image_path] is already mounted.".

globalScope> Global pbs_main (int argc, char *argv[])

If pbs is run with any number of arguments, pbs shall print a readout containing information about the boot sector of the currently mounted disk image.

globalScope> Global pfe_main (int argc, char *argv[])

If pfe is provided with exactly two arguments indicating the start and end entry indices within the FAT table for which to print a range of FAT entries.

If pfe is provided with any number of arguments other than two, pfe shall exit printing, "Invalid argument count; pfe takes the start and end indices of the FAT table indicating a range of FAT entries to print out.".

If any of pfe's two arguments are not a valid number, that argument shall be interpreted as 0.

globalScope> Global pwd_main (int argc, char *argv[])

If the pwd command is invoked with any number of arguments, the current working path of the shell shall be displayed.

globalScope> Global rm_main (int argc, char *argv[])

If rm is given a single argument containing a valid path to a file, rm shall delete that file from the image.

If rm is given any number of arguments other than one, rm shall exit printing, "Invalid argument count; rm takes the path of the file to remove.".

If rm is used successfully, rm shall attempt to collapse the directory it is deleting from.

If rm is given a single argument containing a valid path to a file, however, that file is a subdirectory or long file entry, then rm shall exit printing "Could not rm file [path].".

globalScope> Global rmdir_main (int argc, char *argv[])

If rmdir is provided with a single argument that is a valid path to a directory, rmdir shall remove that folder from the mounted image.

If rmdir is provided with a single argument that is a valid path to a directory containing any file and/or directory, rmdir shall exit printing the message, "Directory still has files.".

If rmdir is provided with a single argument that is a valid path to something other than a subdirectory, or a long file header, rmdir shall exit printing the message, "Specified file [file_name] is not a directory.".

If rmdir is provided with a single argument that is an invalid path rmdir shall exit printing the message, "Directory [path] could not be found!".

If rmdir is provided with a number of arguments other than one, rmdir shall exit printing, "Invalid argument count; rmdir takes the path of the directory to remove.".

If rmdir is directed to delete the current working directory, the working directory has no files within it, and the user name of the current user can be obtained, rmdir shall exit printing "Nice try [user_name], but deleting the directory you are currently in is not allowed.".

If rmdir is directed to delete the current working directory and the working directory has no files within it, rmdir shall exit printing, "Deleting the directory you are currently in is not allowed.".

If rmdir successfully deletes a directory, rmdir shall attempt to collapse the parent directory deleted from.

globalScope> Global touch_main (int argc, char *argv[])

If provided with a single argument containing a non-existent filename, touch shall create a file with the given name within the current working directory.

If provided with a single argument containing a path and culminating in a non-existent filename, touch shall create a file with the given filename within the provided directory.

If provided with anything other than one argument, touch shall exit printing, "Invalid argument count; touch takes the path of the file to create.".

If provided with a single argument containing a valid path to an existing file or directory, touch shall print "File [file_name] touched." and shall update the timestamp of the file or directory.

If provided with ".", "..", touch shall exit printing, "[entry] is not allowed.".

If during the process of trying to create a new file, mkdir cannot allocate a file sector, touch shall exit printing, "Failed to allocate file sector.".

If there is not enough room in a directory to add a new file, a successful touch call shall expand the directory before attempting to create the new file.

If during the process of trying to create a new file, touch cannot allocate a file header, touch shall exit printing, "Failed to allocate file header.".

If successful in creating a file, touch shall add a timestamp accurate to the closest two seconds to the newly created file.

3 Bug List

globalScope> Global read_sector (int sector_number, unsigned char *buffer)

DEPRECATED - use find_sector() instead!

globalScope> Global write_sector (int sector_number, unsigned char *buffer)

DEPRECATED - use find_sector() instead!

4 Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

5 File Index

5.1 File List

Here is a list of all files with brief descriptions:

6 Data Structure Documentation

6.1 BOOT_SECTOR Struct Reference

A struct that stores boot sector information.

`#include <bootsector.h>`

Data Fields

- uint8_t bootstrap_jump [3]
- char OEM_name [8]
- uint16_t bytes_per_sector: 16
- uint8_t sectors_per_cluster: 8
- uint16_t number_of_reserved_sectors: 16
- uint8_t number_of_FATs: 8
- uint16_t max_root_dir_entries: 16
- uint16_t total_sector_count: 16
- uint8_t ignore2: 8
- uint16_t sectors_per_FAT: 16

- uint16_t sectors_per_track: 16
- uint16_t number_of_heads: 16
- uint32_t ignore3: 32
- uint32_t FAT32_total_sector_count: 32
- uint16_t ignore4: 16
- uint8_t boot_signature: 8
- uint32_t volume_id: 32
- char volume_label [11]
- char file_system_type [8]

6.1.1 Detailed Description

A struct that stores boot sector information.

Definition at line 11 of file bootsector.h.

6.1.2 Field Documentation

6.1.2.1 uint8_t boot_signature

Definition at line 36 of file bootsector.h.

6.1.2.2 uint8_t bootstrap_jump[3]

Definition at line 13 of file bootsector.h.

6.1.2.3 uint16_t bytes_per_sector

Definition at line 17 of file bootsector.h.

6.1.2.4 uint32_t FAT32_total_sector_count

Definition at line 32 of file bootsector.h.

6.1.2.5 char file_system_type[8]

Definition at line 41 of file bootsector.h.

6.1.2.6 uint8_t ignore2

Definition at line 24 of file bootsector.h.

6.1.2.7 uint32_t ignore3

Definition at line 30 of file bootsector.h.

6.1.2.8 uint16_t ignore4

Definition at line 34 of file bootsector.h.

6.1.2.9 uint16_t max_root_dir_entries

Definition at line 21 of file bootsector.h.

6.1.2.10 uint8_t number_of_FATs

Definition at line 20 of file bootsector.h.

6.1.2.11 uint16_t number_of_heads

Definition at line 28 of file bootsector.h.

6.1.2.12 uint16_t number_of_reserved_sectors

Definition at line 19 of file bootsector.h.

6.1.2.13 char OEM_name[8]

Definition at line 15 of file bootsector.h.

6.1.2.14 uint8_t sectors_per_cluster

Definition at line 18 of file bootsector.h.

6.1.2.15 uint16_t sectors_per_FAT

Definition at line 26 of file bootsector.h.

6.1.2.16 uint16_t sectors_per_track

Definition at line 27 of file bootsector.h.

6.1.2.17 uint16_t total_sector_count

Definition at line 22 of file bootsector.h.

6.1.2.18 uint32_t volume_id

Definition at line 38 of file bootsector.h.

6.1.2.19 char volume_label[11]

Definition at line 39 of file bootsector.h.

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

- include/bootsector.h

6.2 FILE_DATE Struct Reference

A struct to hold a file date.

`#include <timeanddate.h>`

Data Fields

- unsigned int month: 4
  Four bits to hold month.
- unsigned int year: 7
  Seven bits to hold years (since 1980).

- unsigned int day: 5

  Five bits to hold day.

6.2.1 Detailed Description

A struct to hold a file date.

Definition at line 23 of file timeanddate.h.

6.2.2 Field Documentation

6.2.2.1 unsigned int day

Five bits to hold day.

Definition at line 30 of file timeanddate.h.

6.2.2.2 unsigned int month

Four bits to hold month.

Definition at line 26 of file timeanddate.h.

6.2.2.3 unsigned int year

Seven bits to hold years (since 1980).

Definition at line 28 of file timeanddate.h.

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

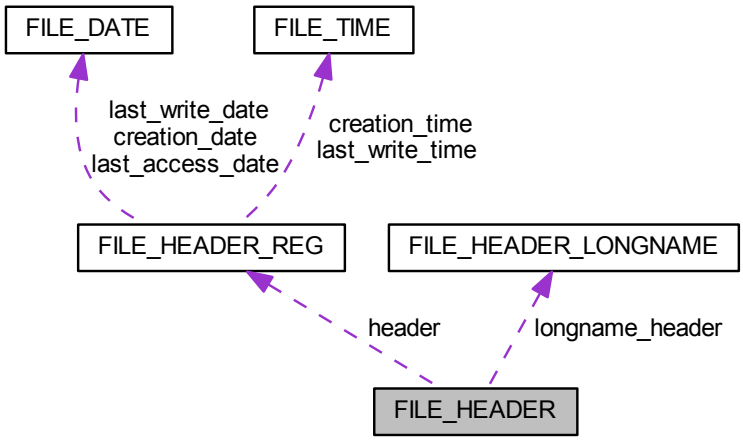
- include/timeanddate.h

6.3 FILE_HEADER Union Reference

A union of the regular 8.1 file header and the long name file header.

```
#include <fileio.h>
```

Collaboration diagram for FILE_HEADER:



Data Fields

- FILE_HEADER_REG header
- FILE_HEADER_LONGNAME longname_header

6.3.1 Detailed Description

A union of the regular 8.1 file header and the long name file header.

Definition at line 62 of file fileio.h.

6.3.2 Field Documentation

6.3.2.1 FILE_HEADER_REG header

Definition at line 64 of file fileio.h.

6.3.2.2 FILE_HEADER_LONGNAME longname_header

Definition at line 65 of file fileio.h.

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

- include/fileio.h

6.4 FILE_HEADER_LONGNAME Struct Reference

A struct to store and manipulate long name file headers.

`#include <fileio.h>`

Data Fields

- uint8_t index
- char16_t name1 [5]

  First five characters of filename.
- uint8_t attributes
- uint8_t type
- uint8_t checksum
- char16_t name2 [6]

  Next six characters of filename.
- uint16_t __pad0__: 16
- char16_t name3 [2]

  Last two characters of filename.

6.4.1 Detailed Description

A struct to store and manipulate long name file headers.

Definition at line 41 of file fileio.h.

6.4.2 Field Documentation

6.4.2.1 uint16_t __pad0__

Definition at line 55 of file fileio.h.

6.4.2.2 uint8_t attributes

Definition at line 48 of file fileio.h.

6.4.2.3 uint8_t checksum

Definition at line 50 of file fileio.h.

6.4.2.4 uint8_t index

Definition at line 43 of file fileio.h.

6.4.2.5 char16_t name1[5]

First five characters of filename.

Definition at line 46 of file fileio.h.

6.4.2.6 char16_t name2[6]

Next six characters of filename.

Definition at line 53 of file fileio.h.

6.4.2.7 char16_t name3[2]

Last two characters of filename.

Definition at line 58 of file fileio.h.

6.4.2.8 uint8_t type

Definition at line 49 of file fileio.h.

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

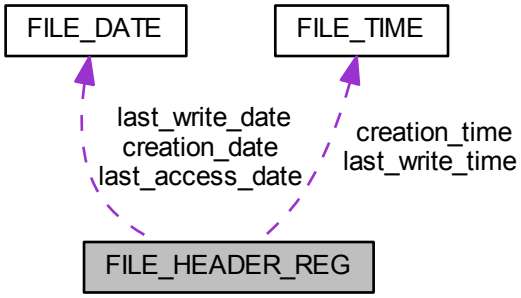
- include/fileio.h

6.5 FILE_HEADER_REG Struct Reference

A struct to store and manipulate long name file headers.

`#include <fileio.h>`

Collaboration diagram for FILE_HEADER_REG:



Data Fields

- char file_name [8]
- char extension [3]
- uint8_t attributes
- uint16_t reserved
- FILE_TIME creation_time
- FILE_DATE creation_date
- FILE_DATE last_access_date
- uint16_t ignore
- FILE_TIME last_write_time
- FILE_DATE last_write_date
- uint16_t first_logical_cluster
- uint32_t file_size

6.5.1 Detailed Description

A struct to store and manipulate long name file headers.

Definition at line 18 of file fileio.h.

6.5.2 Field Documentation

6.5.2.1 uint8_t attributes

Definition at line 23 of file fileio.h.

6.5.2.2 FILE_DATE creation_date

Definition at line 28 of file fileio.h.

6.5.2.3 FILE_TIME creation_time

Definition at line 27 of file fileio.h.

6.5.2.4 char extension[3]

Definition at line 21 of file fileio.h.

6.5.2.5 char file_name[8]

Definition at line 20 of file fileio.h.

6.5.2.6 uint32_t file_size

Definition at line 37 of file fileio.h.

6.5.2.7 uint16_t first_logical_cluster

Definition at line 36 of file fileio.h.

6.5.2.8 uint16_t ignore

Definition at line 31 of file fileio.h.

6.5.2.9 FILE_DATE last_access_date

Definition at line 29 of file fileio.h.

6.5.2.10 FILE_DATE last_write_date

Definition at line 34 of file fileio.h.

6.5.2.11 FILE_TIME last_write_time

Definition at line 33 of file fileio.h.

6.5.2.12 uint16_t reserved

Definition at line 25 of file fileio.h.

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

- include/fileio.h

6.6 FILE_TIME Struct Reference

A struct to hold a file time.

`#include <timeanddate.h>`

Data Fields

- unsigned int doubleseconds: 5

  Five bits to hold number of seconds (divided by 2).

- unsigned int minutes: 6

  Six bits to hold number of minutes.

- unsigned int hours: 5

  Five bits to hold number of hours.

6.6.1 Detailed Description

A struct to hold a file time.

Definition at line 12 of file timeanddate.h.

6.6.2 Field Documentation

6.6.2.1 unsigned int doubleseconds

Five bits to hold number of seconds (divided by 2).

Definition at line 15 of file timeanddate.h.

6.6.2.2 unsigned int hours

Five bits to hold number of hours.

Definition at line 19 of file timeanddate.h.

6.6.2.3 unsigned int minutes

Six bits to hold number of minutes.

Definition at line 17 of file timeanddate.h.

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

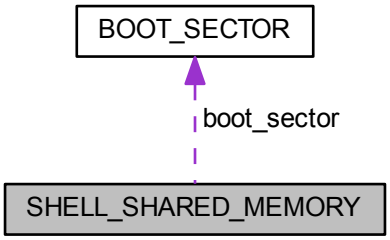
- include/timeanddate.h

6.7 SHELL_SHARED_MEMORY Struct Reference

A structure to facilitate shared data between shell and applications.

`#include <sharedmemory.h>`

Collaboration diagram for SHELL_SHARED_MEMORY:



Data Fields

- BOOT_SECTOR boot_sector

  A full copy of the boot sector in use.
- int current_dir_flc

  First Logical Cluster of current directory. If in root directory this will be 0.
- void * current_dir_offset

  Offset into file system for file entry that describes this subdirectory.
- char image_path [MAX_SHM_PATH_SIZE]

  String holding a path to the current loaded working drive image.
- char working_dir_path [MAX_SHM_PATH_SIZE]

  String holding a path to the current working directory.
- int stack_top_index

  The index of the top entry in the directory stack.
- void * directory_stack [MAX_DIR_STACK_ENTRIES]

  The directory stack used for quick working directory traversal.
- int next_free_fat

  The index of the next free fat.

6.7.1 Detailed Description

A structure to facilitate shared data between shell and applications.

Definition at line 11 of file sharedmemory.h.

6.7.2 Field Documentation

6.7.2.1 BOOT_SECTOR boot_sector

A full copy of the boot sector in use.

Definition at line 14 of file sharedmemory.h.

6.7.2.2 int current_dir_flc

First Logical Cluster of current directory. If in root directory this will be 0.

Definition at line 16 of file sharedmemory.h.

6.7.2.3 void* current_dir_offset

Offset into file system for file entry that describes this subdirectory.

Definition at line 18 of file sharedmemory.h.

6.7.2.4 void* directory_stack[MAX_DIR_STACK_ENTRIES]

The directory stack used for quick working directory traversal.

Definition at line 29 of file sharedmemory.h.

6.7.2.5 char image_path[MAX_SHM_PATH_SIZE]

String holding a path to the current loaded working drive image.

Definition at line 21 of file sharedmemory.h.

6.7.2.6 int next_free_fat

The index of the next free fat.

Definition at line 32 of file sharedmemory.h.

6.7.2.7 int stack_top_index

The index of the top entry in the directory stack.

Directory stack

Definition at line 27 of file sharedmemory.h.

6.7.2.8 char working_dir_path[MAX_SHM_PATH_SIZE]

String holding a path to the current working directory.

Definition at line 23 of file sharedmemory.h.

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

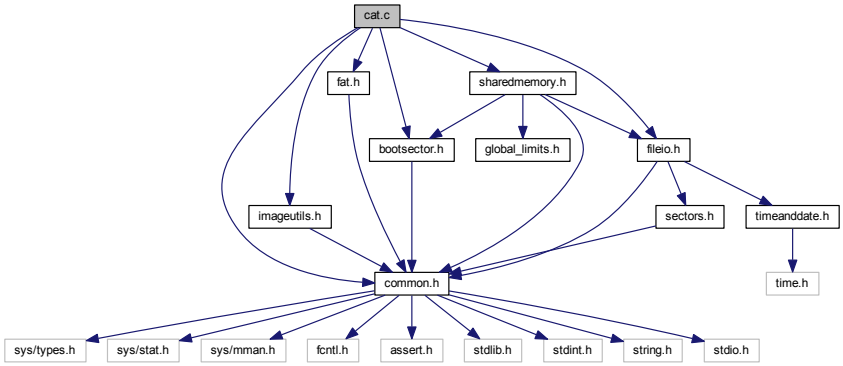
- include/sharedmemory.h

7 File Documentation

7.1 cat.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
#include "fileio.h"
```

Include dependency graph for cat.c:



Functions

- int cat_main (int argc, char ∗argv[ ])

  Main function for cat command.

- int main (int argc, char ∗argv[ ])

7.1.1 Function Documentation

7.1.1.1 int cat_main (int *argc,* char ∗ *argv[ ]*)

Main function for cat command.

Test If given the name of a file path culminating in a file name with an extension, cat shall display file contents.

If given '.' or '..', cat shall print "Cannot use cat on ./...".

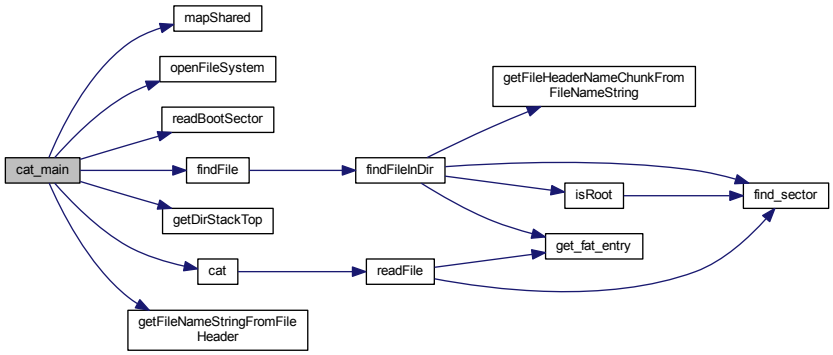
If number of arguments is greater or less than one, cat shall print "Invalid argument count; cat takes a file name to cat.".

If path given points to root, attributes indicate a long filename (should never occur), or a file is not a subdirectory, cat shall print "File %s is incompatible with cat.".

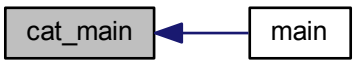
If path given does not lead to a file at all, cat shall print "Could not find file [filename] to cat!".

Definition at line 14 of file cat.c.

Here is the call graph for this function:



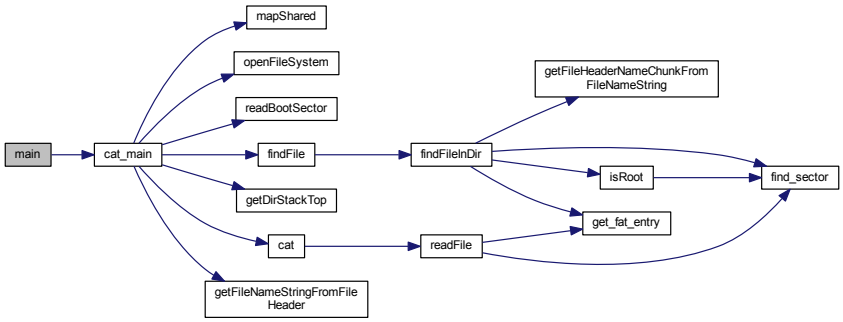
Here is the caller graph for this function:



7.1.1.2 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 62 of file cat.c.

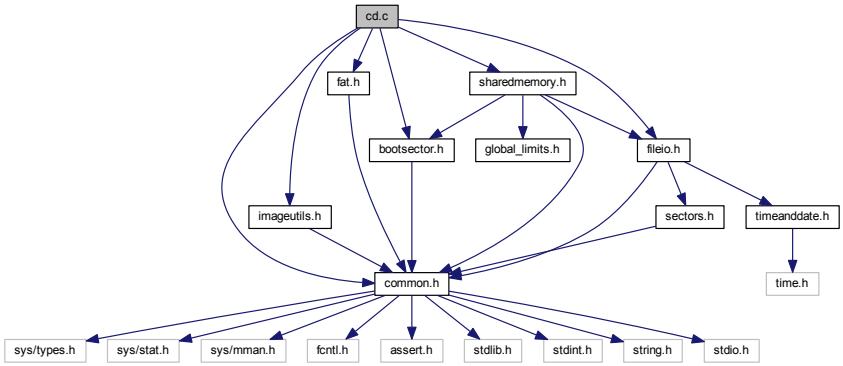
Here is the call graph for this function:



7.2 cd.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
#include "fileio.h"
```

Include dependency graph for cd.c:



Functions

- int cd_main (int argc, char ∗argv[ ])

  Main function for cd command.

- int main (int argc, char ∗argv[ ])

7.2.1 Function Documentation

7.2.1.1 int cd_main (int *argc,* char ∗ *argv[ ]*)

Main function for cd command.

Test If cd is provided with a valid path, the working directory shall be set to that path.

If there is any quantity of arguments other than one for a path, cd shall exit printing, "Invalid argument count; cd takes a file name to search for.".

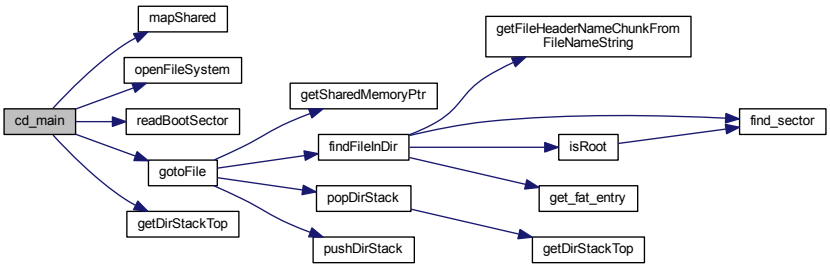
If cd ends up in root, the string "In root!" shall be printed.

If cd cannot move to the path provided, it shall print "Could not find file/folder: [path]".

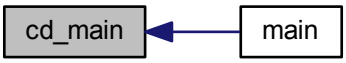
cd shall be able to support . and .. and relative as well as absolute paths.

Definition at line 14 of file cd.c.

Here is the call graph for this function:



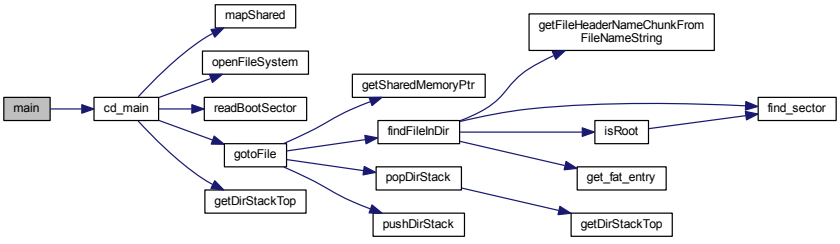
Here is the caller graph for this function:



7.2.1.2 int main (int *argc*, char * *argv[]*)

Definition at line 119 of file cd.c.

Here is the call graph for this function:

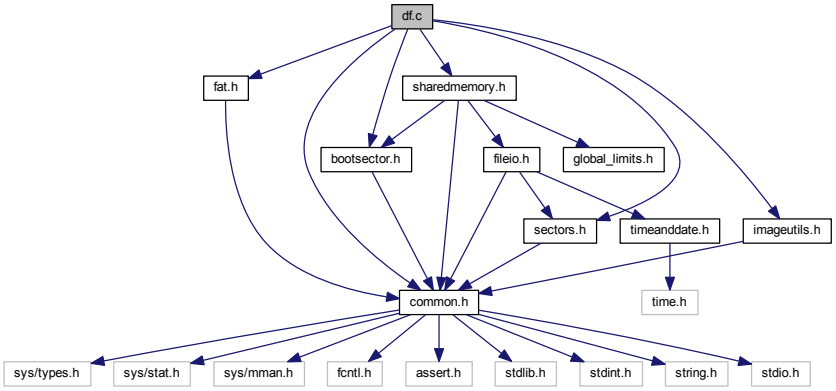


7.3 df.c File Reference

```
#include "common.h"
```

```
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
#include "sectors.h"
```

Include dependency graph for df.c:



Functions

- int df_main (int argc, char ∗argv[ ])

  Main function for df command.

- int main (int argc, char ∗argv[ ])

7.3.1 Function Documentation

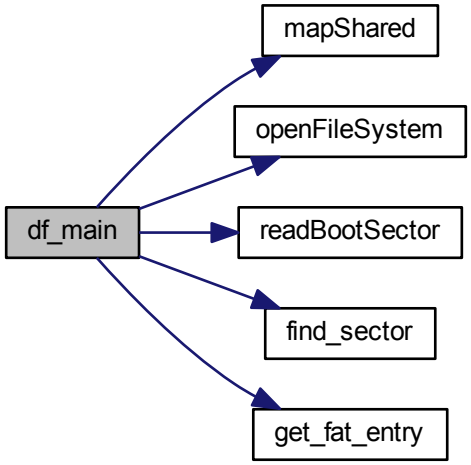
7.3.1.1 int df_main (int *argc*, char ∗ *argv[ ]*)

Main function for df command.

Test If called with any number of arguments, df shall display the number of K-blocks as well as the FAT sector count, the used sector count, the free sector count, and the percentage of sectors used.

Definition at line 10 of file df.c.

Here is the call graph for this function:

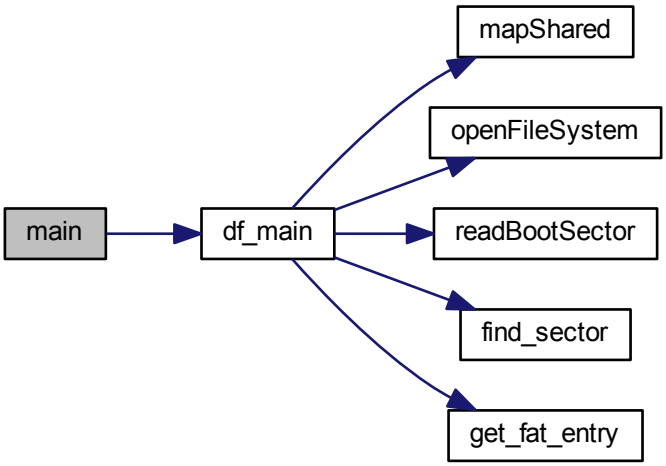


Here is the caller graph for this function:

7.3.1.2 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 47 of file df.c.

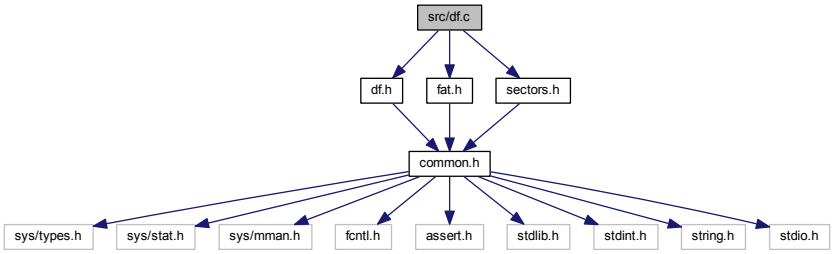
Here is the call graph for this function:



7.4 src/df.c File Reference

```
#include "df.h"
#include "fat.h"
#include "sectors.h"
```

Include dependency graph for df.c:

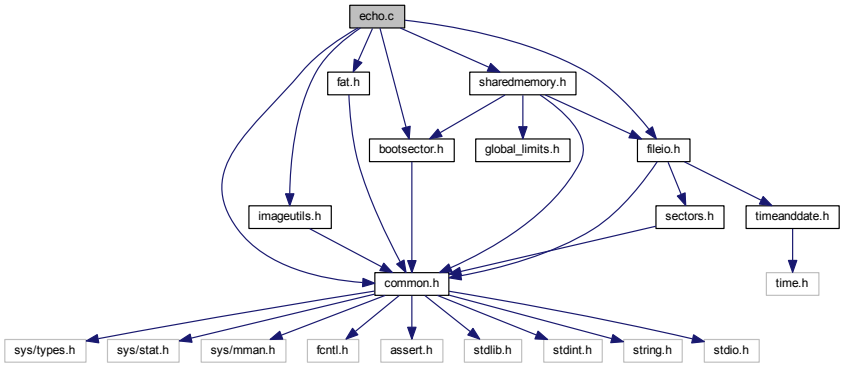


7.5 echo.c File Reference

```
#include "common.h"
```

```
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
#include "fileio.h"
```

Include dependency graph for echo.c:



Functions

- int echo_main (int argc, char ∗argv[ ])

  Main function for echo.

- int main (int argc, char ∗argv[ ])

7.5.1 Function Documentation

7.5.1.1 int echo_main (int *argc,* char ∗ *argv[ ]*)

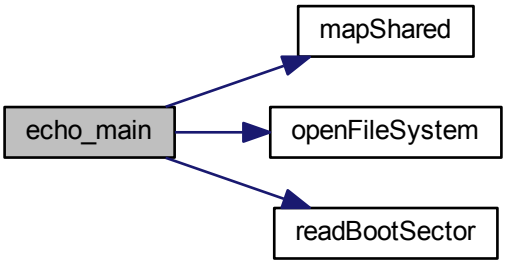
Main function for echo.

Test Echo shall print the first argument provided to it to console.

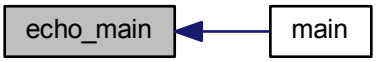
If echo is provided with a number of arguments other than one, echo shall exit printing, "Invalid argument count; need something to echo.".

Definition at line 11 of file echo.c.

Here is the call graph for this function:



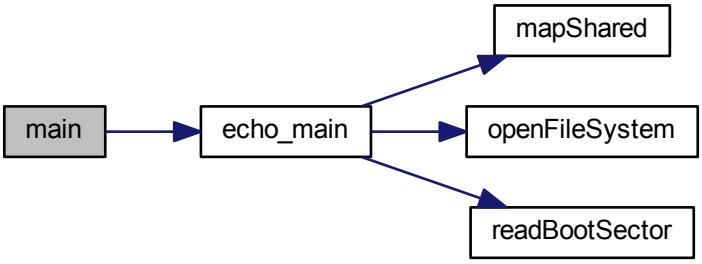
Here is the caller graph for this function:



7.5.1.2 int main (int *argc*, char * *argv[]*)

Definition at line 32 of file echo.c.

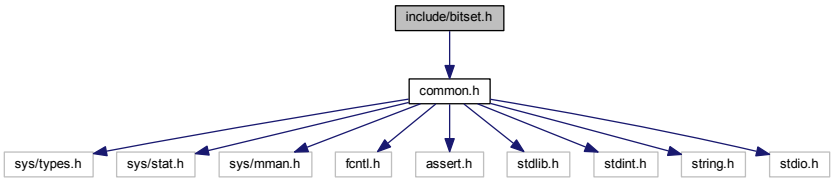
Here is the call graph for this function:



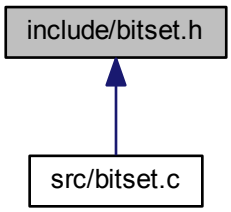
7.6 include/bitset.h File Reference

```
#include "common.h"
```

Include dependency graph for bitset.h:



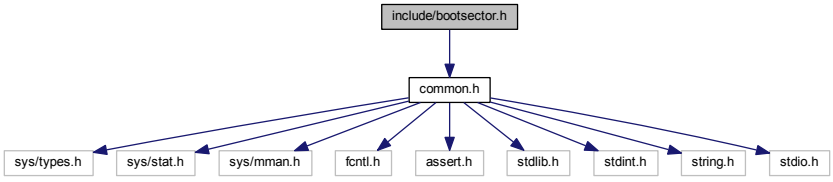
This graph shows which files directly or indirectly include this file:



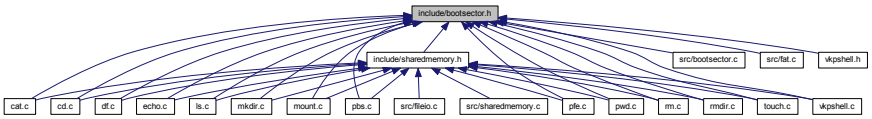
7.7 include/bootsector.h File Reference

```
#include "common.h"
```

Include dependency graph for bootsector.h:



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



Data Structures

- struct BOOT_SECTOR

  A struct that stores boot sector information.

Typedefs

- typedef struct BOOT_SECTOR BOOT_SECTOR

Functions

- void readBootSector ()

  Reads the boot sector from sector 0 on the file system.

- BOOT_SECTOR * getBootSector (uint8_t *fileSystem)
- void printBootSector (BOOT_SECTOR *bootSector)

  Prints the contents of the boot sector to stdout.

Variables

- BOOT_SECTOR PBS_BOOT_SEC
- uint16_t BYTES_PER_SECTOR

7.7.1 Typedef Documentation

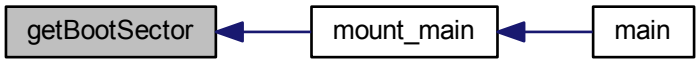
7.7.1.1 typedef struct BOOT_SECTOR BOOT_SECTOR

7.7.2 Function Documentation

7.7.2.1 BOOT_SECTOR* getBootSector (uint8_t * *fileSystem*)

Definition at line 14 of file bootsector.c.

Here is the caller graph for this function:



7.7.2.2 void printBootSector (BOOT_SECTOR * *bootSector*)

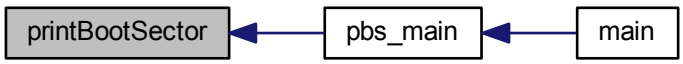
Prints the contents of the boot sector to stdout.

Parameters

in	*bootSector*	A pointer to a BOOT_SECTOR object holding the information to print.

Definition at line 19 of file bootsector.c.

Here is the caller graph for this function:

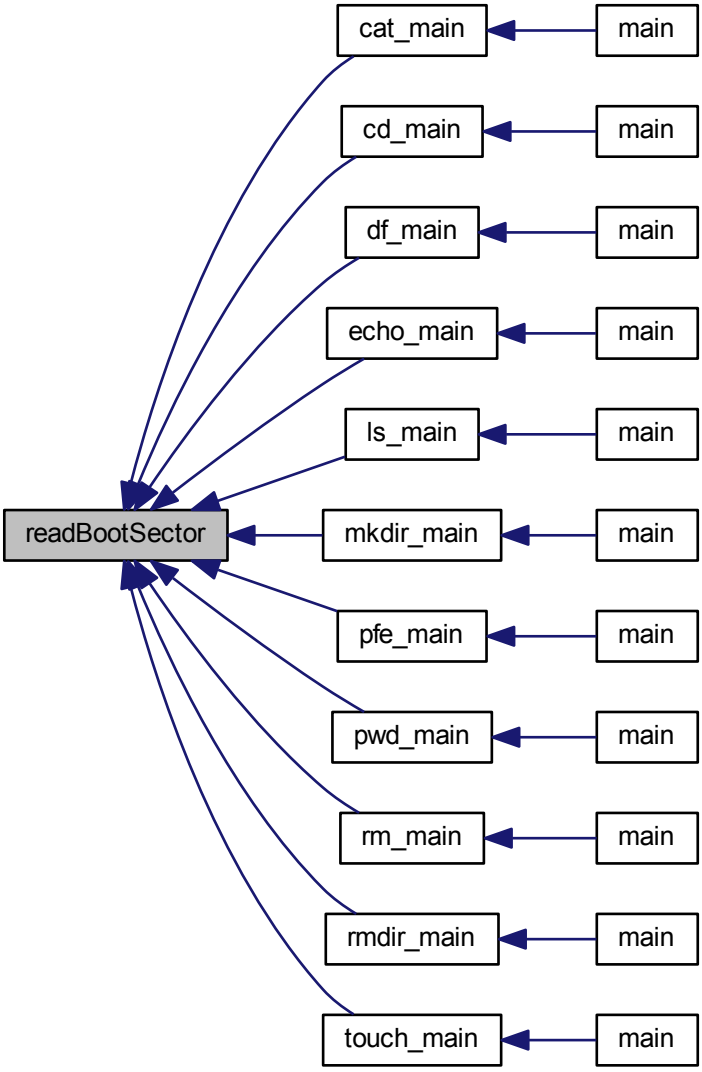


7.7.2.3 void readBootSector ( )

Reads the boot sector from sector 0 on the file system.

Definition at line 8 of file bootsector.c.

Here is the caller graph for this function:



7.7.3 Variable Documentation

7.7.3.1 uint16_t BYTES_PER_SECTOR

Definition at line 6 of file bootsector.c.

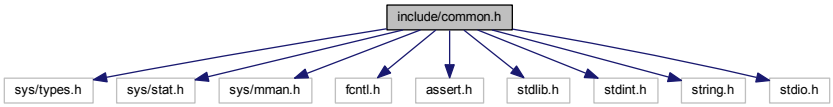
7.7.3.2 BOOT_SECTOR PBS_BOOT_SEC

Definition at line 5 of file bootsector.c.

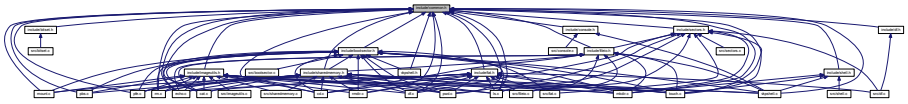
7.8 include/common.h File Reference

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <assert.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <stdio.h>
```

Include dependency graph for common.h:



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



Macros

- #define true 1
- #define false 0

Typedefs

- typedef int bool

Variables

- uint16_t BYTES_PER_SECTOR

  The number of bytes per sector.

7.8.1 Macro Definition Documentation

7.8.1.1 #define false 0

Definition at line 23 of file common.h.

7.8.1.2 #define true 1

Definition at line 22 of file common.h.

7.8.2 Typedef Documentation

7.8.2.1 typedef int bool

Definition at line 21 of file common.h.

7.8.3 Variable Documentation

7.8.3.1 uint16_t BYTES_PER_SECTOR

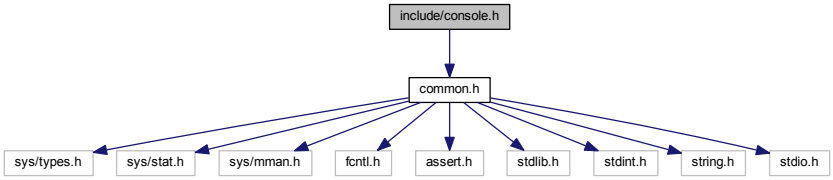
The number of bytes per sector.

Definition at line 6 of file bootsector.c.

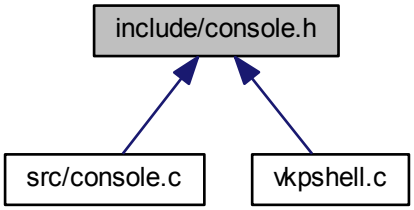
7.9 include/console.h File Reference

```
#include "common.h"
```

Include dependency graph for console.h:



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



Functions

- char * getLine ()

  Gets a line of input from the user.

7.9.1 Function Documentation

7.9.1.1 char* getLine ( )

Gets a line of input from the user.

Returns

Returns a pointer to a C-string.

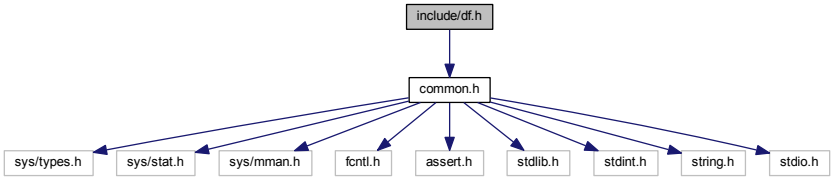
Definition at line 3 of file console.c.

Here is the caller graph for this function:

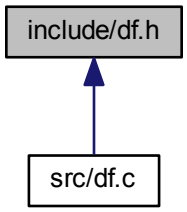
7.10 include/df.h File Reference

```
#include "common.h"
```

Include dependency graph for df.h:



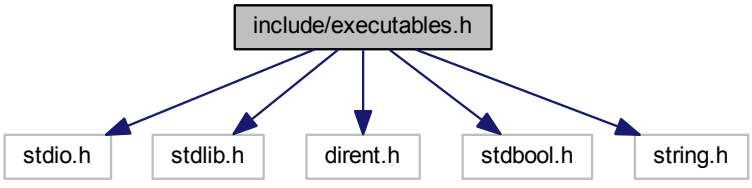
This graph shows which files directly or indirectly include this file:



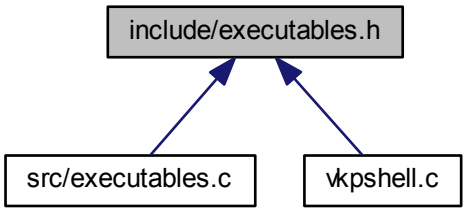
7.11 include/executables.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>
#include <stdbool.h>
#include <string.h>
```

Include dependency graph for executables.h:



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



Macros

- #define _EXECUTABLES_H
- #define EXECUTABLES_ALLOC_CHUNK_SIZE 16

  Allocation chunk size for executables list.
- #define ELF_HEADER_SIZE 4

  The amount of initial bytes needed to tell if a file is an ELF executable.

Functions

- bool isELF (FILE *fp)

  Determines if a file is a valid executable ELF file.
- void freeExecutableList ()

  Frees the executables list.
- void addExecutable (char *name)

  Adds an executable to the executables list.
- void printExecutables ()

  Prints a list of all executables.
- void trimExecutables ()

  Trims off unused executable entries.
- void addDirToExecutableList (char *dir)

  Adds the executables of a directory to the executable list.

Variables

- const unsigned char ELF_HEADER_BYTES [ELF_HEADER_SIZE]
- char ** EXECUTABLES

  An array of strings of executables allowed by the shell.
- size_t EXECUTABLES_SIZE

  Stores the number of entry slots allocated in the executable list.
- size_t NUM_EXECUTABLES

  Stores the actual number of entries populated in the executable list.

7.11.1 Macro Definition Documentation

7.11.1.1 #define _EXECUTABLES_H

Definition at line 3 of file executables.h.

7.11.1.2 #define ELF_HEADER_SIZE 4

The amount of initial bytes needed to tell if a file is an ELF executable.

Definition at line 18 of file executables.h.

7.11.1.3 #define EXECUTABLES_ALLOC_CHUNK_SIZE 16

Allocation chunk size for executables list.

Definition at line 14 of file executables.h.

7.11.2 Function Documentation

7.11.2.1 void addDirToExecutableList (char * *dir*)

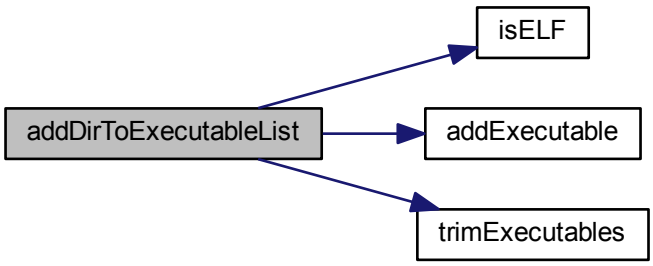
Adds the executables of a directory to the executable list.

Parameters

in	*dir*	A string path to a directory.

Definition at line 91 of file executables.c.

Here is the call graph for this function:



Here is the caller graph for this function:

7.11.2.2 void addExecutable (char * *name*)

Adds an executable to the executables list.

Parameters

in	*name*	A null-terminated character string representing an executable's filename.

Definition at line 44 of file executables.c.

Here is the caller graph for this function:

7.11.2.3 void freeExecutableList ( )

Frees the executables list.

Definition at line 34 of file executables.c.

Here is the caller graph for this function:

7.11.2.4 bool isELF (FILE * *fp*)

Determines if a file is a valid executable ELF file.

Parameters

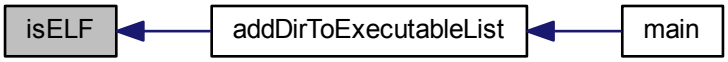
in	*fp*	A FILE pointer to an open file.

Return values

true	The file is a valid ELF.
false	The file is not executable ELF or the file has not been opened.

Definition at line 14 of file executables.c.

Here is the caller graph for this function:

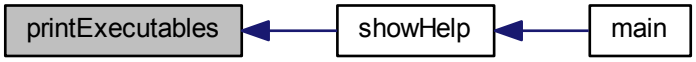


7.11.2.5 void printExecutables ( )

Prints a list of all executables.

Definition at line 69 of file executables.c.

Here is the caller graph for this function:



7.11.2.6 void trimExecutables ( )

Trims off unused executable entries.

Definition at line 81 of file executables.c.

Here is the caller graph for this function:

7.11.3 Variable Documentation

7.11.3.1 const unsigned char ELF_HEADER_BYTES[ELF_HEADER_SIZE]

Definition at line 4 of file executables.c.

7.11.3.2 char** EXECUTABLES

An array of strings of executables allowed by the shell.

Definition at line 6 of file executables.c.

7.11.3.3 size_t EXECUTABLES_SIZE

Stores the number of entry slots allocated in the executable list.

Definition at line 9 of file executables.c.

7.11.3.4 size_t NUM_EXECUTABLES

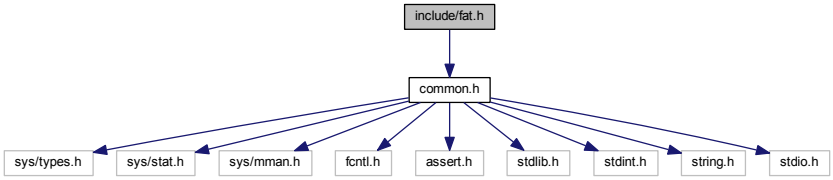
Stores the actual number of entries populated in the executable list.

Definition at line 12 of file executables.c.

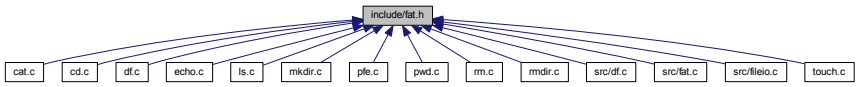
7.12 include/fat.h File Reference

```
#include "common.h"
```

Include dependency graph for fat.h:



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



Functions

- unsigned int get_fat_entry (int fat_entry_number, unsigned char *fat)
- void set_fat_entry (int fat_entry_number, int value, unsigned char *fat)
- uint16_t get_free_sector_count ()

  Gets the number of free sectors on disk.
- void pfe (int start, int end)

  Prints out a human-readable table of all of the FAT entries in the FAT table.
- void freeFatChain (int fatStart, bool zeroMemory)

  Frees a FAT chain.
- unsigned int getNextFreeSector ()

  Returns the number of the next free sector.
- unsigned int appendSector (int startSector)

  Links a sector onto the specified sector and updates the FAT tables to extend the FAT entry chain.

7.12.1 Function Documentation

7.12.1.1 unsigned int appendSector (int *startSector*)

Links a sector onto the specified sector and updates the FAT tables to extend the FAT entry chain.

Parameters

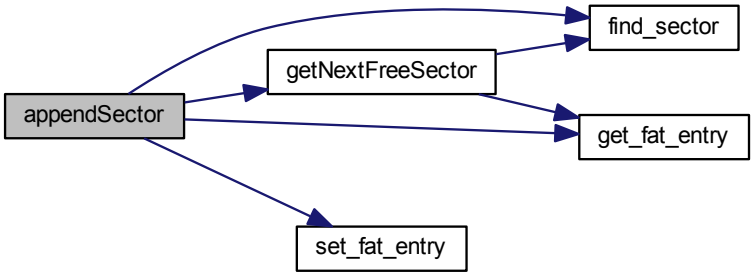
in	*startSector*	The sector number to append to.

Returns

Returns the sector that was allocated and appended to the end.

Definition at line 145 of file fat.c.

Here is the call graph for this function:



Here is the caller graph for this function:

7.12.1.2 void freeFatChain (int *fatStart,* bool *zeroMemory*)

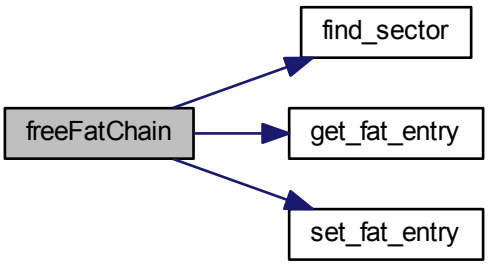
Frees a FAT chain.

Parameters

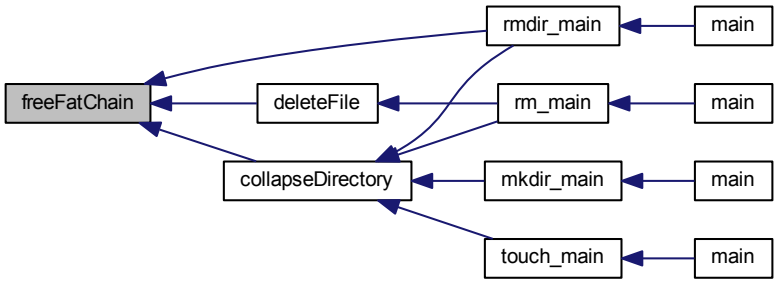
in	*fatStart*	An index of the fat entry to start at.
in	*zeroMemory*	A boolean value indicating whether or not to zero the freed memory.

Definition at line 118 of file fat.c.

Here is the call graph for this function:



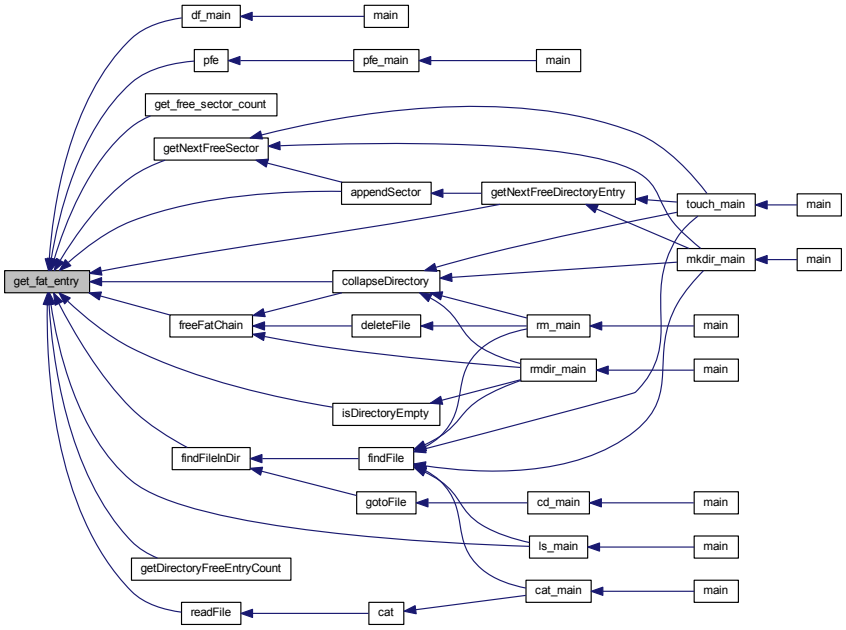
Here is the caller graph for this function:



7.12.1.3 unsigned int get_fat_entry (int *fat_entry_number,* unsigned char * *fat*)

Definition at line 5 of file fat.c.

Here is the caller graph for this function:



7.12.1.4 uint16_t get_free_sector_count ( )

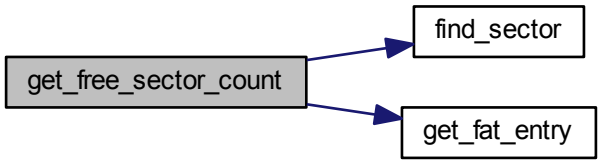
Gets the number of free sectors on disk.

Returns

Returns a uint16_t.

Definition at line 66 of file fat.c.

Here is the call graph for this function:



7.12.1.5 unsigned int getNextFreeSector ( )

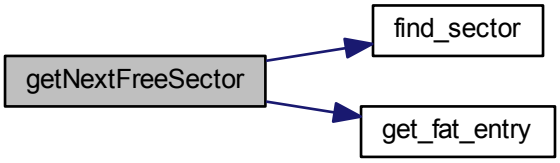
Returns the number of the next free sector.

Returns

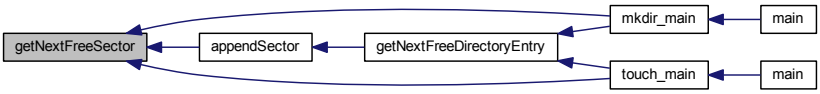
Returns the number of the next free sector as an unsigned int.

Definition at line 102 of file fat.c.

Here is the call graph for this function:



Here is the caller graph for this function:



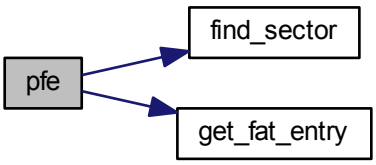
7.12.1.6 void pfe (int *start*, int *end*)

Prints out a human-readable table of all of the FAT entries in the FAT table.

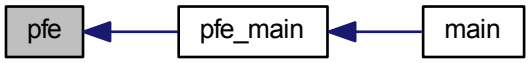
param[in] start The number of the first FAT entry to start reading from (start with 2 since first 2 are unused). param[in] end The number of the last FAT entry to read from (must be at least 2 since first 2 are unused).

Definition at line 84 of file fat.c.

Here is the call graph for this function:



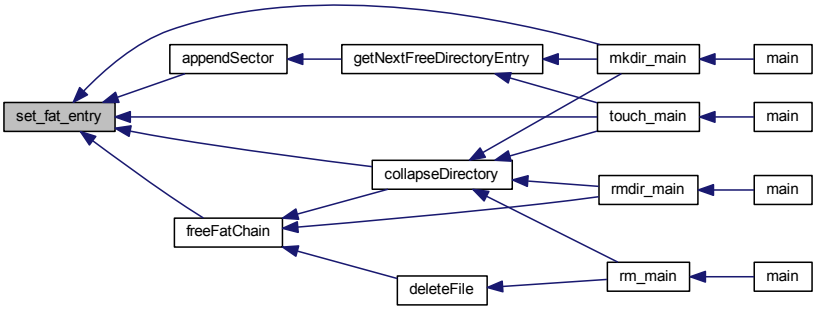
Here is the caller graph for this function:



7.12.1.7 void set_fat_entry (int *fat_entry_number,* int *value,* unsigned char * *fat*)

Definition at line 31 of file fat.c.

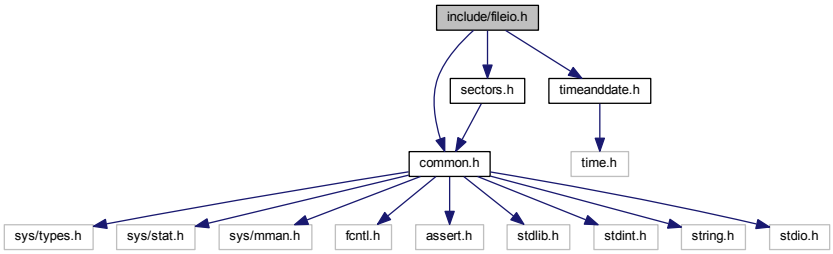
Here is the caller graph for this function:



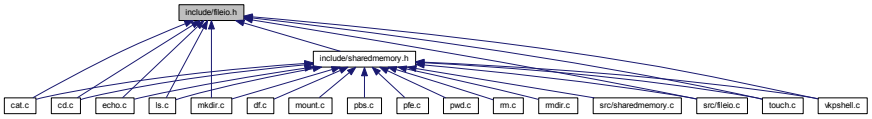
7.13 include/fileio.h File Reference

```
#include "common.h"
#include "timeanddate.h"
#include "sectors.h"
```

Include dependency graph for fileio.h:



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



Data Structures

- struct FILE_HEADER_REG

  A struct to store and manipulate long name file headers.
- struct FILE_HEADER_LONGNAME

  A struct to store and manipulate long name file headers.
- union FILE_HEADER

  A union of the regular 8.1 file header and the long name file header.

Macros

- #define FILE_DELETED_BYTE 0xE5

Typedefs

- typedef uint16_t char16_t

  An OS-independent type to store 16-bit characters.
- typedef struct FILE_HEADER_REG FILE_HEADER_REG

  A struct to store and manipulate long name file headers.
- typedef struct FILE_HEADER_LONGNAME FILE_HEADER_LONGNAME

  A struct to store and manipulate long name file headers.
- typedef union FILE_HEADER FILE_HEADER

  A union of the regular 8.1 file header and the long name file header.
- typedef enum FILE_ATTRIBUTE FILE_ATTRIBUTE

  An enumeration to hold file attribute flags.

Enumerations

- enum FILE_ATTRIBUTE {
  FILE_ATTR_READONLY = 1, FILE_ATTR_HIDDEN = 1 << 1, FILE_ATTR_SYSTEM = 1 << 2, FILE_ATTR↩
  _VOLUME_LABEL = 1 << 3,
  FILE_ATTR_SUBDIRECTORY = 1 << 4, FILE_ATTR_ARCHIVE = 1 << 5 }

  An enumeration to hold file attribute flags.

Functions

- char * getFileHeaderNameChunkFromFileNameString (char *filenameString)

  Gives an 11-byte name and extension block for a file header from a filename string.
- char * getFileNameStringFromFileHeader (FILE_HEADER_REG *header)

  Gives a filename as a string from a file header.
- void getNameFromLongNameFileHeader (const FILE_HEADER_LONGNAME *header, wchar_t *name)

  Takes a pointer to a wide character string (at least 13 characters allocated) and populates it with the filename from a longname file header.
- void printFileHeader (const FILE_HEADER *header)

  Prints out the contents of a file header to a human-readable form in the console.
- void readFile (const FILE_HEADER *header, void **buffer)

  Reads the contents of a file into a function-allocated buffer given a pointer to its file header and a pointer to store the buffer at.
- bool findFile (const char *name, const FILE_HEADER *searchLocation, FILE_HEADER_REG **found)

  Finds a file header with a specified name (and/or path)
- bool findFileInDir (const char *name, const FILE_HEADER *searchLocation, FILE_HEADER_REG **found)

  Finds a file header with a specified name.
- bool gotoFile (const char *name, const FILE_HEADER *searchLocation, FILE_HEADER_REG **found)

  Moves within the directory stack to a file header with a specified name (and/or path)
- void cat (const FILE_HEADER_REG *file)

  Given a regular 8.1 file header, prints out the contents of the file to console.
- void deleteFile (FILE_HEADER *header)

  Deletes a file given a pointer to a file header.
- int getDirectoryFreeEntryCount (FILE_HEADER *directory)

  Gets the number of free entries in a provided directory.
- void collapseDirectory (FILE_HEADER *directory)

  Collapses all files in a directory toward the front then drops any extra sectors.
- FILE_HEADER_REG * getNextFreeDirectoryEntry (FILE_HEADER *directory)

  Gets the next free entry of the provided directory? Will expand directory if required.
- bool isDirectoryEmpty (FILE_HEADER *directory)

  Checks if a directory is empty aside from the . and .. entries along with the long file headers.
- bool isRoot (void *file)

  Determines whether a given file header is a pointer to root.

7.13.1 Macro Definition Documentation

7.13.1.1 #define FILE_DELETED_BYTE 0xE5

Definition at line 81 of file fileio.h.

7.13.2 Typedef Documentation

7.13.2.1 typedef uint16_t char16_t

An OS-independent type to store 16-bit characters.

Definition at line 15 of file fileio.h.

7.13.2.2 typedef enum FILE_ATTRIBUTE FILE_ATTRIBUTE

An enumeration to hold file attribute flags.

7.13.2.3 typedef union FILE_HEADER FILE_HEADER

A union of the regular 8.1 file header and the long name file header.

7.13.2.4 typedef struct FILE_HEADER_LONGNAME FILE_HEADER_LONGNAME

A struct to store and manipulate long name file headers.

7.13.2.5 typedef struct FILE_HEADER_REG FILE_HEADER_REG

A struct to store and manipulate long name file headers.

7.13.3 Enumeration Type Documentation

7.13.3.1 enum FILE_ATTRIBUTE

An enumeration to hold file attribute flags.

Enumerator

FILE_ATTR_READONLY

FILE_ATTR_HIDDEN

FILE_ATTR_SYSTEM

FILE_ATTR_VOLUME_LABEL

FILE_ATTR_SUBDIRECTORY

FILE_ATTR_ARCHIVE

Definition at line 71 of file fileio.h.

7.13.4 Function Documentation

7.13.4.1 void cat (const FILE_HEADER_REG * *file*)

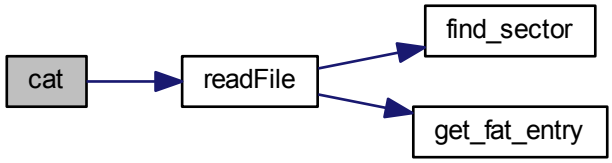
Given a regular 8.1 file header, prints out the contents of the file to console.

Parameters

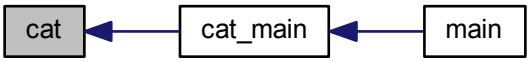
in	*file*	A pointer to a FILE_HEADER_REG.

Definition at line 758 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.2 void collapseDirectory (FILE_HEADER * *directory*)

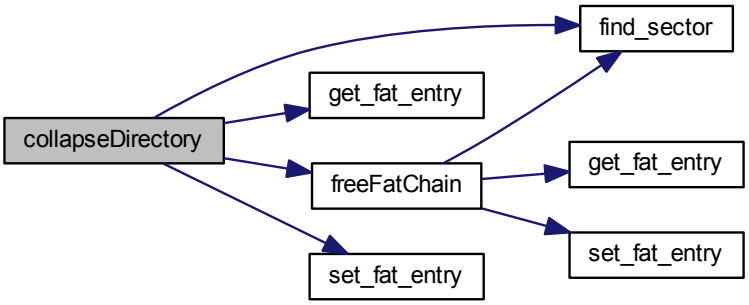
Collapses all files in a directory toward the front then drops any extra sectors.

Parameters

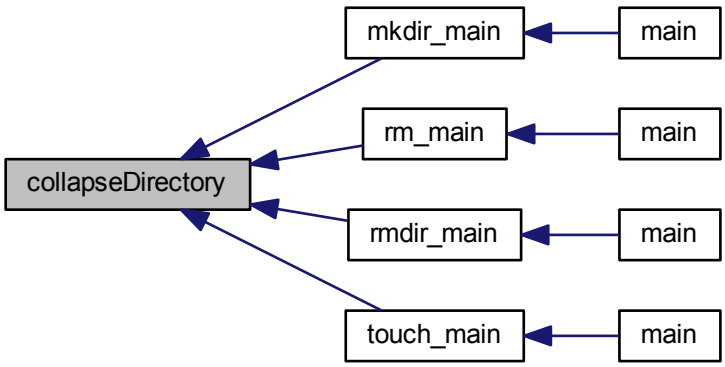
in	*directory*	A pointer to the FILE_HEADER of a directory to collapse.

Definition at line 567 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.3 void deleteFile (FILE_HEADER * *header*)

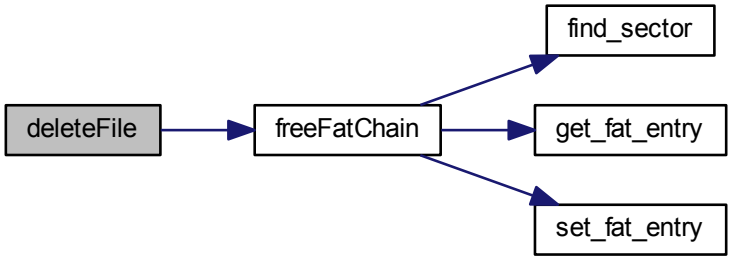
Deletes a file given a pointer to a file header.

Parameters

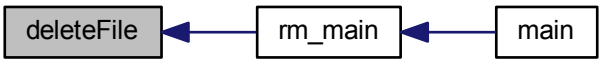
in	*header*	A pointer to the FILE_HEADER of the file to be deleted.

Definition at line 523 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.4 bool findFile (const char * *name*, const FILE_HEADER * *searchLocation*, FILE_HEADER_REG ** *found*)

Finds a file header with a specified name (and/or path)

Parameters

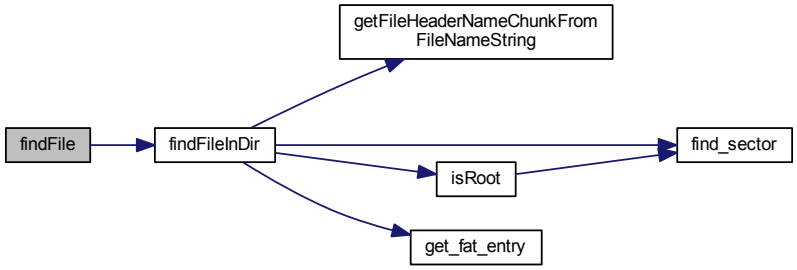
in	*name*	The name of the file to search for.
in	*searchLocation*	A pointer to a FILE_HEADER object to start searching from. This may be NULL to signify a search of the root directory.
out	*found*	A pointer to the file header, if found. This is NULL if root or if not found.

Return values

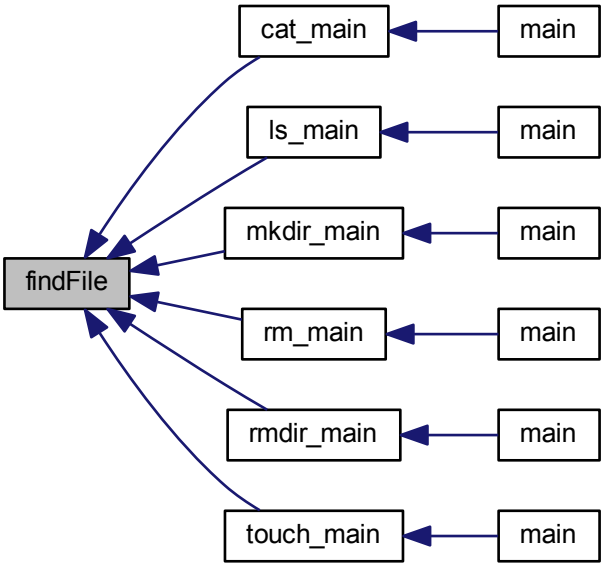
true	A file header with the information given was found. (If found is NULL and the return value is true, the file is root.)
false	The target file header could not be found.

Definition at line 340 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.5 bool findFileInDir (const char * *name*, const FILE_HEADER * *searchLocation*, FILE_HEADER_REG ** *found*)

Finds a file header with a specified name.

Parameters

in	*name*	The name of the file to search for.
in	*searchLocation*	A pointer to a FILE_HEADER object to start searching from. This may be NULL to signify a search of the root directory.
out	*found*	A pointer to the file header, if found. This is NULL if root or if not found.

Return values

true	A file header with the information given was found. (If found is NULL and the return value is true, the file is root.)
false	The target file header could not be found.

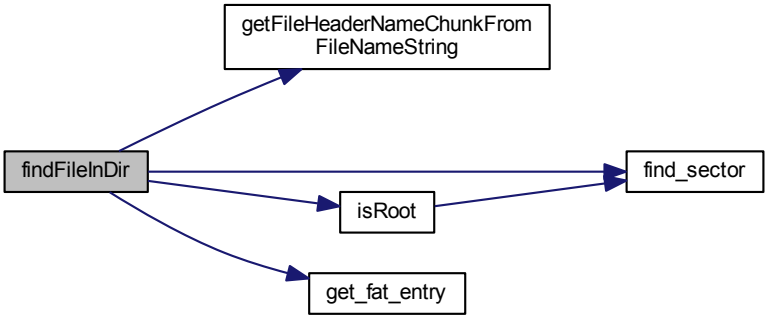
Remarks

Calls findFile().

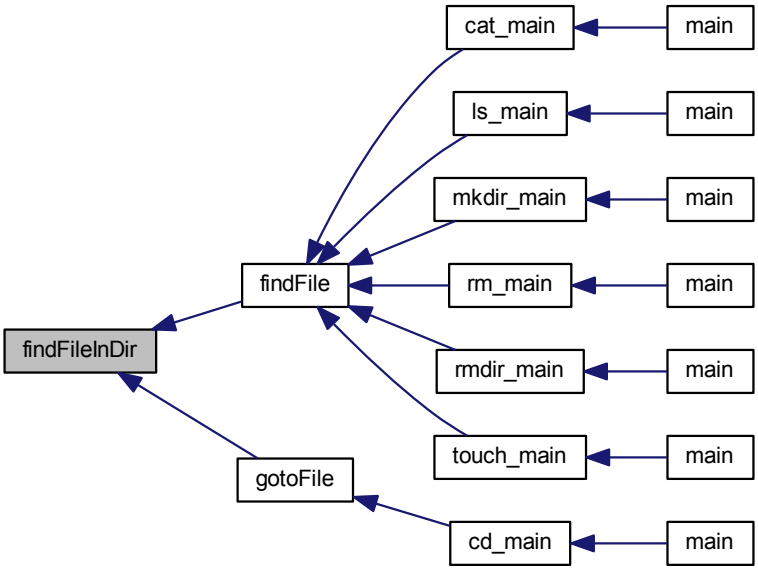
TODO: Check extensions

Definition at line 262 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.6 int getDirectoryFreeEntryCount (FILE_HEADER * *directory*)

Gets the number of free entries in a provided directory.

Parameters

in	*directory*	A pointer to the FILE_HEADER of a directory to get information from.

Returns

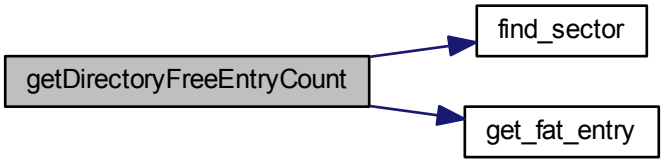
Returns the number of free entries in the given directory.

Return values

-1	Obtaining the free entry count was unsuccessful.

Definition at line 532 of file fileio.c.

Here is the call graph for this function:



7.13.4.7 char* getFileHeaderNameChunkFromFileNameString (char * *filenameString*)

Gives an 11-byte name and extension block for a file header from a filename string.

Remarks

Uses a static internal buffer char[11]. Not thread-safe.

Parameters

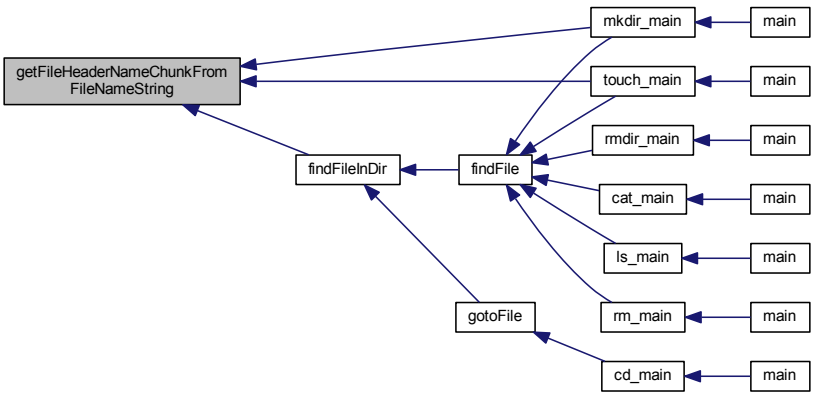
in	*filenameString*	A string containing a file name and extension (ex. "hello.txt").

Returns

Returns a static 11-char buffer that should match a file header's first 11 bytes (name and extension).

Definition at line 13 of file fileio.c.

Here is the caller graph for this function:



7.13.4.8 char* getFileNameStringFromFileHeader (FILE_HEADER_REG * *header*)

Gives a filename as a string from a file header.

Remarks

Uses a static internal string buffer. Not thread-safe.

Parameters

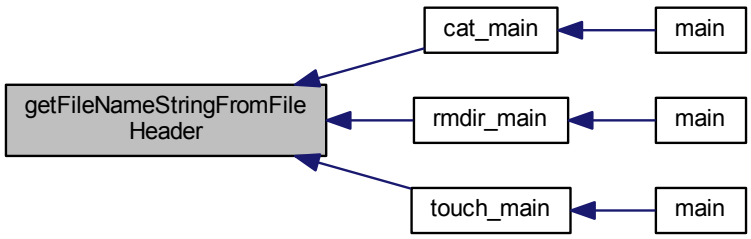
in	*header*	A file header pointer.

Returns

Returns a pointer to a static string buffer containing the file name and extension as a human-readable string.

Definition at line 93 of file fileio.c.

Here is the caller graph for this function:



7.13.4.9 void getNameFromLongNameFileHeader (const FILE_HEADER_LONGNAME * *header*, wchar_t * *name*)

Takes a pointer to a wide character string (at least 13 characters allocated) and populates it with the filename from a longname file header.

Parameters

in	*header*	A pointer to a FILE_HEADER_LONGNAME object.
out	*name*	A pointer to a wchar_t string (32-bits per char on Linux, 16-bits per char on Windows)

Definition at line 132 of file fileio.c.

7.13.4.10 FILE_HEADER_REG* getNextFreeDirectoryEntry (FILE_HEADER * *directory*)

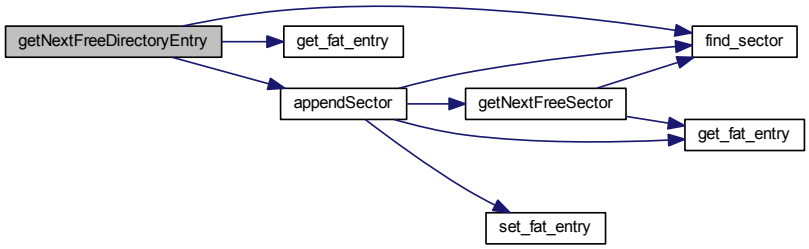
Gets the next free entry of the provided directory? Will expand directory if required.

Parameters

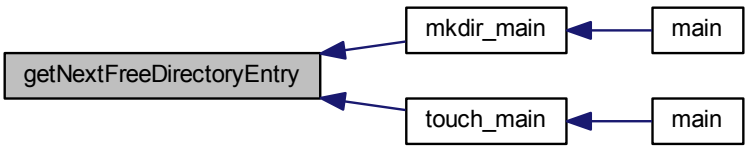
in	*directory*	A pointer to the FILE_HEADER of a directory to get the next free entry of.

Definition at line 671 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.11 bool gotoFile (const char * *name*, const FILE_HEADER * *searchLocation*, FILE_HEADER_REG ** *found*)

Moves within the directory stack to a file header with a specified name (and/or path)

Parameters

in	*name*	The name of the file to search for.
in	*searchLocation*	A pointer to a FILE_HEADER object to start searching from. This may be NULL to signify a search of the root directory.
out	*found*	A pointer to the file header, if found. This is NULL if root or if not found.

Return values

true	A file header with the information given was found. (If found is NULL and the return value is true, the file is root.)

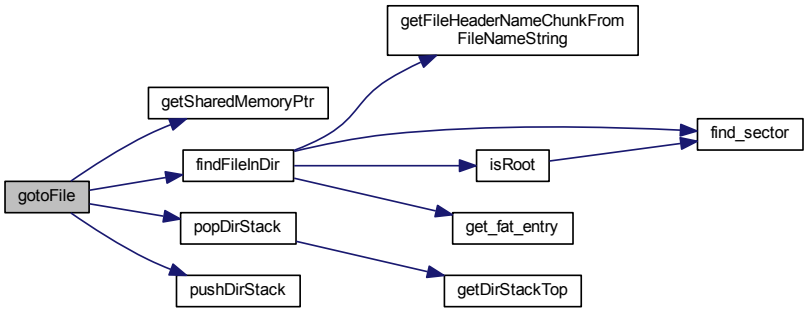
false	The target file header could not be found.

Remarks

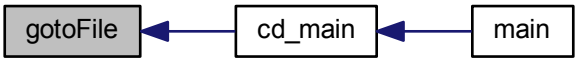
Calls findFile().

Definition at line 412 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.12 bool isDirectoryEmpty (FILE_HEADER * *directory*)

Checks if a directory is empty aside from the . and .. entries along with the long file headers.

Parameters

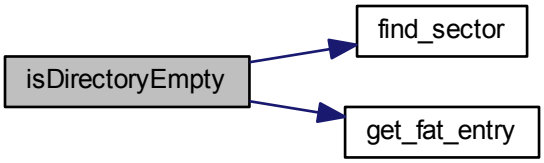
in	*directory*	A pointer to the FILE_HEADER of a directory to check.

Return values

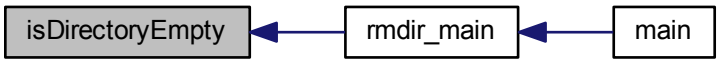
true	The given directory is empty.
false	The given directory contains entries.

Definition at line 724 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.13 bool isRoot (void * *file*)

Determines whether a given file header is a pointer to root.

Parameters

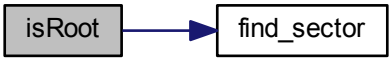
in	*file*	A pointer to a FILE_HEADER.

Returns

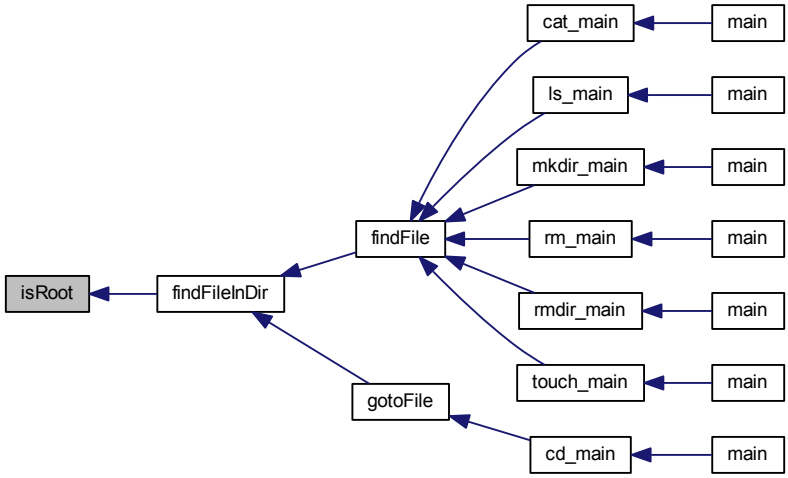
Returns 1 for true and 0 for false.

Definition at line 774 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.13.4.14 void printFileHeader (const FILE_HEADER * *header*)

Prints out the contents of a file header to a human-readable form in the console.

Parameters

in	*header*	A pointer to a FILE_HEADER union. (This could be either a FILE_HEADER_REG or a FILE_HEADER_LONGNAME.)

Definition at line 158 of file fileio.c.

Here is the call graph for this function:

7.13.4.15 void readFile (const FILE_HEADER * *header*, void ** *buffer*)

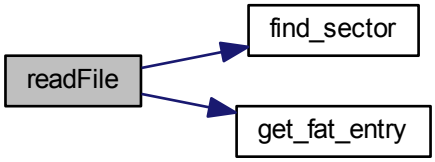
Reads the contents of a file into a function-allocated buffer given a pointer to its file header and a pointer to store the buffer at.

Parameters

in	*header*	A pointer to a FILE_HEADER_REG object.
out	*buffer*	A pointer to a pointer at which a buffer containing the bytes of the file are allocated by the function.

Definition at line 216 of file fileio.c.

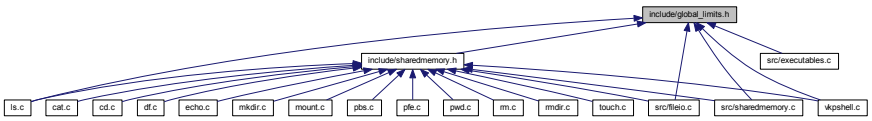
Here is the call graph for this function:



Here is the caller graph for this function:

7.14 include/global_limits.h File Reference

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



Macros

- #define _GLOBAL_LIMITS_H
- #define MAX_PATH_SIZE 512
- #define MAX_SHM_PATH_SIZE 256
- #define MAX_FILES_IN_ROOT_DIR 224

- #define MAX_LISTABLE_FILES 256
- #define MAX_DIR_STACK_ENTRIES 64

7.14.1 Macro Definition Documentation

7.14.1.1 #define _GLOBAL_LIMITS_H

Definition at line 3 of file global_limits.h.

7.14.1.2 #define MAX_DIR_STACK_ENTRIES 64

Definition at line 12 of file global_limits.h.

7.14.1.3 #define MAX_FILES_IN_ROOT_DIR 224

Definition at line 8 of file global_limits.h.

7.14.1.4 #define MAX_LISTABLE_FILES 256

Definition at line 10 of file global_limits.h.

7.14.1.5 #define MAX_PATH_SIZE 512

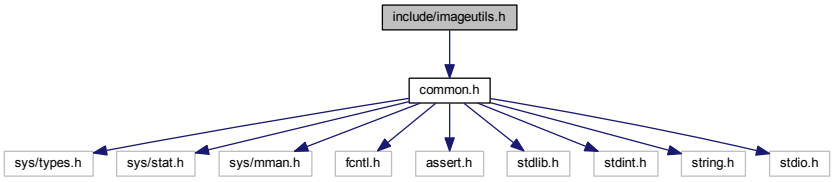
Definition at line 5 of file global_limits.h.

7.14.1.6 #define MAX_SHM_PATH_SIZE 256

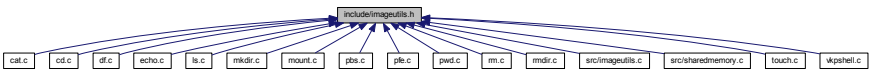
Definition at line 6 of file global_limits.h.

7.15 include/imageutils.h File Reference

`#include "common.h"`
Include dependency graph for imageutils.h:



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



Functions

- bool openFileSystem (const char *path)

  Memory maps the file system to FILE_SYSTEM.

- void closeFileSystem ()

  Closes memory map.

Variables

- uint8_t * FILE_SYSTEM

  Memory map array for file.

7.15.1 Function Documentation

7.15.1.1 void closeFileSystem ( )

Closes memory map.

Definition at line 84 of file imageutils.c.

7.15.1.2 bool openFileSystem (const char * *path*)

Memory maps the file system to FILE_SYSTEM.

Parameters

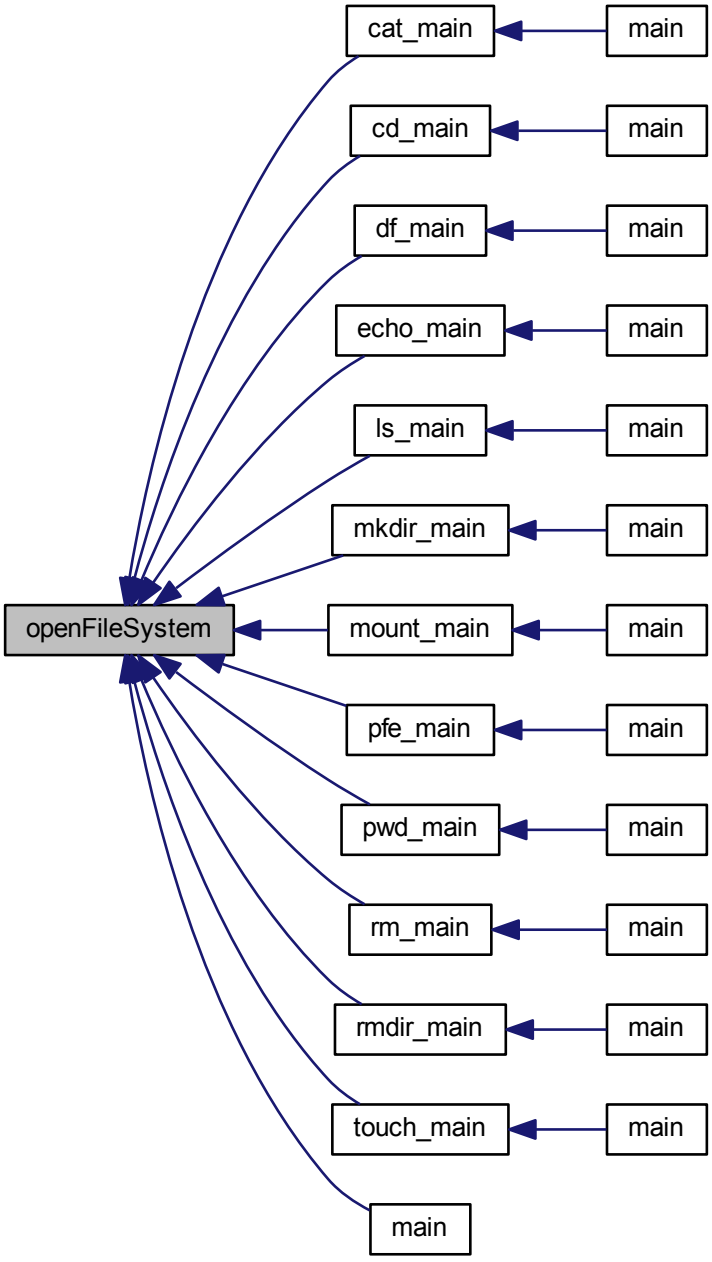
in	*path*	A const character string representing a path to an image file to mount.

Return values

true	The mount is successful.
false	The mount is unsuccessful.

Definition at line 17 of file imageutils.c.

Here is the caller graph for this function:



7.15.2 Variable Documentation

7.15.2.1 uint8_t* FILE_SYSTEM

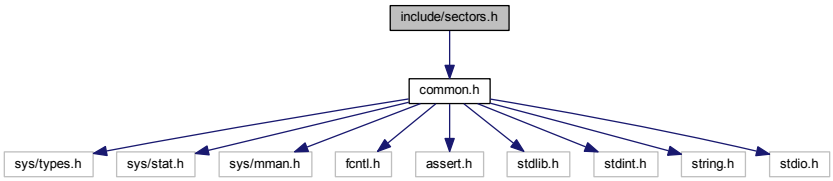
Memory map array for file.

Definition at line 14 of file imageutils.c.

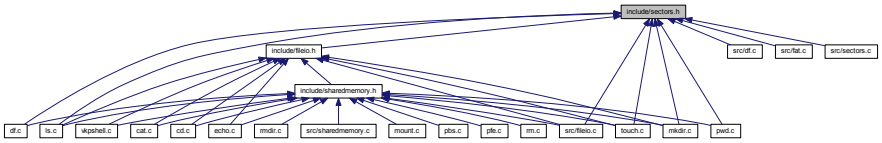
7.16 include/sectors.h File Reference

```
#include "common.h"
```

Include dependency graph for sectors.h:



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



Macros

- #define BOOT_OFFSET 0
- #define FAT1_OFFSET 1
- #define FAT2_OFFSET 10
- #define ROOT_OFFSET 19
- #define DATA_OFFSET 31

Functions

- int read_sector (int sector_number, unsigned char *buffer)

  Reads the contents of a sector given a sector number and places the contents in a user-allocated buffer.
- int write_sector (int sector_number, unsigned char *buffer)

  Writes the contents of a sector provided by the user with a sector number to which to write.
- void * find_sector (uint32_t sector_number)

  Returns a pointer to a sector in the filesystem memory map given a sector number.

7.16.1 Macro Definition Documentation

7.16.1.1 #define BOOT_OFFSET 0

Definition at line 23 of file sectors.h.

7.16.1.2 #define DATA_OFFSET 31

Definition at line 27 of file sectors.h.

7.16.1.3 #define FAT1_OFFSET 1

Definition at line 24 of file sectors.h.

7.16.1.4 #define FAT2_OFFSET 10

Definition at line 25 of file sectors.h.

7.16.1.5 #define ROOT_OFFSET 19

Definition at line 26 of file sectors.h.

7.16.2 Function Documentation

7.16.2.1 void* find_sector (uint32_t *sector_number*)

Returns a pointer to a sector in the filesystem memory map given a sector number.

Parameters

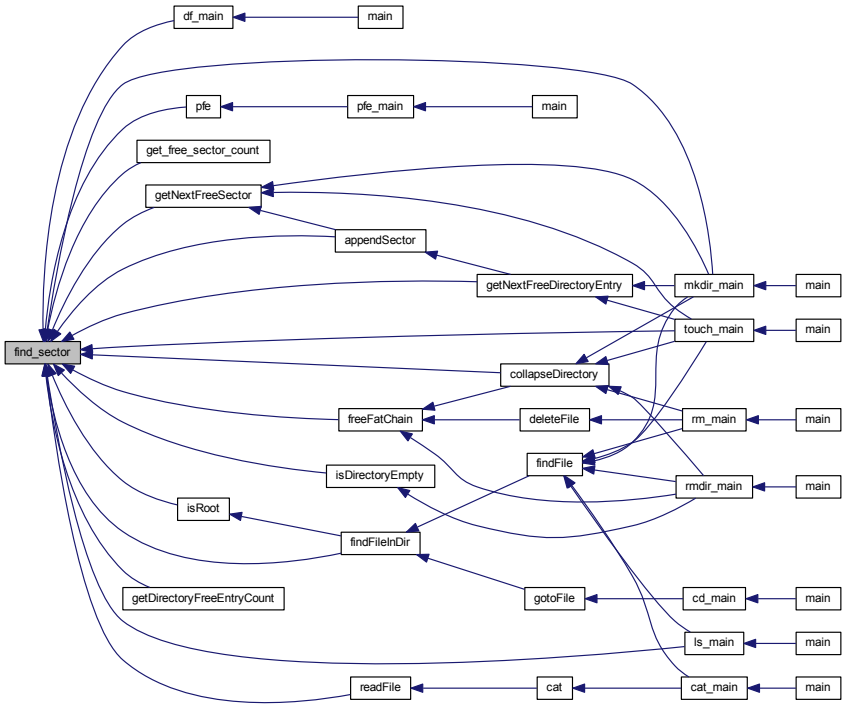
in	*sector_number*	A uint32_t describing the sector number to be found.

Returns

A void pointer pointing to the sector with the given number.

Definition at line 57 of file sectors.c.

Here is the caller graph for this function:



7.16.2.2 int read_sector (int *sector_number,* unsigned char * *buffer*)

Reads the contents of a sector given a sector number and places the contents in a user-allocated buffer.

Bug DEPRECATED - use find_sector() instead!

Parameters

in	*sector_number*	An int describing the number of the sector to read.
in	*buffer*	An unsigned char pointer to a buffer to read the file sector into (allocated by user).

Definition at line 7 of file sectors.c.

7.16.2.3 int write_sector (int *sector_number,* unsigned char * *buffer*)

Writes the contents of a sector provided by the user with a sector number to which to write.

Bug DEPRECATED - use find_sector() instead!

Parameters

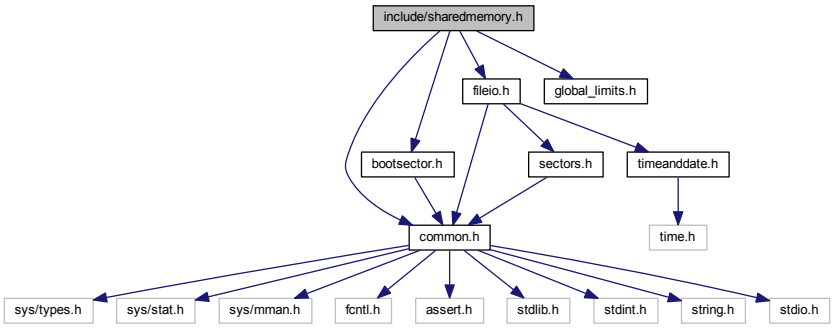
in	*sector_number*	An int describing the number of the sector to write to.
in	*buffer*	A buffer provided by the user containing the sector bytes.

Definition at line 33 of file sectors.c.

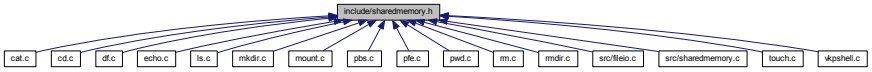
7.17 include/sharedmemory.h File Reference

```
#include "common.h"
#include "bootsector.h"
#include "fileio.h"
#include "global_limits.h"
```

Include dependency graph for sharedmemory.h:



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



Data Structures

- struct SHELL_SHARED_MEMORY

  A structure to facilitate shared data between shell and applications.

Macros

- #define SHMKEY 2467
- #define SHMNAME "/vkpmemspace"

Typedefs

- typedef struct SHELL_SHARED_MEMORY SHELL_SHARED_MEMORY

  A structure to facilitate shared data between shell and applications.

Functions

- void createShared ()

  Allocates an internal shared memory file buffer.
- SHELL_SHARED_MEMORY * mapShared ()

  Gets a memory-mapped pointer to shared memory allocated by a call to createShared().
- SHELL_SHARED_MEMORY * getSharedMemoryPtr ()

  Gets the pointer to shared memory last set up by a call to mapShared().
- void unmapShared ()

  Called to unmap the pointer to shared memory.
- FILE_HEADER * getDirStackTop (SHELL_SHARED_MEMORY *sharedMemory)

  Gets the address of FILE_HEADER at the top of the stored directory stack.
- FILE_HEADER * getDirStackIndex (SHELL_SHARED_MEMORY *sharedMemory, int index)

  Gets the address of a FILE_HEADER at the specified index of the stored directory stack.
- FILE_HEADER * popDirStack (SHELL_SHARED_MEMORY *sharedMemory)

  Pops the directory stack and returns a pointer to the topmost FILE_HEADER popped.
- void pushDirStack (SHELL_SHARED_MEMORY *sharedMemory, FILE_HEADER *header)

  Pushes a pointer to a FILE_HEADER the directory stack.
- void printWorkingDirectory (SHELL_SHARED_MEMORY *sharedMemory)

  Prints the working directory.
- void printWorkingDirectoryPath (SHELL_SHARED_MEMORY *sharedMemory)

  Prints the working directory path.
- const char * getWorkingPathFromStack (SHELL_SHARED_MEMORY *sharedMemory)

  Returns a working path as a string, given a pointer to a SHELL_SHARED_MEMORY object containing a directory stack.

7.17.1 Macro Definition Documentation

7.17.1.1 #define SHMKEY 2467

Definition at line 36 of file sharedmemory.h.

7.17.1.2 #define SHMNAME "/vkpmemspace"

Definition at line 38 of file sharedmemory.h.

7.17.2 Typedef Documentation

7.17.2.1 typedef struct SHELL_SHARED_MEMORY SHELL_SHARED_MEMORY

A structure to facilitate shared data between shell and applications.

7.17.3 Function Documentation

7.17.3.1 void createShared ( )

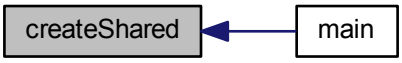
Allocates an internal shared memory file buffer.

Returns

N/A (call mapShared() after this to get a memory-mapped pointer to what this allocates)

Definition at line 32 of file sharedmemory.c.

Here is the caller graph for this function:



7.17.3.2 FILE_HEADER* getDirStackIndex (SHELL_SHARED_MEMORY * *sharedMemory*, int *index*)

Gets the address of a FILE_HEADER at the specified index of the stored directory stack.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.
in	*index*	The index to read from.

Returns

Returns a pointer to the FILE_HEADER.

Definition at line 81 of file sharedmemory.c.

Here is the caller graph for this function:

7.17.3.3 FILE_HEADER* getDirStackTop (SHELL_SHARED_MEMORY * *sharedMemory*)

Gets the address of FILE_HEADER at the top of the stored directory stack.

Parameters

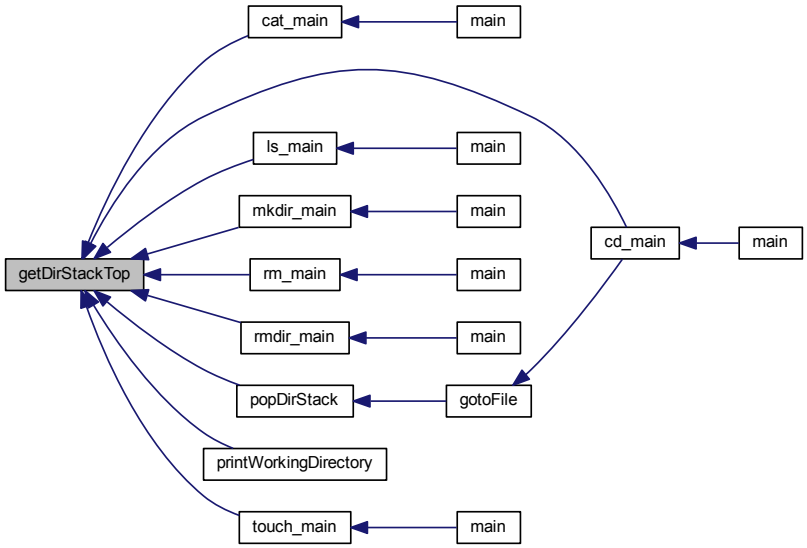
in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Returns

Returns a pointer to the FILE_HEADER.

Definition at line 71 of file sharedmemory.c.

Here is the caller graph for this function:



7.17.3.4 SHELL_SHARED_MEMORY* getSharedMemoryPtr ( )

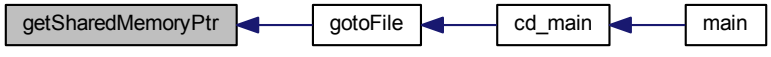
Gets the pointer to shared memory last set up by a call to mapShared().

Returns

Returns a pointer to a SHELL_SHARED_MEMORY struct.

Definition at line 59 of file sharedmemory.c.

Here is the caller graph for this function:



7.17.3.5 const char* getWorkingPathFromStack (SHELL_SHARED_MEMORY * *sharedMemory*)

Returns a working path as a string, given a pointer to a SHELL_SHARED_MEMORY object containing a directory stack.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Returns

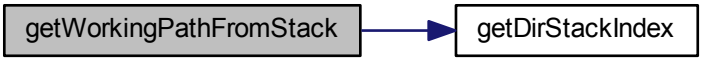
Returns a const char string containing the path.

Warning

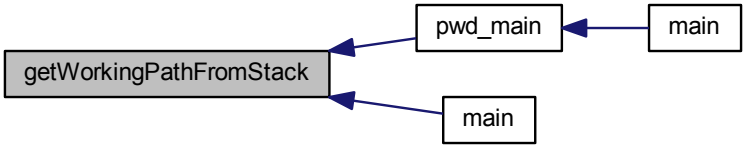
The pointer returned is to a statically allocated buffer within the function and should NOT be freed via free()! A copy should be made (e.g. via strdup()) if any manipulation is to be done.

Definition at line 155 of file sharedmemory.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.17.3.6 SHELL_SHARED_MEMORY* mapShared ( )

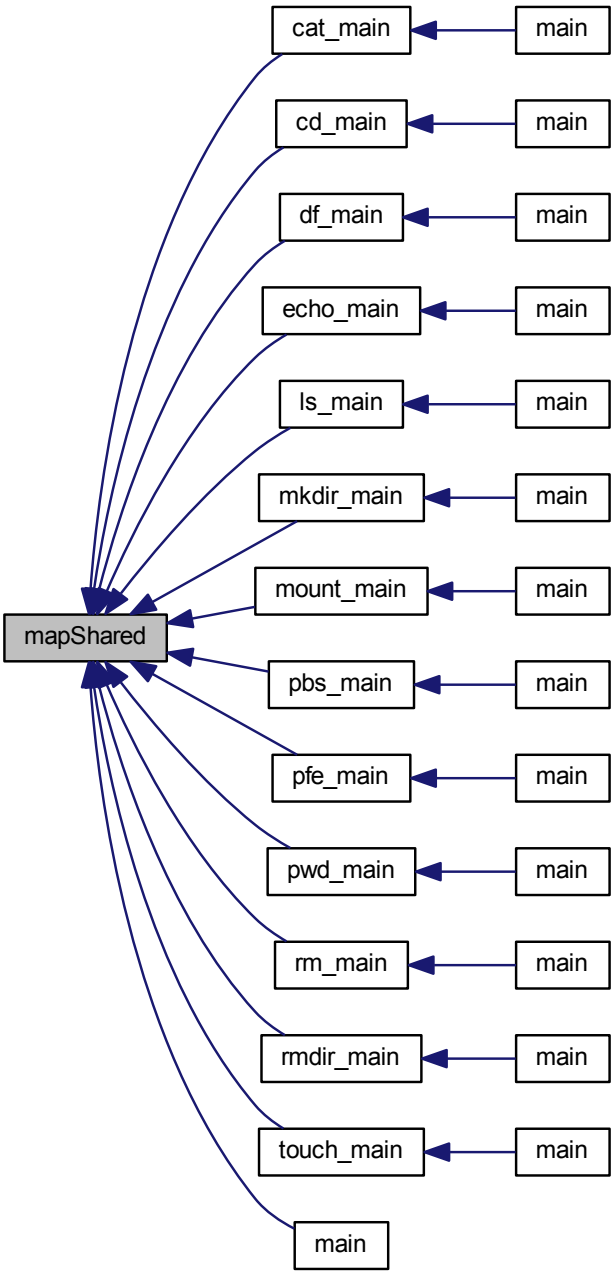
Gets a memory-mapped pointer to shared memory allocated by a call to createShared().

Returns

Returns a pointer to a SHELL_SHARED_MEMORY struct.

Definition at line 42 of file sharedmemory.c.

Here is the caller graph for this function:



7.17.3.7 FILE_HEADER* popDirStack (SHELL_SHARED_MEMORY * *sharedMemory*)

Pops the directory stack and returns a pointer to the topmost FILE_HEADER popped.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to operate on.

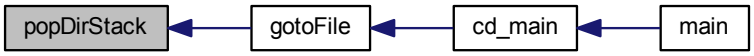
Returns

Returns a pointer to the FILE_HEADER popped.

Definition at line 91 of file sharedmemory.c.

Here is the call graph for this function:

Here is the caller graph for this function:



7.17.3.8 void printWorkingDirectory (SHELL_SHARED_MEMORY * *sharedMemory*)

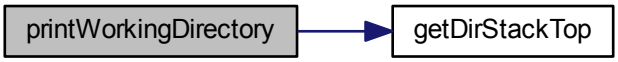
Prints the working directory.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Definition at line 134 of file sharedmemory.c.

Here is the call graph for this function:



7.17.3.9 void printWorkingDirectoryPath (SHELL_SHARED_MEMORY * *sharedMemory*)

Prints the working directory path.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Definition at line 150 of file sharedmemory.c.

7.17.3.10 void pushDirStack (SHELL_SHARED_MEMORY * *sharedMemory*, FILE_HEADER * *header*)

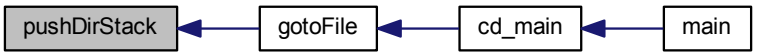
Pushes a pointer to a FILE_HEADER the directory stack.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to operate on.
in	*header*	The FILE_HEADER pointer to be pushed.

Definition at line 119 of file sharedmemory.c.

Here is the caller graph for this function:

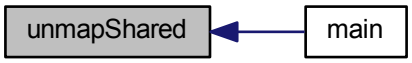


7.17.3.11 void unmapShared ( )

Called to unmap the pointer to shared memory.

Definition at line 64 of file sharedmemory.c.

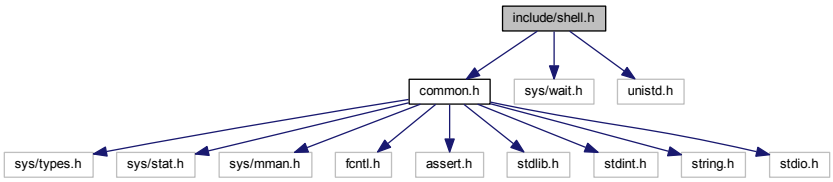
Here is the caller graph for this function:



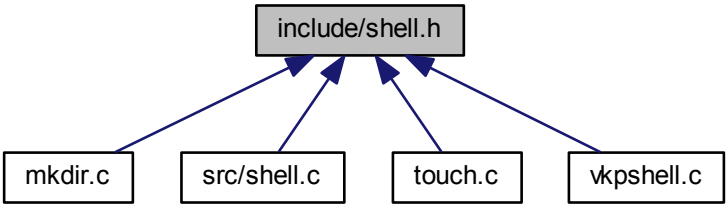
7.18 include/shell.h File Reference

```
#include "common.h"
#include <sys/wait.h>
#include <unistd.h>
```

Include dependency graph for shell.h:



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



Functions

- void execProcess (const char *path, char *arguments[])

  Fork off the shell and execute a process, giving it a list of optional arguments.
- int parseCommand (char *command, char ***commandArr)

  Parses a command from its arguments.
- void parsePathFileExtension (char *fullPath, char **pathOut, char **fileNameOut, char **extensionOut)

7.18.1 Function Documentation

7.18.1.1 void execProcess (const char * *path*, char * *arguments[ ]*)

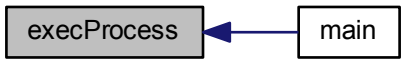
Fork off the shell and execute a process, giving it a list of optional arguments.

Parameters

in	*path*	A C-string holding the path to the executable to be run by the forked off shell.
in	*arguments*	An array of C-string arguments to be passed to the executable to be run by the forked off shell.

Definition at line 3 of file shell.c.

Here is the caller graph for this function:



7.18.1.2 int parseCommand (char * *command*, char *** *commandArr*)

Parses a command from its arguments.

Parameters

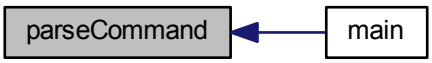
in	*command*	The command to parse.
out	*commandArr*	The command array output.

Returns

Returns the number of arguments delimited by spaces.

Definition at line 25 of file shell.c.

Here is the caller graph for this function:



7.18.1.3 void parsePathFileExtension (char * *fullPath*, char ** *pathOut*, char ** *fileNameOut*, char ** *extensionOut*)

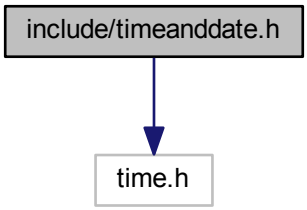
Warning

Made obsolete by getFileHeaderNameChunkFromFileNameString() in fileio.h.

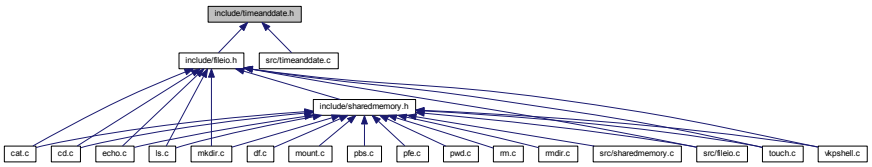
Definition at line 58 of file shell.c.

7.19 include/timeanddate.h File Reference

`#include <time.h>`
Include dependency graph for timeanddate.h:



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



Data Structures

- struct FILE_TIME

  A struct to hold a file time.
- struct FILE_DATE

  A struct to hold a file date.

Typedefs

- typedef struct FILE_TIME FILE_TIME

  A struct to hold a file time.
- typedef struct FILE_DATE FILE_DATE

  A struct to hold a file date.

Functions

- void createFileDateTime (time_t in, FILE_DATE *date, FILE_TIME *time)

  Populates a FILE_TIME and a FILE_DATE from a time_t provided. Both the FILE_TIME and FILE_DATE pointers can be NULL.
- time_t timeDateToCTime (const FILE_DATE *date, const FILE_TIME *time, struct tm *out)

  Populates a tm struct given a FILE_DATE and a FILE_TIME. It is possible to simply put NULL in for either field if unavailable.
- void getHumanReadableDateTimeString (const FILE_DATE *date, const FILE_TIME *time, char *out)

  Populates a pre-allocated string buffer with the date and/or time provided.

7.19.1 Typedef Documentation

7.19.1.1 typedef struct FILE_DATE FILE_DATE

A struct to hold a file date.

7.19.1.2 typedef struct FILE_TIME FILE_TIME

A struct to hold a file time.

7.19.2 Function Documentation

7.19.2.1 void createFileDateTime (time_t *in*, FILE_DATE * *date*, FILE_TIME * *time*)

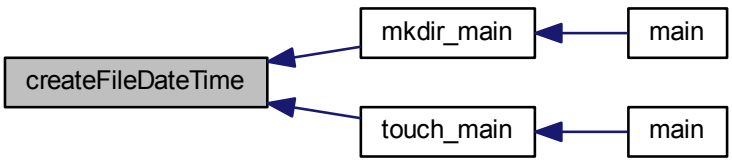
Populates a FILE_TIME and a FILE_DATE from a time_t provided. Both the FILE_TIME and FILE_DATE pointers can be NULL.

Parameters

in	*in*	A time_t object.
out	*time*	A FILE_TIME object to fill. (Can be NULL to ignore.)
out	*date*	A FILE_DATE object to fill. (Can be NULL to ignore.)

Definition at line 20 of file timeanddate.c.

Here is the caller graph for this function:



7.19.2.2 void getHumanReadableDateTimeString (const FILE_DATE * *date*, const FILE_TIME * *time*, char * *out*)

Populates a pre-allocated string buffer with the date and/or time provided.

Parameters

in	*date*	An optional FILE_DATE object. (Use NULL to negate.)
in	*time*	An optional FILE_TIME object. (Use NULL to negate.)
out	*out*	A pre-allocated string buffer large enough to contain the date and/or time string produced.

Definition at line 86 of file timeanddate.c.

Here is the caller graph for this function:

7.19.2.3 time_t timeDateToCTime (const FILE_DATE * *date*, const FILE_TIME * *time*, struct tm * *out*)

Populates a tm struct given a FILE_DATE and a FILE_TIME. It is possible to simply put NULL in for either field if unavailable.

Parameters

in	*date*	A FILE_DATE object.
in	*time*	A FILE_TIME object.
out	*out*	A pointer to an allocated tm struct. Can be NULL.

Returns

Returns a time_t of the time inputted.

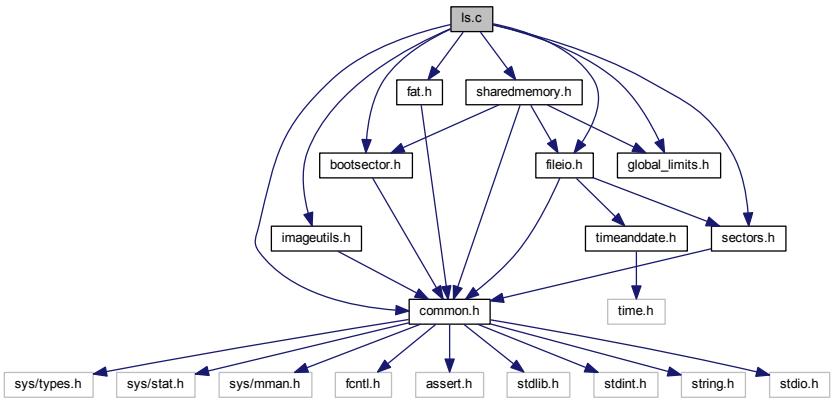
Definition at line 46 of file timeanddate.c.

Here is the caller graph for this function:

7.20 ls.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
#include "fileio.h"
#include "sectors.h"
#include "global_limits.h"
```

Include dependency graph for ls.c:



Functions

- int compareFileHeaderByName (const FILE_HEADER_REG **file1, const FILE_HEADER_REG **file2)

  A function to compare two pointers to FILE_HEADER_REG pointers by their file name and extension contents (alphabetically).
- void listFileEntry (FILE_HEADER_REG *header)

  A function to print a file's information for ls.
- int ls_main (int argc, char *argv[])

  Main function for ls.
- int main (int argc, char *argv[])

Variables

- FILE_HEADER_REG * fileList [MAX_LISTABLE_FILES]

  An array to hold files found.

7.20.1 Function Documentation

7.20.1.1 int compareFileHeaderByName (const FILE_HEADER_REG ** *file1*, const FILE_HEADER_REG ** *file2*)

A function to compare two pointers to FILE_HEADER_REG pointers by their file name and extension contents (alphabetically).

Parameters

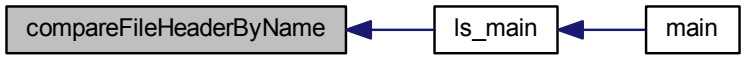
in	*file1*	A pointer to a pointer to a FILE_HEADER_REG.
in	*file2*	A pointer to a pointer to a FILE_HEADER_REG.

Returns

Returns an int value that is 0 if equal, <0 if less, and >0 if greater.

Definition at line 19 of file ls.c.

Here is the caller graph for this function:



7.20.1.2 void listFileEntry (FILE_HEADER_REG * *header*)

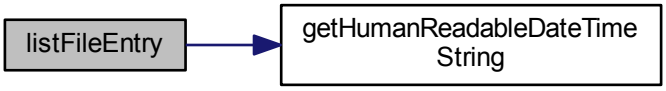
A function to print a file's information for ls.

Parameters

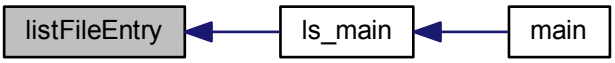
in	*header*	A FILE_HEADER_REG pointer to a file, the contents of which to be displayed.

Definition at line 35 of file ls.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.20.1.3 int ls_main (int *argc*, char * *argv[]*)

Main function for ls.

Test If ls is called with no arguments, ls shall list the files and folders of the current working directory, providing their individual FLCs, sizes, dates, and names.

If ls is called with an argument that is a valid path to a directory, ls shall list the files and folders of the provided directory, displaying their individual FLCs, sizes, dates, and names.

If ls is called with an argument that is a valid path to a file, ls shall print the listing for that individual file, displaying its FLC, size, date, and name.

If ls is called with an argument that is an invalid path to a directory, ls shall exit printing, "Could not find path".

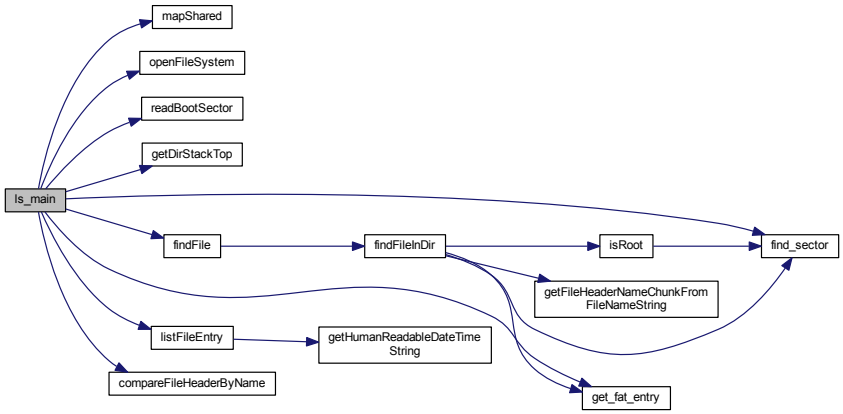
If the arguments provided to ls number more than one, ls shall exit printing, "Too many arguments!".

Any and all file/folder listings provided by ls shall be sorted in alphabetical order by file name and extension if applicable.

ls shall not print any file whose attributes are 0 or are 0x0f under any circumstances.

Definition at line 92 of file ls.c.

Here is the call graph for this function:

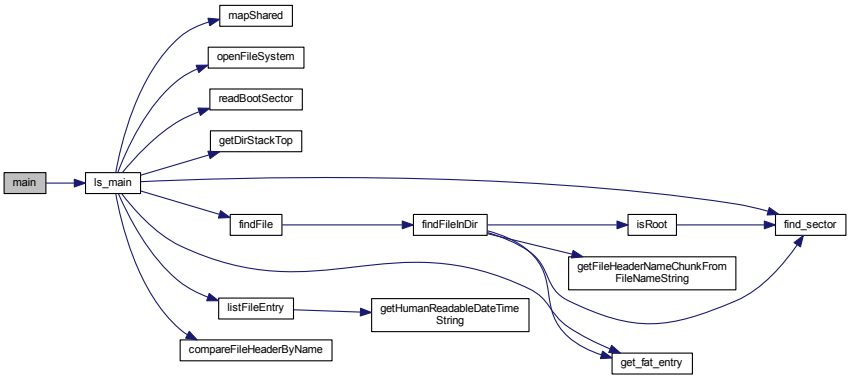


Here is the caller graph for this function:

7.20.1.4 int main (int *argc,* char * *argv[ ]*)

Definition at line 221 of file ls.c.

Here is the call graph for this function:



7.20.2 Variable Documentation

7.20.2.1 FILE_HEADER_REG* fileList[MAX_LISTABLE_FILES]

An array to hold files found.

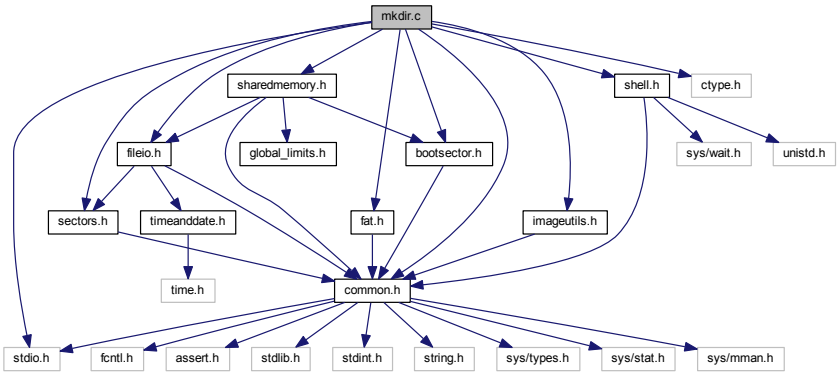
Definition at line 13 of file ls.c.

7.21 mkdir.c File Reference

```
#include "common.h"
```

```
#include "imageutils.h"
#include "bootsector.h"
#include "sectors.h"
#include "fat.h"
#include "sharedmemory.h"
#include "shell.h"
#include "fileio.h"
#include <stdio.h>
#include <ctype.h>
```

Include dependency graph for mkdir.c:



Functions

- int mkdir_main (int argc, char ∗argv[ ])

  Main function for mkdir.

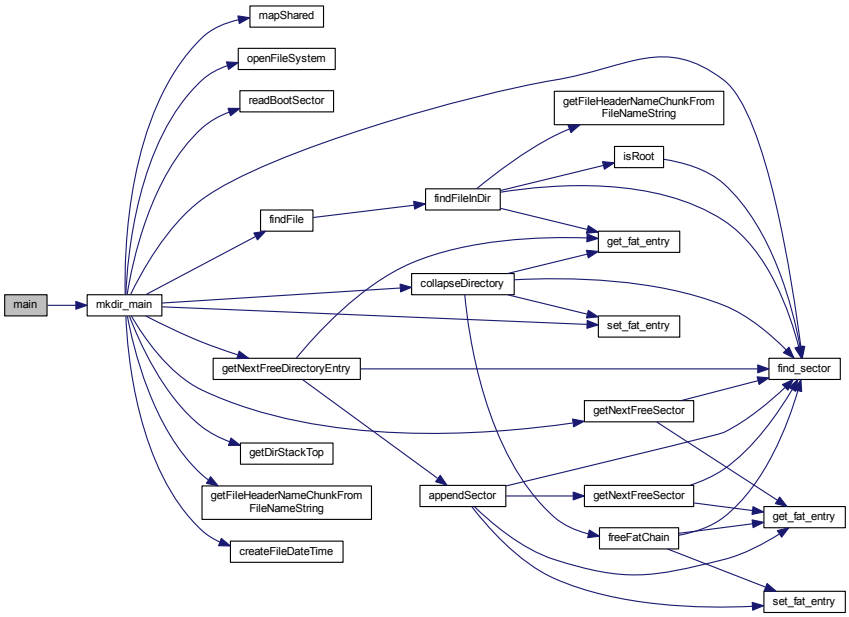
- int main (int argc, char ∗argv[ ])

7.21.1 Function Documentation

7.21.1.1 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 192 of file mkdir.c.

Here is the call graph for this function:



7.21.1.2 int mkdir_main (int *argc*, char * *argv[ ]*)

Main function for mkdir.

Test If provided with a single argument containing a non-existent filename, mkdir shall create a folder with the given name within the current working directory.

If provided with a single argument containing a path and culminating in a non-existent filename, mkdir shall create a folder with the given filename within the provided directory.

If provided with anything other than one argument, mkdir shall exit printing, "Invalid argument count; mkdir takes the path of the directory to create.".

If provided with a single argument containing a valid path to an existing directory, mkdir shall print "File [file_name] already exists.".

If provided with ".", "..", mkdir shall exit printing, "[entry] is not allowed.".

If during the process of trying to create a new directory, mkdir cannot allocate a directory sector, mkdir shall exit printing, "Failed to allocate directory sector.".

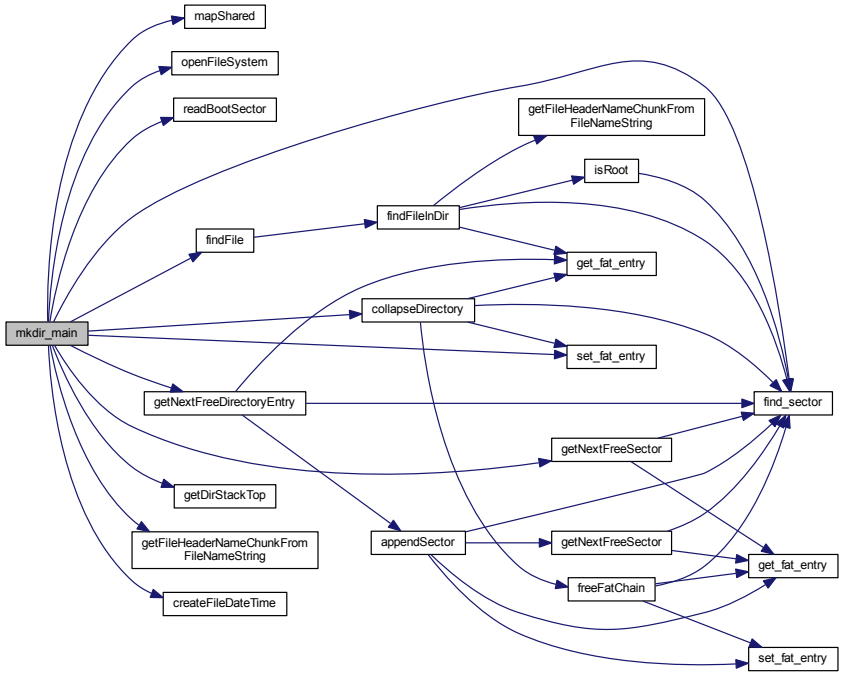
Test If there is not enough room in a directory to add a new directory, a successful mkdir call shall expand the directory before attempting to create the new directory.

Test If during the process of trying to create a new directory, mkdir cannot allocate a directory header, mkdir shall exit printing, "Failed to allocate directory header.".

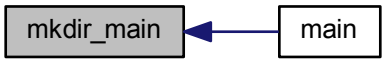
Test If successful in creating a directory, mkdir shall add a timestamp accurate to the closest two seconds to the newly created directory.

Definition at line 25 of file mkdir.c.

Here is the call graph for this function:



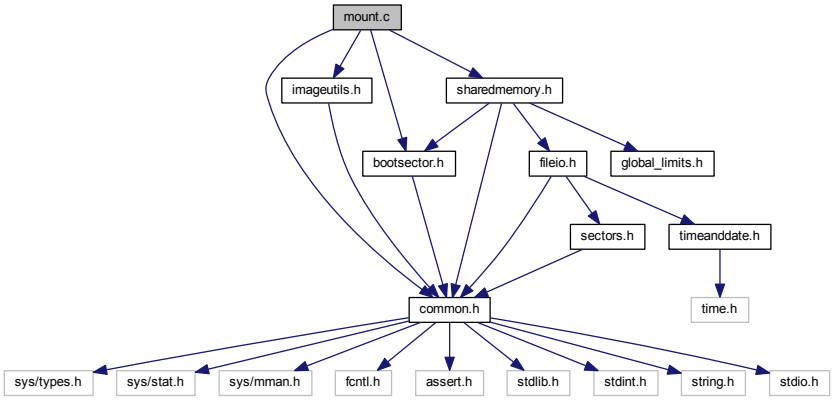
Here is the caller graph for this function:



7.22 mount.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "sharedmemory.h"
```

Include dependency graph for mount.c:



Functions

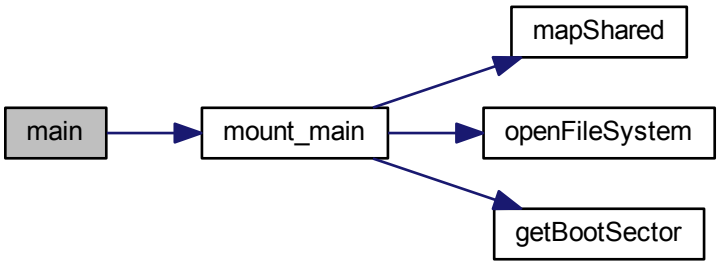
- int mount_main (int argc, char ∗argv[ ])
  Main function for mount command.
- int main (int argc, char ∗argv[ ])

7.22.1 Function Documentation

7.22.1.1 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 55 of file mount.c.

Here is the call graph for this function:



7.22.1.2 int mount_main (int *argc,* char ∗ *argv[ ]*)

Main function for mount command.

Test If mount is provided with a single argument describing a valid OS path to a FAT12-formatted bootable floppy image, mount shall load the image and make it available for use as a file system.

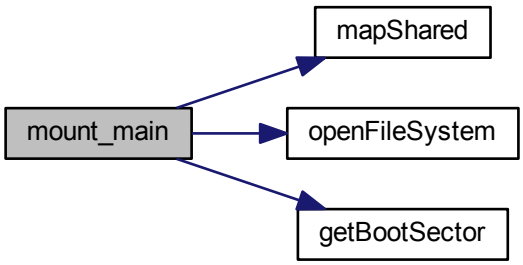
If mount is provided with a single argument describing an invalid OS path, mount shall exit printing, "Could not mount image [image_path]".

If mount is provided with a number of arguments other than one, mount shall exit printing, "Invalid argument count. mount takes the path of the floppy to mount.".

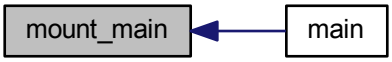
if mount has successfully mounted an image in the past during execution, mount shall exit printing, "File system at [mounted_image_path] is already mounted.".

Definition at line 11 of file mount.c.

Here is the call graph for this function:



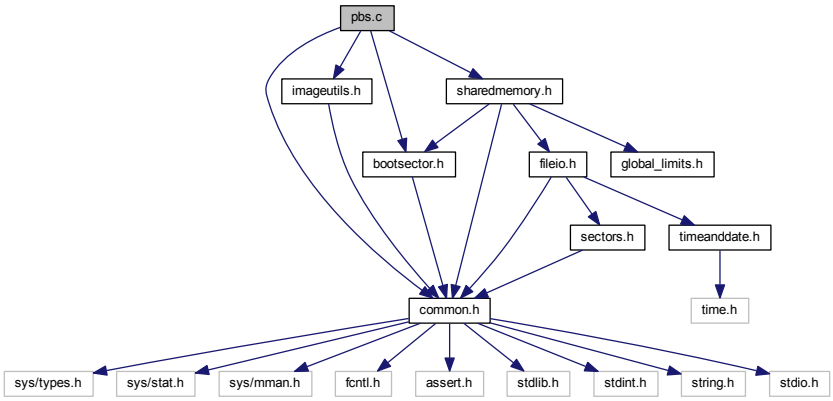
Here is the caller graph for this function:



7.23 pbs.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "sharedmemory.h"
```

Include dependency graph for pbs.c:



Functions

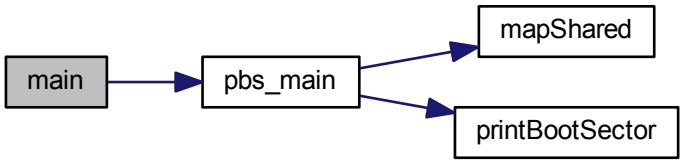
- int pbs_main (int argc, char ∗argv[ ])
  Main function for pbs command.
- int main (int argc, char ∗argv[ ])

7.23.1 Function Documentation

7.23.1.1 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 35 of file pbs.c.

Here is the call graph for this function:



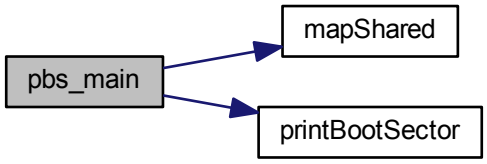
7.23.1.2 int pbs_main (int *argc,* char ∗ *argv[ ]*)

Main function for pbs command.

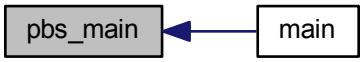
Test If pbs is run with any number of arguments, pbs shall print a readout containing information about the boot sector of the currently mounted disk image.

Definition at line 8 of file pbs.c.

Here is the call graph for this function:



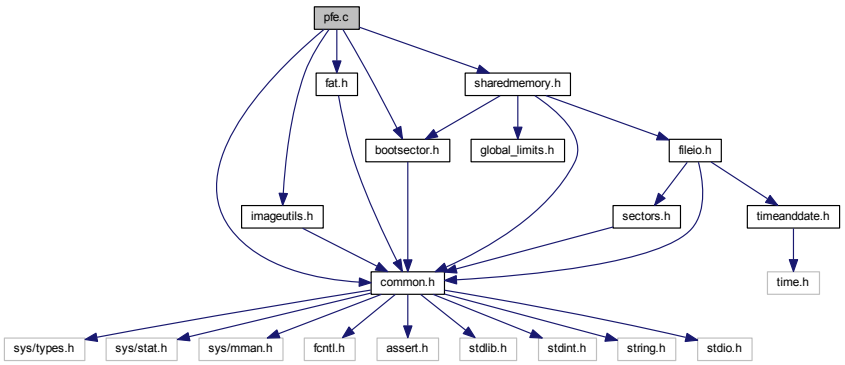
Here is the caller graph for this function:



7.24 pfe.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
```

Include dependency graph for pfe.c:



Functions

- int pfe_main (int argc, char *argv[])

  Main function for pfe command.

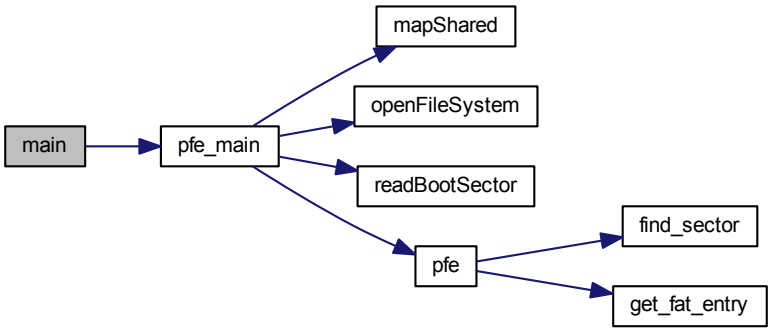
- int main (int argc, char *argv[])

7.24.1 Function Documentation

7.24.1.1 int main (int *argc*, char * *argv[]*)

Definition at line 33 of file pfe.c.

Here is the call graph for this function:



7.24.1.2 int pfe_main (int *argc*, char * *argv[]*)

Main function for pfe command.

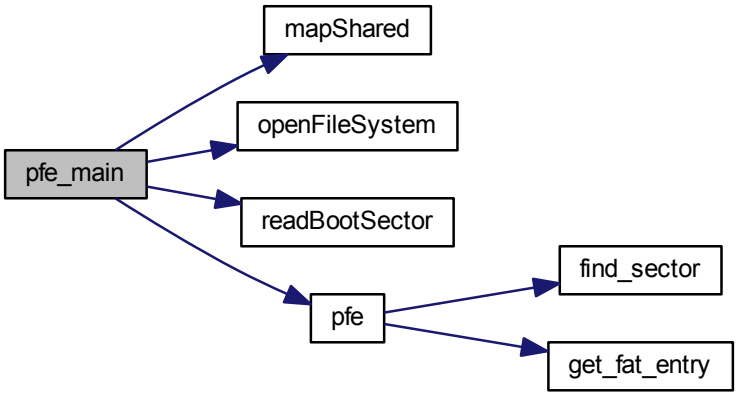
Test If pfe is provided with exactly two arguments indicating the start and end entry indices within the FAT table for which to print a range of FAT entries.

If pfe is provided with any number of arguments other than two, pfe shall exit printing, "Invalid argument count; pfe takes the start and end indices of the FAT table indicating a range of FAT entries to print out.".

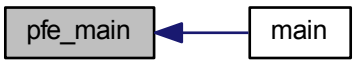
If any of pfe's two arguments are not a valid number, that argument shall be interpreted as 0.

Definition at line 11 of file pfe.c.

Here is the call graph for this function:



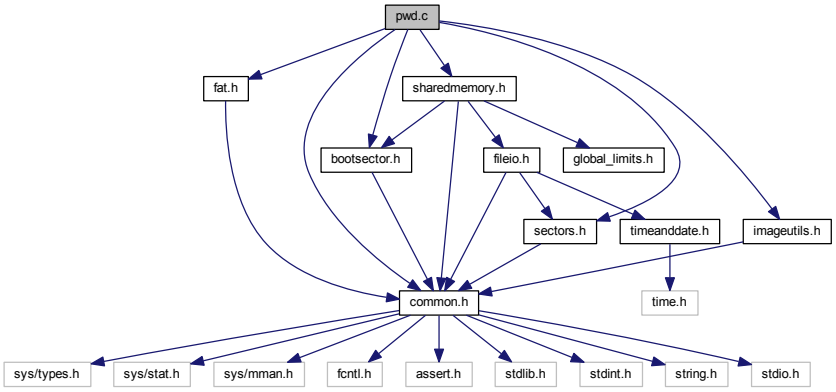
Here is the caller graph for this function:



7.25 pwd.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
#include "sectors.h"
```

Include dependency graph for pwd.c:



Functions

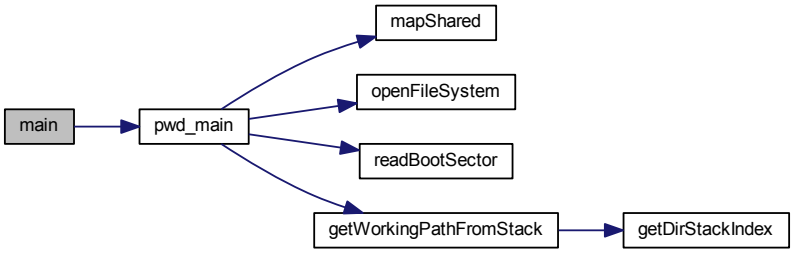
- int pwd_main (int argc, char ∗argv[ ])
  Main function for pwd command.
- int main (int argc, char ∗argv[ ])

7.25.1 Function Documentation

7.25.1.1 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 24 of file pwd.c.

Here is the call graph for this function:



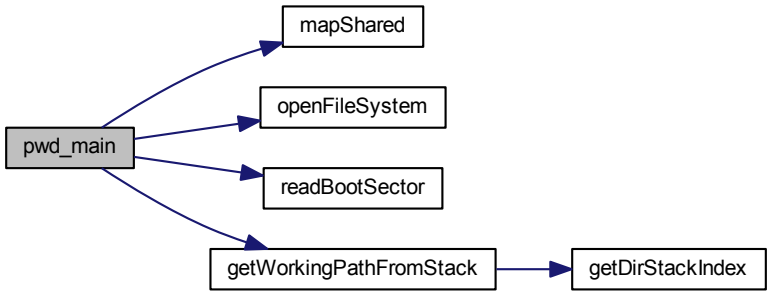
7.25.1.2 int pwd_main (int *argc,* char ∗ *argv[ ]*)

Main function for pwd command.

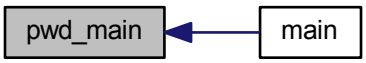
Test If the pwd command is invoked with any number of arguments, the current working path of the shell shall be displayed.

Definition at line 11 of file pwd.c.

Here is the call graph for this function:



Here is the caller graph for this function:

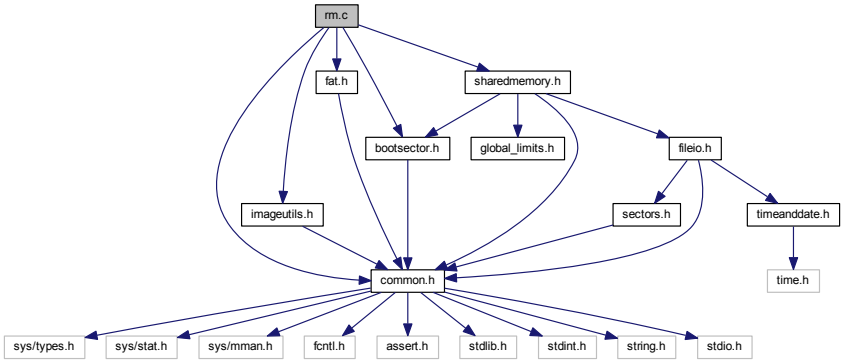


7.26 README.md File Reference

7.27 rm.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
```

Include dependency graph for rm.c:



Functions

- int rm_main (int argc, char ∗argv[ ])

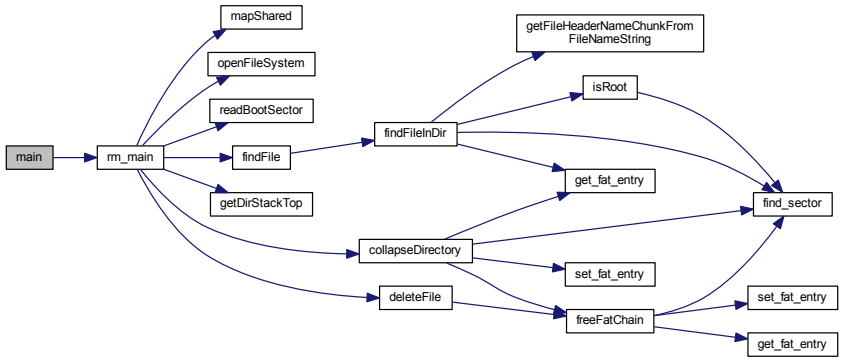
  Main function for rm command.
- int main (int argc, char ∗argv[ ])

7.27.1 Function Documentation

7.27.1.1 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 56 of file rm.c.

Here is the call graph for this function:



7.27.1.2 int rm_main (int *argc,* char ∗ *argv[ ]*)

Main function for rm command.

Test If rm is given a single argument containing a valid path to a file, rm shall delete that file from the image.

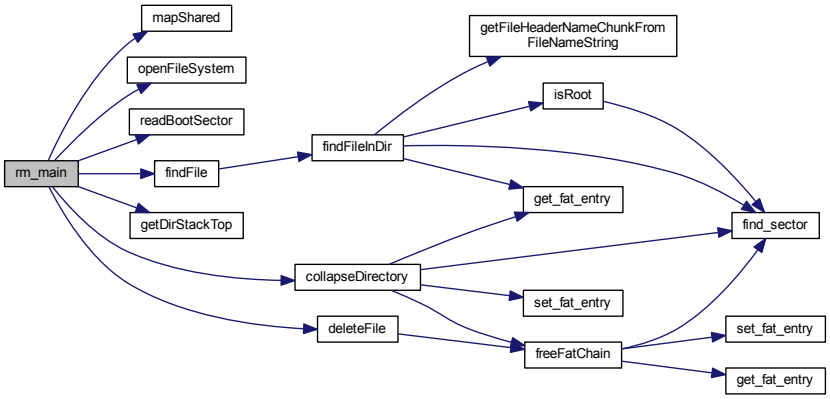
If rm is given any number of arguments other than one, rm shall exit printing, "Invalid argument count; rm takes the path of the file to remove.".

If rm is used successfully, rm shall attempt to collapse the directory it is deleting from.

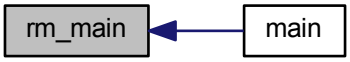
If rm is given a single argument containing a valid path to a file, however, that file is a subdirectory or long file entry, then rm shall exit printing "Could not rm file [path].".

Definition at line 13 of file rm.c.

Here is the call graph for this function:



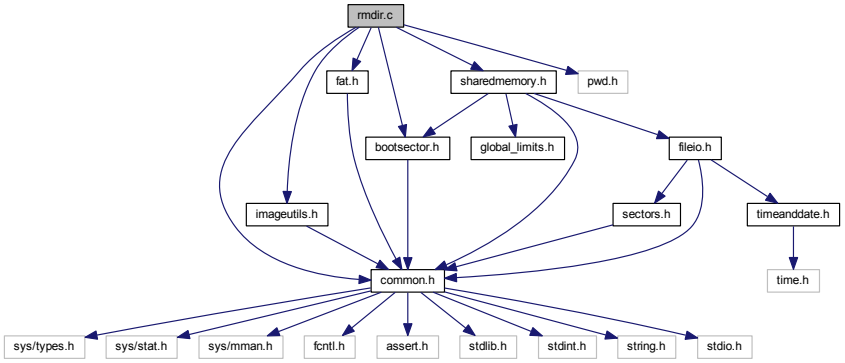
Here is the caller graph for this function:



7.28 rmdir.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "fat.h"
#include "sharedmemory.h"
#include "pwd.h"
```

Include dependency graph for rmdir.c:



Functions

- int rmdir_main (int argc, char ∗argv[ ])

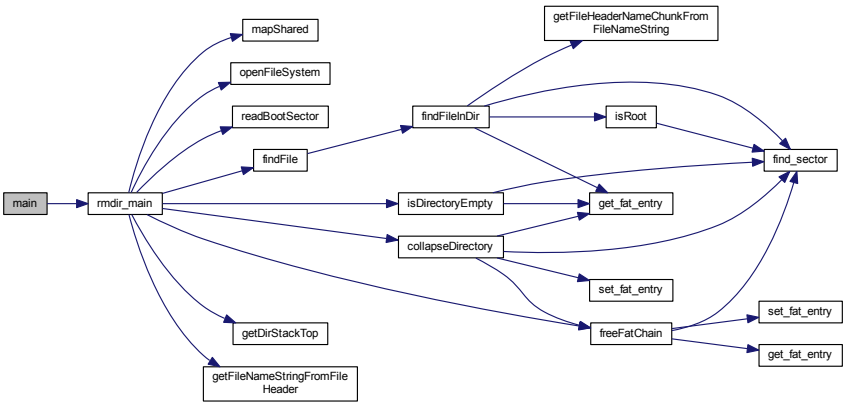
  Main function for rmdir command.
- int main (int argc, char ∗argv[ ])

7.28.1 Function Documentation

7.28.1.1 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 104 of file rmdir.c.

Here is the call graph for this function:



7.28.1.2 int rmdir_main (int *argc,* char ∗ *argv[ ]*)

Main function for rmdir command.

Test If rmdir is provided with a single argument that is a valid path to a directory, rmdir shall remove that folder from the mounted image.

If rmdir is provided with a single argument that is a valid path to a directory containing any file and/or directory, rmdir shall exit printing the message, "Directory still has files.".

If rmdir is provided with a single argument that is a valid path to something other than a subdirectory, or a long file header, rmdir shall exit printing the message, "Specified file [file_name] is not a directory.".

If rmdir is provided with a single argument that is an invalid path rmdir shall exit printing the message, "Directory [path] could not be found!".

If rmdir is provided with a number of arguments other than one, rmdir shall exit printing, "Invalid argument count; rmdir takes the path of the directory to remove.".

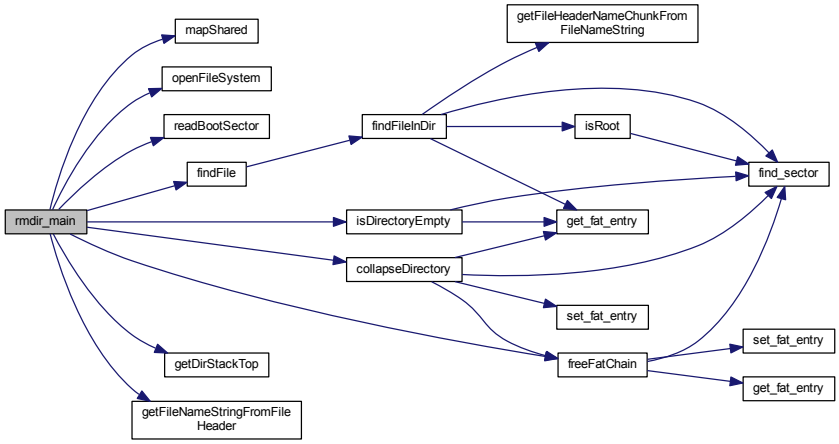
If rmdir is directed to delete the current working directory, the working directory has no files within it, and the user name of the current user can be obtained, rmdir shall exit printing "Nice try [user_name], but deleting the directory you are currently in is not allowed.".

If rmdir is directed to delete the current working directory and the working directory has no files within it, rmdir shall exit printing, "Deleting the directory you are currently in is not allowed.".

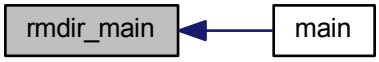
If rmdir successfully deletes a directory, rmdir shall attempt to collapse the parent directory deleted from.

Definition at line 20 of file rmdir.c.

Here is the call graph for this function:



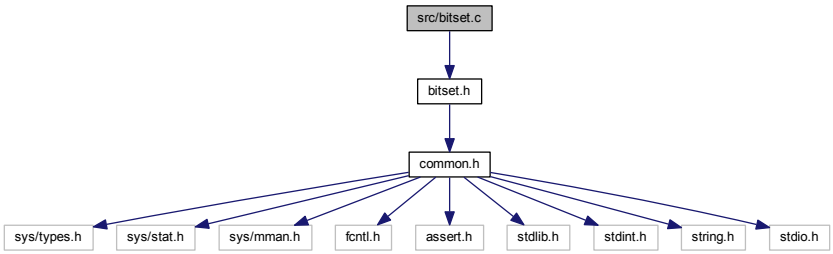
Here is the caller graph for this function:



7.29 src/bitset.c File Reference

```
#include "bitset.h"
```

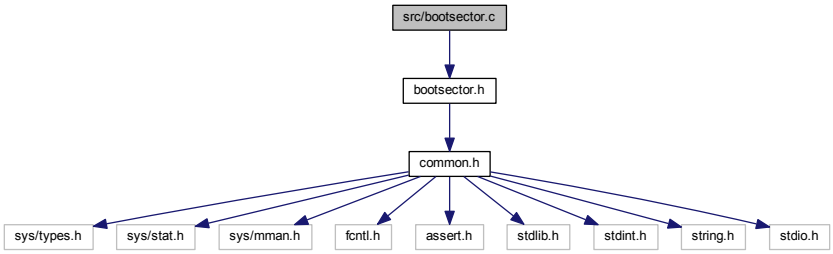
Include dependency graph for bitset.c:



7.30 src/bootsector.c File Reference

```
#include "bootsector.h"
```

Include dependency graph for bootsector.c:



Functions

- void readBootSector ()

  Reads the boot sector from sector 0 on the file system.
- BOOT_SECTOR * getBootSector (uint8_t *fileSystem)
- void printBootSector (BOOT_SECTOR *bootSector)

  Prints the contents of the boot sector to stdout.

Variables

- uint8_t * FILE_SYSTEM

  Memory map array for file.
- BOOT_SECTOR PBS_BOOT_SEC = {0}
- uint16_t BYTES_PER_SECTOR = 0

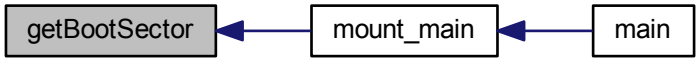
  The number of bytes per sector.

7.30.1 Function Documentation

7.30.1.1 BOOT_SECTOR* getBootSector (uint8_t * *fileSystem*)

Definition at line 14 of file bootsector.c.

Here is the caller graph for this function:



7.30.1.2 void printBootSector (BOOT_SECTOR * *bootSector*)

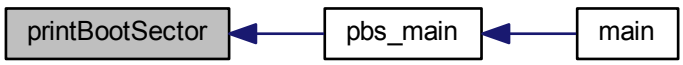
Prints the contents of the boot sector to stdout.

Parameters

in	*bootSector*	A pointer to a BOOT_SECTOR object holding the information to print.

Definition at line 19 of file bootsector.c.

Here is the caller graph for this function:

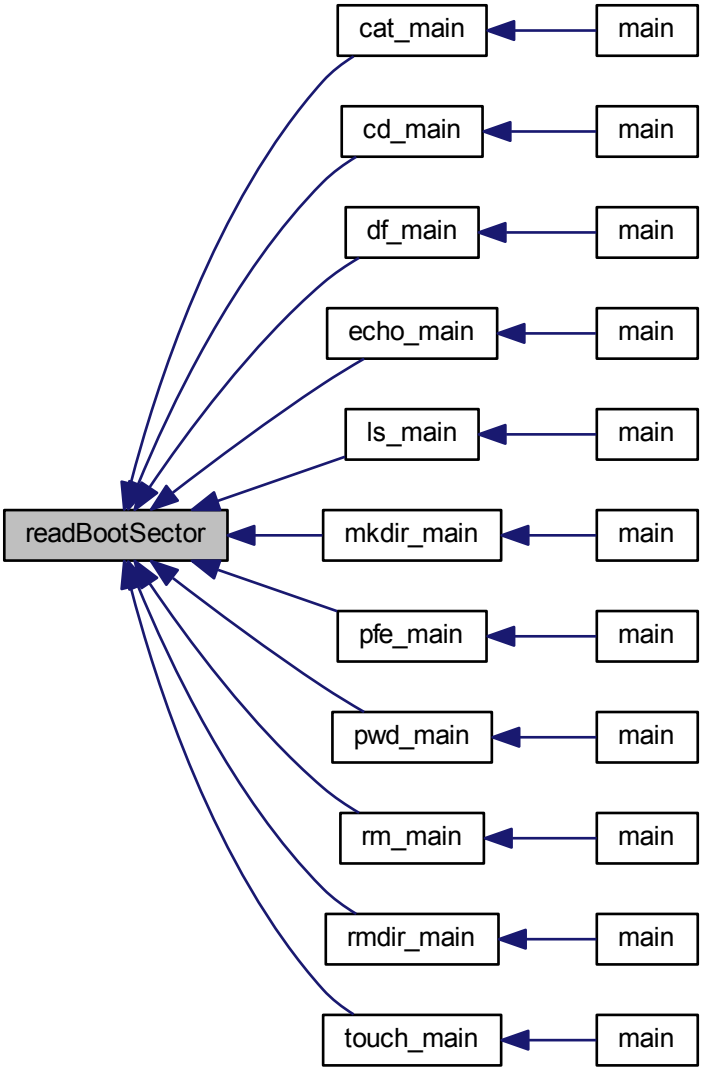


7.30.1.3 void readBootSector ( )

Reads the boot sector from sector 0 on the file system.

Definition at line 8 of file bootsector.c.

Here is the caller graph for this function:



7.30.2 Variable Documentation

7.30.2.1 uint16_t BYTES_PER_SECTOR = 0

The number of bytes per sector.

Definition at line 6 of file bootsector.c.

7.30.2.2 uint8_t* FILE_SYSTEM

Memory map array for file.

Definition at line 14 of file imageutils.c.

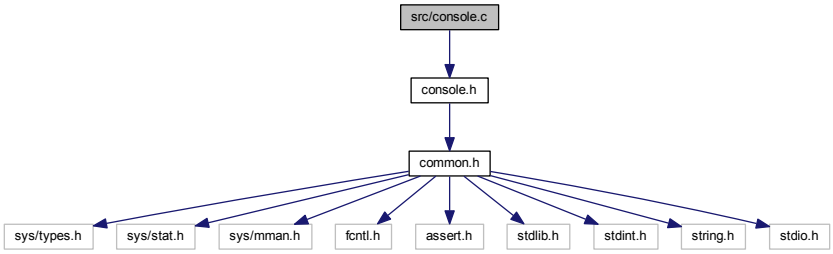
7.30.2.3 BOOT_SECTOR PBS_BOOT_SEC = {0}

Definition at line 5 of file bootsector.c.

7.31 src/console.c File Reference

```
#include "console.h"
```

Include dependency graph for console.c:



Functions

- char * getLine ()

  Gets a line of input from the user.

7.31.1 Function Documentation

7.31.1.1 char* getLine ( )

Gets a line of input from the user.

Returns

Returns a pointer to a C-string.

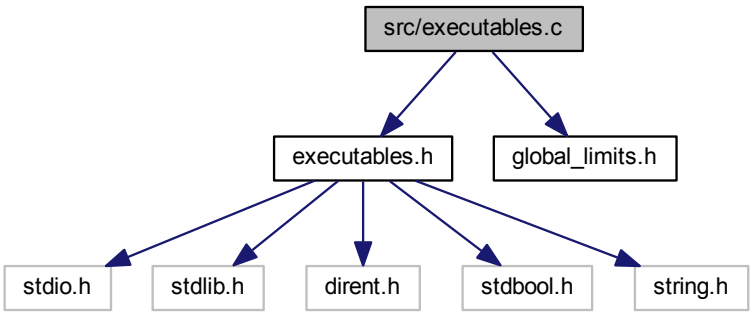
Definition at line 3 of file console.c.

Here is the caller graph for this function:

7.32 src/executables.c File Reference

```
#include "executables.h"
#include "global_limits.h"
```

Include dependency graph for executables.c:



Functions

- bool isELF (FILE *fp)

  Determines if a file is a valid executable ELF file.
- void freeExecutableList ()

  Frees the executables list.
- void addExecutable (char *name)

  Adds an executable to the executables list.
- void printExecutables ()

  Prints a list of all executables.
- void trimExecutables ()

Trims off unused executable entries.

- void addDirToExecutableList (char *indir)

  Adds the executables of a directory to the executable list.

Variables

- const unsigned char ELF_HEADER_BYTES [ELF_HEADER_SIZE] = { 0x7f, 'E', 'L', 'F' }
- char ** EXECUTABLES = NULL

  An array of strings of executables allowed by the shell.

- size_t EXECUTABLES_SIZE = 0

  Stores the number of entry slots allocated in the executable list.

- size_t NUM_EXECUTABLES = 0

  Stores the actual number of entries populated in the executable list.

7.32.1 Function Documentation

7.32.1.1 void addDirToExecutableList (char * *dir*)

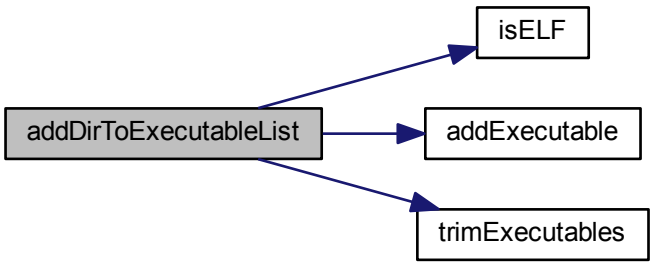
Adds the executables of a directory to the executable list.

Parameters

in	*dir*	A string path to a directory.

Definition at line 91 of file executables.c.

Here is the call graph for this function:



Here is the caller graph for this function:

7.32.1.2 void addExecutable (char * *name*)

Adds an executable to the executables list.

Parameters

in	*name*	A null-terminated character string representing an executable's filename.

Definition at line 44 of file executables.c.

Here is the caller graph for this function:

7.32.1.3 void freeExecutableList ( )

Frees the executables list.

Definition at line 34 of file executables.c.

Here is the caller graph for this function:

7.32.1.4 bool isELF (FILE * *fp*)

Determines if a file is a valid executable ELF file.

Parameters

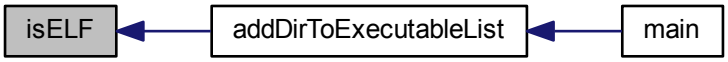
in	*fp*	A FILE pointer to an open file.

Return values

true	The file is a valid ELF.
false	The file is not executable ELF or the file has not been opened.

Definition at line 14 of file executables.c.

Here is the caller graph for this function:

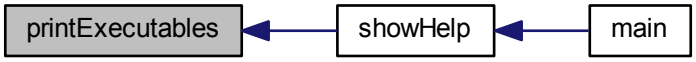


7.32.1.5 void printExecutables ( )

Prints a list of all executables.

Definition at line 69 of file executables.c.

Here is the caller graph for this function:

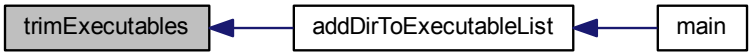


7.32.1.6 void trimExecutables ( )

Trims off unused executable entries.

Definition at line 81 of file executables.c.

Here is the caller graph for this function:



7.32.2 Variable Documentation

7.32.2.1 const unsigned char ELF_HEADER_BYTES[ELF_HEADER_SIZE] = { 0x7f, 'E', 'L', 'F' }

Definition at line 4 of file executables.c.

7.32.2.2 char** EXECUTABLES = NULL

An array of strings of executables allowed by the shell.

Definition at line 6 of file executables.c.

7.32.2.3 size_t EXECUTABLES_SIZE = 0

Stores the number of entry slots allocated in the executable list.

Definition at line 9 of file executables.c.

7.32.2.4 size_t NUM_EXECUTABLES = 0

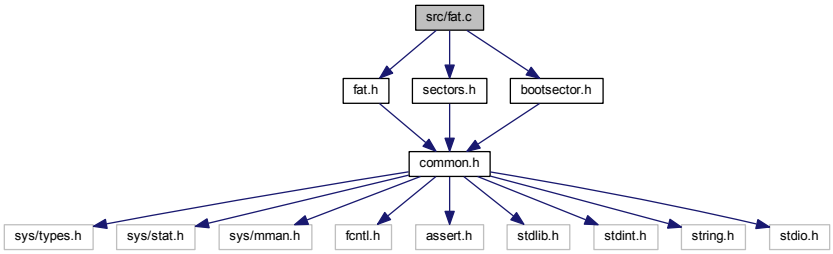
Stores the actual number of entries populated in the executable list.

Definition at line 12 of file executables.c.

7.33 src/fat.c File Reference

```
#include "fat.h"
#include "sectors.h"
#include "bootsector.h"
```

Include dependency graph for fat.c:



Functions

- unsigned int get_fat_entry (int fat_entry_number, unsigned char *fat)
- void set_fat_entry (int fat_entry_number, int value, unsigned char *fat)
- uint16_t get_free_sector_count ()

  Gets the number of free sectors on disk.
- void pfe (int start, int end)

  Prints out a human-readable table of all of the FAT entries in the FAT table.
- unsigned int getNextFreeSector ()

  Returns the number of the next free sector.
- void freeFatChain (int fatStart, bool zeroMemory)

  Frees a FAT chain.
- unsigned int appendSector (int startSector)

  Links a sector onto the specified sector and updates the FAT tables to extend the FAT entry chain.

7.33.1 Function Documentation

7.33.1.1 unsigned int appendSector (int *startSector*)

Links a sector onto the specified sector and updates the FAT tables to extend the FAT entry chain.

Parameters

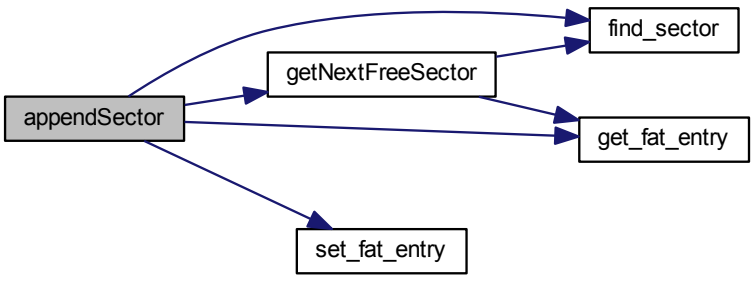
in	*startSector*	The sector number to append to.

Returns

Returns the sector that was allocated and appended to the end.

Definition at line 145 of file fat.c.

Here is the call graph for this function:



Here is the caller graph for this function:

7.33.1.2 void freeFatChain (int *fatStart*, bool *zeroMemory*)

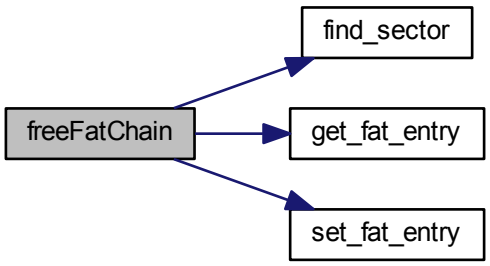
Frees a FAT chain.

Parameters

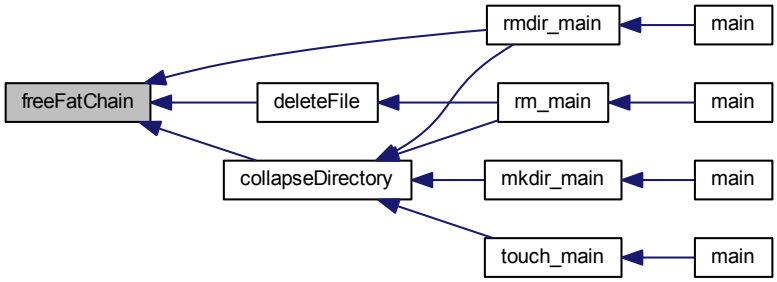
in	*fatStart*	An index of the fat entry to start at.
in	*zeroMemory*	A boolean value indicating whether or not to zero the freed memory.

Definition at line 118 of file fat.c.

Here is the call graph for this function:



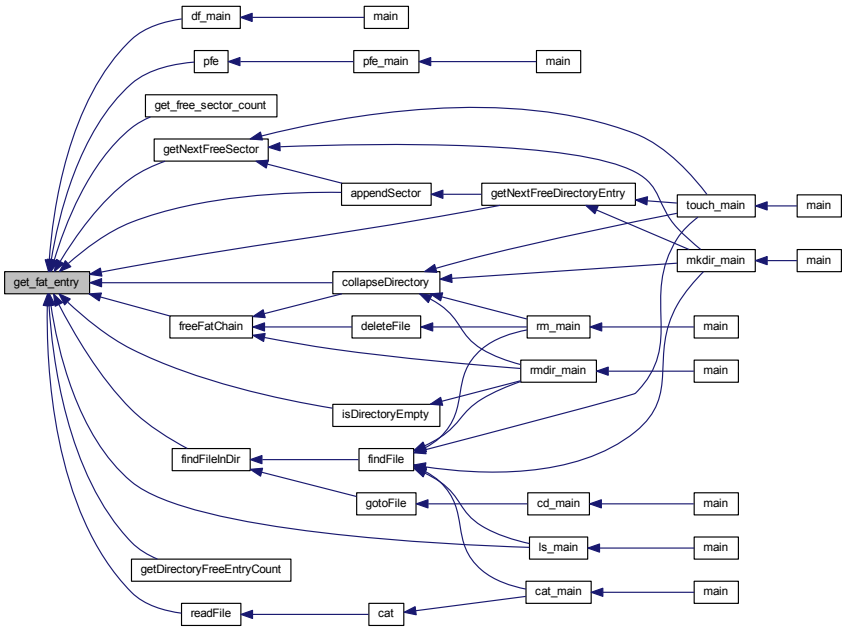
Here is the caller graph for this function:



7.33.1.3 unsigned int get_fat_entry (int *fat_entry_number,* unsigned char * *fat*)

Definition at line 5 of file fat.c.

Here is the caller graph for this function:



7.33.1.4 uint16_t get_free_sector_count ( )

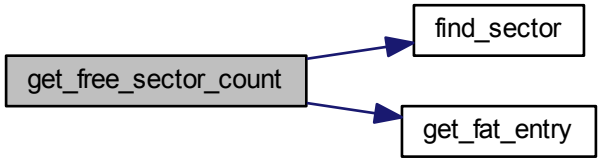
Gets the number of free sectors on disk.

Returns

Returns a uint16_t.

Definition at line 66 of file fat.c.

Here is the call graph for this function:



7.33.1.5 unsigned int getNextFreeSector ( )

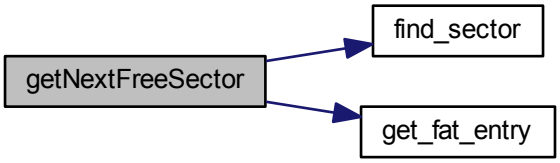
Returns the number of the next free sector.

Returns

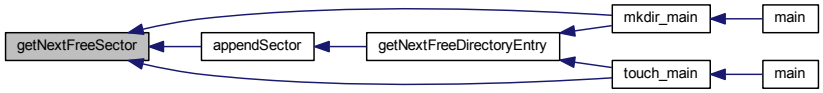
Returns the number of the next free sector as an unsigned int.

Definition at line 102 of file fat.c.

Here is the call graph for this function:



Here is the caller graph for this function:



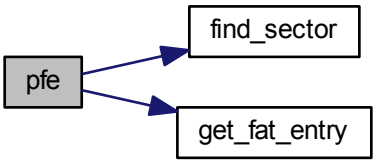
7.33.1.6 void pfe (int *start*, int *end*)

Prints out a human-readable table of all of the FAT entries in the FAT table.

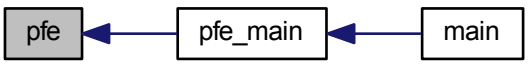
param[in] start The number of the first FAT entry to start reading from (start with 2 since first 2 are unused). param[in] end The number of the last FAT entry to read from (must be at least 2 since first 2 are unused).

Definition at line 84 of file fat.c.

Here is the call graph for this function:



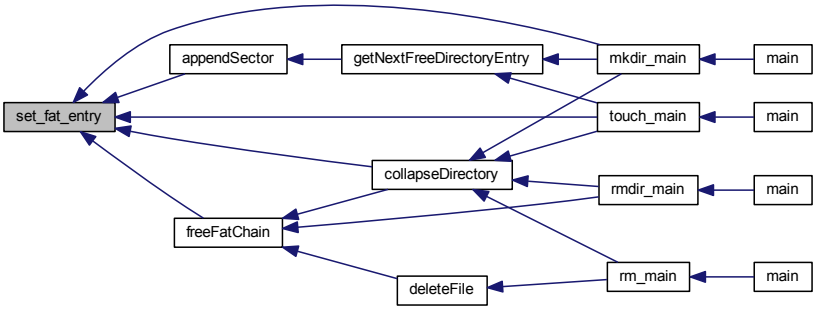
Here is the caller graph for this function:



7.33.1.7 void set_fat_entry (int *fat_entry_number*, int *value*, unsigned char * *fat*)

Definition at line 31 of file fat.c.

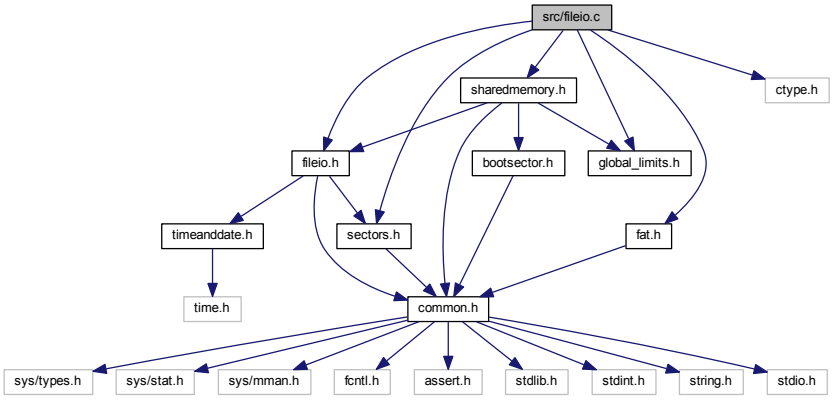
Here is the caller graph for this function:



7.34 src/fileio.c File Reference

```
#include "fileio.h"
#include "fat.h"
#include "sectors.h"
#include "global_limits.h"
#include "sharedmemory.h"
#include <ctype.h>
```

Include dependency graph for fileio.c:



Functions

- char * getFileHeaderNameChunkFromFileNameString (char *filenameString)

  Gives an 11-byte name and extension block for a file header from a filename string.
- char * getFileNameStringFromFileHeader (FILE_HEADER_REG *header)

  Gives a filename as a string from a file header.
- void getNameFromLongNameFileHeader (const FILE_HEADER_LONGNAME *header, wchar_t *name)

Takes a pointer to a wide character string (at least 13 characters allocated) and populates it with the filename from a longname file header.

- void printFileHeader (const FILE_HEADER *header)

  Prints out the contents of a file header to a human-readable form in the console.

- void readFile (const FILE_HEADER *header, void **buffer)

  Reads the contents of a file into a function-allocated buffer given a pointer to its file header and a pointer to store the buffer at.

- bool findFileInDir (const char *name, const FILE_HEADER *searchLocation, FILE_HEADER_REG **found)

  Finds a file header with a specified name.

- bool findFile (const char *name, const FILE_HEADER *searchLocation, FILE_HEADER_REG **found)

  Finds a file header with a specified name (and/or path)

- bool gotoFile (const char *name, const FILE_HEADER *searchLocation, FILE_HEADER_REG **found)

  Moves within the directory stack to a file header with a specified name (and/or path)

- void deleteFile (FILE_HEADER *header)

  Deletes a file given a pointer to a file header.

- int getDirectoryFreeEntryCount (FILE_HEADER *directory)

  Gets the number of free entries in a provided directory.

- void collapseDirectory (FILE_HEADER *directory)

  Collapses all files in a directory toward the front then drops any extra sectors.

- FILE_HEADER_REG * getNextFreeDirectoryEntry (FILE_HEADER *directory)

  Gets the next free entry of the provided directory? Will expand directory if required.

- bool isDirectoryEmpty (FILE_HEADER *directory)

  Checks if a directory is empty aside from the . and .. entries along with the long file headers.

- void cat (const FILE_HEADER_REG *file)

  Given a regular 8.1 file header, prints out the contents of the file to console.

- bool isRoot (void *file)

  Determines whether a given file header is a pointer to root.

Variables

- uint8_t * FILE_SYSTEM

  Memory map array for file.

7.34.1 Function Documentation

7.34.1.1 void cat (const FILE_HEADER_REG * *file*)

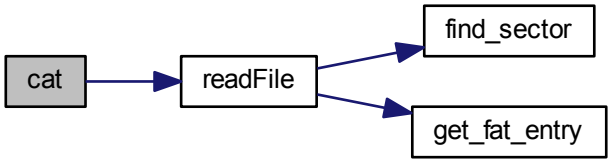
Given a regular 8.1 file header, prints out the contents of the file to console.

Parameters

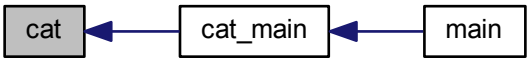
in	*file*	A pointer to a FILE_HEADER_REG.

Definition at line 758 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.2 void collapseDirectory (FILE_HEADER * *directory*)

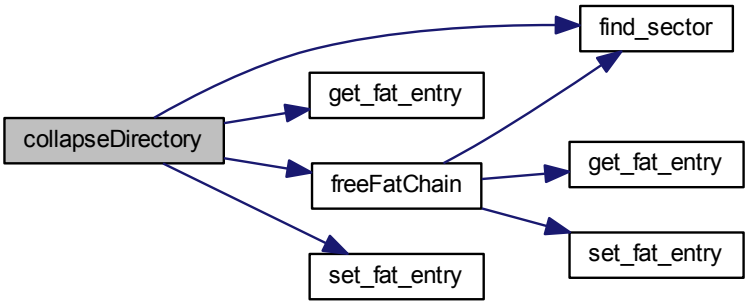
Collapses all files in a directory toward the front then drops any extra sectors.

Parameters

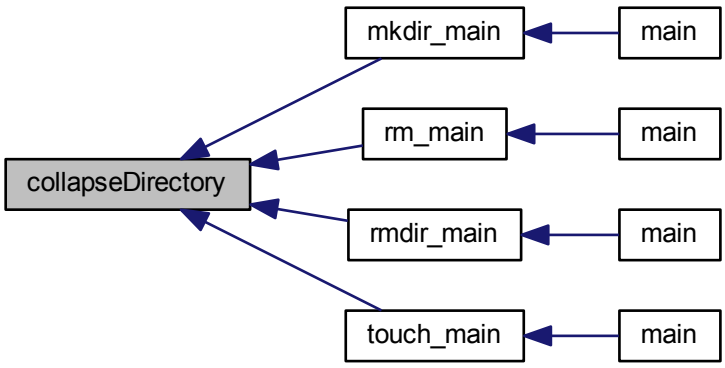
in	*directory*	A pointer to the FILE_HEADER of a directory to collapse.

Definition at line 567 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.3 void deleteFile (FILE_HEADER * *header*)

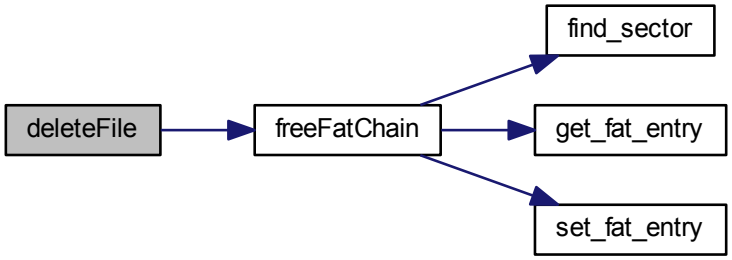
Deletes a file given a pointer to a file header.

Parameters

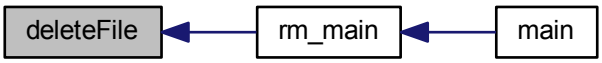
in	*header*	A pointer to the FILE_HEADER of the file to be deleted.

Definition at line 523 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.4 bool findFile (const char * *name*, const FILE_HEADER * *searchLocation*, FILE_HEADER_REG ** *found*)

Finds a file header with a specified name (and/or path)

Parameters

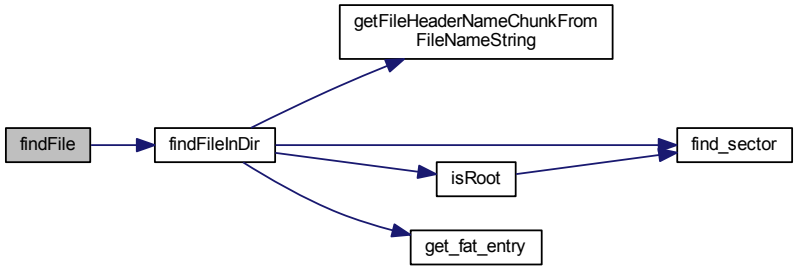
in	*name*	The name of the file to search for.
in	*searchLocation*	A pointer to a FILE_HEADER object to start searching from. This may be NULL to signify a search of the root directory.
out	*found*	A pointer to the file header, if found. This is NULL if root or if not found.

Return values

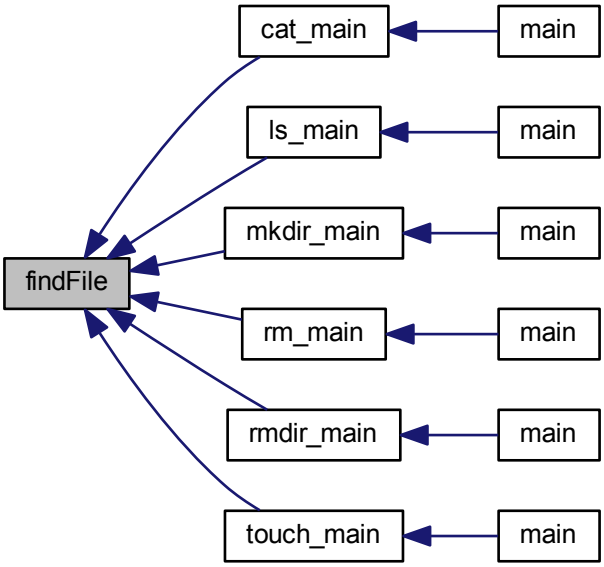
true	A file header with the information given was found. (If found is NULL and the return value is true, the file is root.)
false	The target file header could not be found.

Definition at line 340 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.5 bool findFileInDir (const char * *name*, const FILE_HEADER * *searchLocation*, FILE_HEADER_REG ** *found*)

Finds a file header with a specified name.

Parameters

in	*name*	The name of the file to search for.
in	*searchLocation*	A pointer to a FILE_HEADER object to start searching from. This may be NULL to signify a search of the root directory.
out	*found*	A pointer to the file header, if found. This is NULL if root or if not found.

Return values

true	A file header with the information given was found. (If found is NULL and the return value is true, the file is root.)
false	The target file header could not be found.

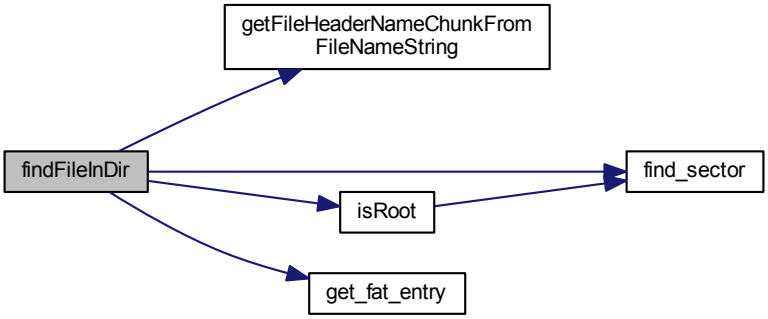
Remarks

Calls findFile().

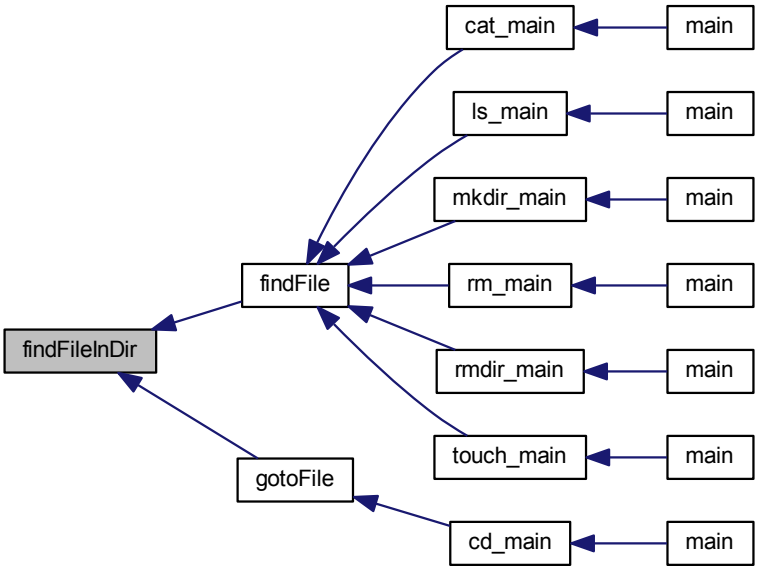
TODO: Check extensions

Definition at line 262 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.6 int getDirectoryFreeEntryCount (FILE_HEADER * *directory*)

Gets the number of free entries in a provided directory.

Parameters

in	*directory*	A pointer to the FILE_HEADER of a directory to get information from.

Returns

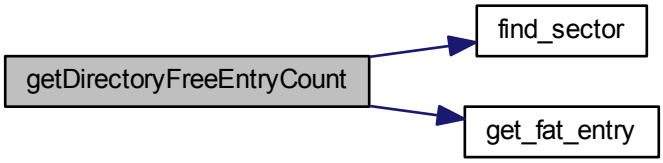
Returns the number of free entries in the given directory.

Return values

-1	Obtaining the free entry count was unsuccessful.

Definition at line 532 of file fileio.c.

Here is the call graph for this function:



7.34.1.7 char* getFileHeaderNameChunkFromFileNameString (char * *filenameString*)

Gives an 11-byte name and extension block for a file header from a filename string.

Remarks

Uses a static internal buffer char[11]. Not thread-safe.

Parameters

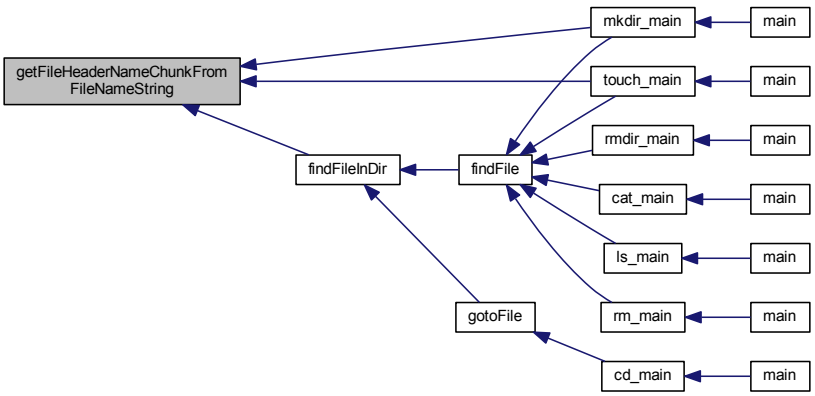
in	*filenameString*	A string containing a file name and extension (ex. "hello.txt").

Returns

Returns a static 11-char buffer that should match a file header's first 11 bytes (name and extension).

Definition at line 13 of file fileio.c.

Here is the caller graph for this function:



7.34.1.8 char* getFileNameStringFromFileHeader (FILE_HEADER_REG * *header*)

Gives a filename as a string from a file header.

Remarks

Uses a static internal string buffer. Not thread-safe.

Parameters

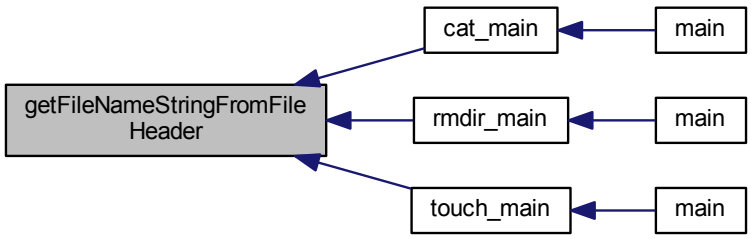
in	*header*	A file header pointer.

Returns

Returns a pointer to a static string buffer containing the file name and extension as a human-readable string.

Definition at line 93 of file fileio.c.

Here is the caller graph for this function:



7.34.1.9 void getNameFromLongNameFileHeader (const FILE_HEADER_LONGNAME * *header,* wchar_t * *name*)

Takes a pointer to a wide character string (at least 13 characters allocated) and populates it with the filename from a longname file header.

Parameters

in	*header*	A pointer to a FILE_HEADER_LONGNAME object.
out	*name*	A pointer to a wchar_t string (32-bits per char on Linux, 16-bits per char on Windows)

Definition at line 132 of file fileio.c.

7.34.1.10 FILE_HEADER_REG* getNextFreeDirectoryEntry (FILE_HEADER * *directory*)

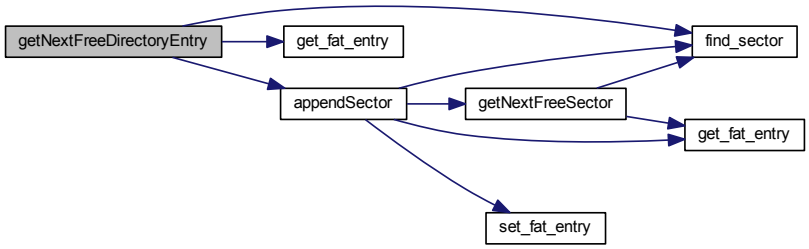
Gets the next free entry of the provided directory? Will expand directory if required.

Parameters

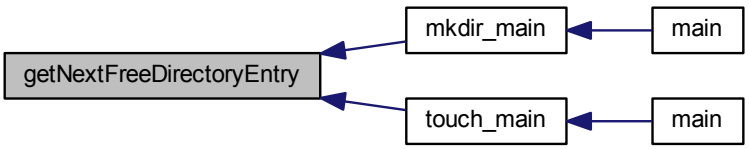
in	*directory*	A pointer to the FILE_HEADER of a directory to get the next free entry of.

Definition at line 671 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.11 bool gotoFile (const char * *name*, const FILE_HEADER * *searchLocation*, FILE_HEADER_REG ** *found*)

Moves within the directory stack to a file header with a specified name (and/or path)

Parameters

in	*name*	The name of the file to search for.
in	*searchLocation*	A pointer to a FILE_HEADER object to start searching from. This may be NULL to signify a search of the root directory.
out	*found*	A pointer to the file header, if found. This is NULL if root or if not found.

Return values

true	A file header with the information given was found. (If found is NULL and the return value is true, the file is root.)

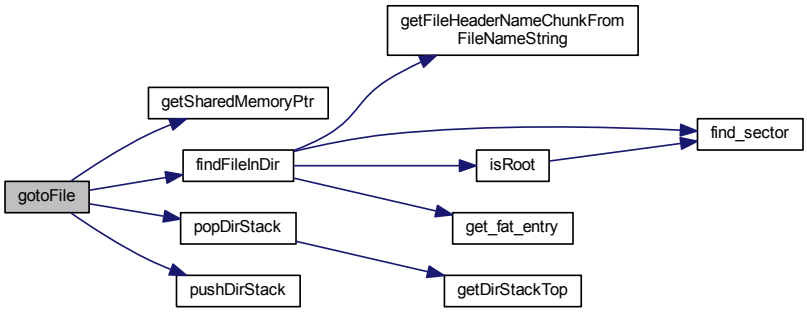
false	The target file header could not be found.

Remarks

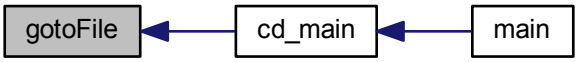
Calls findFile().

Definition at line 412 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.12 bool isDirectoryEmpty (FILE_HEADER * *directory*)

Checks if a directory is empty aside from the . and .. entries along with the long file headers.

Parameters

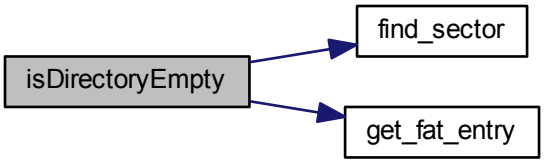
in	*directory*	A pointer to the FILE_HEADER of a directory to check.

Return values

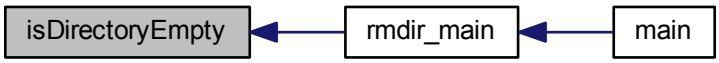
true	The given directory is empty.
false	The given directory contains entries.

Definition at line 724 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.13 bool isRoot (void * *file*)

Determines whether a given file header is a pointer to root.

Parameters

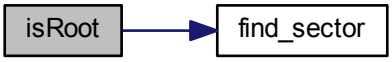
in	*file*	A pointer to a FILE_HEADER.

Returns

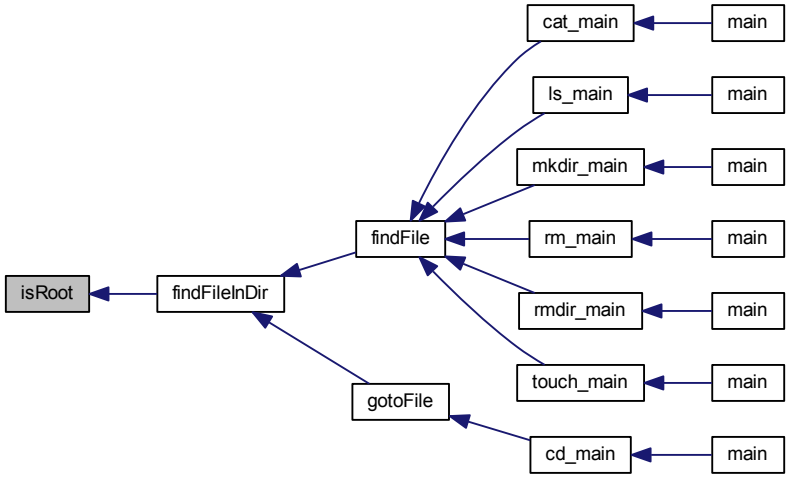
Returns 1 for true and 0 for false.

Definition at line 774 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.34.1.14 void printFileHeader (const FILE_HEADER * *header*)

Prints out the contents of a file header to a human-readable form in the console.

Parameters

in	*header*	A pointer to a FILE_HEADER union. (This could be either a FILE_HEADER_REG or a FILE_HEADER_LONGNAME.)

Definition at line 158 of file fileio.c.

Here is the call graph for this function:

7.34.1.15 void readFile (const FILE_HEADER * *header*, void ** *buffer*)

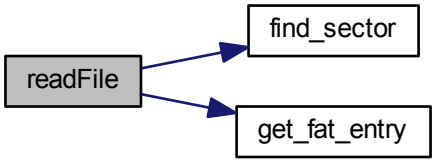
Reads the contents of a file into a function-allocated buffer given a pointer to its file header and a pointer to store the buffer at.

Parameters

in	*header*	A pointer to a FILE_HEADER_REG object.
out	*buffer*	A pointer to a pointer at which a buffer containing the bytes of the file are allocated by the function.

Definition at line 216 of file fileio.c.

Here is the call graph for this function:



Here is the caller graph for this function:

7.34.2 Variable Documentation

7.34.2.1 uint8_t* FILE_SYSTEM

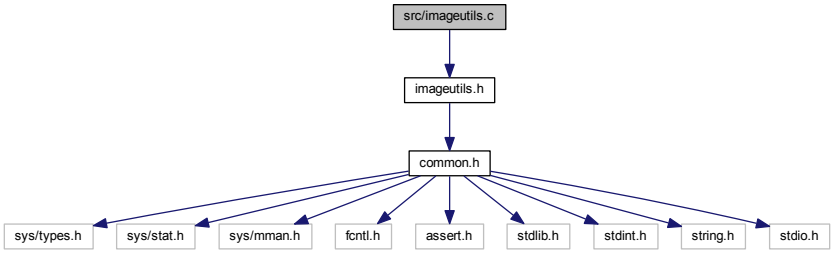
Memory map array for file.

Definition at line 14 of file imageutils.c.

7.35 src/imageutils.c File Reference

```
#include "imageutils.h"
```

Include dependency graph for imageutils.c:



Functions

- bool openFileSystem (const char *path)

  Memory maps the file system to FILE_SYSTEM.
- void closeFileSystem ()

  Closes memory map.

Variables

- uint8_t * FILE_SYSTEM = NULL

  Memory map array for file.

7.35.1 Function Documentation

7.35.1.1 void closeFileSystem ( )

Closes memory map.

Definition at line 84 of file imageutils.c.

7.35.1.2 bool openFileSystem (const char * *path*)

Memory maps the file system to FILE_SYSTEM.

Parameters

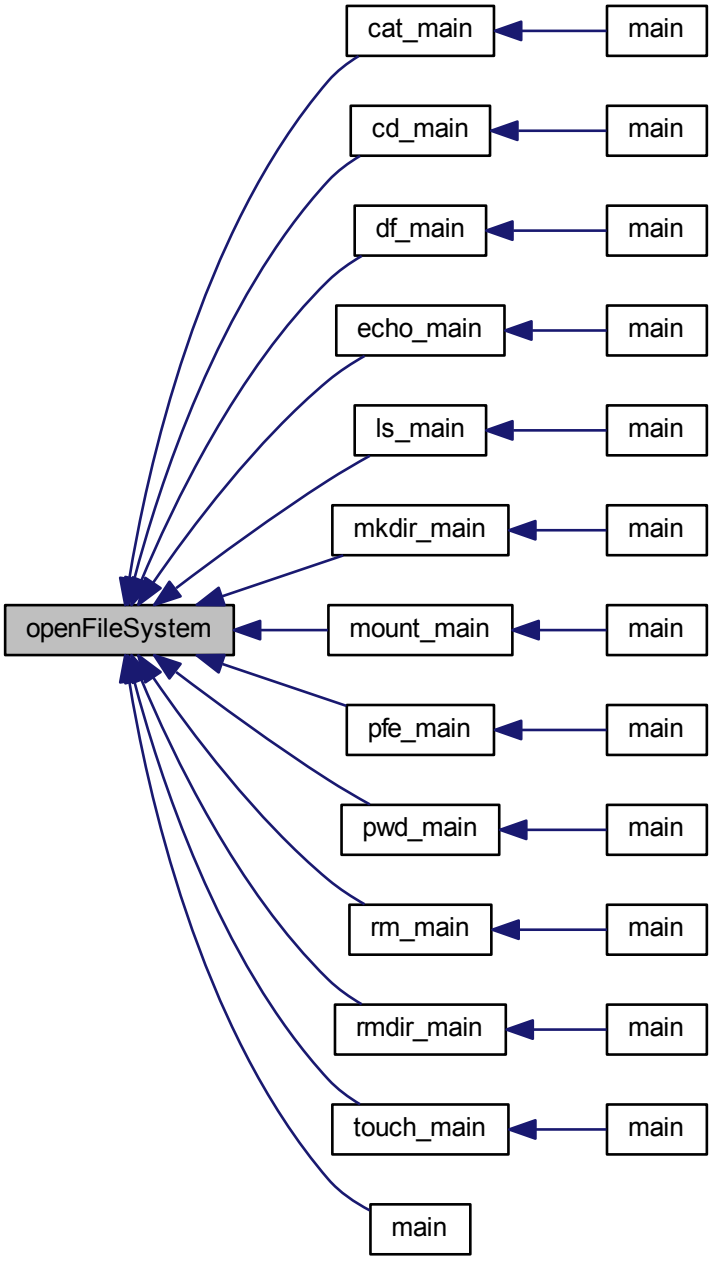
in	*path*	A const character string representing a path to an image file to mount.

Return values

true	The mount is successful.
false	The mount is unsuccessful.

Definition at line 17 of file imageutils.c.

Here is the caller graph for this function:



7.35.2 Variable Documentation

7.35.2.1 uint8_t* FILE_SYSTEM = NULL

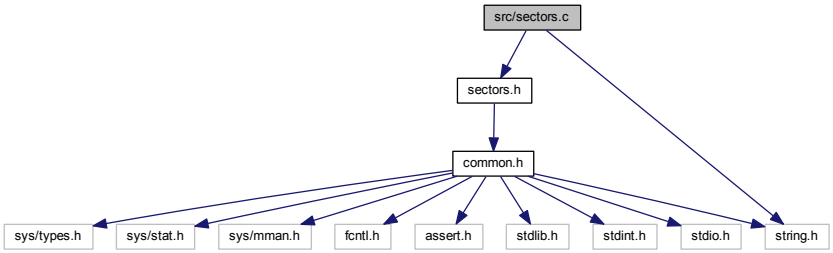
Memory map array for file.

Definition at line 14 of file imageutils.c.

7.36 src/sectors.c File Reference

```
#include "sectors.h"
#include "string.h"
```

Include dependency graph for sectors.c:



Functions

- int read_sector (int sector_number, unsigned char *buffer)

  Reads the contents of a sector given a sector number and places the contents in a user-allocated buffer.
- int write_sector (int sector_number, unsigned char *buffer)

  Writes the contents of a sector provided by the user with a sector number to which to write.
- void * find_sector (uint32_t sector_number)

  Returns a pointer to a sector in the filesystem memory map given a sector number.

Variables

- uint8_t * FILE_SYSTEM

  Memory map array for file.

7.36.1 Function Documentation

7.36.1.1 void* find_sector (uint32_t *sector_number*)

Returns a pointer to a sector in the filesystem memory map given a sector number.

Parameters

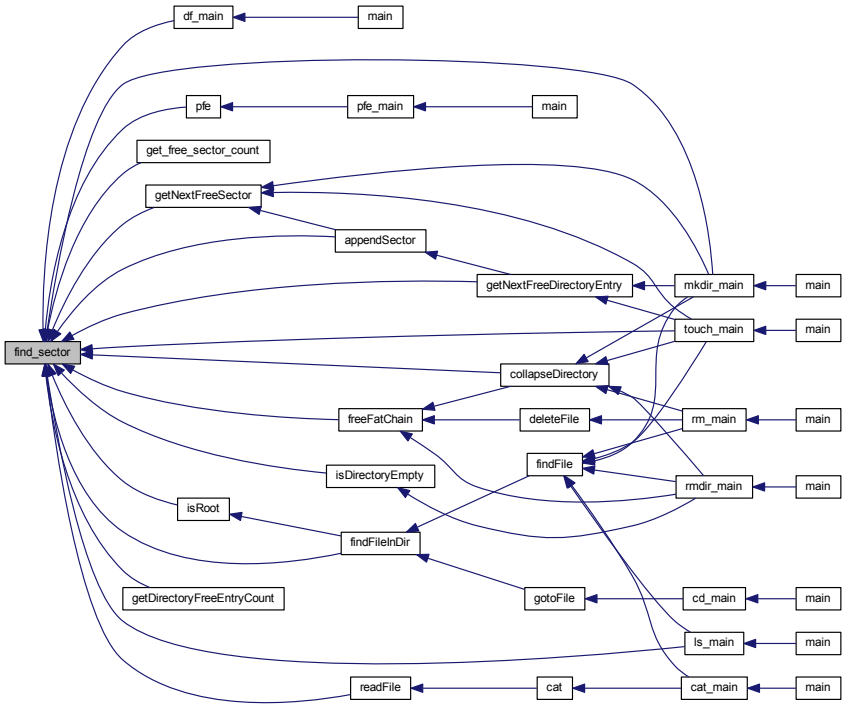
in	*sector_number*	A uint32_t describing the sector number to be found.

Returns

A void pointer pointing to the sector with the given number.

Definition at line 57 of file sectors.c.

Here is the caller graph for this function:



7.36.1.2 int read_sector (int *sector_number*, unsigned char * *buffer*)

Reads the contents of a sector given a sector number and places the contents in a user-allocated buffer.

Bug DEPRECATED - use find_sector() instead!

Parameters

in	*sector_number*	An int describing the number of the sector to read.
in	*buffer*	An unsigned char pointer to a buffer to read the file sector into (allocated by user).

Definition at line 7 of file sectors.c.

7.36.1.3 int write_sector (int *sector_number*, unsigned char * *buffer*)

Writes the contents of a sector provided by the user with a sector number to which to write.

Bug DEPRECATED - use find_sector() instead!

Parameters

in	*sector_number*	An int describing the number of the sector to write to.
in	*buffer*	A buffer provided by the user containing the sector bytes.

Definition at line 33 of file sectors.c.

7.36.2 Variable Documentation

7.36.2.1 uint8_t* FILE_SYSTEM

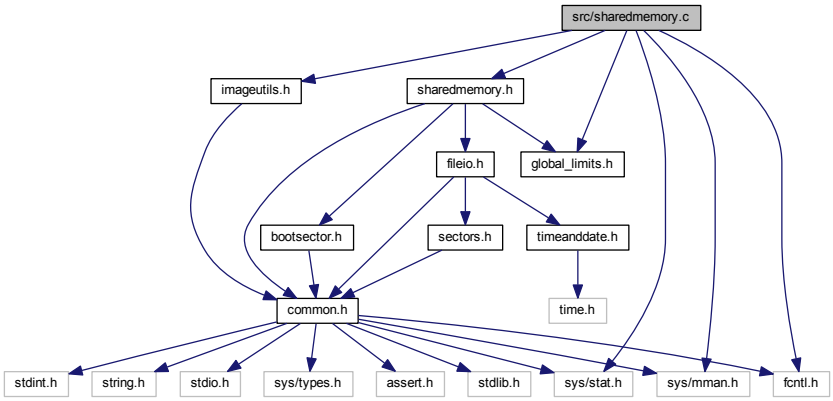
Memory map array for file.

Definition at line 14 of file imageutils.c.

7.37 src/sharedmemory.c File Reference

```
#include "sharedmemory.h"
#include "global_limits.h"
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include "imageutils.h"
```

Include dependency graph for sharedmemory.c:



Functions

- void createShared ()

  Allocates an internal shared memory file buffer.
- SHELL_SHARED_MEMORY * mapShared ()

  Gets a memory-mapped pointer to shared memory allocated by a call to createShared().
- SHELL_SHARED_MEMORY * getSharedMemoryPtr ()

  Gets the pointer to shared memory last set up by a call to mapShared().
- void unmapShared ()

Called to unmap the pointer to shared memory.

- FILE_HEADER * getDirStackTop (SHELL_SHARED_MEMORY *sharedMemory)

  Gets the address of FILE_HEADER at the top of the stored directory stack.
- FILE_HEADER * getDirStackIndex (SHELL_SHARED_MEMORY *sharedMemory, int index)

  Gets the address of a FILE_HEADER at the specified index of the stored directory stack.
- FILE_HEADER * popDirStack (SHELL_SHARED_MEMORY *sharedMemory)

  Pops the directory stack and returns a pointer to the topmost FILE_HEADER popped.
- void pushDirStack (SHELL_SHARED_MEMORY *sharedMemory, FILE_HEADER *header)

  Pushes a pointer to a FILE_HEADER the directory stack.
- void printWorkingDirectory (SHELL_SHARED_MEMORY *sharedMemory)

  Prints the working directory.
- void printWorkingDirectoryPath (SHELL_SHARED_MEMORY *sharedMemory)

  Prints the working directory path.
- const char * getWorkingPathFromStack (SHELL_SHARED_MEMORY *sharedMemory)

  Returns a working path as a string, given a pointer to a SHELL_SHARED_MEMORY object containing a directory stack.

Variables

- SHELL_SHARED_MEMORY * sharedMemoryPtr = NULL

7.37.1 Function Documentation

7.37.1.1 void createShared ( )

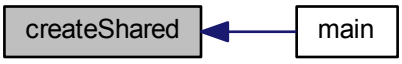
Allocates an internal shared memory file buffer.

Returns

N/A (call mapShared() after this to get a memory-mapped pointer to what this allocates)

Definition at line 32 of file sharedmemory.c.

Here is the caller graph for this function:



7.37.1.2 FILE_HEADER* getDirStackIndex (SHELL_SHARED_MEMORY * *sharedMemory*, int *index*)

Gets the address of a FILE_HEADER at the specified index of the stored directory stack.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.
in	*index*	The index to read from.

Returns

Returns a pointer to the FILE_HEADER.

Definition at line 81 of file sharedmemory.c.

Here is the caller graph for this function:

7.37.1.3 FILE_HEADER* getDirStackTop (SHELL_SHARED_MEMORY * *sharedMemory*)

Gets the address of FILE_HEADER at the top of the stored directory stack.

Parameters

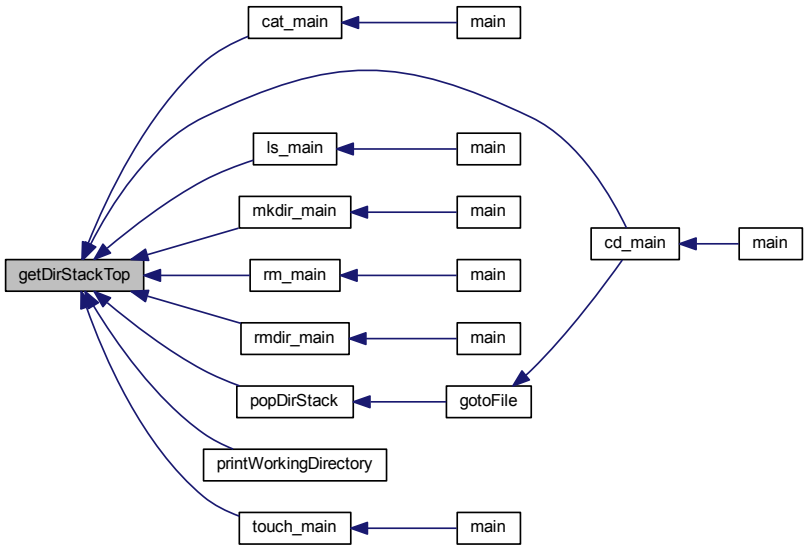
in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Returns

Returns a pointer to the FILE_HEADER.

Definition at line 71 of file sharedmemory.c.

Here is the caller graph for this function:



7.37.1.4 SHELL_SHARED_MEMORY* getSharedMemoryPtr ( )

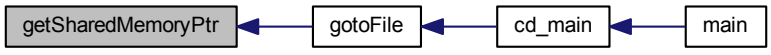
Gets the pointer to shared memory last set up by a call to mapShared().

Returns

Returns a pointer to a SHELL_SHARED_MEMORY struct.

Definition at line 59 of file sharedmemory.c.

Here is the caller graph for this function:



7.37.1.5 const char* getWorkingPathFromStack (SHELL_SHARED_MEMORY * *sharedMemory*)

Returns a working path as a string, given a pointer to a SHELL_SHARED_MEMORY object containing a directory stack.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Returns

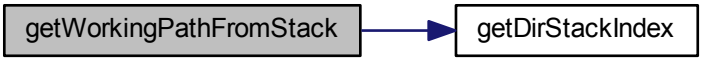
Returns a const char string containing the path.

Warning

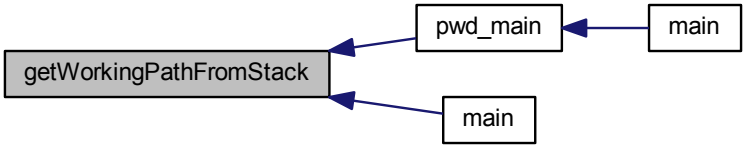
The pointer returned is to a statically allocated buffer within the function and should NOT be freed via free()! A copy should be made (e.g. via strdup()) if any manipulation is to be done.

Definition at line 155 of file sharedmemory.c.

Here is the call graph for this function:



Here is the caller graph for this function:



7.37.1.6 SHELL_SHARED_MEMORY* mapShared ( )

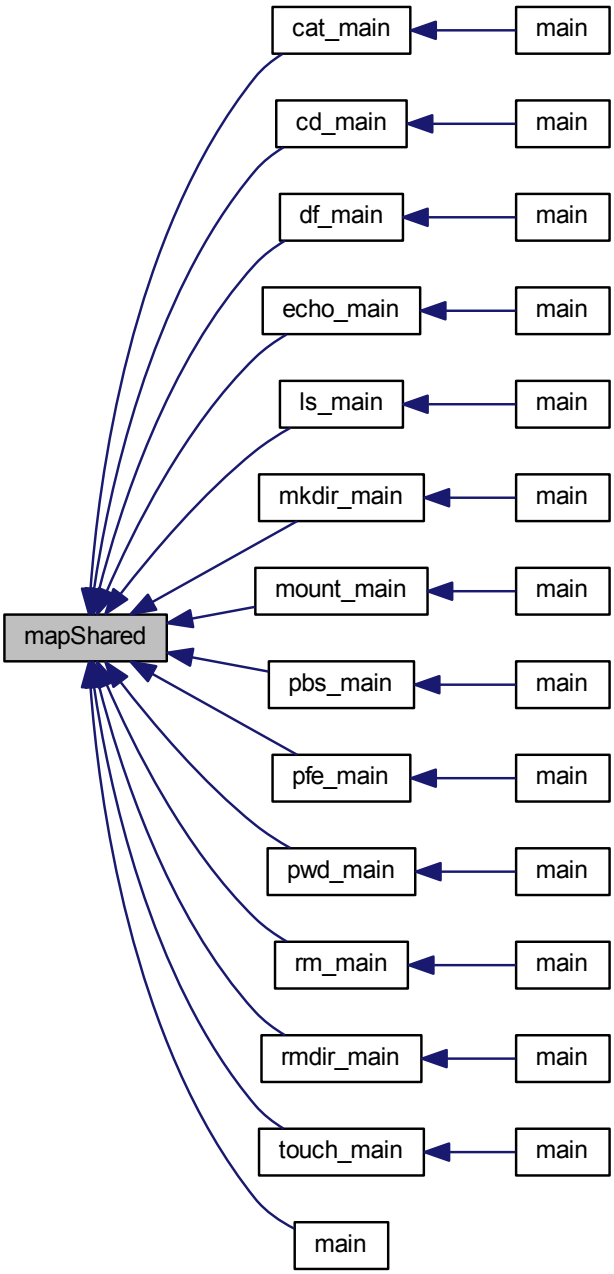
Gets a memory-mapped pointer to shared memory allocated by a call to createShared().

Returns

Returns a pointer to a SHELL_SHARED_MEMORY struct.

Definition at line 42 of file sharedmemory.c.

Here is the caller graph for this function:



7.37.1.7 FILE_HEADER* popDirStack (SHELL_SHARED_MEMORY * *sharedMemory*)

Pops the directory stack and returns a pointer to the topmost FILE_HEADER popped.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to operate on.

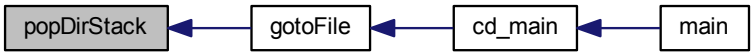
Returns

Returns a pointer to the FILE_HEADER popped.

Definition at line 91 of file sharedmemory.c.

Here is the call graph for this function:

Here is the caller graph for this function:



7.37.1.8 void printWorkingDirectory (SHELL_SHARED_MEMORY * *sharedMemory*)

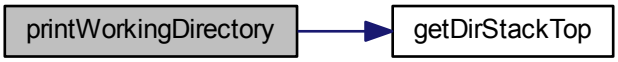
Prints the working directory.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Definition at line 134 of file sharedmemory.c.

Here is the call graph for this function:



7.37.1.9 void printWorkingDirectoryPath (SHELL_SHARED_MEMORY * *sharedMemory*)

Prints the working directory path.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to read from.

Definition at line 150 of file sharedmemory.c.

7.37.1.10 void pushDirStack (SHELL_SHARED_MEMORY * *sharedMemory*, FILE_HEADER * *header*)

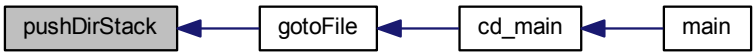
Pushes a pointer to a FILE_HEADER the directory stack.

Parameters

in	*sharedMemory*	The SHELL_SHARED_MEMORY object to operate on.
in	*header*	The FILE_HEADER pointer to be pushed.

Definition at line 119 of file sharedmemory.c.

Here is the caller graph for this function:

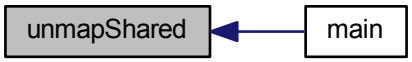


7.37.1.11 void unmapShared ( )

Called to unmap the pointer to shared memory.

Definition at line 64 of file sharedmemory.c.

Here is the caller graph for this function:



7.37.2 Variable Documentation

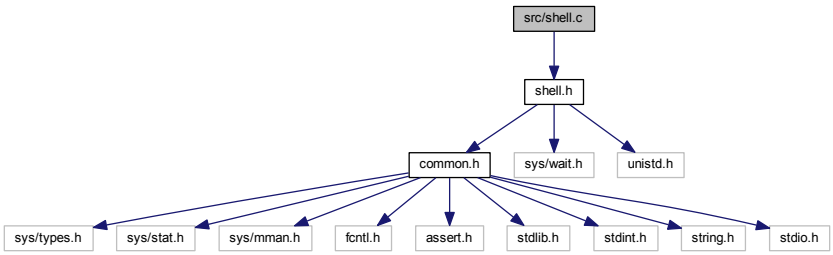
7.37.2.1 SHELL_SHARED_MEMORY* sharedMemoryPtr = NULL

Definition at line 30 of file sharedmemory.c.

7.38 src/shell.c File Reference

```
#include "shell.h"
```

Include dependency graph for shell.c:



Functions

- void execProcess (const char *path, char *arguments[])

  Fork off the shell and execute a process, giving it a list of optional arguments.
- int parseCommand (char *command, char ***commandArr)

  Parses a command from its arguments.
- void parsePathFileExtension (char *fullPath, char **pathOut, char **fileNameOut, char **extensionOut)

7.38.1 Function Documentation

7.38.1.1 void execProcess (const char * *path*, char * *arguments[ ]*)

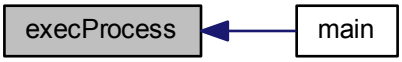
Fork off the shell and execute a process, giving it a list of optional arguments.

Parameters

in	*path*	A C-string holding the path to the executable to be run by the forked off shell.
in	*arguments*	An array of C-string arguments to be passed to the executable to be run by the forked off shell.

Definition at line 3 of file shell.c.

Here is the caller graph for this function:



7.38.1.2 int parseCommand (char * *command*, char *** *commandArr*)

Parses a command from its arguments.

Parameters

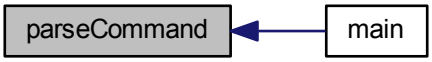
in	*command*	The command to parse.
out	*commandArr*	The command array output.

Returns

Returns the number of arguments delimited by spaces.

Definition at line 25 of file shell.c.

Here is the caller graph for this function:



7.38.1.3 void parsePathFileExtension (char * *fullPath*, char ** *pathOut*, char ** *fileNameOut*, char ** *extensionOut*)

Warning

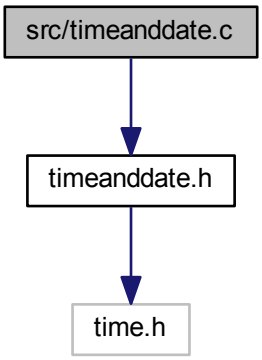
Made obsolete by getFileHeaderNameChunkFromFileNameString() in fileio.h.

Definition at line 58 of file shell.c.

7.39 src/timeanddate.c File Reference

```
#include "timeanddate.h"
```

Include dependency graph for timeanddate.c:



Functions

- void createFileDateTime (time_t in, FILE_DATE *date, FILE_TIME *time)

  Populates a FILE_TIME and a FILE_DATE from a time_t provided. Both the FILE_TIME and FILE_DATE pointers can be NULL.
- time_t timeDateToCTime (const FILE_DATE *date, const FILE_TIME *time, struct tm *out)

  Populates a tm struct given a FILE_DATE and a FILE_TIME. It is possible to simply put NULL in for either field if unavailable.
- void getHumanReadableDateTimeString (const FILE_DATE *date, const FILE_TIME *time, char *out)

  Populates a pre-allocated string buffer with the date and/or time provided.

Variables

- const char * MONTHS_STR [ ]

7.39.1 Function Documentation

7.39.1.1 void createFileDateTime (time_t *in*, FILE_DATE * *date*, FILE_TIME * *time*)

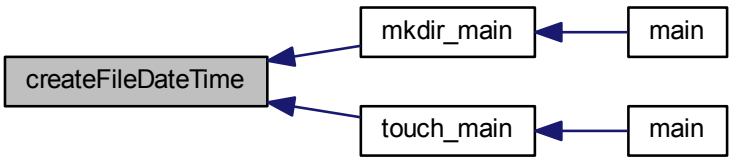
Populates a FILE_TIME and a FILE_DATE from a time_t provided. Both the FILE_TIME and FILE_DATE pointers can be NULL.

Parameters

in	*in*	A time_t object.
out	*time*	A FILE_TIME object to fill. (Can be NULL to ignore.)
out	*date*	A FILE_DATE object to fill. (Can be NULL to ignore.)

Definition at line 20 of file timeanddate.c.

Here is the caller graph for this function:



7.39.1.2 void getHumanReadableDateTimeString (const FILE_DATE * *date*, const FILE_TIME * *time*, char * *out*)

Populates a pre-allocated string buffer with the date and/or time provided.

Parameters

in	*date*	An optional FILE_DATE object. (Use NULL to negate.)
in	*time*	An optional FILE_TIME object. (Use NULL to negate.)
out	*out*	A pre-allocated string buffer large enough to contain the date and/or time string produced.

Definition at line 86 of file timeanddate.c.

Here is the caller graph for this function:

7.39.1.3 time_t timeDateToCTime (const FILE_DATE * *date*, const FILE_TIME * *time*, struct tm * *out*)

Populates a tm struct given a FILE_DATE and a FILE_TIME. It is possible to simply put NULL in for either field if unavailable.

Parameters

in	*date*	A FILE_DATE object.
in	*time*	A FILE_TIME object.
out	*out*	A pointer to an allocated tm struct. Can be NULL.

Returns

Returns a time_t of the time inputted.

Definition at line 46 of file timeanddate.c.

Here is the caller graph for this function:

7.39.2 Variable Documentation

7.39.2.1 const char* MONTHS_STR[ ]

Initial value:

=

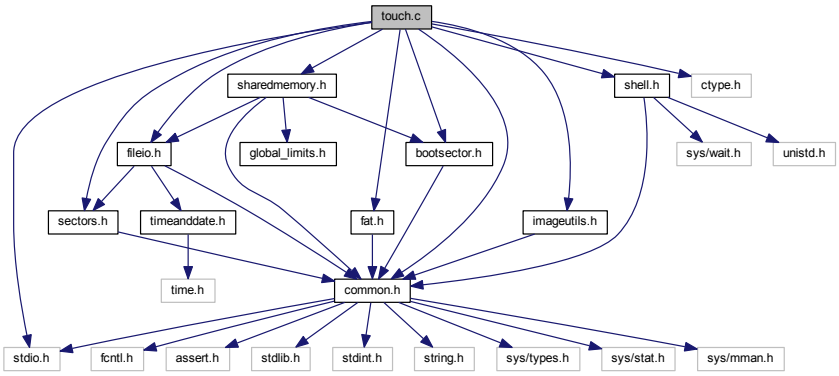
```
{
    "Jan",
    "Feb",
    "Mar",
    "Apr",
    "May",
    "Jun",
    "Jul",
    "Aug",
    "Sep",
    "Oct",
    "Nov",
    "Dec"
}
```

Definition at line 4 of file timeanddate.c.

7.40 touch.c File Reference

```
#include "common.h"
#include "imageutils.h"
#include "bootsector.h"
#include "sectors.h"
#include "fat.h"
#include "sharedmemory.h"
#include "shell.h"
#include "fileio.h"
#include <stdio.h>
#include <ctype.h>
```

Include dependency graph for touch.c:



Functions

- int touch_main (int argc, char *argv[])

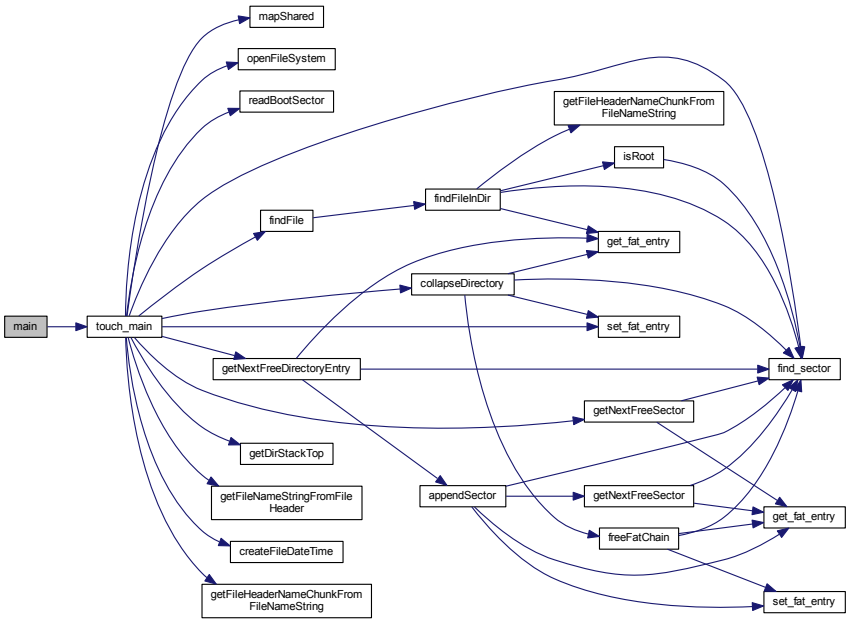
  Main function for touch.
- int main (int argc, char *argv[])

7.40.1 Function Documentation

7.40.1.1 int main (int *argc*, char * *argv[ ]*)

Definition at line 176 of file touch.c.

Here is the call graph for this function:



7.40.1.2 int touch_main (int *argc*, char * *argv[ ]*)

Main function for touch.

Test If provided with a single argument containing a non-existent filename, touch shall create a file with the given name within the current working directory.

If provided with a single argument containing a path and culminating in a non-existent filename, touch shall create a file with the given filename within the provided directory.

If provided with anything other than one argument, touch shall exit printing, "Invalid argument count; touch takes the path of the file to create.".

If provided with a single argument containing a valid path to an existing file or directory, touch shall print "File [file_name] touched." and shall update the timestamp of the file or directory.

If provided with ".", "..", touch shall exit printing, "[entry] is not allowed.".

If during the process of trying to create a new file, mkdir cannot allocate a file sector, touch shall exit printing, "Failed to allocate file sector.".

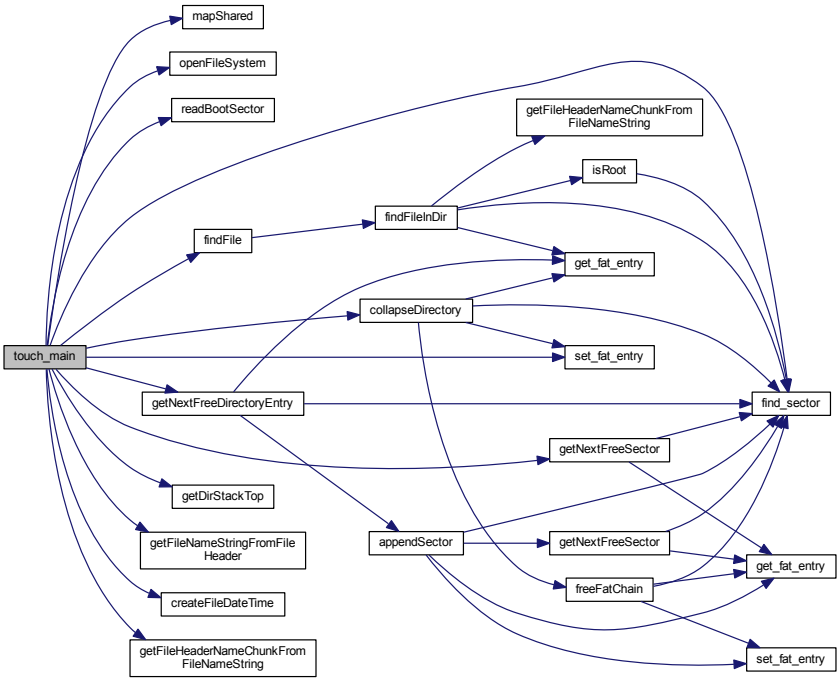
Test If there is not enough room in a directory to add a new file, a successful touch call shall expand the directory before attempting to create the new file.

Test If during the process of trying to create a new file, touch cannot allocate a file header, touch shall exit printing, "Failed to allocate file header.".

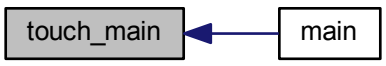
Test If successful in creating a file, touch shall add a timestamp accurate to the closest two seconds to the newly created file.

Definition at line 25 of file touch.c.

Here is the call graph for this function:



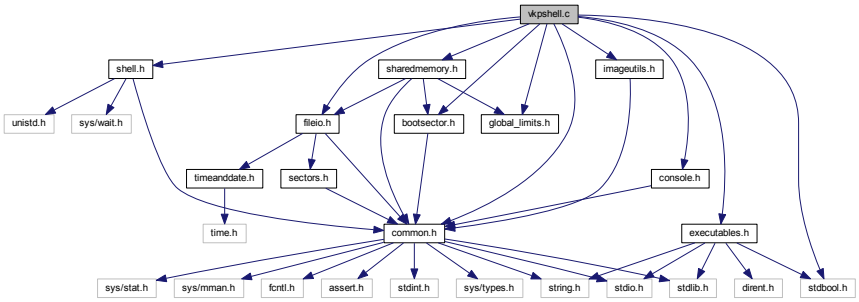
Here is the caller graph for this function:



7.41 vkpshell.c File Reference

```
#include "global_limits.h"
#include "common.h"
#include "shell.h"
#include "bootsector.h"
#include "console.h"
#include "sharedmemory.h"
#include "fileio.h"
#include "imageutils.h"
#include "executables.h"
#include "stdbool.h"
```

Include dependency graph for vkpshell.c:



Macros

- #define SHELL_PROMPT "shell∼"

Functions

- void showHelp ()

  Display a list of shell commands available.
- int main (int argc, char ∗argv[ ])

7.41.1 Macro Definition Documentation

7.41.1.1 #define SHELL_PROMPT "shell∼"

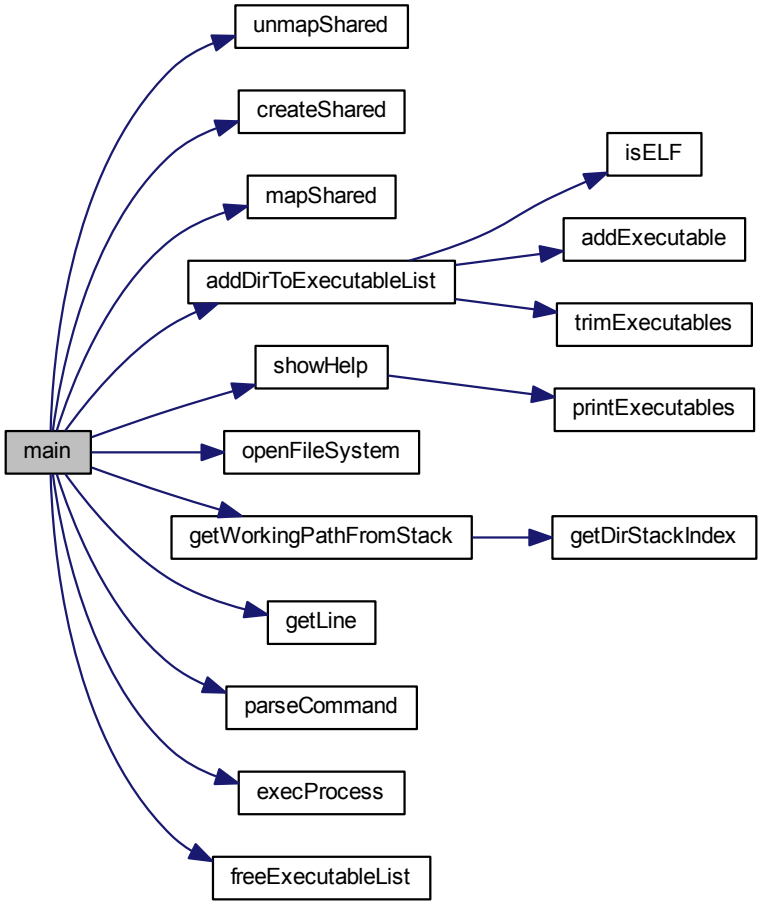
Definition at line 14 of file vkpshell.c.

7.41.2 Function Documentation

7.41.2.1 int main (int *argc,* char ∗ *argv[ ]*)

Definition at line 24 of file vkpshell.c.

Here is the call graph for this function:

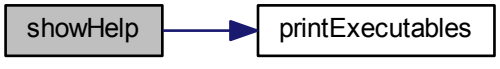


7.41.2.2 void showHelp ( )

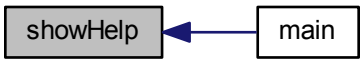
Display a list of shell commands available.

Definition at line 17 of file vkpshell.c.

Here is the call graph for this function:



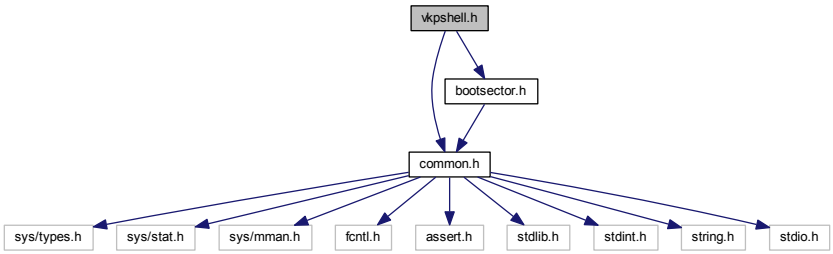
Here is the caller graph for this function:



7.42 vkpshell.h File Reference

```
#include "common.h"
#include "bootsector.h"
```

Include dependency graph for vkpshell.h:



Index