Sistema de Logging Geoposicional en Tiempo Real para Sistemas Empotrados

Generado por Doxygen 1.7.6.1

Miércoles, 2 de Julio de 2014 10:04:37

Índice general

1 Índice de estructura de datos **1**

1.1 Estructura de datos 1

2 Indice de archivos **3**

2.1 Lista de archivos 3

3 Documentación de las estructuras de datos **5**

3.1 Referencia de la Estructura FIL 5

3.1.1 Descripción detallada 5

3.1.2 Documentación de los campos 6

3.1.2.1 buff 6

3.1.2.2 buffindex 6

3.1.2.3 descriptorSector 6

3.1.2.4 dirty 6

3.1.2.5 err 6

3.1.2.6 flag 6

3.1.2.7 fsIndex 6

3.1.2.8 name 6

3.1.2.9 readPointer 7

3.1.2.10 startSector 7

3.1.2.11 writePointer 7

3.2 Referencia de la Estructura FS 7

3.2.1 Descripción detallada 9

3.2.2 Documentación de los campos 9

3.2.2.1 database 9

3.2.2.2 fatbase 9

3.2.2.3 files . . . 9
3.2.2.4 free_sector . . . 9
3.2.2.5 fs_size . . . 9
3.2.2.6 lastDescriptor . . . 10
3.2.2.7 sector_size . . . 10
3.2.2.8 start_address . . . 10
3.2.2.9 volbase . . . 10
3.2.2.10 win . . . 10
3.3 Referencia de la Estructura GPS_MSG . . . 10
3.3.1 Descripción detallada . . . 11
3.3.2 Documentación de los campos . . . 11
3.3.2.1 buffer . . . 11
3.3.2.2 count . . . 11

4 Documentación de archivos **13**
4.1 Referencia del Archivo FS/diskio.c . . . 13
4.1.1 Descripción detallada . . . 14
4.1.2 Documentación de las funciones . . . 15
4.1.2.1 byte_to_uint32 . . . 15
4.1.2.2 disk_initialize . . . 15
4.1.2.3 disk_ioctl . . . 15
4.1.2.4 disk_read . . . 16
4.1.2.5 disk_status . . . 16
4.1.2.6 disk_write . . . 16
4.1.2.7 flashState2FSState . . . 17
4.1.2.8 get_address . . . 17
4.1.2.9 get_start_end_address . . . 17
4.1.2.10 GetSector . . . 18
4.1.2.11 uint32_to_byte . . . 18
4.2 FS/diskio.c . . . 18
4.3 Referencia del Archivo FS/diskio.h . . . 22
4.3.1 Descripción detallada . . . 24
4.3.2 Documentación de los 'typedefs' . . . 24
4.3.2.1 DSTATUS . . . 24

4.3.3 Documentación de las funciones . . . 24
4.3.3.1 disk_initialize . . . 24
4.3.3.2 disk_ioctl . . . 25
4.3.3.3 disk_read . . . 25
4.3.3.4 disk_status . . . 25
4.3.3.5 disk_write . . . 26
4.3.3.6 GetSector . . . 26
4.4 FS/diskio.h . . . 26
4.5 Referencia del Archivo FS/ffconf.h . . . 27
4.5.1 Descripción detallada . . . 28
4.6 FS/ffconf.h . . . 29
4.7 Referencia del Archivo FS/fileSystem.c . . . 29
4.7.1 Descripción detallada . . . 31
4.7.2 Documentación de las funciones . . . 31
4.7.2.1 backup_fs . . . 31
4.7.2.2 byte_2_uint32 . . . 31
4.7.2.3 change_sector . . . 31
4.7.2.4 check_file . . . 32
4.7.2.5 check_fs . . . 32
4.7.2.6 close_all_files . . . 32
4.7.2.7 compare . . . 32
4.7.2.8 copy_file . . . 33
4.7.2.9 f_close . . . 33
4.7.2.10 f_getfree . . . 33
4.7.2.11 f_lseek . . . 34
4.7.2.12 f_mkfs . . . 34
4.7.2.13 f_mount . . . 34
4.7.2.14 f_open . . . 35
4.7.2.15 f_read . . . 35
4.7.2.16 f_sync . . . 35
4.7.2.17 f_truncateStart . . . 36
4.7.2.18 f_write . . . 36
4.7.2.19 loading_files . . . 36
4.7.2.20 read_file_entry . . . 37

4.7.2.21 reset_sector . . . 37
4.7.2.22 uint32_2_byte . . . 37
4.7.2.23 update_file . . . 38
4.8 FS/fileSystem.c . . . 38
4.9 Referencia del Archivo FS/fileSystem.h . . . 45
4.9.1 Descripción detallada . . . 48
4.9.2 Documentación de las enumeraciones . . . 48
4.9.2.1 FRESULT . . . 48
4.9.3 Documentación de las funciones . . . 49
4.9.3.1 f_close . . . 49
4.9.3.2 f_getfree . . . 49
4.9.3.3 f_lseek . . . 49
4.9.3.4 f_mkfs . . . 50
4.9.3.5 f_mount . . . 50
4.9.3.6 f_open . . . 50
4.9.3.7 f_read . . . 51
4.9.3.8 f_sync . . . 51
4.9.3.9 f_truncateStart . . . 51
4.9.3.10 f_write . . . 52
4.9.3.11 reset_sector . . . 52
4.10 FS/fileSystem.h . . . 52
4.11 Referencia del Archivo FS/integer.h . . . 54
4.11.1 Descripción detallada . . . 54
4.12 FS/integer.h . . . 55
4.13 Referencia del Archivo src/common.h . . . 55
4.13.1 Descripción detallada . . . 56
4.14 src/common.h . . . 56
4.15 Referencia del Archivo src/fs_task.c . . . 57
4.15.1 Descripción detallada . . . 57
4.15.2 Documentación de las funciones . . . 58
4.15.2.1 FSStartTask . . . 58
4.16 src/fs_task.c . . . 58
4.17 Referencia del Archivo src/fs_task.h . . . 60
4.17.1 Descripción detallada . . . 61

4.17.2 Documentación de las funciones 61
4.17.2.1 FSStartTask 61
4.18 src/fs_task.h 62
4.19 Referencia del Archivo src/gps_task.c 62
4.19.1 Descripción detallada 63
4.19.2 Documentación de las funciones 63
4.19.2.1 GPSHardwareInit 63
4.19.2.2 GPSStartTask 63
4.20 src/gps_task.c 63
4.21 Referencia del Archivo src/gps_task.h 66
4.21.1 Descripción detallada 67
4.21.2 Documentación de las funciones 67
4.21.2.1 GPSStartTask 67
4.22 src/gps_task.h 67
4.23 Referencia del Archivo src/main.c 67
4.23.1 Descripción detallada 68
4.23.2 Documentación de las funciones 69
4.23.2.1 Delay 69
4.23.2.2 main 69
4.23.2.3 prvSetupHardware 69
4.23.2.4 putChar 69
4.23.2.5 vApplicationStackOverflowHook 70
4.23.2.6 vApplicationTickHook 70
4.24 src/main.c 70

Capítulo 1

Índice de estructura de datos

1.1. Estructura de datos

Lista de estructuras con una breve descripción:

FIL
: Estructura que define un archivo 5

FS
: Estructura que define el Sistema de Archivos 7

GPS_MSG
: Estructura que define el mensaje de la cola de mensaje writeQueue 10

Capítulo 2

Indice de archivos

2.1. Lista de archivos

Lista de todos los archivos documentados y con descripciones breves:

FS/diskio.c
: Módulo que implementa las funciones del archivo diskio.h 13

FS/diskio.h
: Cabecera de la capa de abstracción de operaciones a bajo nivel sobre la memoria 22

FS/ffconf.h
: Fichero de configuración del sistema de archivos 27

FS/fileSystem.c
: Módulo que implementa las funciones del archivo fileSystem.h . . . 29

FS/fileSystem.h
: Cabecera principal del módulo del Sistema de Archivos 45

FS/integer.h
: Tipos de datos abreviados 54

src/common.h
: En este archivo se definen estructuras de datos comunes para todas las tareas 55

src/fs_task.c
: Módulo que implementa las funciones del archivo fs_task.h 57

src/fs_task.h
: Cabecera de la tarea que gestiona el sistema de archivos 60

src/gps_task.c
: Módulo que implementa las funciones del archivo gps_task.h 62

src/gps_task.h
: Cabecera de la tarea que gestiona el GPS 66

src/main.c
: Función principal 67

Capítulo 3

Documentación de las estructuras de datos

3.1. Referencia de la Estructura FIL

Estructura que define un archivo.

`#include <fileSystem.h>`

Campos de datos

- BYTE name [7]
- BYTE flag
- BYTE err
- BYTE dirty
- BYTE fsIndex
- DWORD startSector
- DWORD writePointer
- DWORD readPointer
- DWORD descriptorSector
- WORD buffindex
- BYTE buff [SECTOR_SIZE]

3.1.1. Descripción detallada

Estructura que define un archivo.

Definición en la línea 47 del archivo fileSystem.h.

3.1.2. Documentación de los campos

3.1.2.1. BYTE FIL::buff[SECTOR_SIZE]

buffer de escritura

Definición en la línea 58 del archivo fileSystem.h.

3.1.2.2. WORD FIL::buffindex

Índice del buffer donde se ha escrito por última vez

Definición en la línea 57 del archivo fileSystem.h.

3.1.2.3. DWORD FIL::descriptorSector

Sector del sistema de archivos en el que se encuentra el archivo

Definición en la línea 56 del archivo fileSystem.h.

3.1.2.4. BYTE FIL::dirty

Si hay operaciones de escritura pendientes

Definición en la línea 51 del archivo fileSystem.h.

3.1.2.5. BYTE FIL::err

Estado de error

Definición en la línea 50 del archivo fileSystem.h.

3.1.2.6. BYTE FIL::flag

Estado y permisos del archivo

Definición en la línea 49 del archivo fileSystem.h.

3.1.2.7. BYTE FIL::fsIndex

Índice en el sistema de archivos

Definición en la línea 52 del archivo fileSystem.h.

3.1.2.8. BYTE FIL::name[7]

Nombre del archivo

Definición en la línea 48 del archivo fileSystem.h.

3.1.2.9. DWORD FIL::readPointer

Puntero de lectura. En Bytes

Definición en la línea 55 del archivo fileSystem.h.

3.1.2.10. DWORD FIL::startSector

Sector de inicio del archivo

Definición en la línea 53 del archivo fileSystem.h.

3.1.2.11. DWORD FIL::writePointer

Puntero de escritura del archivo. Se encuentra al final del archivo. En bytes

Definición en la línea 54 del archivo fileSystem.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

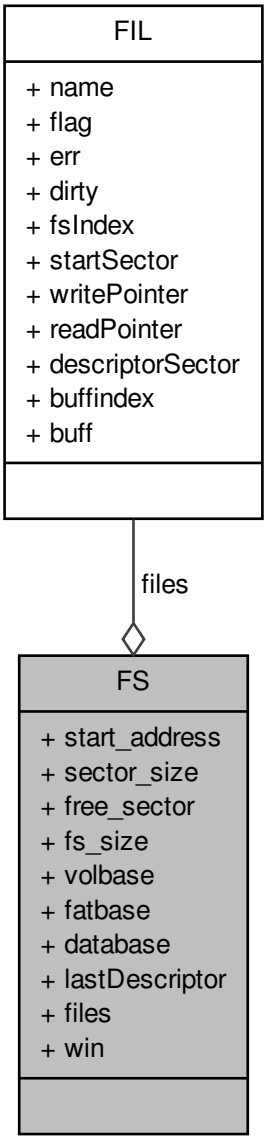
- FS/fileSystem.h

3.2. Referencia de la Estructura FS

Estructura que define el Sistema de Archivos.

```
#include <fileSystem.h>
```

Diagrama de colaboración para FS:



Campos de datos

- uint32_t start_address

- BYTE sector_size
- WORD free_sector
- WORD fs_size
- WORD volbase
- WORD fatbase
- WORD database
- WORD lastDescriptor
- FIL files [_MAX_FILES]
- BYTE win [SECTOR_SIZE]

3.2.1. Descripción detallada

Estructura que define el Sistema de Archivos.

Definición en la línea 64 del archivo fileSystem.h.

3.2.2. Documentación de los campos

3.2.2.1. WORD FS::database

Sector de inicio de la tabla de descriptores

Definición en la línea 71 del archivo fileSystem.h.

3.2.2.2. WORD FS::fatbase

Sector de inicio del sistema de archivos

Definición en la línea 70 del archivo fileSystem.h.

3.2.2.3. FIL FS::files[_MAX_FILES]

Vector de archivos abiertos

Definición en la línea 73 del archivo fileSystem.h.

3.2.2.4. WORD FS::free_sector

Número de sectores libres

Definición en la línea 67 del archivo fileSystem.h.

3.2.2.5. WORD FS::fs_size

Tamaño en sectores lógicos del sistema de archivos

Definición en la línea 68 del archivo fileSystem.h.

3.2.2.6. WORD FS::lastDescriptor

Sector donde se guardó el último descriptor de archivo

Definición en la línea 72 del archivo fileSystem.h.

3.2.2.7. BYTE FS::sector_size

Tamaño de cada sector lógico

Definición en la línea 66 del archivo fileSystem.h.

3.2.2.8. uint32_t FS::start_address

Dirección de inicio del sistema de archivos

Definición en la línea 65 del archivo fileSystem.h.

3.2.2.9. WORD FS::volbase

Sector de inicio del volúmen de datos

Definición en la línea 69 del archivo fileSystem.h.

3.2.2.10. BYTE FS::win[SECTOR_SIZE]

Buffer de lectura del sistema de archivos y de los archivos

Definición en la línea 74 del archivo fileSystem.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- FS/fileSystem.h

3.3. Referencia de la Estructura GPS_MSG

Estructura que define el mensaje de la cola de mensaje writeQueue.

`#include <common.h>`

Campos de datos

- uint16_t count
- uint8_t buffer [80]

3.3.1. Descripción detallada

Estructura que define el mensaje de la cola de mensaje writeQueue.

Definición en la línea 13 del archivo common.h.

3.3.2. Documentación de los campos

3.3.2.1. uint8_t GPS_MSG::buffer[80]

Buffer donde se almacena el mensaje

Definición en la línea 15 del archivo common.h.

3.3.2.2. uint16_t GPS_MSG::count

Número de caracteres del mensaje enviado

Definición en la línea 14 del archivo common.h.

La documentación para esta estructura fue generada a partir del siguiente fichero:

- src/common.h

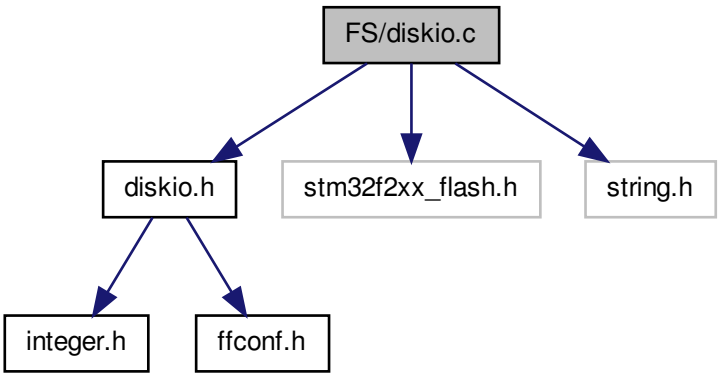
Capítulo 4

Documentación de archivos

4.1. Referencia del Archivo FS/diskio.c

Módulo que implementa las funciones del archivo diskio.h.

`#include "diskio.h" #include <stm32f2xx_flash.h> #include <string.h>` Dependencia gráfica adjunta para diskio.c:



'defines'

- #define **ADDR_FLASH_SECTOR_0** ((uint32_t)0x08000000)
- #define **ADDR_FLASH_SECTOR_1** ((uint32_t)0x08004000)
- #define **ADDR_FLASH_SECTOR_2** ((uint32_t)0x08008000)

- #define **ADDR_FLASH_SECTOR_3** ((uint32_t)0x0800C000)
- #define **ADDR_FLASH_SECTOR_4** ((uint32_t)0x08010000)
- #define **ADDR_FLASH_SECTOR_5** ((uint32_t)0x08020000)
- #define **ADDR_FLASH_SECTOR_6** ((uint32_t)0x08040000)
- #define **ADDR_FLASH_SECTOR_7** ((uint32_t)0x08060000)
- #define **ADDR_FLASH_SECTOR_8** ((uint32_t)0x08080000)
- #define **ADDR_FLASH_SECTOR_9** ((uint32_t)0x080A0000)
- #define **ADDR_FLASH_SECTOR_10** ((uint32_t)0x080C0000)
- #define **ADDR_FLASH_SECTOR_11** ((uint32_t)0x080E0000)

Funciones

- uint32_t GetSector (uint32_t Address)

  Función que devuelve el sector al que hace referencia la dirección dada.
- uint32_t byte_to_uint32 (const BYTE ∗src)

  Convierte un dato BYTE en un dato uint32_t.
- BYTE ∗ uint32_to_byte (uint32_t src, BYTE ∗dst)

  Convierte un dato uint32_t en un dato BYTE.
- DSTATUS get_start_end_address (int pdrv, int count, int sector, uint32_t ∗start-Address, uint32_t ∗endAddress)

  Convierte un número de sector en una dirección física y calcula la dirección de fin a partir del contador.
- uint32_t get_address (int pdrv, DWORD sector)

  Convierte un número de sector en una dirección física.
- DSTATUS disk_initialize (int pdrv)

  Función que incializa la memoria FLASH.
- DSTATUS flashState2FSState (FLASH_Status status)

  Convierte el estado de operación de la flash al estado de operación del sistema de archivos.
- DSTATUS disk_status (int pdrv)

  Función que devuelve el estado de la FLASH.
- DRESULT disk_read (int pdrv, BYTE ∗buff, DWORD sector, UINT count)

  Función que lee datos desde la memoria FLASH.
- DRESULT disk_write (int pdrv, const BYTE ∗buff, DWORD sector, UINT count)

  Función que escribe en la memoria FLASH.
- DRESULT disk_ioctl (int pdrv, BYTE cmd, void ∗buff)

  Función de control de entrada/salida, varias funciones.

4.1.1. Descripción detallada

Módulo que implementa las funciones del archivo diskio.h.

Definición en el archivo diskio.c.

4.1.2. Documentación de las funciones

4.1.2.1. uint32_t byte_to_uint32 (const BYTE * *src*)

Convierte un dato BYTE en un dato uint32_t.

Parámetros

src	Vector de 4 BYTES a convertir

Devuelve

Devuelve el valor correspondiente en uint32_t

Definición en la línea 75 del archivo diskio.c.

4.1.2.2. DSTATUS disk_initialize (int *pdrv*)

Función que incializa la memoria FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado

Devuelve

Código de estado de la operación

Definición en la línea 139 del archivo diskio.c.

4.1.2.3. DRESULT disk_ioctl (int *pdrv,* BYTE *cmd,* void * *buff*)

Función de control de entrada/salida, varias funciones.

Parámetros

pdrv	Identificador del sector físico utilizado
cmd	Comando que especifica qué función realizar
buff	Parámetro de entrada o salida dependiendo de la función a realizar

Devuelve

Código de estado de la operación

```
A continuación se especifican todos los comandos disponibles:
    CTRL_SYNC espera a que termine la última operación de la FLASH
    GET_SECTOR_COUNT Devuelve el número de sectores del sistema de archivos
    GET_SECTOR_SIZE Devuelve el tamaño de cada sector lógico
    GET_BLOCK_SIZE Devuelve el número de sectores físicos del sistema de archivos
```

```
CTRL_ERASE_SECTOR Resetea el sector especificado en buff
```

Definición en la línea 262 del archivo diskio.c.

4.1.2.4. DRESULT disk_read (int *pdrv*, BYTE * *buff*, DWORD *sector*, UINT *count*)

Función que lee datos desde la memoria FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado
buff	Buffer donde devolver los datos leídos
sector	Número de sector lógico donde empezar a leer
count	Número de sectores lógicos a leer

Devuelve

Código de estado de la operación

Definición en la línea 181 del archivo diskio.c.

4.1.2.5. DSTATUS disk_status (int *pdrv*)

Función que devuelve el estado de la FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado

Devuelve

Estado de la FLASH

Definición en la línea 168 del archivo diskio.c.

4.1.2.6. DRESULT disk_write (int *pdrv*, const BYTE * *buff*, DWORD *sector*, UINT *count*)

Función que escribe en la memoria FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado
buff	Stream de datos que se van a escribir
sector	Número de sector lógico donde escribir
count	Número de sectores lógicos a escribir

Devuelve

Código de estado de la operación

Definición en la línea 222 del archivo diskio.c.

4.1.2.7. DSTATUS flashState2FSState (FLASH_Status *status*)

Convierte el estado de operación de la flash al estado de operación del sistema de archivos.

Parámetros

status	Estado de operación de la FLASH

Devuelve

Código de estado de la operación

Definición en la línea 158 del archivo diskio.c.

4.1.2.8. uint32_t get_address (int *pdrv*, DWORD *sector*)

Convierte un número de sector en una dirección física.

Parámetros

pdrv	Identificador del sistema de archivos sector Número de sector lógico de inicio

Devuelve

Código de estado de la operación

Definición en la línea 127 del archivo diskio.c.

4.1.2.9. DSTATUS get_start_end_address (int *pdrv*, int *count*, int *sector*, uint32_t * *startAddress*, uint32_t * *endAddress*)

Convierte un número de sector en una dirección física y calcula la dirección de fin a partir del contador.

Parámetros

pdrv	Identificador del sistema de archivos
count	Número de sectores lógicos de diferencia entre la dirección de inicio y de fin sector Número de sector lógico de inicio
startAddress	Puntero donde se devuelve la dirección de inicio
endAddress	Puntero donde se devuelve la dirección de fin

Devuelve

Código de estado de la operación

Definición en la línea 107 del archivo diskio.c.

4.1.2.10. uint32_t GetSector (uint32_t *Address*)

Función que devuelve el sector al que hace referencia la dirección dada.

Parámetros

Address	Dirección de memoria

Devuelve

Sector físico al que pertenece la dirección dada

Definición en la línea 28 del archivo diskio.c.

4.1.2.11. BYTE* uint32_to_byte (uint32_t *src*, BYTE * *dst*)

Convierte un dato uint32_t en un dato BYTE.

Parámetros

src	Valor uint32_t a convertir
dst	Puntero donde se devolverá el valor

Devuelve

Devuelve el valor correspondiente en BYTE*

Definición en la línea 90 del archivo diskio.c.

4.2. FS/diskio.c

```
00001
00006 #include "diskio.h"    /* FileSystem lower layer API */
00007 #include <stm32f2xx_flash.h>
00008 #include <string.h>
00009
00010 #define ADDR_FLASH_SECTOR_0     ((uint32_t)0x08000000)
00011 #define ADDR_FLASH_SECTOR_1     ((uint32_t)0x08004000)
00012 #define ADDR_FLASH_SECTOR_2     ((uint32_t)0x08008000)
00013 #define ADDR_FLASH_SECTOR_3     ((uint32_t)0x0800C000)
00014 #define ADDR_FLASH_SECTOR_4     ((uint32_t)0x08010000)
00015 #define ADDR_FLASH_SECTOR_5     ((uint32_t)0x08020000)
00016 #define ADDR_FLASH_SECTOR_6     ((uint32_t)0x08040000)
00017 #define ADDR_FLASH_SECTOR_7     ((uint32_t)0x08060000)
00018 #define ADDR_FLASH_SECTOR_8     ((uint32_t)0x08080000)
00019 #define ADDR_FLASH_SECTOR_9     ((uint32_t)0x080A0000)
```

```
00020 #define ADDR_FLASH_SECTOR_10     ((uint32_t)0x080C0000)
00021 #define ADDR_FLASH_SECTOR_11     ((uint32_t)0x080E0000)
00022
00028 uint32_t GetSector(uint32_t Address) {
00029   uint32_t sector = 0;
00030
00031   if ((Address < ADDR_FLASH_SECTOR_1) && (Address >= ADDR_FLASH_SECTOR_0)) {
00032     sector = FLASH_Sector_0;
00033   } else if ((Address < ADDR_FLASH_SECTOR_2)
00034       && (Address >= ADDR_FLASH_SECTOR_1)) {
00035     sector = FLASH_Sector_1;
00036   } else if ((Address < ADDR_FLASH_SECTOR_3)
00037       && (Address >= ADDR_FLASH_SECTOR_2)) {
00038     sector = FLASH_Sector_2;
00039   } else if ((Address < ADDR_FLASH_SECTOR_4)
00040       && (Address >= ADDR_FLASH_SECTOR_3)) {
00041     sector = FLASH_Sector_3;
00042   } else if ((Address < ADDR_FLASH_SECTOR_5)
00043       && (Address >= ADDR_FLASH_SECTOR_4)) {
00044     sector = FLASH_Sector_4;
00045   } else if ((Address < ADDR_FLASH_SECTOR_6)
00046       && (Address >= ADDR_FLASH_SECTOR_5)) {
00047     sector = FLASH_Sector_5;
00048   } else if ((Address < ADDR_FLASH_SECTOR_7)
00049       && (Address >= ADDR_FLASH_SECTOR_6)) {
00050     sector = FLASH_Sector_6;
00051   } else if ((Address < ADDR_FLASH_SECTOR_8)
00052       && (Address >= ADDR_FLASH_SECTOR_7)) {
00053     sector = FLASH_Sector_7;
00054   } else if ((Address < ADDR_FLASH_SECTOR_9)
00055       && (Address >= ADDR_FLASH_SECTOR_8)) {
00056     sector = FLASH_Sector_8;
00057   } else if ((Address < ADDR_FLASH_SECTOR_10)
00058       && (Address >= ADDR_FLASH_SECTOR_9)) {
00059     sector = FLASH_Sector_9;
00060   } else if ((Address < ADDR_FLASH_SECTOR_11)
00061       && (Address >= ADDR_FLASH_SECTOR_10)) {
00062     sector = FLASH_Sector_10;
00063   } else //(Address < FLASH_END_ADDR) && (Address >= ADDR_FLASH_SECTOR_11))
00064   {
00065     sector = FLASH_Sector_11;
00066   }
00067
00068   return sector;
00069 }
00075 uint32_t byte_to_uint32(const BYTE* src) {
00076   uint32_t dst = 0;
00077   dst |= (uint32_t) src[0] « 0;
00078   dst |= (uint32_t) src[1] « 8;
00079   dst |= (uint32_t) src[2] « 16;
00080   dst |= (uint32_t) src[3] « 24;
00081
00082   return dst;
00083 }
00090 BYTE* uint32_to_byte(uint32_t src, BYTE * dst) {
00091   dst[3] = (BYTE) (src » 24);
00092   dst[2] = (BYTE) (src » 16);
00093   dst[1] = (BYTE) (src » 8);
00094   dst[0] = (BYTE) (src » 0);
00095
00096   return dst;
00097 }
00107 DSTATUS get_start_end_address(int pdrv, int count, int sector,
00108     uint32_t* startAddress, uint32_t* endAddress) {
00109   int size;
00110   disk_ioctl(pdrv, GET_SECTOR_SIZE, &size);
00111
00112   if (pdrv == 0) {
00113     startAddress = (uint32_t*) PHYSYCAL_START_ADDRESS + (size * sector);
00114     endAddress = (uint32_t*) startAddress + (size * count);
00115   } else {
00116     startAddress = (uint32_t*) PHYSYCAL_START_ADDRESS2 + (size * sector);
00117     endAddress = (uint32_t*) startAddress + (size * count);
00118   }
```

```
00119   return RES_OK;
00120 }
00127 uint32_t get_address(int pdrv, DWORD sector) {
00128   uint32_t address;
00129   int size;
00130   disk_ioctl(pdrv, GET_SECTOR_SIZE, &size);
00131   if (pdrv == 0)
00132     address = (uint32_t) PHYSYCAL_START_ADDRESS + (size * sector);
00133   else
00134     address = (uint32_t) PHYSYCAL_START_ADDRESS2 + (size * sector);
00135
00136   return address;
00137 }
00138
00139 DSTATUS disk_initialize(int pdrv /* Physical drive number (0..) */
00140 ) {
00141   if (pdrv != 0 && pdrv != 1)
00142     return RES_PARERR;
00143
00144   FLASH_Unlock();
00145
00146   FLASH_ClearFlag(FLASH_FLAG_EOP | FLASH_FLAG_OPERR | FLASH_FLAG_WRPERR |
00147   FLASH_FLAG_PGAERR | FLASH_FLAG_PGPERR | FLASH_FLAG_PGSERR);
00148   FLASH_Lock();
00149
00150   return RES_OK;
00151
00152 }
00158 DSTATUS flashState2FSState(FLASH_Status status) {
00159   if (status == FLASH_ERROR_WRP)
00160     return STA_PROTECT;
00161   return 0; //Status OK
00162 }
00163
00164 /*-----------------------------------------------------------------------*/
00165 /* Get Disk Status                                                       */
00166 /*-----------------------------------------------------------------------*/
00167
00168 DSTATUS disk_status(int pdrv /* Physical drive number (0..) */
00169 ) {
00170   if (pdrv != 0 && pdrv != 1)
00171     return RES_PARERR;
00172   FLASH_Status state = FLASH_GetStatus();
00173
00174   return flashState2FSState(state);
00175 }
00176
00177 /*-----------------------------------------------------------------------*/
00178 /* Read Sector(s)                                                        */
00179 /*-----------------------------------------------------------------------*/
00180
00181 DRESULT disk_read(int pdrv, /* Physical drive number (0..) */
00182 BYTE *buff, /* Data buffer to store read data */
00183 DWORD sector, /* Sector address (LBA) */
00184 UINT count /* Number of sectors to read (1..128) */
00185 ) {
00186
00187   if ((pdrv != 0 && pdrv != 1) || !count)
00188     return RES_PARERR;
00189
00190   DSTATUS status;
00191   uint32_t endAddress = 0, address = 0;
00192   uint32_t read = 0;
00193
00194   address = get_address(pdrv, sector);
00195   endAddress = get_address(pdrv, sector + count);
00196
00197   status = disk_ioctl(pdrv, CTRL_SYNC, (void*) NULL);
00198
00199   if (status != RES_OK)
00200     return status;
00201
00202   do {
00203     read = *(__IO uint32_t*) address;
```

```
00204     uint32_to_byte(read, buff);
00205
00206     buff += 4;
00207     address += 4;
00208
00209     status = disk_ioctl(pdrv, CTRL_SYNC, (void*) NULL);
00210     if (status != RES_OK)
00211       return status;
00212
00213   } while (address < endAddress);
00214
00215   return RES_OK;
00216 }
00217
00218 /*-----------------------------------------------------------------------*/
00219 /* Write Sector(s)                                                       */
00220 /*-----------------------------------------------------------------------*/
00221
00222 DRESULT disk_write(int pdrv, /* Physical drive number (0..) */
00223 const BYTE *buff, /* Data to be written */
00224 DWORD sector, /* Sector address (LBA) */
00225 UINT count /* Number of sectors to write (1..128) */
00226 ) {
00227   DSTATUS status;
00228
00229   if ((pdrv != 0 && pdrv != 1) || !count)
00230     return RES_PARERR;
00231
00232   status = disk_status(pdrv);
00233   if (status & STA_PROTECT)
00234     return RES_WRPRT;
00235
00236   uint32_t endAddress = 0, address = 0;
00237   uint32_t toWrite = 0;
00238   address = get_address(pdrv, sector);
00239   endAddress = get_address(pdrv, sector + count);
00240
00241   do {
00242     toWrite = byte_to_uint32(buff);
00243     FLASH_Unlock();
00244     status = FLASH_ProgramWord((uint32_t) address, toWrite);
00245
00246     FLASH_Lock();
00247
00248     if (status != FLASH_COMPLETE)
00249       return status;
00250
00251     address += 4;
00252     buff += 4;
00253   } while (address < endAddress); //Hasta fin del sector
00254
00255   return RES_OK;
00256 }
00257
00258 /*-----------------------------------------------------------------------*/
00259 /* Miscellaneous Functions                                               */
00260 /*-----------------------------------------------------------------------*/
00261
00262 DRESULT disk_ioctl(int pdrv, /* Physical drive nmuber (0..) */
00263 BYTE cmd, /* Control code */
00264 void *buff /* Buffer to send/receive control data */
00265 ) {
00266   if (pdrv != 0 && pdrv != 1)
00267     return RES_PARERR;
00268
00269   uint32_t address;
00270   DRESULT state;
00271
00272   switch (cmd) {
00273   case CTRL_SYNC:
00274     state = FLASH_WaitForLastOperation();
00275
00276     return flashState2FSState(state);
00277     break;
```

```
00278
00279   case GET_SECTOR_COUNT:
00280     *(DWORD*) buff = (DWORD) FS_SIZE;
00281
00282     return RES_OK;
00283     break;
00284
00285   case GET_SECTOR_SIZE:
00286     *(DWORD*) buff = SECTOR_SIZE;
00287
00288     return RES_OK;
00289     break;
00290
00291   case GET_BLOCK_SIZE:
00292     *(DWORD*) buff = 1;
00293
00294     return RES_OK;
00295     break;
00296
00297   case CTRL_ERASE_SECTOR:
00298     address = get_address(pdrv, (int) buff);
00299     FLASH_Unlock();
00300     state = flashState2FSState(
00301         FLASH_EraseSector(GetSector(address), VoltageRange_3));
00302     FLASH_Lock();
00303
00304     return state;
00305     break;
00306   }
00307
00308   return RES_PARERR;
00309 }
```

4.3. Referencia del Archivo FS/diskio.h

Cabecera de la capa de abstracción de operaciones a bajo nivel sobre la memoria.

`#include "integer.h"` `#include "ffconf.h"` Dependencia gráfica adjunta para diskio.h:

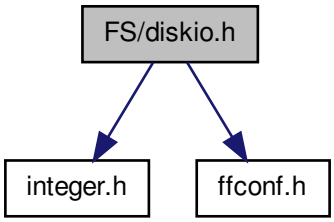
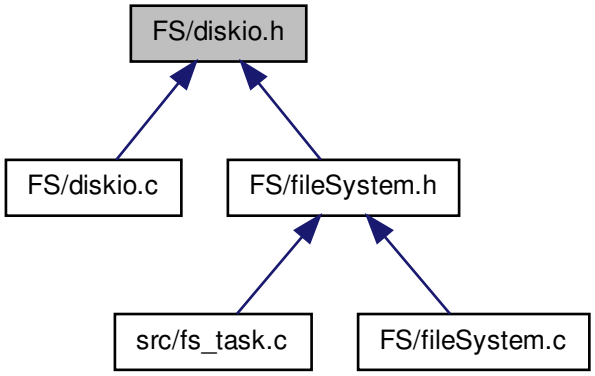


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



'defines'

- #define **STA_NOINIT** 0x01 /∗ Drive not initialized ∗/
- #define **STA_NODISK** 0x02 /∗ No medium in the drive ∗/
- #define **STA_PROTECT** 0x04 /∗ Write protected ∗/
- #define **CTRL_SYNC** 0 /∗ Wait for last operation ∗/
- #define **GET_SECTOR_COUNT** 1 /∗ Get media size (for only f_mkfs()) ∗/
- #define **GET_SECTOR_SIZE** 2 /∗ Get sector size (for multiple sector size (_MAX_SS >= 1024)) ∗/
- #define **GET_BLOCK_SIZE** 3 /∗ Get erase block size (for only f_mkfs()) ∗/
- #define **CTRL_ERASE_SECTOR** 4 /∗ Force erased a block of sectors (for only _USE_ERASE) ∗/

'typedefs'

- typedef BYTE DSTATUS

Enumeraciones

- enum **DRESULT** { **RES_OK** = 0, **RES_ERROR**, **RES_WRPRT**, **RES_NOTRDY**, **RES_PARERR** }

Funciones

- DSTATUS disk_initialize (int pdrv)

  Función que incializa la memoria FLASH.
- DSTATUS disk_status (int pdrv)

  Función que devuelve el estado de la FLASH.
- DRESULT disk_read (int pdrv, BYTE ∗buff, DWORD sector, UINT count)

  Función que lee datos desde la memoria FLASH.
- DRESULT disk_write (int pdrv, const BYTE ∗buff, DWORD sector, UINT count)

  Función que escribe en la memoria FLASH.
- DRESULT disk_ioctl (int pdrv, BYTE cmd, void ∗buff)

  Función de control de entrada/salida, varias funciones.
- uint32_t GetSector (uint32_t Address)

  Función que devuelve el sector al que hace referencia la dirección dada.

4.3.1. Descripción detallada

Cabecera de la capa de abstracción de operaciones a bajo nivel sobre la memoria.

Definición en el archivo diskio.h.

4.3.2. Documentación de los 'typedefs'

4.3.2.1. typedef BYTE DSTATUS

Enumeración de los estados de operación

Definición en la línea 11 del archivo diskio.h.

4.3.3. Documentación de las funciones

4.3.3.1. DSTATUS disk_initialize (int *pdrv*)

Función que incializa la memoria FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado

Devuelve

Código de estado de la operación

Definición en la línea 139 del archivo diskio.c.

4.3.3.2. DRESULT disk_ioctl (int *pdrv*, BYTE *cmd*, void * *buff*)

Función de control de entrada/salida, varias funciones.

Parámetros

pdrv	Identificador del sector físico utilizado
cmd	Comando que especifica qué función realizar
buff	Parámetro de entrada o salida dependiendo de la función a realizar

Devuelve

Código de estado de la operación

```
A continuación se especifican todos los comandos disponibles:
    CTRL_SYNC espera a que termine la última operación de la FLASH
    GET_SECTOR_COUNT Devuelve el número de sectores del sistema de archivos
    GET_SECTOR_SIZE Devuelve el tamaño de cada sector lógico
    GET_BLOCK_SIZE Devuelve el número de sectores físicos del sistema de archivos
    CTRL_ERASE_SECTOR Resetea el sector especificado en buff
```

Definición en la línea 262 del archivo diskio.c.

4.3.3.3. DRESULT disk_read (int *pdrv*, BYTE * *buff*, DWORD *sector*, UINT *count*)

Función que lee datos desde la memoria FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado
buff	Buffer donde devolver los datos leídos
sector	Número de sector lógico donde empezar a leer
count	Número de sectores lógicos a leer

Devuelve

Código de estado de la operación

Definición en la línea 181 del archivo diskio.c.

4.3.3.4. DSTATUS disk_status (int *pdrv*)

Función que devuelve el estado de la FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado

Devuelve

Estado de la FLASH

Definición en la línea 168 del archivo diskio.c.

4.3.3.5. DRESULT disk_write (int *pdrv*, const BYTE * *buff*, DWORD *sector*, UINT *count*)

Función que escribe en la memoria FLASH.

Parámetros

pdrv	Identificador del sector físico utilizado
buff	Stream de datos que se van a escribir
sector	Número de sector lógico donde escribir
count	Número de sectores lógicos a escribir

Devuelve

Código de estado de la operación

Definición en la línea 222 del archivo diskio.c.

4.3.3.6. uint32_t GetSector (uint32_t *Address*)

Función que devuelve el sector al que hace referencia la dirección dada.

Parámetros

Address	Dirección de memoria

Devuelve

Sector físico al que pertenece la dirección dada

Definición en la línea 28 del archivo diskio.c.

4.4. FS/diskio.h

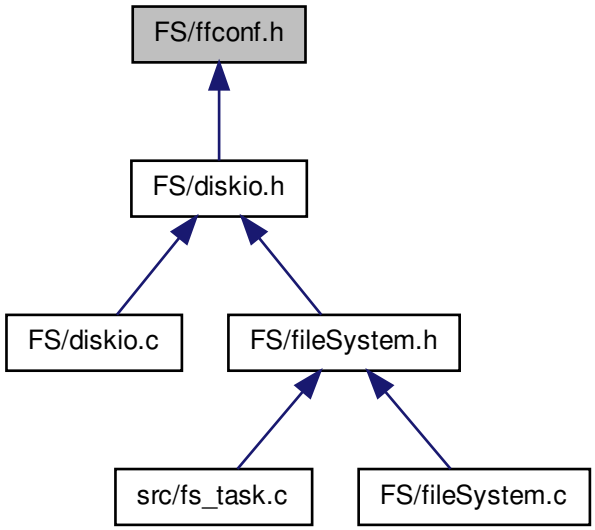
```
00001
00005 #include "integer.h"
00006 #include "ffconf.h"
00007
00008 /* Status of Disk Functions */
00009 typedef BYTE DSTATUS;
00010
00012 typedef enum {
00013   RES_OK = 0, /* 0: Successful */
00014   RES_ERROR,  /* 1: R/W Error */
00015   RES_WRPRT,  /* 2: Write Protected */
00016   RES_NOTRDY, /* 3: Not Ready */
```

```
00017   RES_PARERR  /* 4: Invalid Parameter */
00018 } DRESULT;
00019
00020 /*---------------------------------------*/
00021 /* Prototypes for disk control functions */
00022
00028 DSTATUS disk_initialize(int pdrv);
00029
00035 DSTATUS disk_status(int pdrv);
00036
00045 DRESULT disk_read(int pdrv, BYTE* buff, DWORD sector, UINT count);
00046
00055 DRESULT disk_write(int pdrv, const BYTE* buff, DWORD sector, UINT count);
00056
00072 DRESULT disk_ioctl(int pdrv, BYTE cmd, void* buff);
00073 uint32_t GetSector(uint32_t Address);
00074
00075 /* Disk Status Bits (DSTATUS) */
00076 #define STA_NOINIT    0x01  /* Drive not initialized */
00077 #define STA_NODISK    0x02  /* No medium in the drive */
00078 #define STA_PROTECT    0x04  /* Write protected */
00079
00080 /* Command code for disk_ioctrl function */
00081 #define CTRL_SYNC      0  /* Wait for last operation */
00082 #define GET_SECTOR_COUNT  1  /* Get media size (for only f_mkfs()) */
00083 #define GET_SECTOR_SIZE    2  /* Get sector size (for multiple sector size
       (_MAX_SS >= 1024)) */
00084 #define GET_BLOCK_SIZE    3  /* Get erase block size (for only f_mkfs()) */
00085 #define CTRL_ERASE_SECTOR  4  /* Force erased a block of sectors (for only
       _USE_ERASE) */
```

4.5. Referencia del Archivo FS/ffconf.h

Fichero de configuración del sistema de archivos.

Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



'defines'

- #define **_MAX_FILES** 3 /∗ Number of maximum files supported ∗/
- #define **SECTOR_SIZE** 16 /∗ Bytes ∗/
- #define **FS_SIZE** 8192 /∗ 128KB/16B= 8192 ∗/
- #define **MAX_FILE_SIZE** FS_SIZE/(_MAX_FILES+1)
- #define **PHYSYCAL_START_ADDRESS2** ((uint32_t)0x080A0000)
- #define **PHYSYCAL_START_ADDRESS** ((uint32_t)0x080C0000)

'typedefs'

- typedef unsigned long **uint32_t**

4.5.1. Descripción detallada

Fichero de configuración del sistema de archivos.

Definición en el archivo ffconf.h.

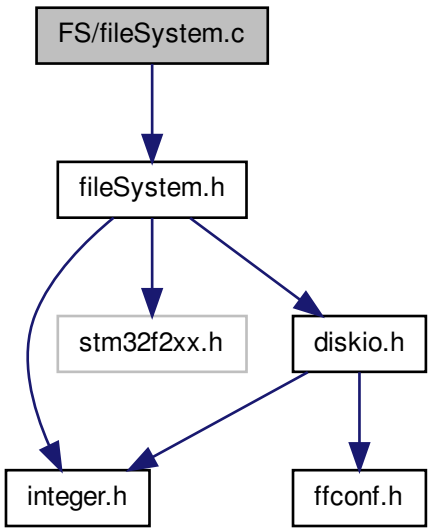
4.6. FS/ffconf.h

```
00001
00005 #define _MAX_FILES 3   /* Number of maximum files supported */
00006 #define SECTOR_SIZE 16 /* Bytes */
00007 #define FS_SIZE 8192  /* 128KB/16B= 8192 */
00008 #define MAX_FILE_SIZE FS_SIZE/(_MAX_FILES+1)
00009
00010 typedef unsigned long uint32_t;
00011
00012 /* Define start physical address,           *
00013  * START_ADDRESS is from 9 flash sector     *
00014  * START_ADDRESS2 is from 10 flash sector   */
00015 #define PHYSYCAL_START_ADDRESS2 ((uint32_t)0x080A0000)  // Sector  9
00016 #define PHYSYCAL_START_ADDRESS ((uint32_t)0x080C0000)  // Sector 10
```

4.7. Referencia del Archivo FS/fileSystem.c

Módulo que implementa las funciones del archivo fileSystem.h.

`#include "fileSystem.h"` Dependencia gráfica adjunta para fileSystem.c:



Funciones

- uint32_t byte_2_uint32 (BYTE *src)

  Convierte un dato BYTE en un dato uint32_t.

- BYTE * uint32_2_byte (uint32_t src, BYTE *dst)

  Convierte un dato uint32_t en un dato BYTE.
- FRESULT check_fs (int fsIndex)

  Comprueba si hay un sistema de archivos en memoria.
- FRESULT check_file (FIL *file)

  Comprueba si el archivo es válido.
- FRESULT read_file_entry (FIL *file, DWORD sector)

  Lee una entrada en la tabla de descriptores de archivos y comprueba si es válida.
- FRESULT loading_files (int fsIndex)

  Busca todos los archivos del sistema de archivos justo después de montarlo con éxito.
- int compare (const TCHAR *path, TCHAR *name)

  Compara dos cadenas correspondientes al nombre de archivos.
- int update_file (FIL *src, FIL *dst)

  Actualiza un archivo. En la práctica crea una copia del mismo.
- FRESULT copy_file (FIL *src, FIL *dst)

  Copia un archivo en otro.
- FRESULT backup_fs (int src, int dst)

  Crea una copia del sistema de archivos en otro sistema de archivos.
- FRESULT close_all_files ()

  Cierra todos los archivos del sistema de archivos actual.
- FRESULT reset_sector (int fsIndex)

  Función que resetea un sector.
- FRESULT f_lseek (FIL *fp, DWORD ofs)

  Función que mueve el puntero de lectura de un archivo.
- FRESULT change_sector (int opt)

  Cambia el sector físico por defecto en el que se crea el sistema de archivos.
- FRESULT f_open (FIL *fp, const TCHAR *path, BYTE mode)

  Función que abre o crea un archivo.
- FRESULT f_close (FIL *fp)

  Función que cierra un archivo.
- FRESULT f_read (FIL *fp, void *buff, UINT btr, UINT *br)

  Función que lee de un archivo.
- FRESULT f_write (FIL *fp, const void *buff, UINT btw, UINT *bw)

  Función que escribe de un archivo.
- FRESULT f_truncateStart (FIL *fp, DWORD ofs)

  Función que elimina parte de un archivo.
- FRESULT f_sync (FIL *fp)

  Función que sincroniza un archivo, escribe en memoria los datos que queden pendientes.
- FRESULT f_getfree (int fsIndex, UINT *mnfs)

  Función que devuelve el número de sectores libres del sistema de archivos.
- FRESULT f_mount (FS *fileSystem, int fsIndex, BYTE opt)

  Función que crea la estructura necesaria para crear un sistema de archivos.
- FRESULT f_mkfs (int fsIndex)

  Función que crea un sistema de archivos.

4.7.1. Descripción detallada

Módulo que implementa las funciones del archivo fileSystem.h.

Definición en el archivo fileSystem.c.

4.7.2. Documentación de las funciones

4.7.2.1. FRESULT backup_fs (int *src*, int *dst*)

Crea una copia del sistema de archivos en otro sistema de archivos.

Parámetros

src	Identificador del sistema de archivos fuente
dst	Identificador del sistema de archivos destino

Devuelve

Código de estado de la operación

Definición en la línea 227 del archivo fileSystem.c.

4.7.2.2. uint32_t byte_2_uint32 (BYTE * *src*)

Convierte un dato BYTE en un dato uint32_t.

Parámetros

src	Vector de 4 BYTES a convertir

Devuelve

Devuelve el valor correspondiente en uint32_t

Definición en la línea 15 del archivo fileSystem.c.

4.7.2.3. FRESULT change_sector (int *opt*)

Cambia el sector físico por defecto en el que se crea el sistema de archivos.

Parámetros

opt	Opción que indica si queremos, o no, har un backup de los archivos de un sistema en el otro. Un 1 indica que queremos hacer backup.

Devuelve

Código de estado de la operación

Definición en la línea 279 del archivo fileSystem.c.

4.7.2.4. FRESULT check_file (FIL * *file*)

Comprueba si el archivo es válido.

Parámetros

file	Puntero del archivo objetivo

Devuelve

Código de estado de la operación

Definición en la línea 75 del archivo fileSystem.c.

4.7.2.5. FRESULT check_fs (int *fsIndex*)

Comprueba si hay un sistema de archivos en memoria.

Parámetros

fsIndex	Índice del sistema de archivos objetivo

Devuelve

Código de estado de la operación

Definición en la línea 50 del archivo fileSystem.c.

4.7.2.6. FRESULT close_all_files ( )

Cierra todos los archivos del sistema de archivos actual.

Devuelve

Código de estado de la operación

Definición en la línea 244 del archivo fileSystem.c.

4.7.2.7. int compare (const TCHAR * *path*, TCHAR * *name*)

Compara dos cadenas correspondientes al nombre de archivos.

Parámetros

path	Cadena correspondiente al nombre del archivo que se quiere abrir
name	Cadena correspondiente al nombre de uno de los archivos del sistema de archivos

Devuelve

Código de estado de la operación

Definición en la línea 169 del archivo fileSystem.c.

4.7.2.8. FRESULT copy_file (FIL * *src*, FIL * *dst*)

Copia un archivo en otro.

Parámetros

src	Archivo fuente
dst	Archivo destino

Devuelve

Código de estado de la operación

Definición en la línea 206 del archivo fileSystem.c.

4.7.2.9. FRESULT f_close (FIL * *fp*)

Función que cierra un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero a cerrar

Devuelve

Código de estado de la operación

Definición en la línea 353 del archivo fileSystem.c.

4.7.2.10. FRESULT f_getfree (int *fsIndex*, UINT * *nclst*)

Función que devuelve el número de sectores libres del sistema de archivos.

Parámetros

fsIndex	Identificador del sistema de archivos
nclst	Puntero donde se devuelve el valor

Devuelve

Código de estado de la operación

Definición en la línea 504 del archivo fileSystem.c.

4.7.2.11. FRESULT f_lseek (FIL * *fp*, DWORD *ofs*)

Función que mueve el puntero de lectura de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero objetivo
ofs	Cantidad de bytes a mover desde la posición actual del puntero de lectura

Devuelve

Código de estado de la operación

Definición en la línea 266 del archivo fileSystem.c.

4.7.2.12. FRESULT f_mkfs (int *fsIndex*)

Función que crea un sistema de archivos.

Parámetros

fsIndex	Identificador del sistema de archivos a crear

Devuelve

Código de estado de la operación

Definición en la línea 561 del archivo fileSystem.c.

4.7.2.13. FRESULT f_mount (FS * *fileSystem*, int *fsIndex*, BYTE *opt*)

Función que crea la estructura necesaria para crear un sistema de archivos.

Parámetros

fileSystem	Puntero de Sistema de Archivos donde devolver el valor
fsIndex	Identificador del sistema de archivos objetivo
opt	Acepta dos opciones: 0, sólo crea la estructura necesaria para crear un sistema de archivos. 1, intenta montar un sistema de archivos existente

Devuelve

Código de estado de la operación

Definición en la línea 529 del archivo fileSystem.c.

4.7.2.14. FRESULT f_open (FIL * *fp*, const TCHAR * *path*, BYTE *mode*)

Función que abre o crea un archivo.

Parámetros

fp	Parámetro de entrada en el que se devolverá el puntero del archivo abierto
path	Nombre del archivo que queremos abrir
mode	Permisos de apertura del archivo

Devuelve

Código de estado de la operación

Definición en la línea 305 del archivo fileSystem.c.

4.7.2.15. FRESULT f_read (FIL * *fp*, void * *buff*, UINT *btr*, UINT * *br*)

Función que lee de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero de donde leer
buff	Buffer donde se devolverán los datos leídos
btr	Dirección de donde empezar a leer
br	Cantidad de bytes a leer

Devuelve

Código de estado de la operación

Definición en la línea 405 del archivo fileSystem.c.

4.7.2.16. FRESULT f_sync (FIL * *fp*)

Función que sincroniza un archivo, escribe en memoria los datos que queden pendientes.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero objetivo

Devuelve

Código de estado de la operación

Definición en la línea 490 del archivo fileSystem.c.

4.7.2.17. FRESULT f_truncateStart (FIL * *fp*, DWORD *ofs*)

Función que elimina parte de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero objetivo
ofs	Cantidad de bytes a eliminar desde la posición cero del archivo

Devuelve

Código de estado de la operación

Definición en la línea 473 del archivo fileSystem.c.

4.7.2.18. FRESULT f_write (FIL * *fp*, const void * *buff*, UINT *btw*, UINT * *bw*)

Función que escribe de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero de donde leer
buff	Stream de datos que se quieren escribir
btw	Dirección de donde empezar a escribir
bw	Cantidad de bytes a leer

Devuelve

Código de estado de la operación

Definición en la línea 435 del archivo fileSystem.c.

4.7.2.19. FRESULT loading_files (int *fsIndex*)

Busca todos los archivos del sistema de archivos justo después de montarlo con éxito.

Parámetros

fsIndex	Identificador del sistema de archivos

Devuelve

Código de estado de la operación

Definición en la línea 135 del archivo fileSystem.c.

4.7.2.20. FRESULT read_file_entry (FIL * *file*, DWORD *sector*)

Lee una entrada en la tabla de descriptores de archivos y comprueba si es válida.

Parámetros

file	Puntero de archivo en el que se devolverá el archivo si es válido
sector	Número de sector donde leer la entrada de la tabla de descriptores

Devuelve

Código de estado de la operación

Definición en la línea 118 del archivo fileSystem.c.

4.7.2.21. FRESULT reset_sector (int *fsIndex*)

Función que resetea un sector.

Parámetros

fsIndex	Identificador del sistema de archivos sobre el que resetear sus sectores

Devuelve

Código de estado de la operación

Definición en la línea 253 del archivo fileSystem.c.

4.7.2.22. BYTE* uint32_2_byte (uint32_t *src*, BYTE * *dst*)

Convierte un dato uint32_t en un dato BYTE.

Parámetros

src	Valor uint32_t a convertir
dst	Puntero donde se devolverá el valor

Devuelve

Devuelve el valor correspondiente en BYTE∗

Definición en la línea 36 del archivo fileSystem.c.

4.7.2.23. int update_file (FIL ∗ *src*, FIL ∗ *dst*)

Actualiza un archivo. En la práctica crea una copia del mismo.

Parámetros

src	Archivo fuente
dst	Archivo destino

Devuelve

Código de estado de la operación

Definición en la línea 184 del archivo fileSystem.c.

4.8. FS/fileSystem.c

```
00001
00005 #include "fileSystem.h"
00006
00007 static FS *fs[2];
00008 static int actual_fs = 0;
00009
00015 uint32_t byte_2_uint32(BYTE* src) {
00016
00017   uint32_t dst = 0;
00018   /*
00019    dst |= (uint32_t) src[0] « 0;
00020    dst |= (uint32_t) src[1] « 8;
00021    dst |= (uint32_t) src[2] « 16;
00022    dst |= (uint32_t) src[3] « 24;
00023    */
00024   dst = (uint32_t) src[0] « 24;
00025   dst |= (uint32_t) src[1] « 26;
00026   dst |= (uint32_t) src[2] « 8;
00027   dst |= (uint32_t) src[3] « 0;
00028   return dst;
00029 }
00036 BYTE* uint32_2_byte(uint32_t src, BYTE * dst) {
00037   dst[3] = (BYTE) (src » 24);
00038   dst[2] = (BYTE) (src » 16);
00039   dst[1] = (BYTE) (src » 8);
00040   dst[0] = (BYTE) (src » 0);
00041
00042   return dst;
00043 }
00044
00050 FRESULT check_fs(int fsIndex) {
00051
00052   FRESULT result = FR_OK;
00053   BYTE check[11] = { 'F', 'S', ' ', 'C', 'O', 'R', 'R', 'E', 'C', 'T', 'O' };
00054   int i;
00055   result = disk_initialize(fsIndex);
00056   if (result != FR_OK)
```

```
00057     return FR_NOT_READY;
00058
00059   result = disk_read(fsIndex, fs[fsIndex]->win, 0, 1); /* Read sector 0 */
00060   if (result != FR_OK)
00061     return FR_DISK_ERR;
00062
00063   for (i = 0; i < 11; i++)
00064     if (fs[fsIndex]->win[i] != check[i])
00065       return FR_NO_FILESYSTEM;
00066
00067   return FR_OK;
00068 }
00069
00075 FRESULT check_file(FIL *file) {
00076
00077   int i;
00078   /* Each file has 1 Byte validity, 4 bytes for start address, 4 bytes for end
      address, 7 bytes for file name */
00079
00080   if (fs[file->fsIndex]->win[0] != 255) /*Invalid entry, read next*/
00081     return FR_INVALID_OBJECT; /* 255 if when a byte is empty*/
00082
00083   file->dirty = 255;
00084
00085   file->startSector = fs[file->fsIndex]->win[1] « 24;
00086   file->startSector |= fs[file->fsIndex]->win[2] « 16;
00087   file->startSector |= fs[file->fsIndex]->win[3] « 8;
00088   file->startSector |= fs[file->fsIndex]->win[4] « 0;
00089
00090   file->writePointer = fs[file->fsIndex]->win[5] « 24;
00091   file->writePointer |= fs[file->fsIndex]->win[6] « 16;
00092   file->writePointer |= fs[file->fsIndex]->win[7] « 8;
00093   file->writePointer |= fs[file->fsIndex]->win[8] « 0;
00094
00095   file->readPointer = 0;
00096   file->flag = 0;
00097   file->err = 0;
00098   file->buffindex = 0;
00099
00100   if ((file->startSector == 4294967295)
00101       && (file->writePointer == 4294967295)) { /* This number is if buff is
      empty*/
00102     file->err = 1;
00103     return FR_NO_FILE; /* No more fat entries, no more files */
00104   }
00105
00106   for (i = 0; i < 7; i++)
00107     file->name[i] = fs[file->fsIndex]->win[i + 9];
00108
00109   return FR_OK;
00110 }
00111
00118 FRESULT read_file_entry(FIL *file, DWORD sector) {
00119
00120   FRESULT result = FR_OK;
00121   result = disk_read(file->fsIndex, fs[file->fsIndex]->win, sector, 1);
00122   if (result != FR_OK)
00123     return FR_DISK_ERR;
00124
00125   result = check_file(file);
00126   if (result == FR_OK)
00127     file->descriptorSector = sector;
00128   return result;
00129 }
00135 FRESULT loading_files(int fsIndex) {
00136
00137   FRESULT result = FR_OK;
00138   DWORD sector;
00139
00140   int nfiles = 0;
00141   for (sector = fs[fsIndex]->fatbase; sector < fs[fsIndex]->database;
00142       sector++) {
00143     fs[fsIndex]->files[nfiles].fsIndex = fsIndex;
00144     result = read_file_entry(&fs[fsIndex]->files[nfiles], sector);
```

```
00145     if (result == FR_OK)
00146       nfiles++;
00147
00148     if (result == FR_NO_FILE) {
00149       fs[fsIndex]->lastDescriptor = sector;
00150       break; /* if no more files then is the last sector */
00151     }
00152     if (nfiles == _MAX_FILES) {
00153       fs[fsIndex]->lastDescriptor = sector + 1;
00154       break; /* if no more files then is the last sector */
00155     }
00156   }
00157   if (nfiles < _MAX_FILES)
00158     for (; nfiles < _MAX_FILES; nfiles++) {
00159       fs[fsIndex]->files[nfiles].err = 1;
00160     }
00161   return FR_OK;
00162 }
00169 int compare(const TCHAR *path, TCHAR *name) {
00170   int i;
00171   for (i = 0; i < 7; i++) { /* 7 is the max length of name */
00172     if (path[i] != name[i])
00173       return 0;
00174   }
00175
00176   return 1;
00177 }
00184 int update_file(FIL *src, FIL *dst) {
00185   dst->buffindex = src->buffindex;
00186   dst->descriptorSector = src->descriptorSector;
00187   dst->dirty = src->dirty;
00188   dst->err = src->err;
00189   dst->flag = src->flag;
00190   dst->readPointer = src->readPointer;
00191   dst->startSector = src->startSector;
00192   dst->writePointer = src->writePointer;
00193   dst->fsIndex = src->fsIndex;
00194   int i;
00195   for (i = 0; i < 7; i++)
00196     dst->name[i] = src->name[i];
00197
00198   return 1;
00199 }
00206 FRESULT copy_file(FIL *src, FIL *dst) {
00207   FRESULT result = FR_OK;
00208   UINT byteRead = 0, byteWrite = 0;
00209   f_open(dst, (TCHAR*) src->name, FA_OPEN_ALWAYS | FA_READ | FA_WRITE);
00210
00211   while (src->readPointer < src->writePointer) { /* ReadPointer se actualiza al
       leer */
00212     f_read(src, fs[src->fsIndex]->win, SECTOR_SIZE, &byteRead);
00213     f_write(dst, fs[src->fsIndex]->win, byteRead, &byteWrite);
00214   }
00215
00216   f_close(src);
00217   f_close(dst);
00218
00219   return result;
00220 }
00227 FRESULT backup_fs(int src, int dst) {
00228   FRESULT result = FR_OK;
00229   int i;
00230   for (i = 0; i < _MAX_FILES; i++) {
00231     if (fs[src]->files[i].err == -1)
00232       break;
00233     result = copy_file(&fs[src]->files[i], &fs[dst]->files[i]);
00234     if (result != FR_OK)
00235       return result;
00236   }
00237
00238   return result;
00239 }
00244 FRESULT close_all_files() {
00245   FRESULT result = FR_OK;
```

```
00246   int i;
00247   for (i = 0; i < _MAX_FILES; i++)
00248     if (fs[actual_fs]->files[i].err == 0)
00249       f_close(&fs[actual_fs]->files[i]);
00250   return result;
00251 }
00252
00253 FRESULT reset_sector(int fsIndex) {
00254   FRESULT result = FR_OK;
00255   uint32_t address;
00256   if (fsIndex == 0)
00257     address = PHYSYCAL_START_ADDRESS;
00258   else
00259     address = PHYSYCAL_START_ADDRESS2;
00260
00261   result = disk_ioctl(fsIndex, CTRL_ERASE_SECTOR, (void *) address);
00262
00263   return result;
00264 }
00265
00266 FRESULT f_lseek(FIL* fp, DWORD ofs) {
00267   if (fp->readPointer + ofs < fp->writePointer)
00268     fp->readPointer += ofs;
00269   else
00270     fp->readPointer = fp->writePointer - 4;
00271   return FR_OK;
00272 }
00273
00279 FRESULT change_sector(int opt) {
00280   FRESULT result = FR_OK;
00281   FS fs1, fs2;
00282   if (actual_fs == 0) {
00283     reset_sector(1);
00284     f_mount(&fs1, 1, 0);
00285     f_mkfs(1);
00286     f_mount(&fs1, 1, 1);
00287     f_mount(&fs2, 0, 1);
00288     actual_fs = 1;
00289     if (opt == 1)
00290       result = backup_fs(0, 1);
00291   } else {
00292     reset_sector(0);
00293     f_mount(&fs1, 0, 0);
00294     f_mkfs(0);
00295     f_mount(&fs1, 0, 1);
00296     f_mount(&fs2, 1, 1);
00297     actual_fs = 0;
00298     if (opt == 1)
00299       result = backup_fs(1, 0);
00300   }
00301   return result;
00302 }
00303
00304 /* Open or create a file */
00305 FRESULT f_open(FIL* fp, /* Pointer to the blank file object */
00306 const TCHAR* path, /* Pointer to the file name */
00307 BYTE mode /* Access mode and file open mode flags */
00308 ) {
00309   FRESULT result = FR_OK;
00310   int i, j;
00311   int find = 0;
00312
00313   if (!fp)
00314     return FR_INVALID_OBJECT;
00315   mode &= FA_READ | FA_WRITE | FA_CREATE_ALWAYS | FA_OPEN_ALWAYS
00316       | FA_CREATE_NEW;
00317
00318   for (i = 0; i < _MAX_FILES && find == 0; i++) {
00319     if (fs[actual_fs]->files[i].err == 1)
00320       break;
00321     if (compare(path, (TCHAR*) fs[actual_fs]->files[i].name) == 1) {
00322       fs[actual_fs]->files[i].flag = mode;
00323       find = 1;
00324       break;
```

```
00325     }
00326   }
00327   if (i == _MAX_FILES && find == 0)
00328     return FR_TOO_MANY_FILES;
00329   if (find == 0
00330       && (mode & (FA_CREATE_ALWAYS | FA_OPEN_ALWAYS | FA_CREATE_NEW))) {
00331
00332     for (j = 0; j < 7; j++) {
00333       fs[actual_fs]->files[i].name[j] = (BYTE) path[j];
00334     }
00335     fs[actual_fs]->files[i].dirty = 1; /* Created is dirty */
00336     fs[actual_fs]->files[i].flag = mode;
00337     fs[actual_fs]->files[i].fsIndex = actual_fs;
00338     fs[actual_fs]->files[i].startSector = fs[actual_fs]->database * (i + 1);
00339     fs[actual_fs]->files[i].readPointer = 0;
00340     fs[actual_fs]->files[i].writePointer = 0;
00341     fs[actual_fs]->files[i].err = 0;
00342     fs[actual_fs]->files[i].buffindex = 0;
00343     fs[actual_fs]->files[i].descriptorSector = 0;
00344
00345   } else if (find == 0)
00346     result = FR_NO_FILE;
00347
00348   update_file(&fs[actual_fs]->files[i], fp);
00349   return result;
00350 }
00351
00352 /* Close an open file object */
00353 FRESULT f_close(FIL *fp /* Pointer to the file object to be closed */) {
00354   FRESULT res = FR_OK;
00355   if (fp->buffindex != 0) {
00356     f_sync(fp);
00357   }
00358
00359   if (fp->dirty != 255) {
00360     if (fp->descriptorSector != 0) { /* Si es 0 quiere decir que no tiene
     ningun descriptor */
00361       fs[fp->fsIndex]->win[0] = 0; /* Para ponerlo como invalido*/
00362       disk_write(fp->fsIndex, fs[fp->fsIndex]->win, fp->descriptorSector,
00363           1);
00364     }
00365
00366     fp->buff[0] = 255;
00367
00368     fp->buff[1] = (BYTE) (fp->startSector » 24);
00369     fp->buff[2] = (BYTE) (fp->startSector » 16);
00370     fp->buff[3] = (BYTE) (fp->startSector » 8);
00371     fp->buff[4] = (BYTE) (fp->startSector » 0);
00372
00373     fp->buff[5] = (BYTE) (fp->writePointer » 24);
00374     fp->buff[6] = (BYTE) (fp->writePointer » 16);
00375     fp->buff[7] = (BYTE) (fp->writePointer » 8);
00376     fp->buff[8] = (BYTE) (fp->writePointer » 0);
00377
00378     fp->buff[9] = fp->name[0];
00379     fp->buff[10] = fp->name[1];
00380     fp->buff[11] = fp->name[2];
00381     fp->buff[12] = fp->name[3];
00382     fp->buff[13] = fp->name[4];
00383     fp->buff[14] = fp->name[5];
00384     fp->buff[15] = fp->name[6];
00385
00386     fp->descriptorSector = fs[fp->fsIndex]->lastDescriptor;
00387
00388     disk_write(fp->fsIndex, fp->buff, fs[fp->fsIndex]->lastDescriptor++, 1); //
     lastDescriptor is the number of sector
00389
00390     fp->dirty = 255;
00391   }
00392
00393   int i;
00394   for (i = 0; i < _MAX_FILES; i++) {
00395     if (compare((TCHAR*) fp->name, (TCHAR*) fs[fp->fsIndex]->files[i].name)
00396         == 1)
```

```
00397       break;
00398     }
00399     update_file(fp, &fs[actual_fs]->files[i]);
00400
00401     return res;
00402 }
00403
00404 /* Read data from a file */
00405 FRESULT f_read(FIL* fp, /* Pointer to the file object */
00406 void* buff, /* Pointer to data buffer */
00407 UINT btr, /* Number of bytes to read */
00408 UINT* br /* Pointer to number of bytes read */
00409 ) {
00410   FRESULT result = FR_OK;
00411   BYTE *rbuff = (BYTE*) buff;
00412   if (((fp->writePointer + fp->buffindex) - fp->readPointer) < btr)
00413     btr = (fp->writePointer + fp->buffindex) - fp->readPointer;
00414   int sector = fp->startSector + (fp->readPointer / SECTOR_SIZE);
00415   int offset = fp->readPointer % SECTOR_SIZE;
00416   UINT counter = 0;
00417
00418   while (counter < btr) {
00419     disk_read(fp->fsIndex, fs[fp->fsIndex]->win, sector++, 1);
00420     while (offset < SECTOR_SIZE) {
00421       rbuff[counter++] = fs[fp->fsIndex]->win[offset++];
00422       if (counter == btr) {
00423         fp->readPointer += counter;
00424         *br = counter;
00425         break;
00426       }
00427     }
00428     offset = 0;
00429   }
00430
00431   return result;
00432 }
00433
00434 /* Write data to a file */
00435 FRESULT f_write(FIL* fp, /* Pointer to the file object */
00436 const void *buff, /* Pointer to the data to be written */
00437 UINT btw, /* Number of bytes to write */
00438 UINT* bw /* Pointer to number of bytes written */
00439 ) {
00440   FRESULT result = FR_OK;
00441   BYTE *rbuff = (BYTE*) buff;
00442   fp->dirty = 1;
00443   DWORD sector = fp->startSector + (fp->writePointer / SECTOR_SIZE);
00444   int index = 0, offset;
00445   offset = fp->writePointer % SECTOR_SIZE;
00446   /* Quiere decir que el ultimo sector escrito no esta completo.
00447    * Si buffindex no es 0 esto ya se ha comprobado antes.
00448    * Puede haber offset!=0 y buffindex!=0 cuando aun no hay suficiente para
       escribir un sector */
00449   if (offset != 0 && fp->buffindex == 0) {
00450     disk_read(fp->fsIndex, fs[fp->fsIndex]->win, sector, 1);
00451     for (; fp->buffindex < SECTOR_SIZE; fp->buffindex++) {
00452       if (fs[fp->fsIndex]->win[fp->buffindex] == 255)
00453         break;
00454       fp->buff[fp->buffindex] = fs[fp->fsIndex]->win[fp->buffindex];
00455     }
00456     fp->writePointer -= offset;
00457   }
00458
00459   while (index < btw) {
00460     fp->buff[fp->buffindex++] = rbuff[index++];
00461     if (fp->buffindex == SECTOR_SIZE) {
00462       fp->buffindex = 0;
00463       disk_write(fp->fsIndex, fp->buff, sector++, 1);
00464       fp->writePointer += 16;
00465     }
00466   }
00467
00468   *bw = btw;
00469   return result;
```

```
00470 }
00471
00472 /* Move file pointer of a file object */
00473 FRESULT f_truncateStart(FIL* fp, /* Pointer to the file object */
00474 DWORD ofs /* File pointer from top of file */
00475 ) {
00476   FRESULT result = FR_OK;
00477
00478   fp->startSector += ofs;
00479   fp->writePointer -= ofs;
00480   if (fp->writePointer < 0)
00481     fp->writePointer = 0;
00482   fp->readPointer -= ofs;
00483   if (fp->readPointer < 0)
00484     fp->readPointer = 0;
00485
00486   return result;
00487 }
00488
00489 /* Flush cached data of a writing file */
00490 FRESULT f_sync(FIL* fp /* Pointer to the file object */
00491 ) {
00492   DRESULT result;
00493   DWORD sector = fp->startSector + (fp->writePointer / SECTOR_SIZE);
00494   fp->writePointer += fp->buffindex;
00495   for (; fp->buffindex < SECTOR_SIZE; fp->buffindex++)
00496     fp->buff[fp->buffindex] = 255;
00497
00498   fp->buffindex = 0;
00499   result = disk_write(fp->fsIndex, fp->buff, sector, 1);
00500   return result;
00501 }
00502
00503 /* Get min free space for a file */
00504 FRESULT f_getfree(int fsIndex, /*Index of fileSystem */
00505 UINT* mnfs /* Pointer to a variable to return number of free sectors */
00506 ) {
00507   FRESULT result = FR_OK;
00508   int i, minFreeSectors = 0, freeSectors = 0;
00509
00510   minFreeSectors = MAX_FILE_SIZE - fs[fsIndex]->lastDescriptor;
00511
00512   for (i = 0; i < _MAX_FILES; i++) {
00513     if (fs[fsIndex]->files[i].err != -1) {
00514       /* i+2 porque se contempla el tamaño de un archivo extra para
       descriptores del FS */
00515       freeSectors =
00516           (MAX_FILE_SIZE * (i + 2))
00517               - ((fs[fsIndex]->files[i].startSector
00518                   + fs[fsIndex]->files[i].writePointer)
00519                   / SECTOR_SIZE);
00520       if (freeSectors < minFreeSectors)
00521         minFreeSectors = freeSectors;
00522     }
00523   }
00524   mnfs = (UINT*) minFreeSectors;
00525   return result;
00526 }
00527
00528 /* Mount/Unmount a logical drive */
00529 FRESULT f_mount(FS * fileSystem, int fsIndex, /*Index of fileSystem */
00530 BYTE opt /* 0:Do not mount (delayed mount), 1:Mount immediately */
00531 ) {
00532   FRESULT result = FR_OK;
00533   fs[fsIndex] = fileSystem;
00534   if (fsIndex == 0)
00535     fileSystem->start_address = PHYSYCAL_START_ADDRESS;
00536   else
00537     fileSystem->start_address = PHYSYCAL_START_ADDRESS2;
00538
00539   fileSystem->sector_size = SECTOR_SIZE;
00540   fileSystem->fs_size = FS_SIZE;
00541   fileSystem->free_sector = FS_SIZE;
00542   fileSystem->volbase = 0;
```

```
00543   fileSystem->fatbase = 2; /* Because only need 1 sector for FS info */
00544   fileSystem->database = FS_SIZE / (_MAX_FILES + 1); /* Same size for fs info
       than file data*/
00545
00546   if (opt != 1)
00547     return FR_OK;
00548
00549   result = check_fs(fsIndex);
00550   if (result != FR_OK)
00551     return FR_NO_FILESYSTEM;
00552
00553   result = loading_files(fsIndex);
00554   if (result != FR_OK)
00555     return FR_NO_FILE;
00556
00557   return FR_OK;
00558 }
00559
00560 /* Create a file system on the volume */
00561 FRESULT f_mkfs(int fsIndex/*Index of fileSystem */) {
00562
00563   FRESULT result = FR_OK;
00564   BYTE buff[SECTOR_SIZE] = { 'F', 'S', ' ', 'C', 'O', 'R', 'R', 'E', 'C', 'T',
00565       'O' };
00566
00567   fs[fsIndex]->win[0] = buff[0];
00568   fs[fsIndex]->win[1] = buff[1];
00569   fs[fsIndex]->win[2] = buff[2];
00570   fs[fsIndex]->win[3] = buff[3];
00571   fs[fsIndex]->win[4] = buff[4];
00572   fs[fsIndex]->win[5] = buff[5];
00573   fs[fsIndex]->win[6] = buff[6];
00574   fs[fsIndex]->win[7] = buff[7];
00575   fs[fsIndex]->win[8] = buff[8];
00576   fs[fsIndex]->win[9] = buff[9];
00577   fs[fsIndex]->win[10] = buff[10];
00578   fs[fsIndex]->win[11] = buff[11];
00579   fs[fsIndex]->win[12] = buff[12];
00580   fs[fsIndex]->win[13] = buff[13];
00581   fs[fsIndex]->win[14] = buff[14];
00582   fs[fsIndex]->win[15] = buff[15];
00583   fs[fsIndex]->win[16] = buff[16];
00584
00585   result = disk_initialize(fsIndex);
00586   if (result != FR_OK)
00587     return FR_NOT_READY;
00588
00589   result = disk_write(fsIndex, fs[fsIndex]->win, 0, 1);
00590   if (result != FR_OK)
00591     return FR_DISK_ERR;
00592
00593   return FR_OK;
00594 }
```

4.9. Referencia del Archivo FS/fileSystem.h

Cabecera principal del módulo del Sistema de Archivos.

`#include "integer.h"` `#include "stm32f2xx.h"` `#include "diskio.-h"` Dependencia gráfica adjunta para fileSystem.h:

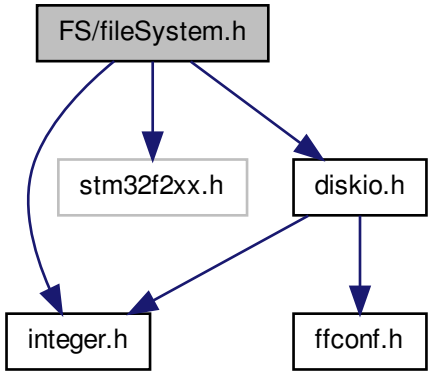
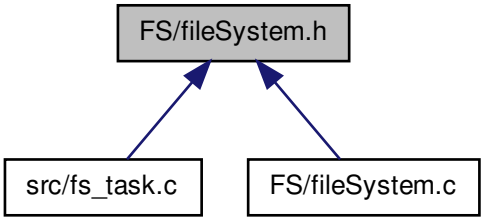


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- struct FIL

  Estructura que define un archivo.

- struct FS

  Estructura que define el Sistema de Archivos.

'defines'

- #define **FA_READ** 0x01
- #define **FA_OPEN_EXISTING** 0x00
- #define **FA_WRITE** 0x02
- #define **FA_CREATE_NEW** 0x04
- #define **FA_CREATE_ALWAYS** 0x08
- #define **FA_OPEN_ALWAYS** 0x10

Enumeraciones

- enum FRESULT { FR_OK = 0, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED, FR_EXIST, FR_INVALID_OBJECT, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_MKFS_ABORTED, FR_TIMEOUT, FR_LOCKED, FR_NOT_ENOUGH_CORE, FR_TOO_MANY_FILES, FR_INVALID_PARAMETER }

  Enumeración de los estados de operación de la memoria FLASH.

Funciones

- FRESULT f_open (FIL *fp, const TCHAR *path, BYTE mode)

  Función que abre o crea un archivo.
- FRESULT f_close (FIL *fp)

  Función que cierra un archivo.
- FRESULT f_read (FIL *fp, void *buff, UINT btr, UINT *br)

  Función que lee de un archivo.
- FRESULT f_write (FIL *fp, const void *buff, UINT btw, UINT *bw)

  Función que escribe de un archivo.
- FRESULT f_truncateStart (FIL *fp, DWORD ofs)

  Función que elimina parte de un archivo.
- FRESULT f_lseek (FIL *fp, DWORD ofs)

  Función que mueve el puntero de lectura de un archivo.
- FRESULT f_sync (FIL *fp)

  Función que sincroniza un archivo, escribe en memoria los datos que queden pendientes.
- FRESULT f_getfree (int fsIndex, UINT *nclst)

  Función que devuelve el número de sectores libres del sistema de archivos.
- FRESULT f_mount (FS *fileSystem, int fsIndex, BYTE opt)

  Función que crea la estructura necesaria para crear un sistema de archivos.
- FRESULT f_mkfs (int fsIndex)

  Función que crea un sistema de archivos.
- FRESULT reset_sector (int fsIndex)

  Función que resetea un sector.

4.9.1. Descripción detallada

Cabecera principal del módulo del Sistema de Archivos.

Definición en el archivo fileSystem.h.

4.9.2. Documentación de las enumeraciones

4.9.2.1. enum FRESULT

Enumeración de los estados de operación de la memoria FLASH.

Valores de enumeraciones:

FR_OK (0) Éxito

FR_DISK_ERR (1) Ha ocurrido un error en la capa de abstracción de E/S

FR_INT_ERR (2) Parámetros erróneos

FR_NOT_READY (3) Memoria ocupada

FR_NO_FILE (4) No se encuentra el archivo

FR_NO_PATH (5) No se encuentra la ruta

FR_INVALID_NAME (6) El nombre o la ruta es inválido

FR_DENIED (7) Acceso denegado o prohibido el acceso al directorio

FR_EXIST (8) Acceso denegado o prohibido el acceso

FR_INVALID_OBJECT (9) El archivo o directorio es inválido

FR_WRITE_PROTECTED (10) La memoria está protegida contra escrituras

FR_INVALID_DRIVE (11) El identificador del medio físco es inválido

FR_NOT_ENABLED (12) El volúmen no tiene área de trabajo

FR_NO_FILESYSTEM (13) No hay un Sistema de Archivos válido

FR_MKFS_ABORTED (14) Abortada la creación de un Sistema de Archivos debido a parámetros erróneos

FR_TIMEOUT (15) Tiempo de espera excedido

FR_LOCKED (16) La operación ha sido rechazada debido a las políticas de acceso

FR_NOT_ENOUGH_CORE (17) No hay espacio disponible para guardar

FR_TOO_MANY_FILES (18) Número de archivos máximo excedido

FR_INVALID_PARAMETER (19) El parámetro dado es inválido

Definición en la línea 22 del archivo fileSystem.h.

4.9.3. Documentación de las funciones

4.9.3.1. FRESULT f_close (FIL * *fp*)

Función que cierra un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero a cerrar

Devuelve

Código de estado de la operación

Definición en la línea 353 del archivo fileSystem.c.

4.9.3.2. FRESULT f_getfree (int *fsIndex*, UINT * *nclst*)

Función que devuelve el número de sectores libres del sistema de archivos.

Parámetros

fsIndex	Identificador del sistema de archivos
nclst	Puntero donde se devuelve el valor

Devuelve

Código de estado de la operación

Definición en la línea 504 del archivo fileSystem.c.

4.9.3.3. FRESULT f_lseek (FIL * *fp*, DWORD *ofs*)

Función que mueve el puntero de lectura de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero objetivo
ofs	Cantidad de bytes a mover desde la posición actual del puntero de lectura

Devuelve

Código de estado de la operación

Definición en la línea 266 del archivo fileSystem.c.

4.9.3.4. FRESULT f_mkfs (int *fsIndex*)

Función que crea un sistema de archivos.

Parámetros

fsIndex	Identificador del sistema de archivos a crear

Devuelve

Código de estado de la operación

Definición en la línea 561 del archivo fileSystem.c.

4.9.3.5. FRESULT f_mount (FS * *fileSystem,* int *fsIndex,* BYTE *opt*)

Función que crea la estructura necesaria para crear un sistema de archivos.

Parámetros

fileSystem	Puntero de Sistema de Archivos donde devolver el valor
fsIndex	Identificador del sistema de archivos objetivo
opt	Acepta dos opciones: 0, sólo crea la estructura necesaria para crear un sistema de archivos. 1, intenta montar un sistema de archivos existente

Devuelve

Código de estado de la operación

Definición en la línea 529 del archivo fileSystem.c.

4.9.3.6. FRESULT f_open (FIL * *fp,* const TCHAR * *path,* BYTE *mode*)

Función que abre o crea un archivo.

Parámetros

fp	Parámetro de entrada en el que se devolverá el puntero del archivo abierto
path	Nombre del archivo que queremos abrir
mode	Permisos de apertura del archivo

Devuelve

Código de estado de la operación

Definición en la línea 305 del archivo fileSystem.c.

4.9.3.7. FRESULT f_read (FIL * *fp*, void * *buff*, UINT *btr*, UINT * *br*)

Función que lee de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero de donde leer
buff	Buffer donde se devolverán los datos leídos
btr	Dirección de donde empezar a leer
br	Cantidad de bytes a leer

Devuelve

Código de estado de la operación

Definición en la línea 405 del archivo fileSystem.c.

4.9.3.8. FRESULT f_sync (FIL * *fp*)

Función que sincroniza un archivo, escribe en memoria los datos que queden pendientes.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero objetivo

Devuelve

Código de estado de la operación

Definición en la línea 490 del archivo fileSystem.c.

4.9.3.9. FRESULT f_truncateStart (FIL * *fp*, DWORD *ofs*)

Función que elimina parte de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero objetivo
ofs	Cantidad de bytes a eliminar desde la posición cero del archivo

Devuelve

Código de estado de la operación

Definición en la línea 473 del archivo fileSystem.c.

4.9.3.10. FRESULT f_write (FIL * *fp*, const void * *buff*, UINT *btw*, UINT * *bw*)

Función que escribe de un archivo.

Parámetros

fp	Parámetro de entrada en el que se especifica el fichero de donde leer
buff	Stream de datos que se quieren escribir
btw	Dirección de donde empezar a escribir
bw	Cantidad de bytes a leer

Devuelve

Código de estado de la operación

Definición en la línea 435 del archivo fileSystem.c.

4.9.3.11. FRESULT reset_sector (int *fsIndex*)

Función que resetea un sector.

Parámetros

fsIndex	Identificador del sistema de archivos sobre el que resetear sus sectores

Devuelve

Código de estado de la operación

Definición en la línea 253 del archivo fileSystem.c.

4.10. FS/fileSystem.h

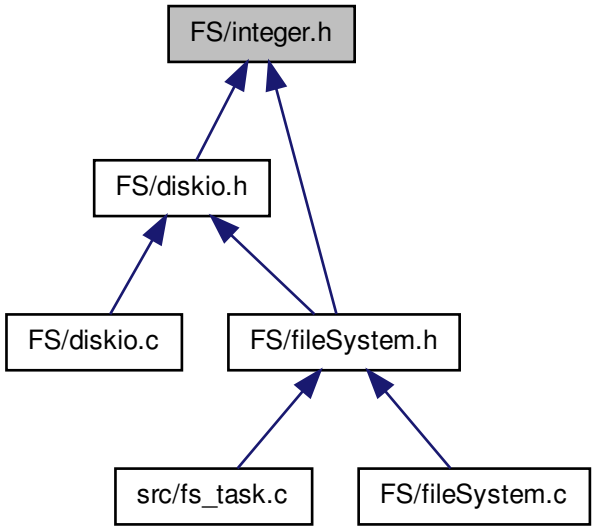
```
00001
00006 #include "integer.h"
00007 #include "stm32f2xx.h"
00008 #include "diskio.h"
00009
00010 /* FILE PERMISSIONS */
00011 #define   FA_READ      0x01
00012 #define   FA_OPEN_EXISTING  0x00
00013
00014 #define  FA_WRITE      0x02
00015 #define   FA_CREATE_NEW    0x04
00016 #define   FA_CREATE_ALWAYS  0x08
00017 #define   FA_OPEN_ALWAYS    0x10
00018
00022 typedef enum {
00023   FR_OK = 0,
00024   FR_DISK_ERR,
00025   FR_INT_ERR,
00026   FR_NOT_READY,
00027   FR_NO_FILE,
00028   FR_NO_PATH,
```

```
00029   FR_INVALID_NAME,
00030   FR_DENIED,
00031   FR_EXIST,
00032   FR_INVALID_OBJECT,
00033   FR_WRITE_PROTECTED,
00034   FR_INVALID_DRIVE,
00035   FR_NOT_ENABLED,
00036   FR_NO_FILESYSTEM,
00037   FR_MKFS_ABORTED,
00038   FR_TIMEOUT,
00039   FR_LOCKED,
00040   FR_NOT_ENOUGH_CORE,
00041   FR_TOO_MANY_FILES,
00042   FR_INVALID_PARAMETER
00043 } FRESULT;
00044 typedef struct {
00048   BYTE name[7];
00049   BYTE flag;
00050   BYTE err;
00051   BYTE dirty;
00052   BYTE fsIndex;
00053   DWORD startSector;
00054   DWORD writePointer;
00055   DWORD readPointer;
00056   DWORD descriptorSector;
00057   WORD buffindex;
00058   BYTE buff[SECTOR_SIZE];
00059 } FIL;
00060
00064 typedef struct {
00065   uint32_t start_address;
00066   BYTE sector_size;
00067   WORD free_sector;
00068   WORD fs_size;
00069   WORD volbase;
00070   WORD fatbase;
00071   WORD database;
00072   WORD lastDescriptor;
00073   FIL files[_MAX_FILES];
00074   BYTE win[SECTOR_SIZE];
00075 } FS;
00076
00084 FRESULT f_open(FIL* fp, const TCHAR* path, BYTE mode);
00085
00091 FRESULT f_close(FIL* fp); /* Close an open file object */
00092
00101 FRESULT f_read(FIL* fp, void* buff, UINT btr, UINT* br); /* Read data from a
       file */
00102
00111 FRESULT f_write(FIL* fp, const void* buff, UINT btw, UINT* bw); /* Write data
       to a file */
00112
00119 FRESULT f_truncateStart(FIL* fp, DWORD ofs); /* Move start file pointer of a
       file object */
00120
00127 FRESULT f_lseek(FIL* fp, DWORD ofs); /* Move read pointer of a file object  */
00128
00134 FRESULT f_sync(FIL* fp); /* Flush cached data of a writing file */
00135
00142 FRESULT f_getfree(int fsIndex, UINT* nclst); /* Get number of free clusters on
       the drive */
00143
00151 FRESULT f_mount(FS* fileSystem, int fsIndex, BYTE opt); /* Mount/Unmount a
       logical drive */
00152
00158 FRESULT f_mkfs(int fsIndex); /* Create a file system on the volume */
00159
00165 FRESULT reset_sector(int fsIndex);
```

4.11. Referencia del Archivo FS/integer.h

Tipos de datos abreviados.

Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



'typedefs'

- typedef unsigned char **BYTE**
- typedef short **SHORT**
- typedef unsigned short **WORD**
- typedef unsigned short **WCHAR**
- typedef int **INT**
- typedef unsigned int **UINT**
- typedef long **LONG**
- typedef unsigned long **DWORD**
- typedef char **TCHAR**

4.11.1. Descripción detallada

Tipos de datos abreviados.

Definición en el archivo integer.h.

4.12. FS/integer.h

```
00001
00006 #ifndef _FF_INTEGER
00007 #define _FF_INTEGER
00008
00009
00010 /* This type MUST be 8 bit */
00011 typedef unsigned char  BYTE;
00012
00013 /* These types MUST be 16 bit */
00014 typedef short      SHORT;
00015 typedef unsigned short  WORD;
00016 typedef unsigned short  WCHAR;
00017
00018 /* These types MUST be 16 bit or 32 bit */
00019 typedef int        INT;
00020 typedef unsigned int  UINT;
00021
00022 /* These types MUST be 32 bit */
00023 typedef long      LONG;
00024 typedef unsigned long  DWORD;
00025 typedef char TCHAR;
00026
00027 #endif
```

4.13. Referencia del Archivo src/common.h

En este archivo se definen estructuras de datos comunes para todas las tareas.

`#include "FreeRTOS.h"` `#include "queue.h"` `#include "stm32f2xx.h"` Dependencia gráfica adjunta para common.h:

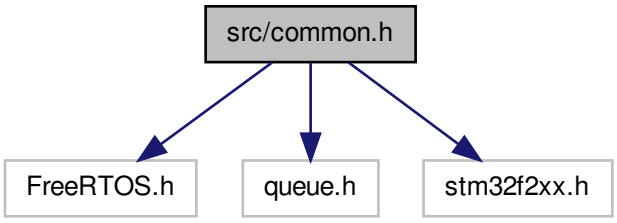
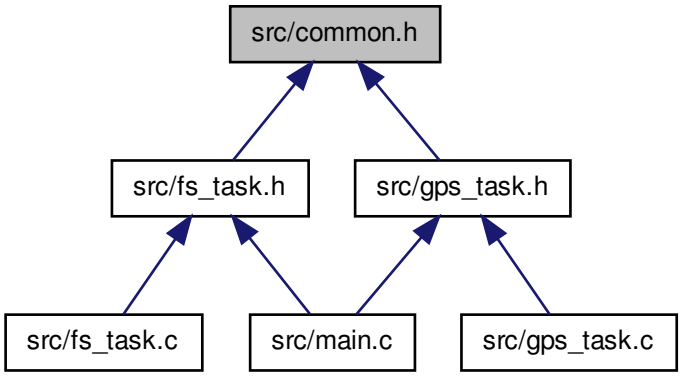


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Estructuras de datos

- struct GPS_MSG

  Estructura que define el mensaje de la cola de mensaje writeQueue.

Variables

- xQueueHandle **writeQueue**

4.13.1. Descripción detallada

En este archivo se definen estructuras de datos comunes para todas las tareas.

Definición en el archivo common.h.

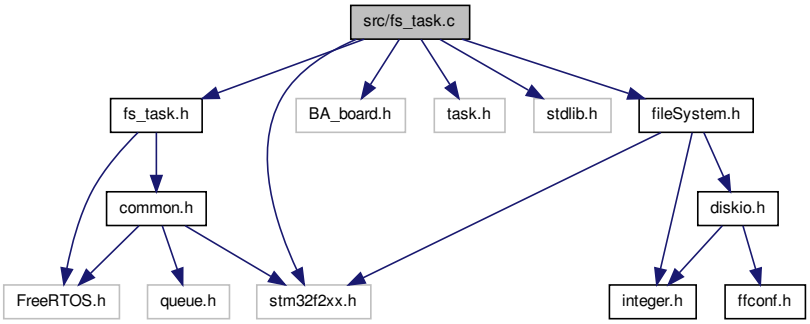
4.14. src/common.h

```
00001
00005 #include "FreeRTOS.h"
00006 #include "queue.h"
00007 #include "stm32f2xx.h"
00008
00009 xQueueHandle writeQueue;
00013 typedef struct {
00014   uint16_t count;
00015   uint8_t buffer[80];
00016 }GPS_MSG;
```

4.15. Referencia del Archivo src/fs_task.c

Módulo que implementa las funciones del archivo fs_task.h.

`#include "fs_task.h" #include "stm32f2xx.h" #include "BA-_board.h" #include "task.h" #include <stdlib.h> #include "fileSystem.h"` Dependencia gráfica adjunta para fs_task.c:



Funciones

- void FSTaskFunc (void ∗pParams)

  Esta es la función principal de la tarea Esta función inicializa la memoria y el sistema de archivos y lanza el método que se queda a la espera de mensajes desde el GPS.
- void FSHardwareInit (void ∗pParam)

  Método que inicializa el hardware de la tarea Esta función resetea un sector y crea en él un sistema de archivos.
- void FSStartTask (unsigned short nStackDepth, unsigned portBASE_TYPE n-Priority, void ∗pParams)

  Método que crea la tarea.
- void read_file ()

  Función que lee de un archivo Esta función no se utiliza en la implementación actual, pero se mantiene para poder ser utilizada cuando se integre en el proyecto Biker - Assistant.
- void ReceiveWriteGPS ()

  Función que escribe en un archivo todos los mensajes recibidos desde la tarea que controla el GPS Esta función, dentro del bucle de ejecución, enciende y apaga un led cada vez que recibe y escribe un dato.

4.15.1. Descripción detallada

Módulo que implementa las funciones del archivo fs_task.h.

Definición en el archivo fs_task.c.

4.15.2. Documentación de las funciones

4.15.2.1. void FSStartTask (unsigned short *nStackDepth*, unsigned portBASE_TYPE *nPriority*, void * *pParams*)

Método que crea la tarea.

Parámetros

sStackDepth	Tamaño de la pila de memoria
nPrioriy	Prioridad de la tarea

Definición en la línea 27 del archivo fs_task.c.

4.16. src/fs_task.c

```
00001
00005 #include "fs_task.h"
00006 #include "stm32f2xx.h"
00007 #include "BA_board.h"
00008 #include "task.h"
00009 #include <stdlib.h>
00010 #include "fileSystem.h"
00011
00012 static FS fileSystem;
00013
00018 void FSTaskFunc(void *pParams);
00019
00020 void FSHardwareInit(void *pParam) {
00021   reset_sector(0);
00022   f_mount(&fileSystem, 0, 0);
00023   f_mkfs(0);
00024   f_mount(&fileSystem, 0, 1);
00025 }
00026
00027 void FSStartTask(unsigned short nStackDepth, unsigned portBASE_TYPE nPriority,
00028     void *pParams) {
00029   xTaskCreate(FSTaskFunc, "FS", nStackDepth, pParams, nPriority, NULL);
00030 }
00031
00036 void read_file() {
00037   FIL fp;
00038   GPS_MSG msg;
00039   f_open(&fp, "ex1.txt", FA_OPEN_ALWAYS | FA_READ | FA_WRITE);
00040
00041   int i;
00042   UINT readed = 0;
00043   for (i = 0; i < 10; i++) {
00044     f_read(&fp, &msg.buffer, 37, &readed);
00045   }
00046 }
00047
00048 // FS
00049 /*
00050 void test_FS() {
00051   FIL fp;
00052
00053   f_open(&fp, "ex1.txt", FA_OPEN_ALWAYS | FA_READ | FA_WRITE);
00054
00055   Delay(500);
```

```
00056   GPIO_ResetBits(LEDS_GPIO_PORT, LEDR_PIN);
00057   Delay(500);
00058   GPIO_SetBits(LEDS_GPIO_PORT, LEDR_PIN);
00059
00060   uint8_t toWrite[100];
00061   uint8_t toRead[100];
00062   int i;
00063   UINT writed, readed;
00064   for (i = 0; i < 100; i++) {
00065     toWrite[i] = '2';
00066     toRead[i] = '1';
00067   }
00068
00069   Delay(500);
00070   GPIO_ResetBits(LEDS_GPIO_PORT, LEDR_PIN);
00071   Delay(500);
00072   GPIO_SetBits(LEDS_GPIO_PORT, LEDR_PIN);
00073
00074   for (i = 0; i < 100; i++) {
00075     f_write(&fp, toWrite, 100, &writed);
00076   }
00077
00078   Delay(500);
00079   GPIO_ResetBits(LEDS_GPIO_PORT, LEDR_PIN);
00080   Delay(500);
00081   GPIO_SetBits(LEDS_GPIO_PORT, LEDR_PIN);
00082
00083   f_close(&fp);
00084
00085   Delay(500);
00086   GPIO_ResetBits(LEDS_GPIO_PORT, LEDR_PIN);
00087   Delay(500);
00088   GPIO_SetBits(LEDS_GPIO_PORT, LEDR_PIN);
00089
00090   f_open(&fp, "ex1.txt", FA_OPEN_ALWAYS | FA_READ | FA_WRITE);
00091
00092   for (i = 0; i < 100; i++) {
00093     f_read(&fp, toRead, 100, &readed);
00094   }
00095
00096   Delay(500);
00097   GPIO_ResetBits(LEDS_GPIO_PORT, LEDR_PIN);
00098   Delay(500);
00099   GPIO_SetBits(LEDS_GPIO_PORT, LEDR_PIN);
00100
00101   f_close(&fp);
00102
00103   Delay(500);
00104   GPIO_ResetBits(LEDS_GPIO_PORT, LEDR_PIN);
00105   Delay(500);
00106   GPIO_SetBits(LEDS_GPIO_PORT, LEDR_PIN);
00107 }
00108 */
00109
00114 void ReceiveWriteGPS(){
00115   int count =10;
00116   UINT writed;
00117   GPS_MSG msg;
00118   FIL fp;
00119   f_open(&fp, "ex1.txt", FA_OPEN_ALWAYS | FA_READ | FA_WRITE);
00120
00121   while (count >0){
00122     Delay(500);
00123     GPIO_SetBits(LEDS_GPIO_PORT, LEDR_PIN);
00124
00125     count--;
00126     xQueueReceive(writeQueue, &msg, 10000);
00127     f_write(&fp, msg.buffer, msg.count, &writed);
00128
00129     Delay(500);
00130     GPIO_ResetBits(LEDS_GPIO_PORT, LEDR_PIN);
00131   }
00132   f_close(&fp);
00133 }
```

```
00134
00135
00136 void FSTaskFunc(void *pParams) {
00137
00138   FSHardwareInit(pParams);
00139   Delay(2000);
00140
00141   while (1) {
00142     //test_FS();
00143     ReceiveWriteGPS();
00144     //read_file();
00145   }
00146
00147 }
```

4.17. Referencia del Archivo src/fs_task.h

Cabecera de la tarea que gestiona el sistema de archivos.

`#include "FreeRTOS.h"` `#include "common.h"` Dependencia gráfica adjunta para fs_task.h:

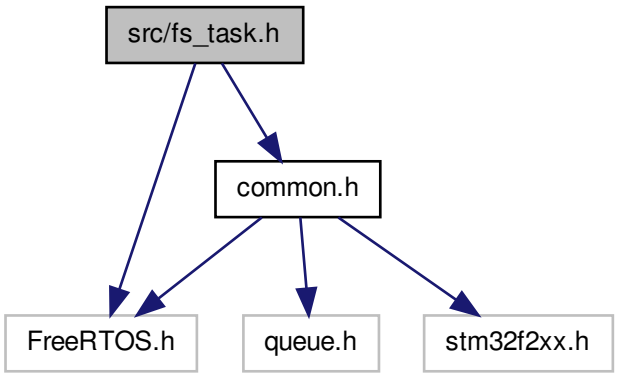
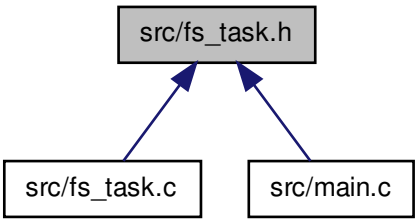


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Funciones

- void FSHardwareInit (void ∗pParam)

  Método que inicializa el hardware de la tarea Esta función resetea un sector y crea en él un sistema de archivos.
- void FSStartTask (unsigned short nStackDepth, unsigned portBASE_TYPE nPriority, void ∗pParams)

  Método que crea la tarea.

4.17.1. Descripción detallada

Cabecera de la tarea que gestiona el sistema de archivos.

Definición en el archivo fs_task.h.

4.17.2. Documentación de las funciones

4.17.2.1. void FSStartTask (unsigned short *nStackDepth,* unsigned portBASE_TYPE *nPriority,* void ∗ *pParams*)

Método que crea la tarea.

Parámetros

sStackDepth	Tamaño de la pila de memoria
nPrioriy	Prioridad de la tarea

Definición en la línea 27 del archivo fs_task.c.

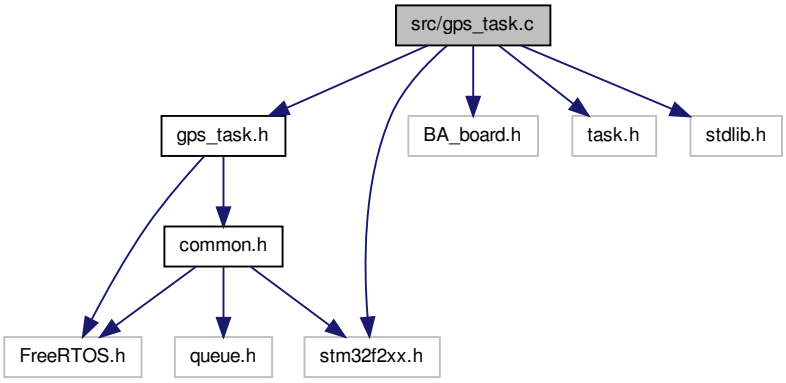
4.18. src/fs_task.h

```
00001
00005 #ifndef CONSOLE_TASK_H_
00006 #define CONSOLE_TASK_H_
00007
00008 #include "FreeRTOS.h"
00009 #include "common.h"
00010
00015 void FSHardwareInit(void *pParam);
00021 void FSStartTask(unsigned short nStackDepth, unsigned portBASE_TYPE nPriority,
      void *pParams);
00022
00023 #endif /* CONSOLE_TASK_H_ */
```

4.19. Referencia del Archivo src/gps_task.c

Módulo que implementa las funciones del archivo gps_task.h.

`#include "gps_task.h" #include "stm32f2xx.h" #include "BA-_board.h" #include "task.h" #include <stdlib.h>` Dependencia gráfica adjunta para gps_task.c:



Funciones

- void GPSTaskFunc (void *pParams)

  Esta es la función principal de la tarea Esta función inicializa el GPS y lanza el método que lee datos del GPS y se los manda a la tarea que gestiona el Sistema de Archivos.
- void setupGPS ()

  Esta es la función que inicializa el GPS.
- void GPSHardwareInit (void *pParam)

  Esta es la función que inicializa todo el hardware que necesita la tarea.

- void GPSStartTask (unsigned short nStackDepth, unsigned portBASE_TYPE nPriority, void ∗pParams)

  Método que crea la tarea.
- void parser_GPS ()

  Esta es la función que lee datos del GPS y, cuando detecta un mensaje correcto, lo envía a la tarea que gestiona el Sistema de Archivos Esta tarea enciende un led cada vez que envía correctamente un mensaje.

4.19.1. Descripción detallada

Módulo que implementa las funciones del archivo gps_task.h.

Definición en el archivo gps_task.c.

4.19.2. Documentación de las funciones

4.19.2.1. void GPSHardwareInit (void ∗ *pParam*)

Esta es la función que inicializa todo el hardware que necesita la tarea.

Método que inicializa el hardware de la tarea.

Definición en la línea 39 del archivo gps_task.c.

4.19.2.2. void GPSStartTask (unsigned short *nStackDepth*, unsigned portBASE_TYPE *nPriority*, void ∗ *pParams*)

Método que crea la tarea.

Parámetros

sStackDepth	Tamaño de la pila de memoria
nPrioriy	Prioridad de la tarea

Definición en la línea 43 del archivo gps_task.c.

4.20. src/gps_task.c

```
00001
00005 #include "gps_task.h"
00006 #include "stm32f2xx.h"
00007 #include "BA_board.h"
00008 #include "task.h"
00009 #include <stdlib.h>
00010
00011 //#define COMn 4
00012 //USART_TypeDef* COM_USART[COMn] = {DBG, BT, GSM, GPS};
00017 void GPSTaskFunc(void *pParams);
00018
00022 void setupGPS() {
00023   USART_InitTypeDef USART_InitStructure;
```

```
00024
00025   USART_InitStructure.USART_BaudRate = 9600;
00026   USART_InitStructure.USART_WordLength = USART_WordLength_8b;
00027   USART_InitStructure.USART_StopBits = USART_StopBits_1;
00028   USART_InitStructure.USART_Parity = USART_Parity_No;
00029   USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
00030   USART_InitStructure.USART_HardwareFlowControl =
00031   USART_HardwareFlowControl_None;
00032
00033   BA_GPSInit(&USART_InitStructure);
00034   Delay(1000);
00035 }
00039 void GPSHardwareInit(void *pParam) {
00040   setupGPS();
00041 }
00042
00043 void GPSStartTask(unsigned short nStackDepth, unsigned portBASE_TYPE nPriority,
00044     void *pParams) {
00045   xTaskCreate(GPSTaskFunc, "GPS", nStackDepth, pParams, nPriority, NULL);
00046 }
00047
00052 void parser_GPS() {
00053   uint8_t answer[80];
00054   uint16_t parser_idx = 0;
00055   uint8_t c;
00056   int state = 0;  // 0-5 -> searching for $GPGGA header,  1-> getting data
00057   GPS_MSG msg;
00058   int i;
00059   // Parse for $GPGGA statements
00060   // $GPGGA,064951.000,2307.1256,N,12016.4438,E,1,8,0.95,39.9,M,17.8,M,,*65\r
00061
00062   while (1) {
00063     GPIO_ResetBits(LEDS_GPIO_PORT, LEDV_PIN);
00064     // BA_DBGSend((uint8_t*)"\r\n New GPGGA reading ... \r\n");
00065     while (state < 7) {
00066       BA_GPSReceive(&c, 1);
00067
00068       switch (state) {
00069       case 0:
00070         if (c == '$') {
00071           answer[parser_idx] = c;
00072           state = 1;
00073           parser_idx++;
00074         } else {
00075           parser_idx = 0;
00076           state = 0;
00077         }
00078         break;
00079
00080       case 1:
00081         if (c == 'G') {
00082           answer[parser_idx] = c;
00083           state = 2;
00084           parser_idx++;
00085         } else {
00086           parser_idx = 0;
00087           state = 0;
00088         }
00089         break;
00090
00091       case 2:
00092         if (c == 'P') {
00093           answer[parser_idx] = c;
00094           state = 3;
00095           parser_idx++;
00096         } else {
00097           parser_idx = 0;
00098           state = 0;
00099         }
00100         break;
00101
00102       case 3:
00103         if (c == 'G') {
00104           answer[parser_idx] = c;
```

```
00105           state = 4;
00106           parser_idx++;
00107         } else {
00108           parser_idx = 0;
00109           state = 0;
00110         }
00111         break;
00112
00113       case 4:
00114         if (c == 'G') {
00115           answer[parser_idx] = c;
00116           state = 5;
00117           parser_idx++;
00118         } else {
00119           parser_idx = 0;
00120           state = 0;
00121         }
00122         break;
00123
00124       case 5:
00125         if (c == 'A') {
00126           answer[parser_idx] = c;
00127           state = 6;
00128           parser_idx++;
00129         } else {
00130           parser_idx = 0;
00131           state = 0;
00132         }
00133         break;
00134
00135       case 6: // get the rest of the message
00136         while (c != '\r') {
00137           BA_GPSReceive(&c, 1);
00138           answer[parser_idx] = c;
00139           parser_idx++;
00140         }
00141         //BA_GPSReceive(&(answer[parser_idx]), 60);
00142         state = 7;
00143         break;
00144
00145       default:
00146         // get the rest of the message
00147         break;
00148       }
00149
00150     }
00151
00152     answer[parser_idx] = '\n';
00153     for (i=0;i<80;i++){
00154       msg.buffer[i]=answer[i];
00155     }
00156     //msg.buffer=(uint8_t[80])answer;
00157     msg.count=parser_idx;
00158
00159     xQueueSend(writeQueue, &msg, 1000);
00160     GPIO_SetBits(LEDS_GPIO_PORT, LEDV_PIN);
00161
00162     Delay(1000);
00163
00164     state = 0;
00165     parser_idx = 0;
00166   }
00167 }
00168
00169 void GPSTaskFunc(void *pParams) {
00170
00171   GPSHardwareInit(pParams);
00172   Delay(2000);
00173
00174   while (1) {
00175     parser_GPS();
00176   }
00177
00178 }
```

4.21. Referencia del Archivo src/gps_task.h

Cabecera de la tarea que gestiona el GPS.

`#include "FreeRTOS.h"` `#include "common.h"` Dependencia gráfica adjunta para gps_task.h:

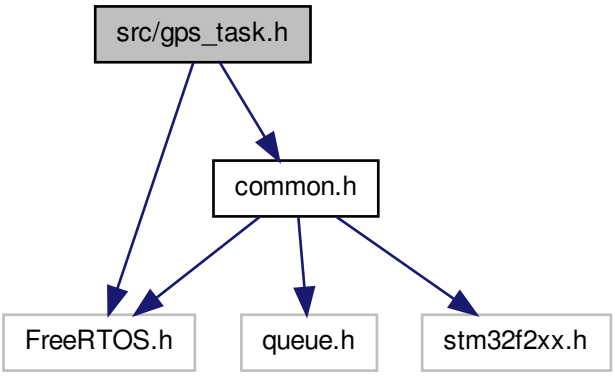
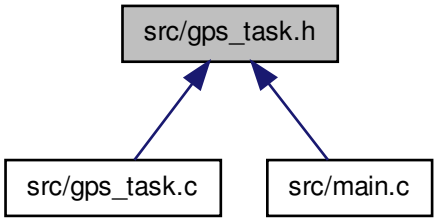


Gráfico de los archivos que directa o indirectamente incluyen a este archivo:



Funciones

- void GPSHardwareInit (void *pParam)

  Método que inicializa el hardware de la tarea.

- void GPSStartTask (unsigned short nStackDepth, unsigned portBASE_TYPE nPriority, void ∗pParams)

  Método que crea la tarea.

4.21.1. Descripción detallada

Cabecera de la tarea que gestiona el GPS.

Definición en el archivo gps_task.h.

4.21.2. Documentación de las funciones

4.21.2.1. void GPSStartTask (unsigned short *nStackDepth,* unsigned portBASE_TYPE *nPriority,* void ∗ *pParams*)

Método que crea la tarea.

Parámetros

sStackDepth	Tamaño de la pila de memoria
nPrioriy	Prioridad de la tarea

Definición en la línea 43 del archivo gps_task.c.

4.22. src/gps_task.h

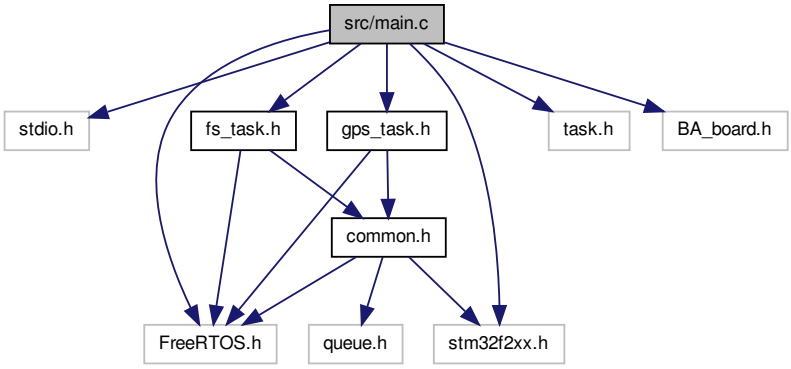
```
00001
00006 #ifndef CONSOLE_TASK_H_
00007 #define CONSOLE_TASK_H_
00008
00009 #include "FreeRTOS.h"
00010 #include "common.h"
00011
00015 void GPSHardwareInit(void *pParam);
00021 void GPSStartTask(unsigned short nStackDepth, unsigned portBASE_TYPE nPriority,
       void *pParams);
00022
00023 #endif /* CONSOLE_TASK_H_ */
```

4.23. Referencia del Archivo src/main.c

Función principal.

#include <stdio.h> #include "FreeRTOS.h" #include "task.h" #include "stm32f2xx.h" #include "BA_board.h" #include "gps_task.h" #include "fs_task.h" Dependencia gráfica adjunta para

main.c:



Funciones

- void prvSetupHardware (void)

  Función que inicializa el hardware.
- int putChar (int ch)
- void setupTasks ()

  Función que crea las tareas del Sistemas Operativo FreeRTOS.
- void setupQueues ()

  Función que crea las colas de mensajes entre tareas del Sistema Operativo FreeRTOS.
- int main (void)

  Función principal del sistema.
- void vApplicationStackOverflowHook (xTaskHandle ∗pxTask, signed portCHAR ∗pcTaskName)

  Función que crea la excepción de desbordamiento de la pila de memoria.
- void vApplicationTickHook (void)

  Función que crea la excepción del reloj.
- void Delay (uint32_t t)

  Función que define la función de espera.

4.23.1. Descripción detallada

Función principal.

Definición en el archivo main.c.

4.23.2. Documentación de las funciones

4.23.2.1. void Delay (uint32_t *t*)

Función que define la función de espera.

Parámetros

t	Tiempo a esperar en milisegundos

Definición en la línea 121 del archivo main.c.

4.23.2.2. int main (void)

Función principal del sistema.

Nota

Esta función inicializa el hardware, crea las colas de mensajes, crea las tareas e inicializa el planificador del Sistema Operativo

Definición en la línea 54 del archivo main.c.

4.23.2.3. void prvSetupHardware (void)

Función que inicializa el hardware.

Nota

Esta función inicializa el vector de interrupciones, configura los niveles de priporidad, establece la fuente del reloj del sistema e inicializa los leds

Definición en la línea 72 del archivo main.c.

4.23.2.4. int putChar (int *ch*)

External dependence needed by printf implementation. Write a character to standard out.

Parámetros

c	Specifies the character to be written.

Devuelve

Returns the character written. No error conditions are managed.

Definición en la línea 112 del archivo main.c.

4.23.2.5. void vApplicationStackOverflowHook (xTaskHandle * *pxTask*, signed portCHAR * *pcTaskName*)

Función que crea la excepción de desbordamiento de la pila de memoria.

Nota

Esta función la llama internamente el Sistema Operativo

Definición en la línea 90 del archivo main.c.

4.23.2.6. void vApplicationTickHook (void)

Función que crea la excepción del reloj.

Nota

Esta función la llama internamente el Sistema Operativo

Definición en la línea 106 del archivo main.c.

4.24. src/main.c

```
00001
00006 /* Standard includes. */
00007 #include <stdio.h>
00008
00009 /* Scheduler includes. */
00010 #include "FreeRTOS.h"
00011 #include "task.h"
00012
00013 /* Library includes. */
00014 #include "stm32f2xx.h"
00015 #include "BA_board.h"
00016
00017 /* Demo app includes. */
00018 #include "gps_task.h"
00019 #include "fs_task.h"
00020
00025 void prvSetupHardware( void );
00026
00033 int putChar( int ch );
00034
00038 void setupTasks() {
00039   GPSStartTask(configMINIMAL_STACK_SIZE*4, 1, NULL);
00040   FSStartTask(configMINIMAL_STACK_SIZE*4, 1, NULL);
00041 }
00045 void setupQueues(){
00046   writeQueue = xQueueCreate(4, sizeof(GPS_MSG));
00047 }
00048
00049 /*-----------------------------------------------------------*/
00054 int main( void )
00055 {
00056   prvSetupHardware();
00057   setupQueues();
00058   setupTasks();
00059
00060
00061     /* Start the scheduler. */
```

```
00062   vTaskStartScheduler();
00063
00064     /* Will only get here if there was insufficient memory to create the idle
00065     task.  The idle task is created within vTaskStartScheduler(). */
00066   for( ;; );
00067
00068   return 0;
00069 }
00070 /*-----------------------------------------------------------*/
00071
00072 void prvSetupHardware( void )
00073 {
00074   /* Set the Vector Table base address at 0x08000000 */
00075   NVIC_SetVectorTable( NVIC_VectTab_FLASH, 0x0 );
00076
00077   NVIC_PriorityGroupConfig( NVIC_PriorityGroup_4 );
00078
00079   /* Configure HCLK clock as SysTick clock source. */
00080   SysTick_CLKSourceConfig( SysTick_CLKSource_HCLK );
00081   BA_LEDsInit();
00082
00083 }
00084
00085 /*-----------------------------------------------------------*/
00090 void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *
      pcTaskName )
00091 {
00092   /* This function will get called if a task overflows its stack.   If the
00093   parameters are corrupt then inspect pxCurrentTCB to find which was the
00094   offending task. */
00095
00096   ( void ) pxTask;
00097   ( void ) pcTaskName;
00098
00099   for( ;; );
00100 }
00101 /*-----------------------------------------------------------*/
00106 void vApplicationTickHook( void )
00107 {
00108 }
00109
00110 /*-----------------------------------------------------------*/
00111
00112 int putChar(int ch)
00113 {
00114   return ch;
00115 }
00116
00121 void Delay(uint32_t t) {
00122   vTaskDelay(t / portTICK_RATE_MS);
00123 }
```