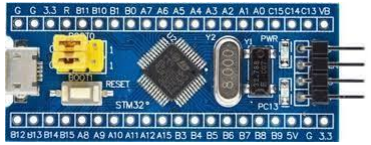
Bus SPI
Sistema de archivos FAT
Memorias MicroSD

Motivación: ¿Qué queremos gestionar?

Memorias SD o microSD desde nuestro embebido

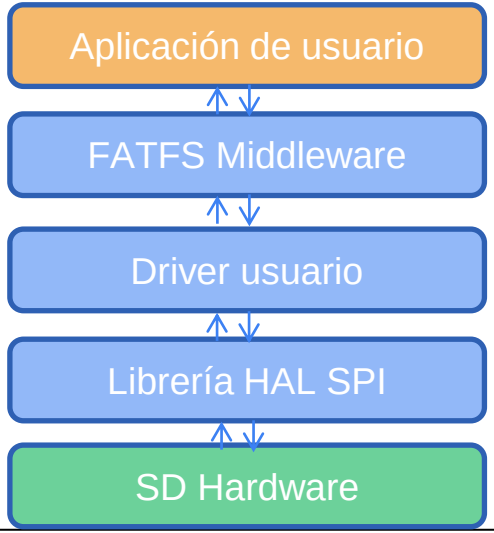


- No volátil
- Gran capacidad
- Fácil acceso y económicas
- Interfaz SPI
- Sistema de archivos FAT
- Escritura en bloques

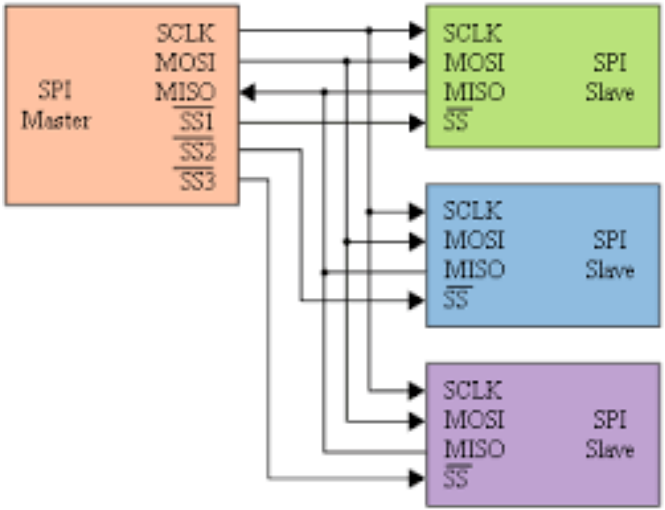
Contexto actual para su gestión

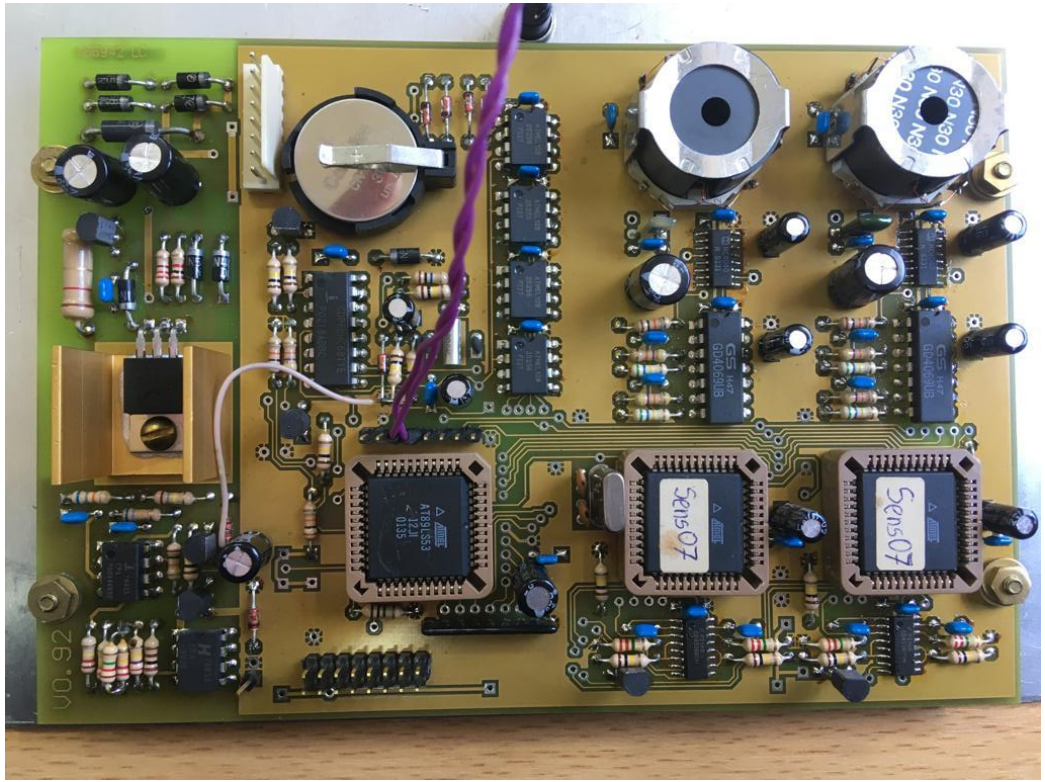
- Existe un proyecto open source denominado FATFS desarrollado por ChanN que está siendo portado a distintas plataformas.
- En STM32 está disponible como firmware en una librería.
- En algunos casos se apoya en el periférico SDIO en otros en SPI.

Arquitectura de software

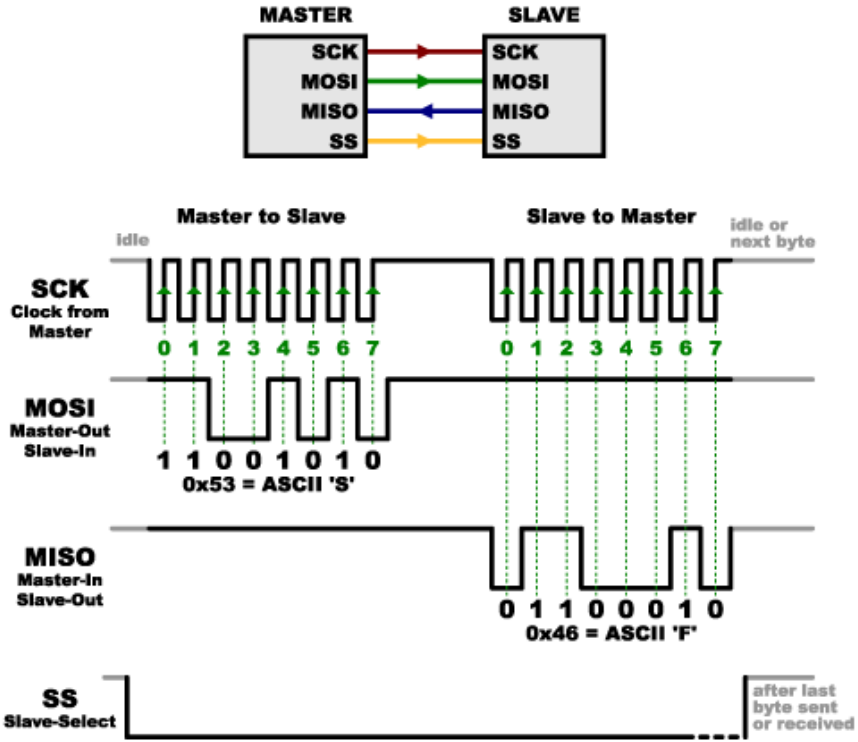


SPI: Serial Peripheral Interface





SPI: Serial Peripheral Interface



Sistema de archivos FAT:

- El significado de la sigla FAT es tabla de ubicación de archivos.
- El sector es el segmento de datos más pequeño del sistema de archivos.
- Cada sector puede almacenar 512 bytes de datos y no puede ser direccionado directamente.
- Debe ser accedido como parte de un cluster (conjunto de sectores).
- El número máximo de clusters que puede manejar el sistema FAT 16 es 65535 y para poder manejar discos más grandes debe incrementarse el tamaño de los clusters.
- El tamaño de los clusters depende del tamaño del disco. Por ejemplo, un disco de 512 MB utiliza clusters de 8KB, uno de 850 MB utiliza clusters de 16 KB, y uno de 1.2 GB utiliza clusters de 32 KB.
- Como podemos ver en el ejemplo, con un disco de 1.2 GB el tamaño de los cluster es tan grande que el sistema de archivos se hace ineficiente, especialmente si tenemos muchos archivos pequeños. No importa lo chico que sea el archivo, éste ocupará un cluster. Por ejemplo, si almacenamos un archivo de 1KB en un disco de 1.2 GB, éste pequeño archivo consumirá 32 KB de memoria. Los 31 KB restantes estarán vacíos. Debido a la ineficiencia del sistema FAT 16 para discos grandes, se establece un máximo en 2 GB.

FAT32 mejora en la eficiencia:

Ahora que tenemos un entendimiento básico de la forma en la que opera FAT y algunos de sus problemas, seremos capaces de comprender las ventajas que ofrece FAT 32. Para comenzar es importante remarcar que FAT 32 está basado en el sistema FAT original y opera en forma similar para mantener compatibilidad. La mejora más significativa en FAT 32 es la habilidad para manejar eficientemente los espacios de almacenamiento en los discos más grandes.

Tamaño de clusters más pequeño:

Para mejorar la eficiencia en el almacenamiento, el sistema FAT 32 utiliza clusters de 4 KB para todos los discos por debajo de 8 GB. Esto reduce el espacio muerto en el disco cuando se almacenan archivos muy pequeños. Por ejemplo, en un disco de 1GB usando el sistema FAT antiguo, un archivo de 1 KB ocupa 32 KB, como lo mencionamos anteriormente. De la misma forma, un archivo de 1KB en el mismo disco usando FAT 32 ocupa solo 4 KB de espacio, ahorrando 28 KB. Esto puede parecer trivial, pero cuando tratamos con un disco entero que posee miles de archivos, el ahorro es importante.

Cómo se guarda un archivo en FAT

- Ejemplo de disco de 8 GB, formateado en FAT32.
- La estructura de un disco en FAT 32, consta de un directorio principal de tamaño fijo.
- Dos tablas llamadas FAT (File Access Table). Cada tabla de 960000 elementos cada uno de 4 Bytes (FAT 32). Cada elemento de esta tabla referencia a un '*cluster*' (agrupación de sectores) del disco.
- Cada *cluster* es de 32768 bytes (es decir, de 64 sectores de 512 bytes).
- Hay unos elementos reservados al principio de dicha tabla para referencias a los *clusters* ocupados por el directorio raíz del disco.
- Se define que si un elemento de la tabla contiene en binario una determinada marca (el hexadecimal 0x00000000 en FAT32), esto indica al sistema operativo que ese *cluster* está libre.

Veamos ahora como se guarda un archivo:

1) El sistema operativo calcula cuantos *clusters* va a ocupar. Para ello, divide el tamaño del archivo con el tamaño del *cluster*, y al dato obtenido le redondea a la unidad superior. Supongamos 90KB de archivo con clusters de 32768 B -> ocupará 3 clusters.

2) Una vez calculado el número de *clusters* a ocupar, el sistema operativo, lee la FAT, buscando un *cluster* libre. Es decir, lee cada elemento de la FAT hasta que encuentra la marca de cluster libre citada anteriormente. Imaginemos que lo encuentra en el elemento 537 de la FAT. Esto le indica que el *cluster* 537 del disco está libre.

3) En ese *cluster* graba los primeros 32768 bytes del archivo, y marca el elemento 537 de la FAT con un cero.

4) Como le queda todavía datos a grabar, vuelve a leer la FAT para localizar otro elemento con la marca libre. Imaginemos que es el elemento numero 612. Por tanto, le indica que el *cluster* 612 del disco está libre. Ahora va a ese *cluster*, y graba los siguiente 32768 bytes del fichero. Igualmente ahora, va a la FAT, y en 612, graba un cero. Hasta aquí, todo es igual que cuando ha grabado el primer *cluster* del archivo.

Pero en este caso, además de lo anterior, el sistema 'recuerda' cual es el elemento de la FAT ultimo grabado(el 537), y en ese elemento, le pone ahora el número 612 (del *cluster* actual).

5) Queda todavía por grabar un *cluster*. Bien, volvemos a repetir los cálculos: se vuelve a leer la FAT para localizar otro *cluster* libre. Imaginemos que es el elemento 1020.

6) En ese *cluster* graba por fin los últimos 32768 bytes del fichero, y marca el elemento 1020 de la FAT con un cero. Ahora va al elemento anterior (el 612) y allí graba el número 1020.

7) Por último, guarda en el directorio raíz del disco el nombre de archivo, y además allí se guarda la fecha, el tamaño, y lo que es más importante: el número del primer elemento de la FAT que apunta al archivo guardado, es decir: 537.

Bien, después de toda esta historia, veamos como está ahora nuestro archivo en disco:

· A nivel del directorio principal, tenemos su nombre, y un número mágico: 537. Esto indica que el archivo empieza en el *cluster* 537. Igualmente, el elemento de la FAT 537, contiene el segundo número mágico: 612, y así sucesivamente hasta el cluster final.

· Lo anterior, nos debe hacer meditar en un tema muy importante: Cualquier cosa que hagamos para leer o grabar un archivo, en principio implica una sucesión de lecturas en el disco. Además como la FAT está al principio del disco, esto implicará que la lectura tiene que dar muchos saltos entre la FAT y el cluster de datos a leer para ir trayéndose poco a poco los datos a memoria.

Zonas de la FAT.

Los dispositivos de almacenamiento tienen su espacio de almacenamiento dividido en cuatro zonas:

- *Sector de Booteo:* contiene los datos de la estructura del disco.
- *Sector FAT:* contiene un mapa de los archivos contenidos en el disco.
- *Directorio Raíz:* contiene información de cada archivo contenido en el disco.
- *Zona de datos:* aloja los datos de cada archivo.

Sector de Booteo

Está ubicado siempre en el primer sector de la unidad. En la siguiente tabla se muestran los diferentes campos que componen el sector de booteo para FAT 16.

Sección	Offset	Tamaño	Descripción
Código	0000h	3 bytes	Código de salto al inicio del código de booteo.
Nombre S.O	0003h	8 bytes	Nombre del Sistema Operativo.
Bloque de Param. BIOS.	000Bh	2 bytes	Bytes por sector.
	000Dh	1 byte	Sectores por cluster. Normalmente 512 bytes por sector.
	000Eh	2 bytes	Sectores reservados desde el inicio del dispositivo.
	0010h	1 byte	Cantidad de copias de la FAT. Normalmente se usan 2 copias de la FAT.
	0011h	2 bytes	Número de entrada en el directorio raíz. Generalmente 512.
	0013h	2 bytes	Número de cluster pequeño. Se usa si el tamaño del disco es menor a 32 MB.
	0015h	1 byte	Descriptor del dispositivo.
	0016h	2 bytes	Sectores por FAT.
	0018h	2 bytes	Sectores por Track.
	001Ah	2 bytes	Número de Cabezas.
	001Ch	4 bytes	Sectores Ocultos.
	0020h	4 bytes	Cantidad de sectores. Si el disco es mayor de 32 MB.
Bloque de Parámetros Extendidos de la BIOS.	0024h	1 byte	Número de disco.
	0025h	1 byte	Reservado.
	0026h	1 byte	Marca del sector de booteo extendido.
	0027h	4 bytes	Número de serie del disco.
	002Bh	11 bytes	Rótulo del disco.
	0036h	8 bytes	Tipo de FAT. En este caso FAT 16.
Código	003Eh	448 Bytes	Código de Booteo.
Marca	01FEh	2 Bytes	Fin de sector de Booteo. Los dos últimos bytes deben ser ‘AA55’

Ejemplo: Contenido del sector de booteo en una MMC de 32 MB.

Sección	Offset	Tamaño	Descripción
Código	0000h	3 bytes	0xE9,0x00,0x00
Nombre S.O	0003h	8 bytes	“Pablo 01” (8 bytes ASCII)
Bloque de Param. BIOS.	000Bh	2 bytes	0x00;0x02 (512 little endian)
	000Dh	1 byte	0x04(4 sectores de 512 bytes, o sea cada cluster 2048 bytes)
	000Eh	2 bytes	0x01;0x00 (1)
	0010h	1 byte	0x02 (2) La MMC usa 2 copias de la FAT. FAT 1 y FAT 2.
	0011h	2 bytes	0x00;0x02 (512).
	0013h	2 bytes	0xE0;0xF4 (62688 sectores disponibles para datos)
	0015h	1 byte	0xF8
	0016h	2 bytes	0x3E;0x00
	0018h	2 bytes	0x20;0x00
	001Ah	2 bytes	0x04;0x00
	001Ch	4 bytes	0x20;0x00;0x00;0x00
	0020h	4 bytes	0x00;0x00;0x00;0x00
Bloque de Parámetros Extendidos de la BIOS.	0024h	1 byte	0x00
	0025h	1 byte	0x00
	0026h	1 byte	0x00
	0027h	4 bytes	4*0x00
	002Bh	11 bytes	11*0x00
	0036h	8 bytes	“FAT 16 ”(8 bytes ASCII)
Código	003Eh	448 Bytes	448*0x00
Marca	01FEh	2 Bytes	0x55;0xAA

La FAT

Por lo general, en cada disco se tienen dos copias idénticas y consecutivas de la FAT, la FAT 1 y la FAT 2. Para FAT16 la FAT 1 comienza en la posición de memoria 512 y termina en la 32225 (62 sectores), y la FAT 2 comienza en la posición de memoria 32256 y finaliza en la posición 63339 (otros 62 sectores).
La FAT funciona de la siguiente manera. En ella se realiza una lista con los clusters que ocupa cada archivo en el mapa de datos. Los dos primeros datos, que representan a los dos primeros clusters del disco, son fijos y valen 0xF8FF; 0xFFFF. Estos aparecen como reservados. El primer cluster de datos válidos es el número 2. El inicio de archivo se indica simplemente anotando el número de cluster de inicio del archivo, y la finalización del archivo se indica con la marca 0xFFFF. Por ejemplo, si tenemos un archivo que ocupa tres clusters y comienza en el cluster 2 en la FAT tendremos:

Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
0xF8FF	0xFFFF	0x0300	0x0400	0xFFFF	0x0000	0x0000

La codificación de la FAT es little endian, los últimos bits son los más significativos

Directorio Raíz

Para FAT16 el directorio raíz comienza en la posición de memoria 64000 y termina en la 80383. Contiene información acerca del nombre de los archivos, fecha de creación, tamaño en bytes de los mismos, etc. En nuestra aplicación, por cada archivo utilizamos 32 bytes de información en el directorio raíz según la siguiente tabla (32 bytes DOS 8.3):

Offset	Tamaño	Descripción
00h	8 bytes	Nombre del archivo. 8 caracteres ASCII en Mayúsculas.
08h	3 bytes	Extensión del archivo. 3 caracteres ASCII en mayúsculas.
0Bh	1 bytes	Atributo del archivo.
0Ch	1 bytes	Reservado para Windows NT (0x00h)
0Dh	1 bytes	Tiempo de creación en ms.
0Eh	2 bytes	Hora de creación del archivo
10h	2 bytes	Fecha de creación.
12h	2 bytes	Fecha de último acceso al archivo.
14h	2 bytes	Reservado para FAT 32.(0x0000h)
16h	2 bytes	Hora de última esritura)
18h	2 bytes	Fecha de última escritura.
1Ah	2 bytes	Cluster de inicio.
1Ch	4 bytes	Tamaño en bytes del archivo. (Little endian)

Zona de Datos

Para FAT16 comienza en la posición 80384 de la memoria. Es una sucesión continua de datos.

Nota: Para aprender bien la estructura del sistema de archivos de la memoria, se recomienda formatearla en la PC, copiar un archivo cualquiera que ocupe más de un cluster y luego inspeccionar el sector de booteo, la FAT 1 y 2, el directorio raíz y la zona de datos utilizando el programa WinHex.

Hagamos el análisis para una memoria micro SD de 8GB formateada en FAT32 con clusters de 32KB conteniendo un único archivo de texto de nombre “ArchivoA.txt” de 85KB.

Sector booteo usando winhex

Sistema operativo: MSDOS5.0

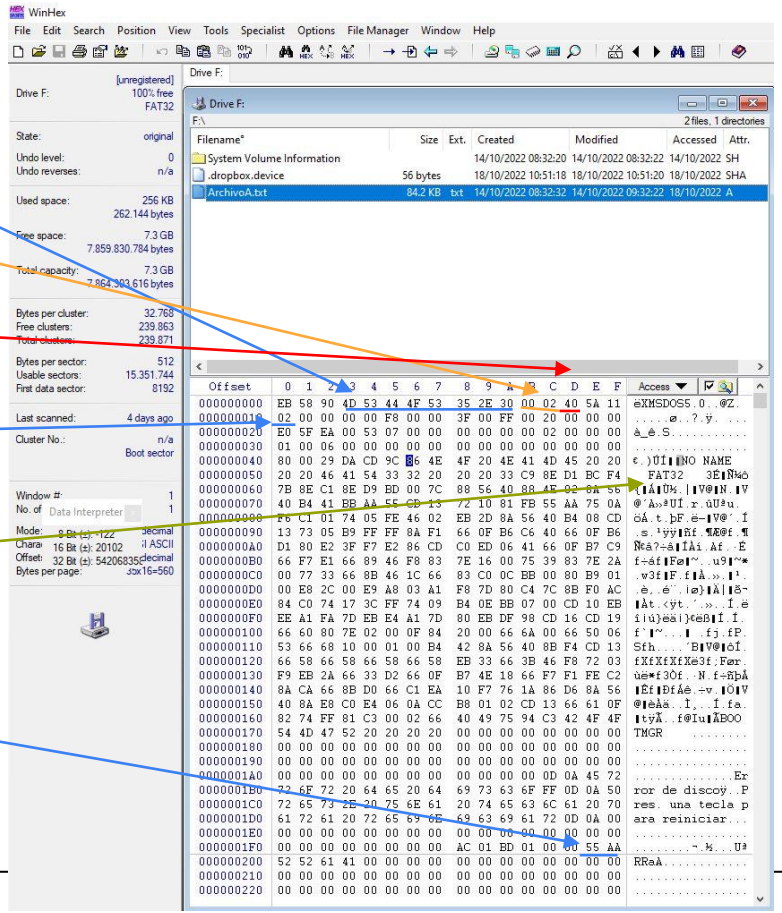
Bytes por sector: 512 =0x0200

Sectores por cluster: 64=0x40

Numero de FATs: 2

Sistema FAT: FAT32

Fin de sector de booteo: 0x55AA



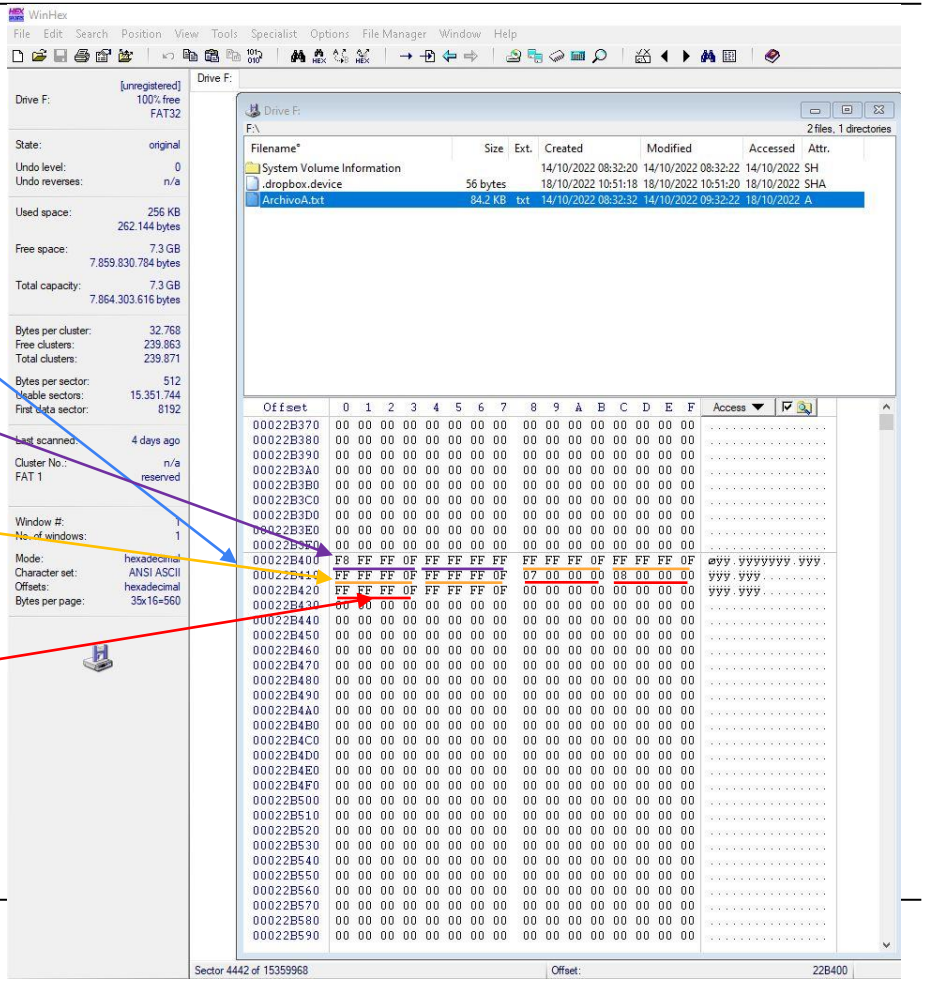
FAT1

Arranca en sector 4442

Cluster 0 y 1: reservados

Cluster 2: Root directory
Cluster 3: system volumen info
Cluster 4: WP settings
Cluster 5: indexer volumen grid

ArchivoA.txt:
Cluster 6: tiene el 7
Cluster 7: tiene el 8
Cluster 8: tiene el fin



FAT2: arranca en sector 6317 (copia de FAT1)

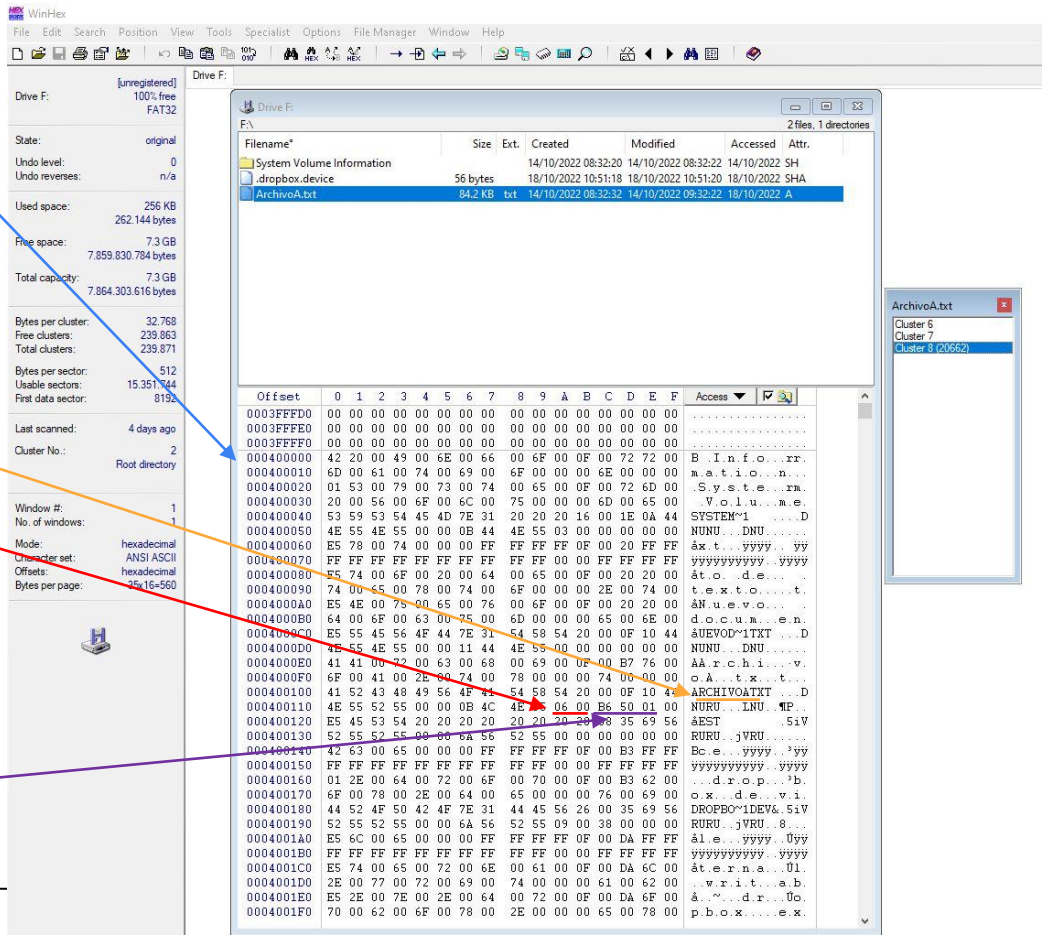
Directorio raíz

A diferencia de FAT16 arranca en cluster 2 (sector 8192)

ArchivoA.txt

Cluster de inicio: 6= 0x0600

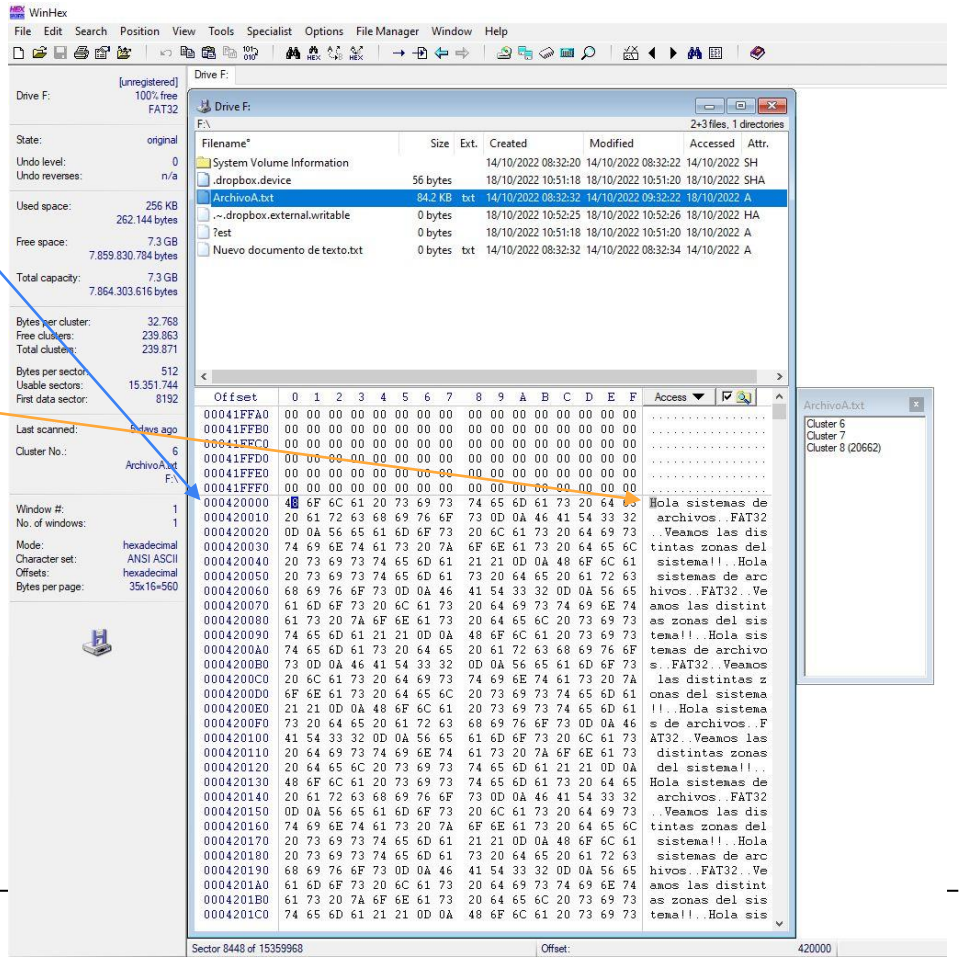
Tamaño de archivo:86198 B=0x0150B6



Zona de datos

Arranca en sector 8448 (cluster 6)

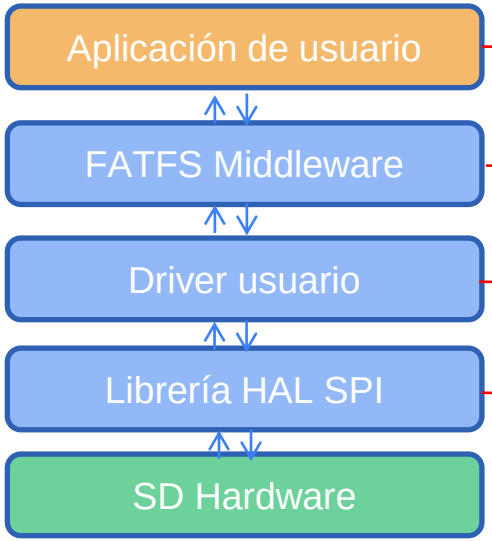
ArchivoA.txt



Consigna integradora

- Haciendo uso de una micro SD formateada en FAT32 vamos a generar un proyecto que nos permita gestionar la misma desde nuestro embebido.
- Como primer ejercicio deberemos leer con nuestro embebido un archivo de texto existente en la memoria que ha sido generado desde la PC.
- Como segundo ejercicio deberemos generar un nuevo archivo de texto en la memoria haciendo uso de nuestro embebido.
- Haciendo uso de un terminal y el puerto serie implementaremos la interfaz de usuario que nos permitirá visualizar el archivo leído, la información de la memoria y generar contenido para el archivo a escribir.

¿Por donde arrancamos?.....ubicándonos!



Acá escribiremos toda la funcionalidad que la consigna nos demanda

En STM32 está disponible como firmware en una librería middleware (proyecto open source FATFS de ChanN)

Es un adaptador que basado en el SPI resuelve el conjunto de funciones que requiere el FATFS para funcionar (código disponible por kiwih)

Deberemos configurar un SPI para manejar la memoria

Seguimos….generando un nuevo proyecto

1) Arrancamos desde el asistente CubeMX con el clock por defecto en 8MHz.
2) Incluimos un SPI configurado en modo “Full-Duplex Master” con hardware NSS signal disabled. El data size en 8 bits y el prescaler en 32 (la SD arranca en low speed 250Kbits/s, luego cambiamos a high speed).
3) Agregamos un pin cualquiera para el chip select (PB1) y le ponemos un user label SD_CS.
4) Agregamos un puerto serie a la velocidad de 115200 bps.
5) Vamos a la parte de middleware para habilitar FATFS en el modo “user-defined”
6) Generamos el código desde CubeMx y realizamos el conexionado (atención que el zócalo SD se alimenta con 5V)

Adaptamos el driver

1) Descargamos los archivos de kiwih de nombre user_diskio_spi.c y user_diskio_spi.h y los agregamos a nuestro proyecto.
2) Enlazamos el driver al FATFS por medio de los siguientes cambios en el archivo user_diskio.c

```
/* USER CODE BEGIN DECL */ /* Includes ------------------------------------------------------
----------*/
#include <string.h>
#include "ff_gen_drv.h"
#include "user_diskio_spi.h"
```

```
/* Private functions ---------------------------------------------------------*/
/**
@brief Initializes a Drive
@param pdrv: Physical drive number (0..) *
@retval DSTATUS: Operation status */
 DSTATUS USER_initialize ( BYTE pdrv )
 {
return  USER_SPI_initialize(pdrv); //ADD THIS LINE
 }
```

Adaptamos el driver

```
DSTATUS USER_status ( BYTE pdrv )
{
        return USER_SPI_status(pdrv); //ADD THIS LINE
}
```

Hacemos lo mismo para las funciones:

DRESULT **USER_read** (BYTE pdrv, BYTE ***buff**, DWORD sector, UINT count)....

DRESULT **USER_write** (BYTE pdrv, **const** BYTE ***buff**, DWORD sector, UINT count).....

DRESULT **USER_ioctl** (BYTE pdrv, BYTE cmd, **void** ***buff**).....

Adaptamos el driver

En la parte superior de user_diskio_spi.c

```
extern SPI_HandleTypeDef SD_SPI_HANDLE;
```

En la zona private defines de main.h

```
/* USER CODE BEGIN Private defines */
#define SD_SPI_HANDLE hspi2
/* USER CODE END Private defines */
```

```
#define FCLK_SLOW() { MODIFY_REG(SD_SPI_HANDLE.Instance->CR1, SPI_BAUDRATEPRESCALER_256, SPI_BAUDRATEPRESCALER_32); }
 /* Set SCLK = slow, approx 280 KBits/s*/

 #define FCLK_FAST() { MODIFY_REG(SD_SPI_HANDLE.Instance->CR1, SPI_BAUDRATEPRESCALER_256, SPI_BAUDRATEPRESCALER_2); }
 /* Set SCLK = fast, approx 4.5 MBits/s */
```

Probamos el driver desde main: 1. Montamos el disco

Código de referencia con funciones útiles:

```
//variables for FatFs
 FATFS FatFs;            //Fatfs handle
 FIL fil;                //File handle
 FRESULT fres; //Result after operations

 //Open the file system
 fres = f_mount(&FatFs, "", 1); //1=mount now
 if (fres != FR_OK) {
                myprintf("f_mount error (%i)\r\n", fres);
                while(1);
 }
```

Probamos el driver desde main: 2. Inspeccionamos la SD

Código de referencia con funciones útiles:

```
//Let's get some statistics from the SD card
 DWORD free_clusters, free_sectors, total_sectors;

 FATFS* getFreeFs;

 fres = f_getfree("", &free_clusters, &getFreeFs);
 if (fres != FR_OK) {
                myprintf("f_getfree error (%i)\r\n", fres);
                while(1);
 }

 //Formula comes from ChaN's documentation
 total_sectors = (getFreeFs->n_fatent - 2) * getFreeFs->csize;
 free_sectors = free_clusters * getFreeFs->csize;
```

Probamos el driver desde main: 3. Abrimos y leemos archivo

Código de referencia con funciones útiles:

```
//Now let's try to open file "test.txt"
  fres = f_open(&fil, "test.txt", FA_READ);
  if (fres != FR_OK) {
                myprintf("f_open error (%i)\r\n");
                while(1);
  }
  myprintf("I was able to open 'test.txt' for reading!\r\n");

  //Read 30 bytes from "test.txt" on the SD card
  BYTE readBuf[30];

  //We can either use f_read OR f_gets to get data out of files
  //f_gets is a wrapper on f_read that does some string formatting for us
  TCHAR* rres = f_gets((TCHAR*)readBuf, 30, &fil);
```

Probamos el driver desde main: 4. Creamos y escribimos archivo

```
//Now let's try and write a file "write.txt"
  fres = f_open(&fil, "write.txt", FA_WRITE | FA_OPEN_ALWAYS | FA_CREATE_ALWAYS);
  if(fres == FR_OK) {
                myprintf("I was able to open 'write.txt' for writing\r\n");
  } else {
                myprintf("f_open error (%i)\r\n", fres);
  }

  //Copy in a string
  strncpy((char*)readBuf, "a new file is made!", 19);
  UINT bytesWrote;
  fres = f_write(&fil, readBuf, 19, &bytesWrote);
  if(fres == FR_OK) {
                myprintf("Wrote %i bytes to 'write.txt'!\r\n", bytesWrote);
  } else {
                myprintf("f_write error (%i)\r\n");
  }

 f_close(&fil);
```

Verificación final

Usando el programa WinHex explorar las distintas zonas de la unidad analizando la FAT, el contenido de los dos archivos en memoria y el root directory con la información disponible de los archivos