

An abstract graphic design featuring three concentric blue circles of varying sizes. The largest circle is in the top right, a medium one in the middle right, and a large one in the bottom right. Thin blue lines intersect these circles, creating a geometric pattern. The background is white.

Système de fichiers sur Carte SD

[Tapez le sous-titre du document]

Adrien BON, Valentin DORISON,
William KLEIN, Romain LIEVRE

**Electronique option SE
2011-2012**

Introduction

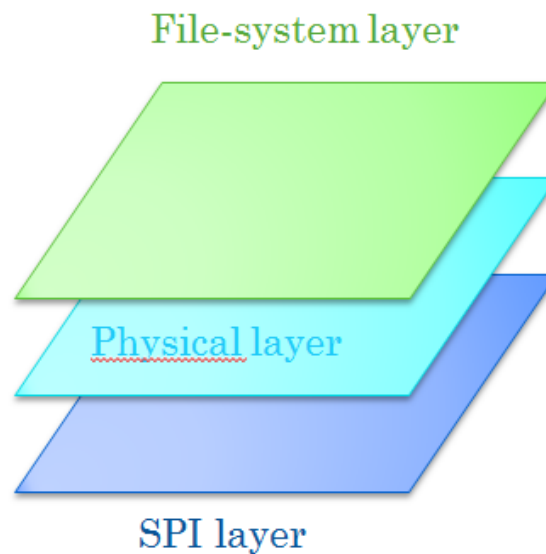
Le but du projet est d'implémenter une API pour gérer un système de fichier sur une carte SD.

Il faudra que notre API gère les systèmes de fichiers suivants : FAT12, FAT16 et FAT32. A ces systèmes de fichiers, on ajoute le VFAT, qui gère les noms de fichiers longs. Ceci en raison du fait que le formatage d'une carte SD par windows (msdos5.0) ou linux (mkdosfs) donne un système de fichier en VFAT.

Au début du projet, nous avons découpé ce projet en 3 couches :

- La couche bas-niveau qui contrôle le SPI
- La couche médiane qui s'occupe de la communication avec la carte
- La couche haut-niveau qui s'occupe de gérer le système de fichier qui est présent à l'intérieur de la carte.

Ce découpage, qui a aussi donné lieu à une explication CLAIRE des fonctions de chaque couche, a permis en fin de projet de rassembler les trois couches immédiatement et sans problème de collisions de données.



Le support de ce projet est un AT91SAM7X, donc un microcontrôleur 32bits avec architecture ARM. On le programme en JTAG => possibilité de debug avec points d'arrêts.

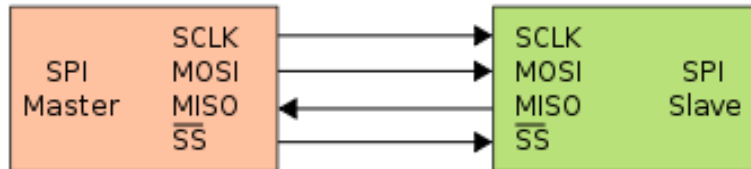
L'autre objectif du projet est d'évaluer la place que va prendre notre programme et de voir s'il est adaptable sur d'autres microcontrôleurs ayant moins de capacité FLASH.

Table des matières

I- SPI	4
a) Les Init du SPI	4
b) Les fonctions du SPI	4
II- La carte SD	6
a) Présentation de la carte	6
b) Les commandes et les réponses	6
1- Les commandes	6
2- Les réponses	7
c) Les fonctions	8
1- Initialisation	8
2- Lecture	10
3- Ecriture	11
III- Structure du système de fichier FAT12/16	13
a) Structure physique de la mémoire	13
b) Structure du système de fichier	13
1- Secteur de boot	14
2- Table de FAT	15
3- Mémoire de masse	16
c) Nom de fichier long	17
Structure d'un header de nom de fichier long	17
IV- Implémentation du système de fichiers	18
a) La structure de fichier	19
b) Les structures de carte SD et de dossier	19
c) Les fonctions	20
1- Create_file	20
2- List_next_file	20
3- List_files	21
4- Open_file	21
d) Précisions	22
CONCLUSION :	23

I- SPI

Le SPI (Serial Peripheral Interface) est un bus de données série synchrone qui opère en full-duplex. Lors d'une communication SPI, il y a forcément un maître et un esclave. Il peut y avoir plusieurs esclaves connectés au même maître, la sélection se faisant grâce au fil chip select, reliant physiquement le maître et l'esclave. La figure ci-dessous résume le SPI :



Ainsi, lors d'une écriture et d'une lecture, il est indispensable que le maître donne la clock à l'esclave, puisque celle-ci est dirigée du maître vers l'esclave.

Abordons maintenant le SPI sur l'AT91SAM7X :

a) Les Init du SPI

Pour pouvoir utiliser le SPI sur notre carte, il y a plusieurs choses à voir :

- Mettre l'alim et la clock sur le SPI, par l'intermédiaire du PMC
- « Disable » l'utilisation des pins du SPI pour des IO normal, avec le PIO
- « Enable » le SPI dans le bloc SPI de l'AT91
- Mettre l'AT91 en master mode dans ce même bloc
- Pour la phase et la polarité, cela dépend de la carte SD. En effet, l'acquisition du niveau logique par la carte et le chargement des données sur front montant ou descendant dépend de celle-ci. C'est à l'AT91 de s'adapter à la logique de la carte SD. En pratique, la phase est à 0, ainsi que la polarité, pour la plupart des cartes SD.
- Pour la clock, on l'a mise à 400Khz (voir initialisation de la carte SD), soit 120 dans le registre 8 bits (avec une clock CPU de 50Mhz).

b) Les fonctions du SPI

Pour les fonctions bas niveaux, il s'agit juste d'écrire les fonctions qui vont utiliser le SPI pour envoyer/lire des données.

Les données que le SPI aura à traiter sont soit sur 16bits, soit sur 8bits. Il est néanmoins obligatoire d'envoyer à la carte des octets. Ainsi, les envois/réceptions de 16bits seront décomposés en deux envois/réceptions de 8bits. Donc le SPI est mis de base en mode 8bits.

Pour écrire un octet :

- On regarde si le registre d'envoi est vide, en checkant le bit correspondant dans le SPI StatusRegister. On écrit nos 8bits dans le registre d'envoi TDR.
- Après ceci, il faut attendre que le registre de réception RDR soit rempli (car la donnée émise est shiftée jusqu'à ce registre après envoi), de même par check dans le SPI SR. Enfin, on lit RDR pour dumper la valeur qu'il y a dedans.

Pour lire un octet :

- Tout d'abord, on écrit 0xFF dans le registre d'envoi TDR de manière à envoyer la clock à la carte (0xFF car MOSI est tiré à Vcc, il faut donc que la carte ne croit pas qu'on lui envoie des données, on écrit donc que des 1 logiques).
- Il suffit maintenant d'attendre que le registre de réception soit plein et on lit la donnée.

De manière générale, il est à noter que l'on désactive les interruptions dans le « StatusRegister » avant de lire ou d'écrire un octet. Ceci pour éviter de se rater dans la récupération/l'écriture des données.

Pour « assert/deassert » de la carte SD :

Pour parler à la carte SD, il faut assigner le bon chip select du SPI (qui en possède 4) à celui qui est physiquement relié à la carte SD, soit le 1 (NPCS1 sur le schéma de la carte).

Ainsi, lorsque l'on veut parler à la carte (« assert_cs »), on met NPCS1 à 0 (car tiré à Vcc en temps normal), et lorsque l'on veut l'ignorer, on le remet à 1 (ceci n'est utilisé qu'au reset de la carteSD pour passer en mode SPI, sinon en temps normal on parle toujours à la carte).

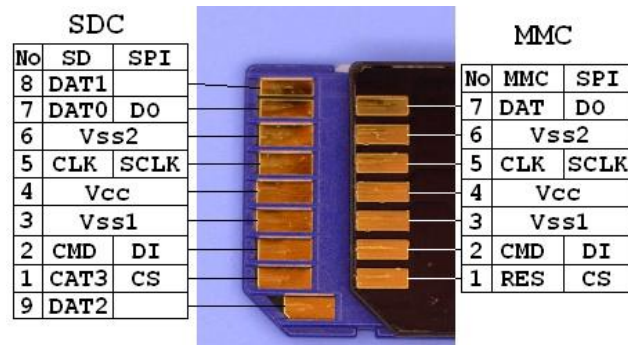
Nous avons fini d'explicitier l'étage le plus bas, c'est-à-dire le SPI. Maintenant, la prochaine couche traitera de l'utilisation du SPI pour parler à la carte SD.

II- La carte SD

a) Présentation de la carte

La carte en elle-même est constituée de la partie stockage à proprement parlé, mais il y a aussi un microcontrôleur à l'intérieur de celle-ci. Ce microcontrôleur gère la partie mémoire de la carte. Donc afin de communiquer avec cette carte, il faut définir des protocoles de communication. Il existe deux modes de communication :

- le mode SDC (Secure Digital Memory Card)
- le mode MMC (Multi Media Card).



Description des IO de la carte SD pour les deux modes de communication

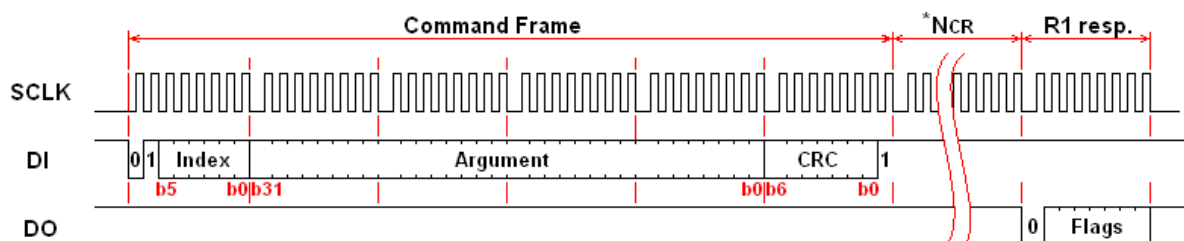
Dans notre cas, nous allons utiliser la mode MMC qui fait appel au SPI présenté dans la partie précédente.

La communication avec la carte via le mode SPI se fait en utilisant des commandes particulières.

b) Les commandes et les réponses

1- Les commandes

Le mode SDC fait lui aussi appel à des commandes, mais il est à noter que les commandes ne sont pas du même type suivant le mode de communication privilégié. Le format de la commande est donné ci-dessous :



Trame d'une commande

La trame est constituée de 6 octets.

Premier octet :

- 2 bits de start « 01 »
- 6 bits correspondant au numéro de la commande

Les octets 2 à 5, sont les octets qui correspondent à l'argument de la commande si celle-ci en possède.

Dernier octet :

- 7 bits de CRC
- 1 bit de stop qui est à « 1 »

Il existe plusieurs commande que la carte peut interpréter afin de faire certaines choses comme l'initialisation de la carte ou encore l'écriture et la lecture.

Dans le cadre du projet, nous avons utilisé que les quelques commandes de base.

Command Index	Argument	Response	Data	Abbreviation	Description
CMD0	None (0)	R1	No	GO_IDLE_STATE	Software reset.
CMD1	None (0)	R1	No	SEND_OP_COND	Initiate initialization process.
ACMD41(*1)	*2	R1	No	APP_SEND_OP_COND	For only SDC. Initiate initialization process.
CMD8	*3	R7	No	SEND_IF_COND	For only SDC V2. Check voltage range.
CMD9	None (0)	R1	Yes	SEND_CSD	Read CSD register.
CMD10	None (0)	R1	Yes	SEND_CID	Read CID register.
CMD12	None (0)	R1b	No	STOP_TRANSMISSION	Stop to read data.
CMD16	Block length[31:0]	R1	No	SET_BLOCKLEN	Change R/W block size.
CMD17	Address[31:0]	R1	Yes	READ_SINGLE_BLOCK	Read a block.
CMD18	Address[31:0]	R1	Yes	READ_MULTIPLE_BLOCK	Read multiple blocks.
CMD23	Number of blocks[15:0]	R1	No	SET_BLOCK_COUNT	For only MMC. Define number of blocks to transfer with next multi-block read/write command.
ACMD23(*1)	Number of blocks[22:0]	R1	No	SET_WR_BLOCK_ERASE_COUNT	For only SDC. Define number of blocks to pre-erase with next multi-block write command.
CMD24	Address[31:0]	R1	Yes	WRITE_BLOCK	Write a block.
CMD25	Address[31:0]	R1	Yes	WRITE_MULTIPLE_BLOCK	Write multiple blocks.
CMD55(*1)	None (0)	R1	No	APP_CMD	Leading command of ACMD<n> command.
CMD58	None (0)	R3	No	READ_OCR	Read OCR.

*1:ACMD<n> means a command sequence of CMD55-CMD<n>.

*2: Rsv(0)[31], HCS[30], Rsv(0)[29:0]

*3: Rsv(0)[31:12], Supply Voltage(1)[11:8], Check Pattern(0xAA)[7:0]

Tableau recensant les principales commandes

Les commandes entourées en rouge sont les commandes que nous avons utilisées pendant le projet, on note donc qu'il y a encore quelques chemins à explorer (optimisation de la lecture/écriture avec les commandes multiple block read/ multiple block write).

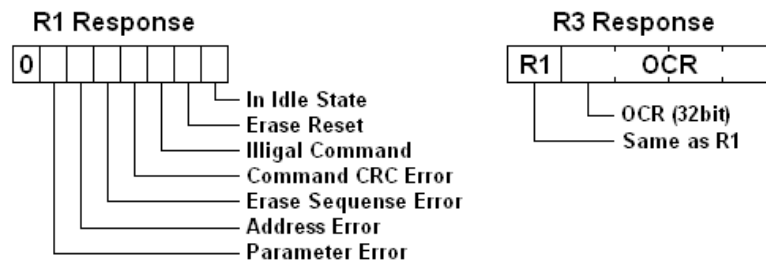
Comme on le voit sur la figure ci-dessus, chaque commande possède un numéro, ce numéro est la valeur à donner dans le champ « index » lors de l'envoi d'une trame.

On peut aussi voir qu'il y a des commandes du type ACMD<n>. Pour ce genre de commande, le principe est le même que les commandes de types CMD<n>, mais il faut au préalable envoyer la commande CMD55 pour informer la Carte SD que la commande est de type ACMD.

2- Les réponses

Comme pour une fonction, l'envoi de commande induit une réponse du microcontrôleur présent dans la carte. Cette réponse est différente en fonction de la

commande, il est donc important de savoir qu'elle est le type de réponse attendu en fonction de la commande envoyée.



Format de la réponse de la carte SD en SPI

Il y a trois types de réponse à un envoi de commande, les réponses de type R1, les réponses de type R2, et les types de réponse R3.

Il en existe une quatrième qui est R1b. Elle est du même type que R1 mais informe le contrôleur de la carte que la tâche est effectuée. Cette tâche s'effectue avec un « busy flag ».

Ces réponses permettent au contrôleur de pouvoir récupérer différentes informations sur l'état de la carte.

c) Les fonctions

1- Initialisation

L'initialisation de la carte en mode SPI s'effectue de la façon suivante :

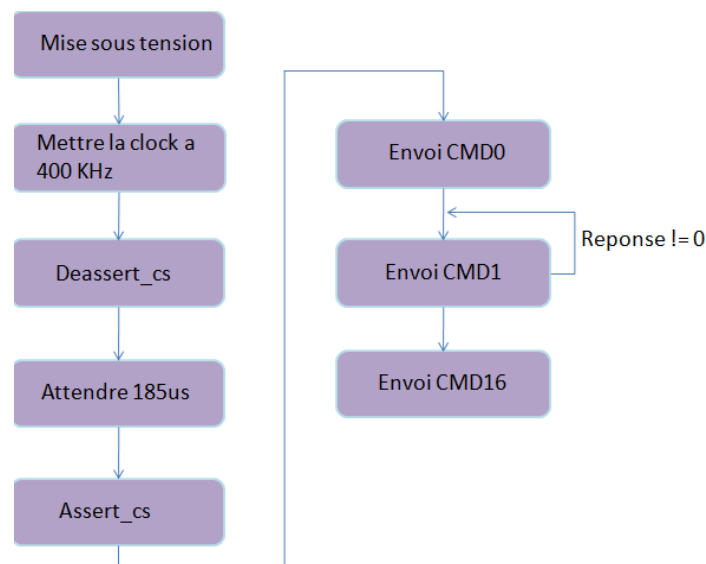


Diagramme d'état de l'initialisation

Ce diagramme est le diagramme qui est actuellement implémenté dans le code. Il est à noter que pour l'envoi des commandes, nous avons fait une fonction « send_frame » dont le prototype est le suivant :

```
uint8_t send_frame(uint8_t cmd , uint8_t cmd_R, uint32_t argument, uint8_t* response)
```


Cette fonction prend en paramètre le type de commande, le type de retour de la commande, l'argument à mettre dans la commande ainsi qu'une réponse. Elle permet d'envoyer la commande avec le bon formatage ainsi que les bonnes temporisations permettant à la carte de faire ce qu'elle a à faire. C'est en outre dans cette fonction qu'il y a l'astuce d'envoyer un octet sans aucune information afin de donner la « clock » à la carte.

Cette astuce nous a permis de pouvoir parler à la carte. Cependant, cette astuce n'est pas donnée dans l'application note. De plus comme on peut le voir sur le diagramme d'initialisation suivant (celui de l'application note) :

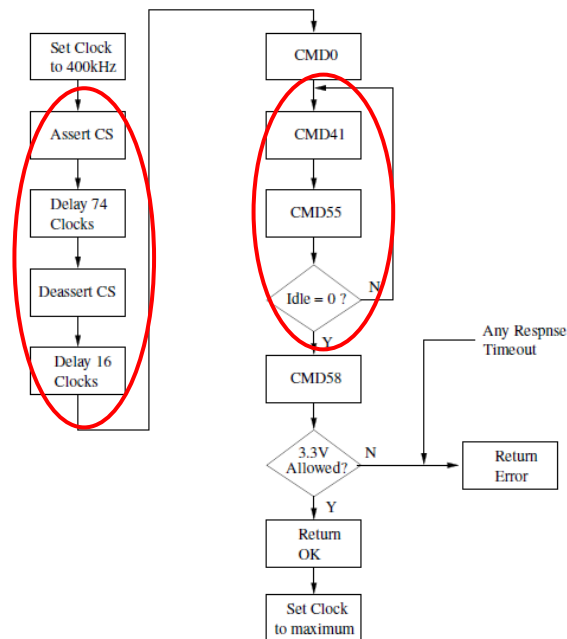


Diagramme d'initialisation de l'application note

Dans un premier temps, on peut voir que les « assert_cs() » et « deassert_cs() » ne sont pas dans le bon sens. Les temporisations ont cependant le bon ordre de grandeur.

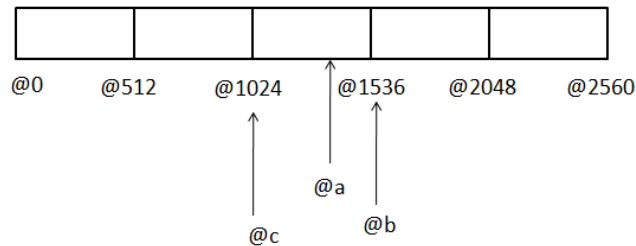
Par la suite il est conseillé d'envoyer la commande CMD41 puis la commande CMD55, or comme on l'a vu précédemment il faut envoyer dans un premier temps la commande CMD55 puis la commande CMD41 afin que la carte comprenne que la commande est bien la commande ACMD41.

A la place d'envoyer cette commande qui est spécifique au mode SDC (donc pas le notre), nous émettons la commande CMD1 qui est spécifique au mode MMC.

Dans la suite l'application note nous conseille d'envoyer la commande CMD58. Cette commande lit le registre OCR de la carte dans lequel il y a les tensions d'alimentations supportées par la carte. Nous ne faisons pas cette vérification mais il serait possible de la rajouter dans l'initialisation de la carte SD.

Dans notre initialisation, nous envoyons la commande CMD16 qui nous permet de définir la taille des blocs auxquels nous accéderons.

Attention, le fait de définir la taille des blocs n'est pas anodin, en effet lors d'une écriture ou d'une lecture il est seulement possible de placer une adresse qui soit un multiple de la taille du bloc.



Représentation de la mémoire dans la Carte SD pour des blocs de 512

Si on veut accéder à l'adresse 1400 directement, ce n'est pas possible. L'accès est seulement autorisé aux adresses les plus proches soit 1024 et 1536.

Lors de l'initialisation de la carte SD, la taille des blocs a été définie à 512. Il y aura donc une gymnastique à avoir lors de la lecture et de l'écriture sur la carte de façon à écrire et lire à la bonne adresse (différente d'un multiple de 512).

2- Lecture

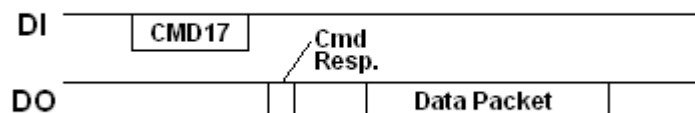
Pour la lecture, nous avons écrit une fonction dont le prototype est le suivant :

```
uint32_t read_bytes (uint64_t address, uint8_t *byte_array, uint32_t nb_read_bytes)
```

La fonction prend en paramètre l'adresse à partir de laquelle on veut lire, le pointeur où sera stocké le résultat de la lecture et le nombre d'octet à lire.

La gymnastique sur les adresses est gérée dans la fonction de lecture, et la fonction retourne bien le nombre d'octets effectivement lus.

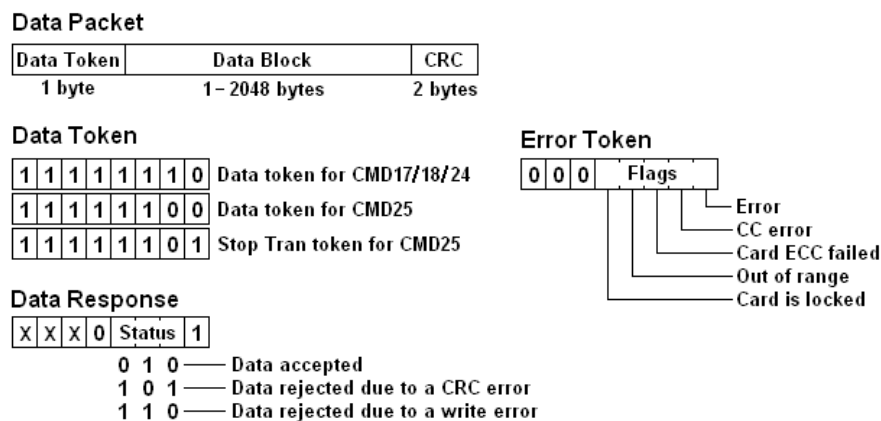
Le protocole d'envoi et de réception des données suit la trame suivante :



Trame d'envoi et de réception pour la lecture.

La lecture d'une donnée se fait donc par l'envoi de la commande CMD17, suivit de l'adresse sur 32bits. Il est donc impossible d'adresser plus de 4 Go de mémoire. Nous ne l'avons pas développé lors de ce projet, mais il est possible d'utiliser la commande CMD18 qui permet de recevoir plusieurs blocs.

La trame qui est renvoyée par la carte est de la forme suivante :



Donnée renvoyée par la carte

La carte nous renvoi donc une multitude d'info sur son état. S'il y a une erreur dans les données envoyées, à la place d'un « Data Packet » on reçoit un « Error Token ». A l'heure actuelle, seule l'erreur « out of range » est vérifiée.

3- Ecriture

Pour l'écriture, il y a deux fonctions dont voici les prototypes:

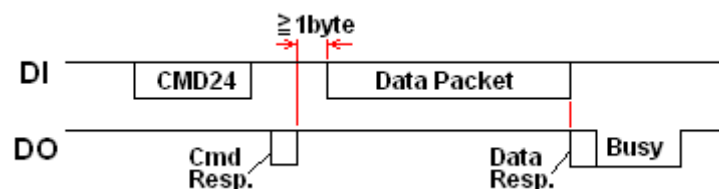
```
uint8_t write_block(uint64_t address_block, uint64_t address, uint8_t *data, int16_t size, uint8_t type)
```

```
uint32_t write_bytes(uint64_t address, uint8_t *byte_array, uint32_t size)
```

La fonction « write_bytes » regarde dans combien de blocs il faut écrire les données placées en paramètre. Ensuite en fonction de cela elle fait appel à la fonction « write_block » en lui donnant l'adresse du block (ici un multiple de 512), l'adresse de début ou de fin d'écriture sur le bloc, la donnée à écrire, sa taille et un paramètre qui permet à la fonction « write_block » de savoir si on est à la fin de l'écriture.

La fonction « write_block » quant à elle utilise la fonction « read_bytes » afin de réécrire les données présentes sur la carte dans le cas où nos données à écrire sont plus petites que la taille du bloc.

Comme pour la lecture, l'écriture fait appel à une commande particulière qui est la commande CMD24.



Trame d'envoi et de réception pour l'écriture

Il faut savoir que la fonction « write_block » est bloquante, c'est-à-dire que tant que la carte est dans le mode « busy », on ne sort pas de la fonction.

Il faut aussi noter que la fonction « write_byte » renvoi le nombre d'octets écrits. Mais s'il y a un problème pendant l'exécution de cette fonction, la fonction retourne 0. Dans le cas d'une erreur, la fonction « write_byte » ne garantie pas que des données n'ont pas été écrite à l'adresse placée en paramètre. Il se peut donc que des données soient perdues lors d'une erreur d'écriture.

III- Structure du système de fichier FAT12/16

a) Structure physique de la mémoire

Dans le cas du système de fichiers FAT, on distingue trois unités de mesures :

- L'octet
- Le secteur, composé de plusieurs octets
- Le cluster, composé de plusieurs secteurs

Ainsi, il y a deux paramètres importants : le nombre d'octets par secteur et le nombre de secteur par cluster.

Ces paramètres sont fixés lors du formatage du système de fichier. Ils dépendent, entre autre, du système d'exploitation utilisé. Par exemple sur Linux, on pourra les renseigner individuellement grâce à la commande `mkdosfs` (voir `man mkdosfs`). Sur Windows, on ne pourra pas les paramétrer aussi précisément.

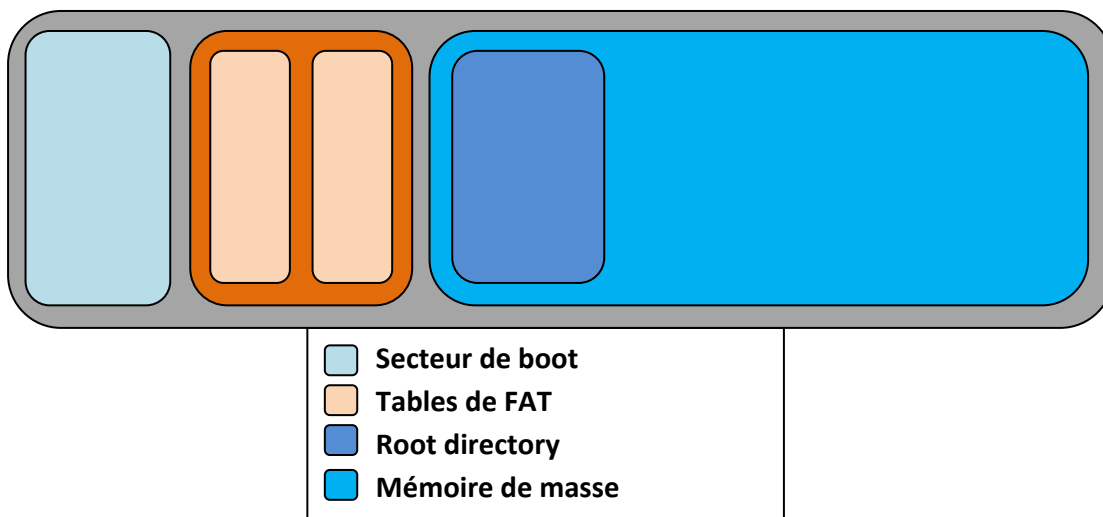
Ces deux paramètres sont très importants afin de pouvoir se repérer dans la mémoire. Nous verrons dans la suite du rapport comment récupérer ces paramètres et comment les utiliser.

On remarquera que dans le cas de la FAT, on utilise le codage little endian. Ainsi, lorsque l'on a deux octets qui se suivent, l'octet de poids fort est à droite.

b) Structure du système de fichier

Le système de fichier se décompose en trois parties principales :

- Le secteur de boot (ou BPB, pour Boot Parameter Block)
- Les tables de FAT, qui sont une carte de la mémoire de masse
- La mémoire de masse, qui dans le cas de la FAT12/16 contient une partie spéciale, le root directory



1- Secteur de boot

Dans cette partie de la mémoire sont recensés plusieurs paramètres important concernant le système de fichier.

Le secteur de boot débute à l'adresse 0x00 et s'étend sur un secteur. Cependant, dans le cas de la FAT12/16 seuls les 62 premiers octets sont utilisés. Le reste du secteur n'est pas utilisé.

Position (octet)	Taille (octets)	Description
0	3	Saut vers un programme qui va charger l'OS
3	8	Nom du programme qui a formaté le disque
11	2	Nombre d'octet par secteur (512, 1024, 2048 ou 4096)
13	1	Nombre de secteur par cluster (1, 2, 4, 8, 16, 32, 64 ou 128)
14	2	Nombre de secteurs réservés en comptant le secteur de boot (32 par défaut pour FAT32, 1 par défaut pour FAT12/16)
16	1	Nombre de FATs (2 par défaut)
17	2	Nombre de header dans le root directory
19	2	Nombre total de secteur 16-bits (0 pour FAT32)
21	1	Type de disque (0xF8 = disque dur, 0xF0 = disquette)
22	2	Nombre de secteur par FAT
24	2	Nombre de secteur par piste
26	2	Nombre de têtes
28	4	Secteurs cachés (0 par défaut si le disque n'est pas partitionné)
32	4	Nombre de secteurs 32-bits (seulement si le nombre de secteurs 16-bits est égal à 0)
36	1	Identifiant du disque (à partir de 0x00 pour disques amovibles, à partir de 0x80 pour les disques fixes)
37	1	Reservé
38	1	Signature (0x29 par défaut)
39	4	Numéro de série du disque
43	11	Nom du disque sur 11 caractères
54	8	Type de systèmes de fichier (FAT, FAT12, FAT16)

Les principales informations que l'on utilise sont surlignées. On ne peut pas se fier au champ indiquant le type de formatage, on doit calculer « à la main » le nombre total de cluster pour déterminer quel est le type de la FAT (voir fonction getFATtype).

On pourra également calculer les adresses de fin du secteur de boot, c'est-à-dire de début des tables de FAT, et de début du root directory. Le calcul de ces adresses sont exposés dans la partie sur la fonction init_file_system.

2- Table de FAT

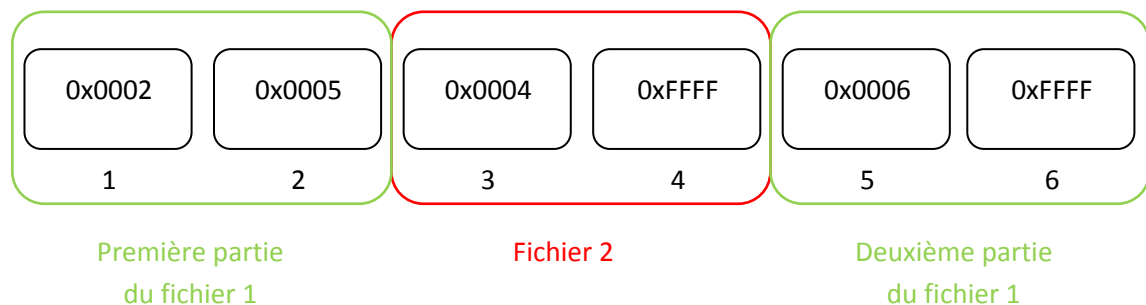
Généralement au nombre de 2, pour des raisons de redondance, les tables de FAT permettent de cartographier la mémoire de masse.

Chaque élément de cette table correspond à un cluster et dans le cas d'un fichier qui occupe plusieurs clusters, de connaître les différents clusters qui le composent.

Un élément de la FAT comprend 16 bits (2 octets) dans le cas de la FAT16 ou 12 bits (1.5 octets) dans le cas de la FAT12 et contient l'une des valeurs suivantes :

FAT12	FAT16	Description
0x000	0x0000	Cluster vide
0x001	0x0001	Cluster réservé
0x002 – 0xFEFF	0x0002 – 0xFFEF	Cluster utilisé, pointeur vers le cluster suivant du fichier
0xFF0 – 0xFF6	0xFFF0 – 0xFFF6	Valeurs réservés
0xFF7	0xFFF7	« mauvais cluster »
0xFF8 – 0xFFFF	0xFFF8 – 0xFFFF	Cluster utilisé, dernier cluster du fichier

On donne ci-dessous un exemple (FAT16) avec deux fichiers, le premier est fragmenté alors que le second est contigu dans la mémoire :



En réalité, les premières entrées de la table ne sont pas disponibles. Elles correspondent aux clusters utilisés pour le secteur de boot et les tables de FAT. On peut déterminer le nombre d'éléments réservés au début de la FAT avec la relation suivante :

3- Mémoire de masse

Dans le cas de la FAT12 et de la FAT16, le début de la mémoire de masse est spécial. Il s'agit du root directory, le répertoire racine qui contient la description de tous les fichiers et sous-répertoires de la partition.

On appelle header la description de ces fichiers / sous-répertoires. Le nombre de header dans le root directory est stocké dans le BPB (offset 17). Chaque header est composé de 32 octets :

Offset	Taille	Description		
0x00	8	Nom du fichier (rempli par des espaces), le premier octet peut avoir l'une des valeurs spéciales suivantes :		
		0x00	Entrée disponible, marque la fin du répertoire	
		0x05	Le nom de fichier commence en fait par le caractère ASCII 0xE5	
		0x2E	Entrée '.' ou '..'	
		0xE5	Entrée supprimée	
0x08	3	Extension du fichier (rempli par des espaces)		
0x0B	1	Attributs du fichier :		
		Bit	Masque	Description
		0	0x01	Lecture seule
		1	0x02	Fichier caché
		2	0x04	Fichier système
		3	0x08	Nom du volume
		4	0x10	Sous-répertoire
		5	0x20	Archive
		6	0x40	Device (utilisé en interne, jamais sur le disque)
		7	0x80	inutilisé
La valeur 0x0F est utilisé pour désigné un nom de fichier long.				
0x0C	1	Réservé (utilisé par NT)		
0x0D	1	Heure de création : par unité de 10ms (0 à 199)		
0x0E	2	Heure de création :		
		Bits	Description	
		15-11	Heures (0-23)	
		10-5	Minutes (0-59)	
		4-0	Secondes (0-59)	
0x10	2	Date de création :		
		Bits	Description	
		15-9	Année – 1980 (0 = 1980, 127 = 2107)	
		8-5	Mois (1 = Janvier, 12 = Décembre)	
		4-0	Jour (1-31)	
0x12	2	Date du dernier accès (voir offset 0x10)		
0x14	2	Index EA (utilisé par OS/2 et NT) pour FAT12/16, octets de poids fort du numéro du premier cluster pour FAT32		
0x16	2	Heure de dernière modification (voir offset 0x0E)		
0x18	2	Date de dernière modification (voir offset 0x10)		
0x1A	2	Numéro du premier cluster du fichier pour FAT12/16, octets de poids		

		faible du numéro du premier cluster pour FAT32
0x1C	4	Taille du fichier en octets (maximum = 4Go)

A partir du numéro du premier cluster du fichier, on peut déterminer l'adresse dans la mémoire de ce fichier (voir fonction `open_file`).

Dans le projet, on aura uniquement un répertoire racine, et pas de sous-répertoire. Un sous-répertoire est semblable à un fichier classique : à l'intérieur de ce fichier on retrouve des headers comme ceux du répertoire racine.

c) Nom de fichier long

On ne gère que les noms de fichiers courts, c'est-à-dire de la forme 8+3 (8 caractères pour le nom et 3 pour l'extension). Cependant, on est obligé de créer un header supplémentaire car c'est ainsi que les systèmes d'exploitation repèrent les fichiers dans la partition. Il s'agit du système VFAT, qui s'applique à toutes les versions de FAT à partir de la FAT12.

A chaque header classique s'ajoute un ou plusieurs header spéciaux, repérés grâce à la valeur 0x0F à l'offset 0x0B.

Le nom du fichier est stocké dans le header, chaque caractère étant codé sur 2 octets (codage UTF-16). Comme on utilise uniquement des caractères ASCII, l'octet de poids fort sera toujours à 0.

Structure d'un header de nom de fichier long :

Offset	Taille	Description
0x00	1	Numéro de séquence (voir ci-dessous)
0x01	10	Nom du fichier (5 caractères)
0x0B	1	Marqueur de header de nom de fichier long (toujours 0x0F)
0x0C	1	Type (toujours 0x00)
0x0D	1	Checksum du nom de fichier DOS (toujours 0x00)
0x0E	12	Nom du fichier (6 caractères)
0x1A	2	Numéro du premier cluster (toujours 0x0000)
0x1C	4	Nom du fichier (2 caractères)

On peut ainsi stocker 13 caractères par header supplémentaire (en comptant l'extension).

Le numéro de séquence identifie cette suite de caractère par rapport au nom du fichier. On donne l'exemple suivant, pour coder le nom du fichier « Un nom de fichier vraiment long.txt » :

Numéro de séquence	Partie du nom stockée
0x43	« long.txt »
0x02	« hier vraiment »
0x01	« Un nom de fic »
	Entrée classique 8+3 (nom stocké : « UNNOMD~1.TXT »)

Dans notre cas, on utilisera seulement un header supplémentaire puisque l'on se limite aux noms de 8 caractères (+3 pour l'extension).

IV- Implémentation du système de fichiers

Cette API est un ensemble de fonctions permettant à un utilisateur de gérer simplement un système de fichier FAT 12, 16 ou 32. Les fonctionnalités demandées dans le cahier des charges étaient les suivantes :

Fonctionnalités minimales (dans une version light):

- créer un fichier
- remplir un fichier (au choix : création ou erreur si inexistant)
- lire un fichier (par octet ou par bloc (taille max définie à la compilation))
- lister les fichiers présents sur la carte SD (noms en 8.3, dans un ordre quelconque)
- fermeture propre de fichier (en faisant le moins d'accès possible à la carte)

Fonctionnalités optionnelles (dans une version full):

- retourner la capacité totale et/ou restante
- effacer un fichier
- navigation dans les répertoires existants
- créer/effacer un répertoire

Lors de la réalisation de ces fonctionnalités nous avons dû garder à l'esprit les contraintes inhérentes à la programmation sur microcontrôleur (code utilisant peu de ressources), et une contrainte fixée par l'encadrant : l'utilisation de types de données portables.

Voici les prototypes des principales fonctions que nous avons construites :

- void init_file_system();
- void quit_file_system();
- uint8_t getFATtype();
- file_SD * create_file(char name[8], char ext[3], uint8_t file_attributes);
- file_SD * open_file(char name[8], char ext[3], uint8_t append_mode);
- uint32_t write_file(file_SD * file, uint8_t * data, uint32_t size);
- uint32_t read_file(file_SD * file, uint8_t * data, uint32_t size);
- file_SD * list_next_file(uint32_t base);
- uint32_t list_files(uint32_t base, file_table * array);
- void close_file(file_SD * file);
- void clear_file(file_SD * file);
- void delete_file(file_SD * file);

a) La structure de fichier

Le type `file_SD` est une structure contenant les différents champs utiles à la gestion d'un fichier dans l'API :

```
typedef struct file_SD
{
    char name[8]; //assumed in capital letters
    char ext[3]; //assumed in capital letters
    uint64_t header_addr;
    uint32_t first_cluster_number;
    uint8_t file_attributes;
    uint32_t file_size; //in bytes
    uint32_t read_cursor; //cursor on the next reading
    uint32_t write_cursor; //cursor on the next writing
} file_SD;
```

Le champ "name" correspond au nom du fichier sur huit caractères en majuscule complété par des espaces (à la fin), "ext" correspond à son extension avec les mêmes règles.

"header_addr" est l'adresse du header du fichier dans le dossier qui le contient, ainsi il suffit de lire 32 octets à cette adresse pour retrouver le contenu du header.

Le "first_cluster_number" est le numéro du premier cluster où se situe le contenu du fichier. Il est utile pour commencer à lire le contenu du fichier et chercher, dans la FAT, le numéro du cluster contenant la suite du fichier.

Le champ "attributes" contient les attributs du fichier tels qu'ils ont été lus dans son header (ou tels qu'on veut qu'ils soient créés). Ce champ n'est pas utilisé dans le code.

Concernant "size", il contient la taille du fichier en octets. La valeur de ce champ peut être différente de celle contenue dans le header du fichier car le header n'est effectivement mis à jour que dans la fonction "create_file" ou "close_file".

"read_cursor" et "write_cursor" sont le numéro de l'octet du fichier qui sera respectivement lu ou écrit lors du prochain appel de "read_file" ou "write_file".

b) Les structures de carte SD et de dossier

Comme pour les fichiers, il existe une structure pour la carte SD ("SD_card") et une autre pour les dossiers ("dir_SD").

Celle de la carte contient les informations utiles, notamment des adresses, pour naviguer dans les différentes parties de la carte : l'adresse de la première FAT, celle du root, le taille d'un cluster en octet, ... etc. Cette structure contient également un tableau listant les fichiers ouverts.

La structure de dossier est très similaire à celle des fichiers (car ils sont identiques à un flag près dans le système FAT), mais comme nous n'avons pas géré la navigation dans les dossiers, elle est très peu utilisée.

c) Les fonctions

Nous ne détaillerons que quelques fonctions certaines étant relativement simples et d'autres ne faisant que des appels. Cependant il y a une description succincte de chacune en commentaires dans `file_system.h`.

1- Create_file

La fonction `"create_file"` renvoi un pointeur vers la structure `file_SD` qui correspond au fichier créé.

On doit lui passer en paramètre le nom, l'extension et les attributs du fichier à créer (voir offset 0x0B dans la description d'un header)

Elle réalise les opérations suivantes :

- Recherche d'un cluster libre dans la FAT. C'est la fonction `"get_free_cluster"` qui renvoi le numéro du premier cluster libre dans la FAT.
- Recherche de deux headers libres consécutifs dans le root directory (un pour le header en lui-même et un autre pour le nom de fichier VFAT). C'est la fonction `"getFreeHeader"` qui renvoie l'adresse d'un header valide.
- Initialisation de la structure `"file_SD"` et mise à zéro de la taille du fichier.
- Ecriture des octets correspondants au fichier sur le disque (dans le root directory et dans la FAT).

Elle utilise une autre fonction, `"write_dir"`. C'est dans cette dernière que l'on réalisera les quatre opérations mentionnées ci-dessus.

La fonction `"create_file"` n'est pas totalement déboguée, elle risque donc de poser problème. Voici quelques pistes pour la rendre totalement fonctionnelle :

On écrit la valeur 0xFFFF (pour signifier la fin du fichier) dans une seule des tables de FAT, alors qu'il faudrait l'écrire dans toutes les tables.

On se base sur la taille en octet du `"root directory"` pour connaître le nombre d'entrées disponibles, alors qu'en fait ce nombre d'entrée total est donné dans le BPB (offset 17)

Une solution alternative consiste à créer ses fichiers depuis un système d'exploitation classique et de les remplir grâce à la fonction `"write_file"`, qui elle est fonctionnelle.

2- List_next_file

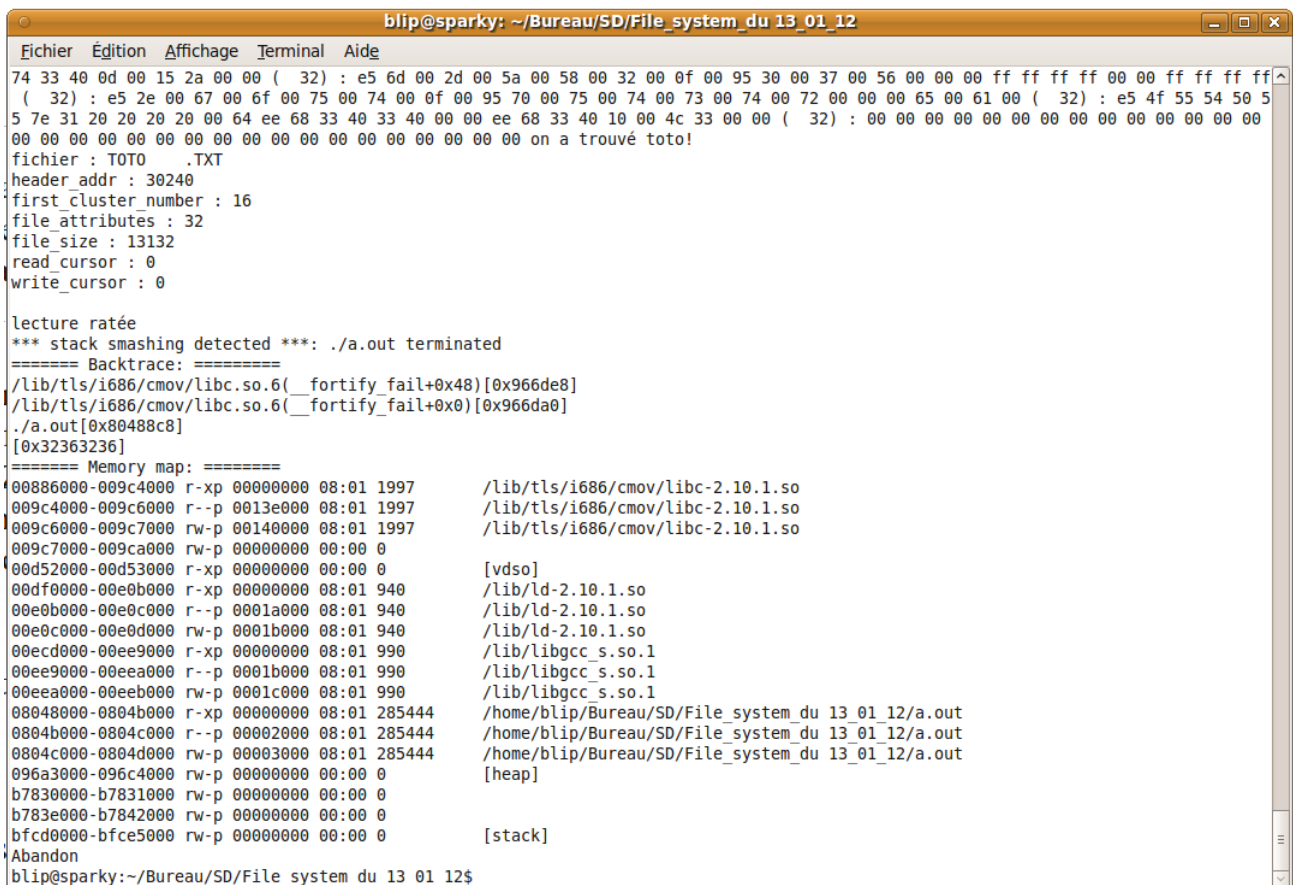
`"list_next_file"` est une fonction qui retourne le pointeur vers un `file_SD` qui a été créé puis rempli avec les informations de son header. On cherche ce header dans le dossier courant, soit au début de ce dossier si le paramètre `"base"` vaut 1 soit à la suite du header précédent si `"base"` vaut 0. Cette fonction parcourt les headers du dossier jusqu'à trouver un fichier valide ou un emplacement libre qui marque la fin du répertoire (elle renvoie alors NULL). Un header est considéré comme valide s'il n'est pas marqué comme supprimé ou s'il

n'est pas un nom de fichier long (actuellement repéré par une taille nulle mais il serait plus judicieux de vérifier que les attributs sont 0x0F).

3- List_files

La fonction "list_files" utilise "list_next_file" pour remplir un tableau de fichiers. Elle s'arrête lorsque la fin du dossier est atteinte ou que le tableau a une taille supérieure ou égale à la constante "MAX_SIZE_LIST_TABLE". En effet, cette fonction peut être gourmande en RAM s'il y a un nombre important de fichiers dans un même dossier, l'utilisateur a donc la possibilité de limiter la consommation de ressources en réduisant cette constante.

De même, de façon générale il faut éviter d'allouer des tableaux de taille trop importante, car une sur-utilisation des ressources peut provoquer des erreurs :



```
blip@sparky: ~/Bureau/SD/File_system_du_13_01_12
Fichier  Édition  Affichage  Terminal  Aide
74 33 40 0d 00 15 2a 00 00 ( 32) : e5 6d 00 2d 00 5a 00 58 00 32 00 0f 00 95 30 00 37 00 56 00 00 00 ff ff ff ff 00 00 ff ff ff ff
( 32) : e5 2e 00 67 00 6f 00 75 00 74 00 0f 00 95 70 00 75 00 74 00 73 00 74 00 72 00 00 00 65 00 61 00 ( 32) : e5 4f 55 54 50 5
5 7e 31 20 20 20 20 00 64 ee 68 33 40 33 40 00 00 ee 68 33 40 10 00 4c 33 00 00 ( 32) : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 on a trouvé toto!
fichier : TOT0 .TXT
header_addr : 30240
first_cluster_number : 16
file_attributes : 32
file_size : 13132
read_cursor : 0
write_cursor : 0

lecture ratée
*** stack smashing detected ***: ./a.out terminated
===== Backtrace: =====
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0x966de8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0x966da0]
./a.out[0x8048c8]
[0x32363236]
===== Memory map: =====
00886000-009c4000 r-xp 00000000 08:01 1997      /lib/tls/i686/cmov/libc-2.10.1.so
009c4000-009c6000 r--p 0013e000 08:01 1997      /lib/tls/i686/cmov/libc-2.10.1.so
009c6000-009c7000 rw-p 00140000 08:01 1997      /lib/tls/i686/cmov/libc-2.10.1.so
009c7000-009ca000 rw-p 00000000 00:00 0
00d52000-00d53000 r-xp 00000000 00:00 0      [vdso]
00df0000-00e0b000 r-xp 00000000 08:01 940      /lib/ld-2.10.1.so
00e0b000-00e0c000 r--p 0001a000 08:01 940      /lib/ld-2.10.1.so
00e0c000-00e0d000 rw-p 0001b000 08:01 940      /lib/ld-2.10.1.so
00ecd000-00ee9000 r-xp 00000000 08:01 990      /lib/libgcc_s.so.1
00ee9000-00eea000 r--p 0001b000 08:01 990      /lib/libgcc_s.so.1
00eea000-00eeb000 rw-p 0001c000 08:01 990      /lib/libgcc_s.so.1
08048000-0804b000 r-xp 00000000 08:01 285444    /home/blip/Bureau/SD/File_system_du_13_01_12/a.out
0804b000-0804c000 r--p 00002000 08:01 285444    /home/blip/Bureau/SD/File_system_du_13_01_12/a.out
0804c000-0804d000 rw-p 00003000 08:01 285444    /home/blip/Bureau/SD/File_system_du_13_01_12/a.out
096a3000-096c4000 rw-p 00000000 00:00 0      [heap]
b7830000-b7831000 rw-p 00000000 00:00 0
b783e000-b7842000 rw-p 00000000 00:00 0
bfcd0000-bfcd5000 rw-p 00000000 00:00 0      [stack]
Abandon
blip@sparky:~/Bureau/SD/File_system_du_13_01_12$
```

Erreur rigolote quand elle se produit sur un PC, beaucoup moins visible sur carte...

4- Open_file

Le comportement de la fonction "open_file" est assez naturel : elle exécute un "list_files", puis cherche s'il y a un fichier de nom identique dans la liste retournée (avec "file_array_contains"). Si le fichier a été trouvé, le "file_SD" correspondant (avec les curseurs de lecture/écriture correctement positionnés) est retourné, sinon "NULL" est retourné.

d) Précisions

Ces fonctions ont été dans un premier temps développées sur une image (fichier texte) de la carte de 16 Mo. Elles ont ensuite été testées sur le kit de développement. La fonction "write_file" n'a été testée qu'avec le kit. Ce code ne gère qu'une seule FAT, la seconde n'étant que peu utilisée par les systèmes classiques.

Il est à noter qu'afin de pouvoir utiliser "malloc" il faut allouer de la mémoire au "heap" (avec le logiciel IAR c'est le fichier "AT91SAM7X256_RAM.xcl").

CONCLUSION :

Au niveau de la finalité du projet, nous sommes plutôt satisfaits du travail rendu puisque les couches SPI et SD sont totalement fonctionnels. Nous pouvons donc parler avec toutes les cartes SD de moins de 4Go (car on a que 32bits d'adressage sur nos commandes). Les polarités et phases sont ajustables suivant la carte SD en changeant le paramètre de la fonction `initSPI(int PhaPol)`.

Sur le système de fichiers, toutes les fonctions demandées sont fonctionnels à part celle pour créer un fichier. En effet, ouvrir un fichier, lister les fichiers sur la carte, et enfin lire/écrire dans un fichier sont complètement fonctionnels. L'API est donc presque complète.

Au niveau du déroulement du projet, tout n'a pas été parfait. En effet, nous avons été obligés de travailler une quinzaine d'heures en plus du temps imparti au projet. Il aurait été préférable de tester la fonctionnalité des couches bas niveaux SPI avant de coder la couche Physical Layer. Du coup, nous avons dû faire un debug général après avoir tout codé, ce qui n'était pas forcément une manière de faire optimisée.

La raison à ceci est que nous ne soupçonnions pas que la documentation sur le SPI nous conduirait sur une fausse voie. En effet, il aura été difficile de trouver une doc correcte à ce niveau-là, nous vous indiquons donc l'adresse du site qui nous aura permis de nous debugger : elm-chan.org/docs/mmc/mmc_e.html

Au niveau de l'apport personnel, cela fait une nouvelle expérience en équipe. En effet, c'est le seul projet à l'ENSEIRB qui se fait à 4. Ceci implique donc de l'organisation supplémentaire et un travail d'équipe accru. A ce sujet, il faut noter que cela s'est très bien passé puisque l'on a judicieusement décomposé le travail au début du projet et mis 1 binôme sur les couches SPI/SD et 1 sur la gestion du système de fichiers. C'est toujours important d'être 2 sur quelque chose car cela permet une double vérification immédiate, d'où un gain de temps pour le debug (On fait théoriquement moins de bug quand on travaille à 2 plutôt que tout seul).

Vous jetterez un œil sur le site elm-chan.org/fsw/ff/00index_e.html ...