Stockage de masse non-volatile : un block device pour MultiMediaCard

J.-M Friedt, S. Guinot
friedtj@free.fr, simon.guinot@alcove.fr
Association Projet Aurore, Besançon, France

1 Introduction

De nombreux systèmes embarqués peuvent profiter d'un mode de stockage de masse non volatile fournissant une grande quantité de mémoire (plusieurs MB), sans partie mobile et de consommation réduite. Nous proposons ici d'utiliser à ces fins une MultiMediaCard (MMC et MMC+) telles que disponibles dans le commerce grand public pour des prix dérisoires compte tenu de leur capacité et répondant à nos contraintes.

Nous proposons ici de nous familiariser dans un premier temps avec les connexions électriques des MMC et MMC+ et les modes de communication entre un processeur et ses périphériques. Ayant été capables de communiquer avec les mémoires, nous aborderons le vif du sujet, à savoir implémenter un *block device* qui nous permette d'accéder aux données stockées par des commandes unix classiques, et ce grâce à un formatage de la carte. Ainsi, nous dépassons le stade du simple enregistreur de données se limitant à stocker des octets successifs, pour réellement proposer un support de stockage souple d'utilisation.

Nous avons initialement développé ce driver pour le processeur Coldfire 5282 équipant la carte SSV DNP/5280 fonctionnant sous uClinux (noyau 2.4.x). Ce circuit fournit en effet une plateforme réellement embarquée, de consommation réduite si on considère le nombre de périphériques disponibles. Ce processeur fournit notamment une implémentation matérielle du bus SPI utilisé pour la communication avec les MMC et MMC+. Cependant, afin de faire profiter à un public aussi vaste que possible de nos développements, nous avons implémenté de façon logicielle le protocole SPI *via* le port parallèle d'ordinateurs personnels compatibles IBM fonctionnant sous GNU/Linux (noyau 2.4.x). Bien que cette solution soit peu performante en terme de bande passante du fait de la lenteur du port, elle ouvre de nombreuses perspectives, telles qu'équiper des ordinateurs trop anciens pour posséder un port USB avec des espaces de stockage de données de plusieurs centaines de MB, ou fournir les bases d'un portage de ce driver à tout matériel offrant 4 broches d'entée-sortie généralistes (*General Purpose I/O*, GPIO) qui peuvent être utilisées pour une émulation logicielle du bus SPI (fig. 3).

2 Aspects matériels

La carte MMC se présente comme un support de mémoire de petites dimensions ($32\times24\times1{,}4$ mm^3 [1]) équipé de 7 broches. La MMC+ présente des dimensions identiques mais une rangée de connecteurs supplémentaires dont nous ne ferons pas usage ici. Notons que les développements qui vont suivre ne sont *pas* appliquables aux cartes SD : bien que de dimensions identiques, ces cartes (équipées de 8 contacts) nécessitent un mode d'initialisation différent que nous n'avons pas encore appréhendé. L'espacement entre les broches est de 2,54 mm, pas standard dans de nombreuses cartes enfichables. De plus, son épaisseur à peine inférieure aux 1,6 mm standard de circuit imprimé nous incite à rechercher dans les supports de cartes d'extensions informatiques un support peu coûteux et plus facilement accessible que les connecteurs spécifiquement dédiés aux MMC que nous n'avons pas été capables de nous procurer en petites quantités.

3 Le bus SPI

Le bus SPI est un bus synchrone (le maître fournit un signal d'horloge aux esclaves), équipé de deux voies unidirectionnelles (MOSI – *Master Out Slave In*, et MISO – *Master In Slave Out*) et d'un signal d'activation de chaque périphérique que nous nommerons CS# (*Chip Select*, actif au niveau bas) bien que de nombreuses datasheets de MMC nomment cette broche SS. Cette dernière broche est toujours celle la plus proche du coin biseauté de la carte qui sert de détrompeur dans les montages commerciaux.

Il faut de plus fournir une alimentation stable de 3,3 V à la MMC par un régulateur de tension connecté à un générateur de tension externe (le port parallèle est incapable de fournir le courant nécessaire au bon fonctionnement d'une MMC) : suivant l'idée proposée il y a longtemps (1994 [2]) par la webcam Connectix Quickcam N&B, nous prendrons l'alimentation sur le port clavier afin de s'affranchir d'une batterie additionnelle [3]. Nous avons pour ceci utilisé un convertisseur LE33CZ [1]

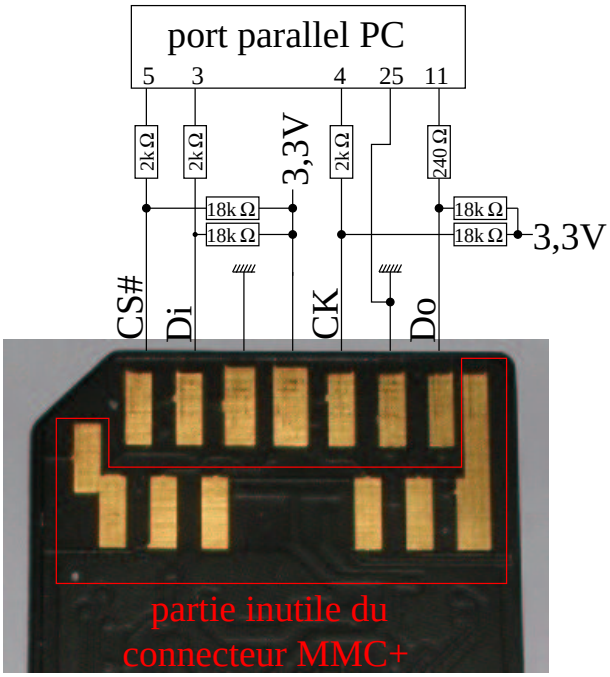


FIG. 1 – Connecteur MMC+ et brochage au port parallèle avec conversion des signaux de TTL (5 V) à 3,3 V par des composants passifs uniquement. Le brochage avec l'uClinux se fait directement broche à broche (D_i à MOSI, D_o à MISO, horloge et activation du composant). Dans le cas d'une MMC, seule la partie utile du connecteur de MMC+ est accessible.

Dans l'ordre depuis le coin biseauté de la MMC (Fig. 1), les signaux sont CS#, la communication du processeur vers la carte Data In (D_i), la masse, l'alimentation à 3,3 V, horloge CK du protocole synchrone de communication, à nouveau la masse, et finalement la communication de la carte vers le processeur D_o.

3.1 La QSPI du Coldfire 5282

Le bus SPI et le protocole synchrone associé sont implémentés dans de nombreux microcontrôleurs. C'est notamment le cas dans le Coldfire 5282, dans une version évoluée nommée QSPI qui fournit au niveau matériel une mémoire tampon de 16 éléments définissant chacun la

[1] 0,80 euros/pièce TTC chez Lextronic ou chez Farnell sous la référence 9755349.

configuration du bus (délais entre les fronts, activation des divers signaux CS# présents) et la donnée associée à transmettre. Notez que le bus du circuit DNP5280 fourni par SSV ne donne accès qu'à un signal d'activation CS#, à savoir CS0# dans la nomenclature Coldfire (Fig. 2).

Il nous faut dans un premier temps initialiser le port QSPI du Coldfire, en définir la configuration avant de pouvoir placer des données à transmettre dans les mémoires tampons. L'ensemble de ces opérations sont décrites en détail dans [4, chap.22], implémentées sous uClinux selon le code présenté dans la section 5.1.2, dont nous résumons ici les étapes qui ont été nécessaires :

- définir le nombre de bits/transfert (8 bits), la vitesse de transfert et la polarité, indiquer que le Coldfire est maître des transactions avec la MMC. *A priori* la vitesse peut-être aussi élevée que possible, mais il faut prendre soin de définir le transfert de données sur le front *montant* de l'horloge (bit CPHA de QMR à 0) tel que décrit dans [4, Fig.22-4]. Toutes ces opérations se définissent dans le registre QMR du Coldfire à l'adresse MCF_IPSBAR+0x340 (où MCF_IPSBAR=0x40000000). Les délais tels que définis dans QDLYR sont mis à leur valeur minimale de 1.
- une interruption est associée aux transactions sur le bus SPI afin de limiter les attentes vides et ainsi réveiller notre module lorque l'opération requise est achevée : nous plaçons QIR à 0xB10D afin de générer une interruption en cas de collision et en fin de transaction (QIR est localisé en MCF_IPSBAR+0x34C).
- pour chaque nouvelle transaction, nous disposons de deux files de 16 mots chacunes, l'une contenant les commandes associées à chaque transaction et la seconde aux données à transférer. Le fait de définir le mot à transférer ou la commande associée est sélectionné par la valeur du registre QAR qui indexe le registre QDR contenant les commandes/données : en deça de 0x20 QDR contient les données, au delà il s'agit de commandes. Le nombre de transactions actives est défini dans QWR (en MCF_IPSBAR+0x348), qui sera en pratique égal à 16 afin d'optimiser le nombre de transactions par appel au module de gestion de la QSPI, mais dont la valeur peut varier en fonction de la quantité de données à transférer. La gestion des commandes associées à chaque transfert est très souple puisque nous nous contentons de placer, pour transférer un mot placé dans le $i$ème emplacement de QDR, de définir dans le mot situé au $i+32$ème octet de QDR une série de bits correspondant à l'état de chacune des lignes de Chip Select (dans notre cas la MMC requiert presque toujours – à l'exception de la phase d'initialisation – un Chip Select actif au niveau bas donc un bit correspondant à 0) : nous prenons CONT=BITSE=0 pour indiquer que le transfert est sur 8 bits, DT=DSK=1 et finalement, puisque seul le Chip Select 0 est accessible sur la carte SSV DNP5280, le niveau du bit de poids le plus faible de QSPI_CS pour définir l'état des broches d'activation des périphériques.
- une fois les registres sous QDR convenablement remplis, la transaction est activée par la mise à 1 du bit de poids le plus fort de QDLYR (à l'adresse MCF_IPSBAR+0x344). Cette transaction prendra fin avec la génération d'une interruption qui permet au processeur de prendre soin d'autres tâches pendant le transfert.

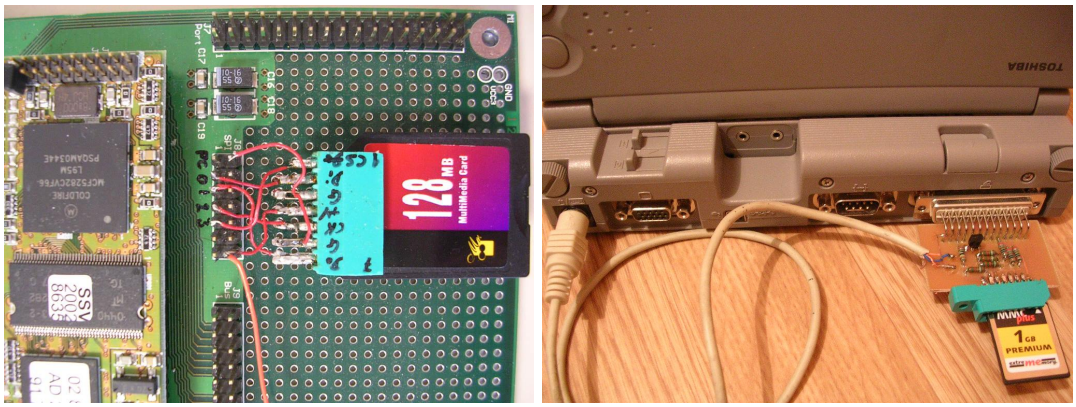
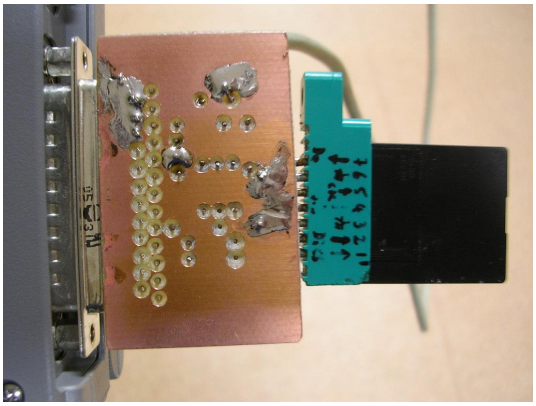


Fig. 2 – Gauche : montage d'une MMC sur la carte de développement associée au DNP5280 de SSV. Notez l'utilisation d'un bout de connecteur ISA (au pas de 2,54 mm) pour assurer la connexion électrique avec la MMC. Les Coldfire ainsi que la MMC fonctionnent tous deux sous une tension de 3,3 V. Droite : adaptation d'une MMC+ au port parallèle d'un Libretto 100CT. Notez ici l'utilisation de la sortie 5 V du connecteur PS2 pour alimenter la carte, évitant ainsi l'ajout d'une batterie externe.



En complément à l'article de Denis Bodor sur la réalisation des circuits imprimés [5], nous proposons ici une alternative à la réalisation de circuits imprimés double-face lorsque la face inférieure se contente de router des signaux de masse. Ici, nous avons réalisé un circuit simple face sur plaque pré-sensibilisée double face, en définissant la face inférieure comme plan de masse uniformément couvert de cuivre. Les quelques composants nécessitant un perçage (la technique est surtout valide pour des circuits comprenant une majorité de composants CMS) sont isolés du plan de masse par un petit fraisage par un forêt de perceuse de quelques millimètres (>5 mm) de diamètre.

3.2 Émulation logicielle du SPI sur port parallèle de PC

Le processeur Coldfire n'étant pas suffisamment répandu auprès des développeurs de circuits embarqués, et afin de nous familiariser avec les MMC sur un environnement de développement plus facile d'accès et plus répandu que ce microcontrôleur, nous avons développé une implémentation matérielle et logicielle du protocole SPI sur port parallèle de PC compatible IBM [6]. La seule subtilité ici consiste à convertir les tensions issues du port parallèle entre 0 et 5 V (électronique

compatible TTL) en tensions entre 0 et 3,3 V telles que requises par la MMC. Bien que des composants dédiés à cette tâche existent, nous avons simplifié au maximum le montage en n'utilisant que des résistances de limitation de courant.

L'émulation logicielle du protocole SPI par le port parallèle du PC est simplifiée d'un point de vue matériel du fait que ce bus inclut deux lignes distinctes de transmission du maître vers l'esclave (Master Out Slave IN : MOSI) et de l'esclave vers le mître (MISO). Du point de vue électronique, nous aurons donc une broche en sortie depuis le PC (la broche 3 du port parallèle, signal Data1) et une broche en entrée (broche 11, signal Status7). Le signal d'horloge est imposé par le maître (le PC) sur une broche de sortie (broche 4, signal Data2). Finalement le signal de sélection de la carte CS# est connecté à la broche 5 – Data3 – du port parallèle.

Le seul point important dans l'implémentation du procotole synchrone est que l'échantillonage de la ligne de données se fait sur le front *montant* de la ligne d'horloge. Il faut donc placer l'horloge au niveau bas, définir le bit de données, puis faire monter le signal d'horloge. L'autre point important est qu'une lecture et une écriture sont identiques du point de vue SPI : toute transition montante du signal d'horloge s'accompage soit d'une lecture du signal MOSI par l'esclave (signal imposé par le maître), soit d'une lecture du signal MISO par le maître. Ainsi, en terme d'implémentation logicielle, chaque transition de l'horloge s'accompage d'une transmission sur MOSI et d'une lecture sur MISO, avec éventuellement un octet aléatoire transmis sur MOSI lors d'une lecture par le maître. Toutes les transmissions sur le bus SPI se font par octet. À ce niveau, nous ne nous préoccupons donc pas de l'endianness du processeur : ce problème réapparaîtra plus tard dans le cas particulier de la transmission d'adresses sur 32 bits.

4 Communication avec les MMC et MMC+

Ayant maîtrisé le bus de communication SPI, il nous faut maintenant y implémenter le protocole d'initialisation et de communication avec la MMC. Nous rappelons ici que ce protocole n'est *pas* compatible avec les cartes de type SD, bien que les différences semblent d'après les documentations relativement mineures.

4.1 Initialisation de la MMC

Les diverses étapes de l'initialisation et de la communication avec une MMC ont déjà été décrites par ailleurs [7] et nous n'en reprenons ici que les grandes lignes. La seule subtilité est de tenir compte de l'endianness du processeur sur lequel s'exécute le code lors de la conversion d'une adresse sur la carte sur un entier (4 octets) qui doivent être transmis octet de poids fort (MSB) en premier.

Dans un premier temps, nous informons la MMC que nous communiquerons avec elle selon un protocole SPI en envoyant 80 transitions sur le fil d'horloge CK tout en maintenant CS# au niveau haut.

Suit la première commande de réinitialisation de la carte, CMD0. Nous développons ici la nomenclature quelque peu originale des commandes telles que décrites dans les manuels de MMC. Toute commande nommée CMDi se compose de 6 octets : le numéro de la commande i, où i est exprimé en *décimal*. Ce numéro de commande doit être masqué *via* un OU logique avec 0x40 avant d'être transmis à la MMC. Suivent 4 octets qui sont les arguments de la commande, et finalement un code redondant cyclique (CRC) qui dans le cas de la communication SPI est ignoré et que nous fixerons donc arbitrairement à 0xFF.

Une exception est la première commande de réinitialisation de la carte lors du passage en mode SPI – CMD0 – pour laquelle le CRC est pris en compte mais est précalculé : il vaut dans ce cas 0x95 si tous les arguments sont définis à 0x00. Ainsi l'initialisation de la carte s'obtient par la transmission de la séquence de 6 octets :

0x40 0x00 0x00 0x00 0x00 0x95

La MMC nous répond 0x01 pour acquitter la CMD0 (GO_IDLE_STATE qui correspond à une réinitialisation logicielle). Suit la commande CMD1 à laquelle la MMC doit répondre par 0x00.

Ayant transmis les commandes CMD0 et CMD1, nous avons un système à notre écoute et fonctionnel. En prévision de l'application à l'écriture d'un *block device* capable de supporter n'importe quelle carte MMC ou MMC+, nous allons implémenter une commande supplémentaire qui nous informe sur la géométrie de la carte et notamment sur sa capacité mémoire.

4.2 Le registre CSD de la MMC

Un registre interne à la carte est le CSD (*Card Specific Data register*) qui contient sur 128 bits un certain nombre d'informations sur les tension et courant nécessaires au bon fonctionnement de la carte, les temps d'accès mais surtout la géométrie et notamment le nombre de blocs accessibles. Les éléments qui vont nous intéresser pour renseigner les structures de données internes à un *block device* sont :

- la taille de chaque bloc physique dans la MMC, *a priori* toujours égale à 512 octets, nommé $READ\_BL\_LEN$ (bits 83-80 du CSD)
- le nombre de blocs disponibles, qui nous renseigne donc sur la taille totale de la MMC, nommé C_SIZE (bits 73-62 du CSD)
- un facteur de pondération nommé C_SIZE_MULT (bits 49-47 du CSD)

Le CSD est obtenu par la commande CMD9 (*SEND_CSD*), qui nous renvoit en réponse l'octet 0xFE de début de transmission de la réponse, suivi de 16 octets (128 bits) du CSD lui même (octet le plus significatif en premier), puis deux octets de checksum que nous ignorons pour le moment. Le calcul de la capacité de la carte en fonction des valeurs lues dans le CSD est décrit dans [1, p.36] :

$$taille = (C\_SIZE + 1) \times 2^{C\_SIZE\_MULT+2} \times 2^{READ\_BL\_LEN}$$

À titre d'application numérique, pour une carte de 64 MB nous lisons un CSD de `48 0e 01 2a 0f f9 81 e9 f6 da 01 e1 8a 40 0 c1` dont nous déduisons que $READ\_BL\_LEN = 9$ soit des blocs de 512 octets, $C\_SIZE = \$7A7 = 1959$ et $C\_SIZE\_MULT = 4$ soit au total $(1959 + 1) \times 512 \times 2^{4+2} = 64225280$ octets.

Exactement selon le même protocole nous pouvons lire le CID (*Card IDentification register*) de la MMC au moyen de CMD10. Ce registre nous informe sur le fabriquant de la carte, sa date de fabrication et fournit un identifiant qui nous permettra éventuellement de détecter un changement de carte.

4.3 Écriture et lecture dans la MMC

La MMC s'attend à recevoir toutes les adresses de blocs octet de poids le plus fort (MSB) en premier : il faut donc penser à convertir une requête exprimée en `long` (telle qu'imposée par un *block device*) via la fonction `htonl()` ou, dans sa version noyau, `__cpu_to_be32()`. La commande de lecture est CMD17 (READ_SINGLE_BLOCK) et la commande d'écriture est CMD24 (WRITE_BLOCK). Ces commandes prennent en argument 4 octets qui représentent l'adresse de début du bloc (nécessairement multiple de 512 si $READ\_BL\_LEN = 9$) puis un dernier octet de validation (CRC) si cette fonction est activée : nous l'ignorons pour le moment. Le protocole diffère ensuite selon que l'on soit en écriture ou en lecture :

- dans le cas d'une lecture, la MMC nous répond si elle accepte notre requête en répondant 0xFE. Toute autre réponse indique le refus de la requête (notamment un réponse de type 0x60 indique que l'on fournit une adresse invalide – soit au delà de la capacité de la carte, soit qui n'est pas multiple de 512 – erreur possible si l'on a inversé l'ordre des octets représentant l'adresse). Suivent la réception des 512 octets de données, puis 2 octets de validation (CRC) que nous ignorons pour le moment.
- dans le cas d'une écriture, nous recevons 0x00 comme acquittement de la requête, auquel nous répondons en envoyant 0xFE suivi des 512 octets à stocker, et finalement les deux octets de CRC si cette option est activée – nous les ignorons encore pour le moment (envoi d'une

valeur quelconque). La transaction s'achève par la lecture du statut de la carte : une réponse finissant par 5 indique que le bloc est en cours d'écriture, opération qui sera complétée lorsque le bit de données du bus SPI passe de l'état bas (0x00) à haut (0xFF).

L'ensemble du protocole implémenté jusqu'ici nous fournit les éléments de base d'un *character device*, à savoir lire et écrire une donnée à une adresse donnée. Nous désirons maintenant dépasser ce stade et implémenter un *block device* facilitant l'accès au périphérique de stockage de masse.

5 Implémentation d'un *block device*

Nous sommes désormais capables de lire et d'écrire des blocs de 512 octets en n'importe quel emplacement de la MMC. L'utilisation d'une énorme quantité de mémoire de cette façon est cependant fastidieuse, et nous allons désormais proposer le développement d'un *block device*, `mmc_qspi_mod` permettant de formater la carte avec un système de fichier et d'y accéder par les commandes unix standards. Le code source de ce pilote est disponible sur http://www.sequanux.org et http://jmfriedt.free.fr.

Dans toute la présentation qui va suivre – qu'il s'agisse de l'implémentation du driver sous GNU/Linux ou uClinux – la communication entre l'espace utilisateur et le noyau se fera par le block device `/dev/mmca` créé au moyen de `mknod /dev/mmca b 254 0` tandis que l'accès aux partitions de ce périphérique se font par les devices `mmca`*i* obtenus par `mknod /dev/mmca`*i* `b 254` *i* avec $i \in [1:4]$.

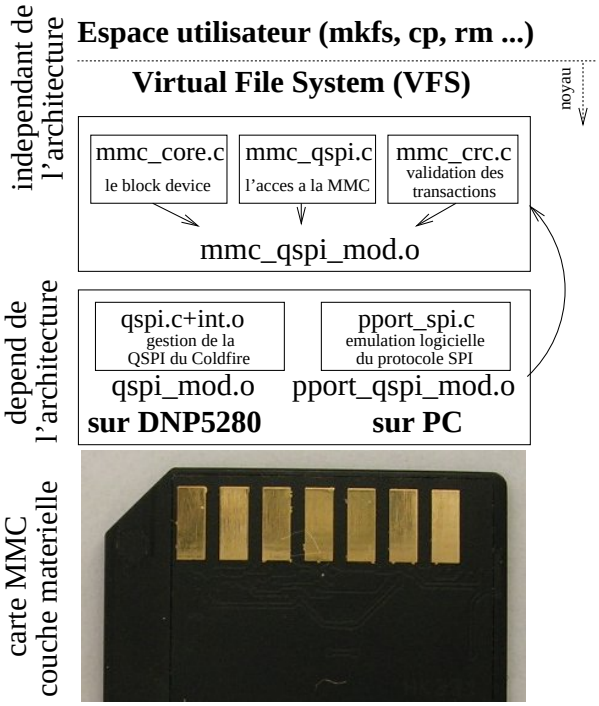


FIG. 3 – Architecture générale des éléments que nous nous proposons de décrire dans ce document : une couche de communication fortement dépendante du matériel. Elle fournit ses méthodes à une couche de plus haut niveau indépendante de l'architecture. Cette dernière gère l'accès aux périphériques et l'interface avec le *Virtual File System* (VFS) de GNU/Linux.

5.1 Architecture du pilote

Le pilote de la MMC est composé de deux modules : `mmc_qspi_mod` et `qspi_mod`. Les intérêts de cette modularisation sont multiples. Bien séparer les fonctionnalités du pilote en plusieurs modules permet de clarifier le développement et de simplifier le débuggage. Chaque module apparaît comme une "boîte noire" et peut être testé de manière indépendante. Un autre intérêt à cette modularisation est de rendre le pilote portable à moindre coût. L'adapter à un nouveau support matériel ne nécessite que la réécriture du module bas niveau. Lors de l'émulation SPI du port parallèle, nous avons pu expérimenter cette flexibilité. Le port de `qspi_mod` en `pport_qspi_mod` ne nous a pris qu'une petite heure !

5.1.1 Couche haut niveau

Le module `mmc_qspi_mod` constitue la partie haute du pilote. Il doit être capable de remplir deux fonctions :

1. la première est de s'interfacer avec le système de fichier virtuel (*Virtual File System – VFS*) et de fournir au noyau les méthodes pour écrire réellement sur la MMC,
2. la seconde est d'implémenter tout le protocole nécessaire à la communication avec la MMC.

Au vue de ces fonctions et aussi pour simplifier le développement et les évolutions futures, nous avons décidé de séparer `mmc_qspi_mod` en deux éléments distincts.
Le premier, `mmc_core` est très proche d'un pilote bloc classique et son travail est de réaliser l'interfaçage avec le noyau et le VFS. Le second, `mmc_qspi` implémente les différentes commandes utiles pour dialoguer avec la MMC en mode SPI.
La communication entre ces deux éléments est à sens unique. `mmc_core` sollicite les fonctions de `mmc_qspi` pour lire ou écrire des données sur la MMC.

Voici le fichier d'entête `mmc_qspi.h` déclarant les fonctions exportées par `mmc_qspi` :

```
#define CSD_SIZE 16

int open_mmc (void);     /* initializate the MMC */
char *get_csd (void);    /* return the CSD register */
int release_mmc (void);  /* free the MMC */
                         /* method to read a data block on the MMC */
int read_block (unsigned char, unsigned char, unsigned char, unsigned char, char *);
                         /* method to write a data block on the MMC */
int write_block (unsigned char, unsigned char, unsigned char, unsigned char, char *);
```

Nous verrons un peu plus tard l'implémentation de quelques unes de ces fonctions.

5.1.2 Couche bas niveau

Le module `qspi_mod` constitue la partie basse du pilote. Son travail est d'échanger des octets avec les périphériques connectés au bus SPI. À la différence d'un *char device* classique, `qspi_mod` n'exporte pas de méthodes à destination des processus en espace utilisateur. Les méthodes qu'il définit seront utilisées par d'autres pilotes du noyau souhaitant communiquer *via* le bus SPI.
Cela dit, nous avons également développé une version "caractère" de `qspi_mod`. Ce module s'appele `qspi_char_mod` et est fourni avec les sources du pilote. Il nous a d'ailleurs beaucoup servi lors des premiers contacts avec la MMC. Grâce à lui nous avons pu commencer par implémenter et débugger les commandes MMC depuis des programmes utilisateurs...

Dans le cas du port parallèle, le module `pport_spi_mod` fournira cette interface bas niveau. Bien que les communications sur le bus SPI et sur le port parallèle d'un PC soient très différentes, rien ne change du point de vue du module `mmc_qspi_mod`. Les modules "bas niveau" masquent ces différences en lui offrant une API unique.

Voici le fichier d'entête `qspi.h` déclarant les méthodes fournies par les modules `qspi_mod` et `pport_spi_mod` :

```
#define CMD_CS_LOW 0
#define CMD_CS_HIGHT 1
#define CMD_CLK 2

#ifndef QSPI_CHAR
extern int qspi_open (int);
extern int qspi_close (int);
extern size_t qspi_read (int, char *, size_t);
extern size_t qspi_write (int, char *, size_t);
extern int qspi_ioctl (int, unsigned int, unsigned long);
#endif
```

Bien que le *hardware*, les performances et l'implémentation soient très différentes, les modules bas niveaux `qspi_mod` et `pport_spi_mod` présentent une même API, compatible avec `mmc_qspi_mod`. Que ce soit pour le module `pport_spi_mod` ou le module `qspi_mod`, les fonctions qu'ils exportent ont un comportement identique... du moins en apparence...
Par exemple, une différence de taille est le mode utilisé pour transmettre physiquement des données à la MMC. Pour le bus SPI, `qspi_mod` utilise une interruption matérielle pour valider les transferts alors que `pport_spi_mod` utilise le mode *polling*, l'interruption du port parallèle n'étant pas exploitable pour cette application. La qualité du service offert par ces deux modules est donc très différente. Dans le cas de `pport_spi_mod`, le mode *polling* va bloquer l'ensemble du système pendant les opérations de lecture et d'écriture. Ce genre d'implémentation dégrade considérablement les performances du système tout entier. Il en résultera un manque d'interactivité de l'ensemble des processus. L'utilisation d'un noyau préemptible pourrait en partie compenser cette implémentation. Cependant un pilote se voulant un minimum portable ne peut décemment pas se reposer sur l'hypothèse de préemptibilité du système qui l'héberge. La seule alternative est donc de procéder à des appels à la fonction `schedule()` pour casser certaines boucles de *polling* un peu longues. La commande `schedule()` appelle l'ordonnanceur et donne une chance à un autre processus de s'exécuter. Le module `qspi_mod` n'est lui pas sujet à ces limitations. Associer une interruption à l'achèvement d'un transfert lui permet de s'endormir et de relacher le processeur à chaque transaction. Lorsque le transfert est complet, l'interruption est prise en charge par un gestionnaire dont le rôle principal est de réveiller le processus endormi...

L'interaction entre le gestionnaire d'interruption `qspi_irqhandler()` et la méthode `qspi_write()` donne un bon aperçu de l'implémentation de `qspi_mod`. Le lecteur curieux de détails supplémentaires est bien sur invité à consulter les sources du pilote.

```
/* specific qspi device settings */
struct qspi_dev
{
        unsigned int in_use;
        unsigned short chip_select;
};

/* global qspi bus settings */
struct qspi_bus
{
        unsigned int open_count;
        unsigned int irq_exit_code;

        /* transfert information */
        unsigned char bits_per_transfer;
        unsigned char baud_rate;
        unsigned char clock_polarity_bit; /* 0 or 1 */
        unsigned char clock_phase_bit; /* 0 or 1*/
        unsigned char clock_delay;
        unsigned char transfer_delay;
```

```
        struct semaphore sem; /* to protect this struct */
        wait_queue_head_t queue;
        struct irq_info irq_info;

        /* global array of qspi device private structure */
        struct qspi_dev qspi_dev_array[QSPI_NB_DEVICE];
};

/* qspi bus device descriptor */
static struct qspi_bus qspi_bus_dev;
static void qspi_irqhandler (int irq, void *dev_id, struct pt_regs *regs)
{
        struct qspi_bus *bus_dev = (struct qspi_bus *) dev_id;

        /* case of a transfer abort by clearing the QDLYR[SPE] bit */
        if (QIR && ABORT_FLAG)
        {
                QIR |= ABORT_FLAG;
                bus_dev->irq_exit_code = ABORT_FLAG;
        }
        /* attempt to write to the ram entry during the transfer */
        if (QIR && WRITE_COLLISION_FLAG)
        {
                QIR |= WRITE_COLLISION_FLAG;
                bus_dev->irq_exit_code = WRITE_COLLISION_FLAG;
        }
        /* normal case of a transfer completion */
        if (QIR && COMPLETION_FLAG)
        {
                QIR |= COMPLETION_FLAG;
                bus_dev->irq_exit_code = COMPLETION_FLAG;
        }

        wake_up_interruptible (&bus_dev->queue);

        return;
}

size_t qspi_write (int device_num, char *kbuffer, unsigned int bytes_to_write)
{
        struct qspi_bus *bus_dev = &qspi_bus_dev;
        struct qspi_dev *dev;
        unsigned int bytes_write = 0;
        unsigned int transfer_size = 0;
        int i, retval = 0;

        /* is device_num a correct device number ? */
        if (!(device_num < QSPI_NB_DEVICE))
        {
                retval = -ENODEV;
                goto exit;
        }

        /* lock the bus */
        if (down_interruptible (&bus_dev->sem))
        {
                retval = -ERESTARTSYS;
                goto exit;
        }

        dev = &bus_dev->qspi_dev_array[device_num];
        if ((!dev) || (!dev->in_use))
        {
                retval = -ENODEV;
                goto exit_bus_sem;
        }
```

```
        if (!bytes_to_write)
                goto exit_bus_sem;

        /* limit the number of transfer to the queued spi lenght */
        bytes_to_write = (bytes_to_write < QSPI_MAX_TRANSFER_SIZE) ?
                bytes_to_write : QSPI_MAX_TRANSFER_SIZE;

        dbg("%s - %d bytes to write", __FUNCTION__, bytes_to_write);
        while (bytes_to_write)
        {
                transfer_size = (bytes_to_write < QSPI_QUEUE_SIZE) ?
                        bytes_to_write : QSPI_QUEUE_SIZE;

                dbg("%s - transfer size : %d",
                        __FUNCTION__, transfer_size);

                /* select the first command entry */
                QAR = 0x20;

                for (i = 0; i < transfer_size; i++)
                {
                        /* set device chip select (low or hight) */
                        /* 3 set the DT and DSCK bits... enable or disable
                         * the delay before and after transfer */
                        QDR = (3 << 12) | dev->chip_select;
                }

                /* select the first data entry */
                QAR = 0x00;

                for (i = bytes_write; i < (bytes_write + transfer_size); i++)
                {
                        QDR = kbuffer[i];
                }

                /* set the queue begining at the entry 0 and the end
                 * at nb_transfer */
                QWR = (transfer_size - 1) << 8;

                /* enable the transfer */
                QDLYR |= (1 << 15);

                /* sleep until a interruption send a wake up signal */
                retval = wait_event_interruptible (bus_dev->queue,
                                bus_dev->irq_exit_code);

                if (retval)
                {
                        retval = -ERESTARTSYS;
                        goto exit_bus_sem;
                }

                bus_dev->irq_exit_code = 0;

                bytes_to_write -= transfer_size;
                bytes_write += transfer_size;
        }

        retval = bytes_write;
        dbg("%s - %d bytes successfully writes",
                        __FUNCTION__, bytes_write);
exit_bus_sem :
        /* unlock the bus */
        up (&bus_dev->sem);

exit :
        return (retval);
```

```
}
```

5.2 Implémentation du block device `mmc_qspi_mod`

5.2.1 Introduction

L'objectif de cette partie est de présenter l'implémentation du module bloc `mmc_qspi_mod`. Mais auparavant, nous allons introduire le fonctionnement d'un *block device* ainsi que son implémentation pour un noyau 2.4. Il ne s'agit pas d'entrer dans les détails mais plutôt de fournir des recettes utiles à l'écriture d'un tel pilote. Le lecteur interessé par la théorie sur la gestion des périphériques blocs par un noyau pourra se référer à l'excellent article de la série *Conception d'OS*, paru dans le Linux Magazine numéro 80 [9]. D'un point de vue plus pratique, le livre *Linux Device Drivers* [8] est sans conteste une bible pour toute personne souhaitant se lancer dans le développement de pilotes blocs.

5.2.2 Généralités sur les *block devices*

Les périphériques blocs possèdent une structure très proche de celle des pilotes caractères. Ils exportent des méthodes à destination des processus utilisateurs. Ces méthodes peuvent être appelées *via* des fichiers spéciaux type bloc. Ces fichiers sont généralement contenus dans le répertoire `/dev`.

```
# ls -l /dev | grep mmc
brw-r--r--    1 0        0        254,   0 Nov 30 00:08 mmca
brw-r--r--    1 0        0        254,   1 Nov 30 00:08 mmca1
brw-r--r--    1 0        0        254,   2 Nov 30 00:08 mmca2
brw-r--r--    1 0        0        254,   3 Nov 30 00:08 mmca3
```

Le nombre majeur (ici 254) sert à désigner le pilote et le nombre mineur (ici de 0 à 3) est utilisé par le pilote en interne. Dans le cas des périphériques blocs, le mineur servira notamment à différencier les partitions. Les méthodes du pilote sont contenues dans une structure `struct block_device_operations`. À quelques exceptions près, cette dernière est très semblable à la structure `struct file_operations` utilisée par les pilotes caractères.

```
static struct block_device_operations mmc_blk_ops =
{
        open:           mmc_blk_open,
        ioctl:          mmc_blk_ioctl,
        release:        mmc_blk_release,
        check_media_change: mmc_blk_change,
        revalidate:     mmc_blk_revalidate
};
```

Cette structure est enregistrée auprès du VFS *via* la fonction `register_blkdev()`. Encore une fois, cette fonction remplit le même rôle que `register_chrdev()` pour les pilotes caractères. Pour les méthodes `open`, `release` et `ioctl`, rien de nouveau, elles permettent respectivement d'ouvir, fermer et configurer le périphérique. Les méthodes `check_media_change()` et `revalidate()` sont elles spécifiques au pilote bloc.

Le rôle de `check_media_change()` est de contrôler si le périphérique a été changé. Si tel est le cas cette fonction doit retourner 1, et 0 sinon. À noter que suivant le type de périphérique, ce genre de contrôle n'est pas toujours aisé à implémenter. S'il n'y a aucun moyen d'être sûr de la validité du périphérique, une politique par défaut correcte est de retourner systématiquement 1.

```
static int mmc_blk_change (kdev_t i_rdev)
{
        int minor = MINOR(i_rdev);
        int device_num = minor >> DEVICE_SHIFT;
```

```
        struct mmc_blk *dev;

        info("%s - minor : %d", __FUNCTION__, minor);

        if ((device_num < 0) ||
            (device_num >= nb_dev))
        {
                err("%s : DEV for minor %d don't exits",
                                __FUNCTION__, device_num);
                return (1);
        }

        dev = (struct mmc_blk *) (mmc_blk_dev + device_num);

        /* as we can't test if the MMC device has changed or not...
         * by default, we mark the device as expired */
        return (1);
}
```

La fonction `revalidate()` est appelée à chaque fois qu'un changement de périphérique est signalé. Par voie de conséquence, un retour de `check_media_change()` positif entraînera toujours un appel à la méthode `revalidate()` du pilote. Le travail de cette fonction est de réinitialiser l'ensemble des structures de données dépendantes du périphérique. Le plus souvent, cela se traduira uniquement par une relecture de la table des partitions.

```
static int mmc_blk_revalidate (kdev_t i_rdev)
{
        int minor = MINOR(i_rdev);
        int device_num = minor >> DEVICE_SHIFT;
        int part1 = (device_num << DEVICE_SHIFT) + 1;
        int npart = (1 << DEVICE_SHIFT) - 1;
        struct mmc_blk *dev;

        if ((device_num < 0) ||
            (device_num >= nb_dev))
        {
                err("%s : device %d don't exits",
                                __FUNCTION__, device_num);
                return (-ENODEV);
        }

        dev = (struct mmc_blk *) (mmc_blk_dev + device_num);

        /* first clear old partition information */
        memset (mmc_blk_gendisk.sizes + part1, 0, npart * sizeof(int));
        memset (mmc_blk_gendisk.part + part1, 0, npart * sizeof (struct hd_struct));
        mmc_blk_gendisk.part[device_num << DEVICE_SHIFT].nr_sects =
                mmc_blk_size[device_num << DEVICE_SHIFT]
                * mmc_blk_blocksizes[device_num << DEVICE_SHIFT]
                / mmc_blk_sectorsizes[device_num << DEVICE_SHIFT];

        /* then fill new info */
        info("(device %d) check partition", device_num);

        register_disk (&mmc_blk_gendisk,
                       MKDEV(mmc_blk_major, (device_num << DEVICE_SHIFT)),
                       npart, &mmc_blk_ops,
                       mmc_blk_partitions[device_num << DEVICE_SHIFT].nr_sects);

        return (0); /* still valid */
}
```

5.2.3 Quelques initialisations

Afin d'être utilisable, un pilote bloc doit également renseigner un certain nombre de variables globales. Le noyau les utilise pour s'informer sur les caractéristiques du périphérique (taille, etc...). Ces variables sont en fait des tableaux indexés par les nombres majeurs et mineurs des pilotes. Chaque pilote doit donc renseigner ces tableaux pour l'indice correspondant à son majeur. Les nombres mineurs permettront de désigner les différentes partitions.

- `int blk_size[][]` : taille du périphérique en nombre de blocs
- `int blkblk_size[][]` : taille d'un bloc en octets
- `int hardsect_size[][]` : taille d'un secteur en octets. Dans le cas d'une MMC, un secteur vaudra 512 octets.
- `read_ahead[]` : nombre de secteurs pouvant être lus par anticipation. Un `read_ahead` important permet d'améliorer les performances sur les périphériques aux temps d'accès assez longs. Ce tableau dépend uniquement du major du périphérique. Il est évident que l'anticipation est liée à des contraintes matérielles et ne dépend pas du partitionnement.

La distinction entre bloc et secteur n'est pas toujours évidente. Du point de vue du VFS on parlera de bloc, alors que du point de vue du matériel, on parlera de secteur. Il est fortement conseillé d'initialiser `blkblk_size[][]` à 1024 octets. Une valeur différente gênera le noyau lors du calcul de la taille du périphérique. Par exemple, pour une taille de bloc de 512 octets, le noyau divisera régulièrement la valeur contenue dans `blk_size[][]` par 2.

5.2.4 La méthode `ioctl()`

La méthode `ioctl()` d'un pilote bloc mérite également qu'on si attarde quelques instants. Le noyau et certains programmes utilisateurs, comme les outils de partitionnement, peuvent utiliser cette méthode. Elle doit être capable de fournir une réponse à quelques commandes :

- `BLKGETSIZE` : demande le renvoi de la taille du périphérique en octets
- `BLKRRPART` : demande la relecture de la table des partitions du périphérique
- `HDIO_GETGEO` : demande le renvoi de la géométrie du périphérique. Cette commande n'a de sens que pour les périphérique de type disque dur. En effet, une MMC ne dispose ni de plateaux, ni de têtes. On se cantonera donc à retourner une valeur plausible...

Pour toutes les autres commandes, le noyau met à disposition une méthode `blk_ioctl`. Elle retournera des valeurs par défaut appropriées.

Voici l'implémentation de la méthode `ioctl()` telle que proposée par le pilote `mmc_qspi_mod` :

```
static int mmc_blk_ioctl (struct inode *inode, struct file *file,
        unsigned int cmd, unsigned long arg)
{
        int minor = MINOR(inode->i_rdev);
        int device_num = minor >> PAGE_SHIFT;
        long size;
        struct hd_geometry geo;

        if ((device_num < 0) ||
            (device_num >= nb_dev))
        {
                err("%s : DEV for minor %d don't exits",
                                __FUNCTION__, device_num);
                return -ENODEV;
        }

        switch (cmd)
        {
                /* return device size in sector */
                case BLKGETSIZE :

                        if (!access_ok (VERIFY_WRITE, (void *) arg, sizeof (unsigned long)))
                                return -EFAULT;

                        size =  mmc_blk_size[minor] *
```

```
                                mmc_blk_blocksizes[minor] /
                                mmc_blk_sectorsizes[minor];

                        if (copy_to_user ((unsigned long *) arg, &size, sizeof (unsigned long)))
                                return (-EFAULT);

                        return (0);

                /* read again partion table */
                case BLKRRPART :

                        return (mmc_blk_revalidate (inode->i_rdev));

                /* return the device geometry */
                case HDIO_GETGEO :

                        if (!access_ok (VERIFY_WRITE, (void *) arg, sizeof (unsigned long)))
                                return -EFAULT;

                        size =  mmc_blk_size[device_num] *
                                mmc_blk_blocksizes[device_num] /
                                mmc_blk_sectorsizes[device_num];

                        geo.cylinders = (size & ~0x3f) >> 6;
                        geo.heads = 4;
                        geo.sectors = 16;
                        geo.start = 4;

                        if (copy_to_user ((struct hd_geometry *) arg, &geo,
                                        sizeof (struct hd_geometry)))
                                return (-EFAULT);

                        return (0);

                /* let the kernel handle for us the unkown command */
                default :

                        return (blk_ioctl (inode->i_rdev, cmd, arg));
        }

        return (-ENOTTY);
}
```

5.2.5 File d'attente des requêtes

La file d'attente des requêtes est un composant fondamental de tout pilote bloc. Les opérations de lectures ou d'écritures à destination du périphérique vont être présentées au *driver* sous la forme de requêtes (`struct request`). Ces requêtes sont elles mêmes placées dans la file d'attente des requêtes du pilote chargé de les traiter. Cette file se présente sous la forme d'une structure `struct request_queue`.

```
#include <linux/blkdev.h>

typedef struct request_queue request_queue_t;

struct request_queue
{
        [...]
        struct list_head queue_head;
        request_fn_proc * request_fn;
        make_request_fn * make_request_fn;
        [...]
};
```

Le pointeur `queue_head` désigne la tête de la liste chaînée des requêtes. La structure `struct request_queue` contient également de nombreux pointeurs de fonctions. Nous nous intéresserons

tout particulièrement aux fonctions `request()` et `make_request()` désignées respectivement par les pointeurs `request_fn` et `make_request_fn`. Le pilote initialise et libère une file de requêtes en utilisant les fonctions suivantes :

```
#include <linux/blkdev.h>

extern void blk_init_queue(request_queue_t *, request_fn_proc *);
extern void blk_cleanup_queue(request_queue_t *);
```

Exemple d'initialisation par un pilote de sa file par défaut :

```
blk_init_queue(BLK_DEFAULT_QUEUE(majeur_du_pilote), fonction_request_du_pilote);
```

Une remarque importante peut être faite au sujet de la concurrence à laquelle la file d'attente des requêtes est sujette. Toute manipulation sur cette file devra être réalisée après acquisition du verrou global `io_request_lock`. À noter que lorsque la méthode `request()` du pilote est appelée, le noyau s'est déjà chargé de l'obtention du verrou...

5.2.6 La méthode `request()`

Le lecteur attentif aura probablement remarqué la principale originalité de la structure `struct block_device_operations`. Il s'agit de l'absence de méthodes `write()` ou `read()`. En effet les entrées-sorties sur les périphériques blocs ne sont pas directes et sont optimisées par l'utilisation des tampons et de pages du noyau [9]. Les processus ne peuvent donc pas accéder aux fonctions d'entrées-sorties du pilote directement. Pour les accès au périphérique, le pilote met à disposition du noyau une unique méthode appelée `request()`. Lorsque le noyau souhaite échanger des données avec le matériel, c'est cette fonction qu'il sollicite.

```
void (request_fn_proc) (request_queue_t *q);
```

Elle reçoit en paramètre d'entrée un pointeur sur la file d'attente des requêtes du pilote.

Le rôle de la méthode `request()` est de traiter les demandes du noyau pour lire ou écrire des blocs de données sur le périphérique. Une requête se présente sous la forme d'une structure `struct request`.

```
#include <linux/blkdev.h>

struct request {
        [...]
        struct list_head queue;
        kdev_t rq_dev;
        int cmd;                /* READ or WRITE */
        unsigned long sector;
        unsigned long nr_sectors;
        char * buffer;
        struct buffer_head * bh;
        request_queue_t *q;
        [...]
};
```

Cette structure est principalement une liste chaînée de structures `struct buffer_head`.

```
#include <linux/fs.h>

struct buffer_head {
        [...]
        unsigned short b_size;          /* block size */
        kdev_t b_rdev;                  /* Real device */
        /* Time when (dirty) buffer should be written */
        unsigned long b_flushtime;
        struct buffer_head *b_reqnext;  /* request queue */
```

```
        char * b_data;                  /* pointer to data block */
        void (*b_end_io)(struct buffer_head *bh, int uptodate); /* I/O completion */
        unsigned long b_rsector;        /* Real buffer location on disk */
        [...]
};
```

Un élément de type `struct buffer_head` désigne un bloc de données que le pilote doit traiter de manière atomique. Chaque tampon de données manipulé par les caches du noyau est représenté par une telle structure. Cela permet au noyau de conserver une information sur la position physique des données sur le périphérique. En effet, des blocs adjacents sur le périphérique ne le sont plus forcément en mémoire après manipulation. Le noyau va donc se servir des informations contenues dans les différentes structures `struct buffer_head` pour soumettre au pilote des requêtes concernant des blocs proches ou adjacents. Ce modèle est particulièrement adapté pour des périphériques de stockage comme les disques durs. En effet le noyau va tenter de minimiser les déplacements de la tête de lecture du disque. Pour cela des algorithmes dits "en acenseur" sont utilisés pour construire les requêtes. L'idée est de déplacer la tête de lecture du disque le plus longtemps possible dans la même direction. Comme nous le verrons plus tard, ces optimisations ne sont pas spécialement intéressantes dans le cas d'une MMC.

À noter qu'une structure `struct request` reflète les caractéristiques du premier tampon `struct buffer_head` de la liste. Les champs `buffer`, `sector` et `nr_sector` d'une requête sont en fait une simple recopie des champs `b_data`, `b_sector` et `b_size` du premier tampon sur la liste. Le champ `b_end_io` de la structure `struct buffer_head` pointe sur une fonction de complétion. Cette dernière doit être appelée chaque fois qu'une opération d'entrée/sortie sur ce tampon s'achève. Cela permet de signaler au noyau le résultat de l'opération. La conséquence peut être le réveil d'un processus en attente.

Une dernière précision importante au sujet de la méthode `request()` est que cette dernière n'est pas forcément appelée dans l'environnement d'un processus. En effet, les accès aux périphériques blocs sont asynchrones et peuvent être traités dans le contexte d'une interruption matérielle. La fonction `request()` est donc soumise aux restrictions du genre. Elle ne doit pas de manière directe ou indirecte provoquer un réordonnancement des processus (appel à `schedule()`). Cela signifie qu'elle doit être exécutée de manière atomique. Ce qui exclut par exemple l'usage de sémaphores...

5.2.7 Gestion des partitions

Pour conclure cette rapide introduction sur les *block devices*, nous allons montrer comment un pilote peut organiser le partitionnement de ses périphériques.
Tout d'abord, l'emploi du nombre mineur doit être précisé. Un pilote est censé pouvoir gérer plusieurs périphériques de même type simultanément. Chaque périphérique est lui même susceptible de contenir plusieurs partitions. Le nombre mineur doit donc refléter ces deux informations. Une idée est de séparer l'octet du nombre mineur en deux groupes de 4 bits. Les 4 bits de poids fort désigneront le périphérique et les 4 bits de poids faible, les numéros de partition.

Une des tâches supplémentaires incombant à un pilote supportant les partitions est l'initialisation d'une structure `struct gendisk` :

```
#include <linux/genhd.h>

struct gendisk {
        int major;                      /* major number of driver */
        const char *major_name;         /* name of major driver */
        int minor_shift;                /* number of times minor is shifted to
                                           get real minor */
        int max_p;                      /* maximum partitions per device */

        struct hd_struct *part;         /* [indexed by minor] */
        int *sizes;                     /* [idem], device size in blocks */
```

```
        int nr_real;                    /* number of real devices */

        void *real_devices;             /* internal use */
        struct gendisk *next;
        struct block_device_operations *fops;

        devfs_handle_t *de_arr;         /* one per physical disc */
        char *flags;                    /* one per physical disc */
};

extern void add_gendisk(struct gendisk *gp);
extern void del_gendisk(struct gendisk *gp);
```

La signification de quelques uns des champs de la structure `struct gendisk` mérite d'être précisée :

- le champ `major_name` est en fait le nom du périphérique tel qu'il apparaîtra dans les logs du noyau. Le noyau complètera automatiquement ce nom par une lettre et un chiffre. La lettre désignera le périphérique et le chiffre la partition. Par exemple, si `major_name` est initialisé avec la chaîne de caractères `mmc`, alors `mmca2` désignera la deuxième partition du premier périphérique,
- `sizes` doit contenir exactement les mêmes informations que `blk_size[major][]`. Le pilote pourra d'ailleur s'arranger pour les faire pointer sur la même zone mémoire,
- le champ `minor_shift` indique au noyau l'offset qu'il doit appliquer au mineur pour récupérer le numéro du périphérique. Si les 4 bits de poids forts du mineur contiennent cette information, `minor_shift` doit être initialisé à 4,
- le pointeur `part` contient l'adresse de la table des partitions du périphérique. Le pilote devra juste allouer cette structure. Il sera très rarement amené à la consulter.

Une fois cette structure initialisée, le pilote n'a plus qu'à l'ajouter à la liste chaînée des structures `struct gendisk` en utilisant la fonction `add_gendisk()`. Le suite est très simple. La gestion des partitions est complètement transparente pour le pilote. Il devra juste de temps en temps demander au noyau de relire la table des partions sur le périphérique pour mettre à jour sa structure `struct hd_struct`. Pour cela il suffira au pilote d'utiliser la fonction `register_disk()`.

```
#include <linux/blkdev.h>

extern void register_disk(struct gendisk *dev, kdev_t first,
        unsigned minors, struct block_device_operations *ops, long size);
```

Un appel à `register_disk()` est toujours suivi par un appel à la méthode `request()` du pilote. En effet, le noyau doit lire sur le périphérique pour en extraire la table des partitions. Les appels à `register_disk()` peuvent être réalisés pendant l'initialisation du périphérique, depuis la fonction `init()` ou depuis la méthode `open()`. La méthode `revalidate()`, appelée suite à un changement de périphérique, est également un bon emplacement pour forcer la relecture de la table des partitions

5.2.8 Présentation du module `mmc_qspi_mod`

Il est maintenant temps de voir comment `mmc_qspi_mod` fait usage des techniques exposées précédement.

Tout d'abord, comme nous l'avons précisé plus tôt, l'utilisation des files d'attente de requêtes permet souvent d'améliorer les performances d'un pilote bloc. En mode SPI, la MMC propose des commandes de transfert dites multiblocs. Il est ainsi possible de transférer en une opération plusieurs blocs physiquement adjacents sur la carte. Cependant, notre pilote ne supporte pas encore ces commandes. Les opérations de lectures-écritures ne se font donc que par transferts de secteur unique (512 octets). Recevoir des requêtes sur des blocs adjacents n'offre aucun gain de temps. Pire encore, construire les requêtes en classant les structures `struct buffer_head` en fonction de la

position des données sur le disque ralentit notablement les transferts. Partant de ce constat, nous avons décidé de ne pas pas utiliser de file d'attente de requêtes. Le pilote n'implémente donc pas de méthode `request()`. À la place, il définit une fonction `make_request ()`. Habituellement cette fonction construit les requêtes destinées au pilote puis appelle la méthode `request()` de ce dernier. Le noyau met à disposition des pilotes une fonction `make_request()` générique. Nous allons tout simplement remplacer cette fonction par la nôtre. Son travail sera de transférer directement les données sur la MMC. Cette technique est couramment employée dans l'écriture des pilotes pour des périphériques ne tirant pas avantage des optimisations. Un pilote pour `ramdisk` constitue un bon exemple. Les transferts sont pour lui de simples copies mémoires. Construire des requêtes le ralentirait considérablement. Un exemple d'implémentation de pilote `ramdisk` est consultable dans le fichier ∼`/linux/drivers/block/rd.c` des sources du noyau Linux.

```
#include <linux/blkdev.h>

typedef int (make_request_fn) (request_queue_t *q, int rw, struct buffer_head *bh);
extern void blk_queue_make_request(request_queue_t *, make_request_fn *);
```

La fonction `blk_queue_make_request` est utilisée pour associer une fonction `make_request ()` à une file d'attente de requête :

```
blk_queue_make_request (BLK_DEFAULT_QUEUE(mmc_blk_major),
                mmc_blk_make_request);
```

La fonction `make_request()` chargée d'exécuter les lectures et les écriture sur la MMC constitue le cœur du pilote. Elle se trouve dans l'élément `mmc_core` du module `mmc_qspi_mod`. Voici son implémentation :

```
static int mmc_blk_make_request (request_queue_t *queue, int rw, struct buffer_head *bh)
{
        int minor = MINOR(bh->b_rdev);
        int device_num = minor >> DEVICE_SHIFT;
        struct mmc_blk *dev;
        unsigned char *block_addr_ptr[MAX_BLK_NB];
        unsigned long block_addr[MAX_BLK_NB];
        int i, size = 0, nb_block, status = 1, count;

        if ((device_num < 0) ||
            (device_num >= nb_dev))
        {
                err("%s : device %d don't exits",
                                __FUNCTION__, device_num);
                status = 0;
                goto exit;
        }

        dev = (struct mmc_blk *) (mmc_blk_dev + device_num);

        /* only handle modulo BLKSIZE_SIZE request */
        if ((bh->b_size%DFLT_HARDSECT_SIZE) != 0)
        {
                err("%s - can't handle a %d bytes request (not a x%d size)",
                                __FUNCTION__, bh->b_size, DFLT_BLKSIZE_SIZE);
                status = 0;
                goto exit;
        }

        nb_block = bh->b_size / DFLT_HARDSECT_SIZE;

        /* don't allow to treat more than MAX_BLK_NB blocks
         * nb_block depend from MMC size and from filesystem type */
        if (nb_block > MAX_BLK_NB)
        {
                err("%s - request to handle %d blocks (max = %d)",
                                __FUNCTION__, nb_block, MAX_BLK_NB);
```

```
                status = 0;
                goto exit;
        }

        for (i = 0; i < nb_block; i++)
        {
                /* use DFLT_BLKSIZE_SIZE instead of
                 * mmc_blk_blocksizes[device_num]
                 * hard MMC block == 512 */
                block_addr[i] = (mmc_blk_partitions[minor].start_sect +
                         bh->b_rsector) * mmc_blk_sectorsizes[device_num] +
                        (i * DFLT_HARDSECT_SIZE);

                block_addr[i] = __cpu_to_be32(block_addr[i]); /* endianness */

                block_addr_ptr[i] = (unsigned char *) &block_addr[i];

                /* don't allow MMC overflow access */
                if ((bh->b_size) > (mmc_blk_partitions[minor].nr_secta
                                * mmc_blk_sectorsizes[device_num]))
                {
                        err("%s : device %d overflow",
                                        __FUNCTION__, device_num);
                        status = 0;
                        goto exit;
                }
        }

#if CONFIG_HIGHMEM
        bh = create_bounce (rw, bh);
#endif

        switch (rw)
        {
                case READ :
                case READA :

                for (i = 0; i < nb_block; i++)
                {
                        count = 2;
                        do
                        {
                                size = read_block (block_addr_ptr[i][0],
                                                block_addr_ptr[i][1],
                                                block_addr_ptr[i][2],
                                                block_addr_ptr[i][3],
                                                bh->b_data + (i * DFLT_HARDSECT_SIZE));
                        }
                        while ((size != DFLT_HARDSECT_SIZE) && (count--));

                        if (size != DFLT_HARDSECT_SIZE)
                        {
                                err("failed to read block %d - %d - %d - %d",
                                                block_addr_ptr[i][0],
                                                block_addr_ptr[i][1],
                                                block_addr_ptr[i][2],
                                                block_addr_ptr[i][3]);
                                status = 0;
                                goto exit;
                        }
                }
                goto exit;
                        break;

                case WRITE :

                refile_buffer (bh);
```

```
                for (i = 0; i < nb_block; i++)
                {
                        count = 2;
                        do
                        {
                                size = write_block (block_addr_ptr[i][0],
                                        block_addr_ptr[i][1],
                                        block_addr_ptr[i][2],
                                        block_addr_ptr[i][3],
                                        bh->b_data + (i * DFLT_HARDSECT_SIZE));
                        }
                        while ((size != DFLT_HARDSECT_SIZE) && (count--));

                        if (size != DFLT_HARDSECT_SIZE)
                        {
                                err("failed to write block %d - %d - %d - %d",
                                                block_addr_ptr[i][0],
                                                block_addr_ptr[i][1],
                                                block_addr_ptr[i][2],
                                                block_addr_ptr[i][3]);
                                status = 0;
                                goto exit;
                        }
                }
                mark_buffer_uptodate (bh, 1);
                goto exit;
                break;

                default :
                        status = 0;
                        goto exit;
                        break;
        }

exit :
        bh->b_end_io (bh, status);
        return (0);
}
```

Afin d'illustrer les transferts de données avec la MMC, nous allons présenter la fonction `read_block()` ainsi que quelques unes des fonctions qu'elle utilise. Comme nous venons de le voir, `read_block()` est appelée par la fonction `mmc_blk_make_request()` chaque fois que le pilote doit procéder à une lecture sur la MMC. Elle est définie dans l'élément `mmc_qspi` du module `mmc_qspi_mod`

```
int read_block (unsigned char a1, unsigned char a2, unsigned char a3, unsigned char a4, char *block)
{
        int nb_bytes_read, retval;
        unsigned int count = 10;
        unsigned char answer, status[2];
        unsigned short checksum, block_crc;

        dbg("%s : block %d %d %d %d", __FUNCTION__, a1, a2, a3, a4);

        if (!block)
        {
                err("%s - first argument is a NULL pointer", __FUNCTION__);
                return (-EINVAL);
        }

        mmc_cmd_array[CMD_17].a1 = a1;
        mmc_cmd_array[CMD_17].a2 = a2;
        mmc_cmd_array[CMD_17].a3 = a3;
        mmc_cmd_array[CMD_17].a4 = a4;

        /* send the single read block command */
        retval = send_cmd (CMD_17, &answer);
```

```
if (retval < 0)
{
        err("%s - failed to send CMD_17", __FUNCTION__);
        return (retval);
}


while (((answer & 0xff) != 0xfe) && (count--))
{
        retval = read_answer (&answer);
        if (retval < 0)
        {
                return (retval);
        }
}

if (answer != 0xfe)
{
        err("%s - CMD_17 failure (answer = 0x%x)",
                        __FUNCTION__, answer & 0xff);
        return (-EPROTO);
}

retval = qspi_read (MMC_DEVICE, block,
                MMC_BLOCK_SIZE * sizeof (unsigned char));
if (retval < 0)
{
        err("%s - failed to read block", __FUNCTION__);
        return (retval);
}
if (retval != MMC_BLOCK_SIZE)
{
        err("%s - %d bytes read instead of %d",
                __FUNCTION__, retval, MMC_BLOCK_SIZE);
}
nb_bytes_read = retval;

/*  read the checksum (2 bytes) */
retval = qspi_read (MMC_DEVICE, (char *) &checksum,
                sizeof (unsigned short));
if (retval < 0)
{
        err("%s - failed to read checksum", __FUNCTION__);
        return (retval);
}
if (retval != sizeof (unsigned short))
{
        err("%s - failed to read checksum", __FUNCTION__);
        return (-EIO);
}
/* compute the block checksum and compare with
 * the checksum send by the MMC */
block_crc = (unsigned short) crc16 (block, MMC_BLOCK_SIZE);
block_crc = __cpu_to_be16(block_crc); /* endianness */
if (block_crc != checksum)
{
        err("%s - bad checksum : compute %d instead of %d",
                        __FUNCTION__,
                        block_crc,
                        checksum);
        return (-EIO);
}
/* take a look on the MMC status */
retval = get_status (status);
if (retval < 0)
{
        err("%s - unable to get MMC status", __FUNCTION__);
```

```
                return (retval);
        }
        if ((status[0]) || (status[1]))
        {
                err("%s - status 0x%x 0x%x", __FUNCTION__,
                                status[0] & 0xff, status[1] & 0xff);
                return (-EPROTO);
        }
        return (nb_bytes_read);
}

static int read_answer (unsigned char *answer)
{
        int retval = 0;
        unsigned int count = 1000;

        do
        {
                retval = qspi_read (MMC_DEVICE, answer, sizeof (unsigned char));
                if (retval < 0)
                        return (retval);

                /* must handle the case retval == 0 */
        }
        while ((*answer == 0xff) && (count--));
        /* if the MMC is not ready, 0xff is returned as answer...
         * we retry count times... */

        if (!count)
                return (-EBUSY);

        dbg("%s - answer : 0x%x count : %d", __FUNCTION__, *answer & 0xff, 1000 - count);

        return (0);
}

static int send_cmd (unsigned int cmd_num, unsigned char *answer)
{
        int retval;

        /* test cmd_num */
        mmc_cmd_array[cmd_num].checksum = 0;
        mmc_cmd_array[cmd_num].checksum =
                crc7 (&mmc_cmd_array[cmd_num].cmd, 5);

        dbg("%s - cmd %d - checksum : 0x%x", __FUNCTION__,
                        cmd_num, mmc_cmd_array[cmd_num].checksum & 0xff);

        retval = qspi_write (MMC_DEVICE,
                        (char*) &mmc_cmd_array[cmd_num].cmd, CMD_SIZE);
        if (retval < 0)
        {
                return (retval);
        }
        if (retval != CMD_SIZE)
        {
                err("%s - incomplete command send (%d bytes instead of %d)",
                                __FUNCTION__, retval, CMD_SIZE);
                return (-EIO);
        }
        return (read_answer (answer));
}
```

6 Améliorations du système de base

Nous avons, à force de tester notre pilote dans diverses configurations, constaté la présence de quelques erreurs de communications, particulièrement visibles lors du stockage de fichiers volumineux (qui ne sont pas nécessairement le standard dans des applications embarquées mais sont devenues courants dans une utilisation quotidienne d'un ordinateur). Afin d'identifier ces erreurs et d'éventuellement les corriger, nous avons implémenté plusieurs méthodes que nous décrirons ici :

- implémentation du code de redondance cyclique (CRC) visant à valider les blocs lus
- implémentation d'une lecture après toute écriture sur la carte mémoire : cette méthode nous permet de valider le stockage de données sans nécessairement implémenter toute la chaîne de validation qui va suivre
- implémentation des CRC en lecture comme en écriture qui nécessite par conséquent le calcul du CRC lors de l'envoi de commandes. Rappelons que par défaut la vérification de CRC est désactivée en mode SPI et nous la réactivons manuellement au moyen de la commande CMD59. Ceci nécessite donc, en plus de l'implémentation du CRC16 utilisé lors de transissions de blocs de données tel que validé dans la phase précédente, d'implémenter le CRC7 utilisé lors de la transmission de commandes.

L'intérêt additionnel de valider les communications certifiées par CRC est d'offrir la perspective d'une implémentation complète du mode natif de communication des MMC, plus robuste et plus rapide que le mode SPI (mais nécessitant toutes les vérifications de CRC fastidieuses à valider sans passer par le mode SPI).

6.1 Validation des blocs de données : CRC16

La vérification qui implique le moins de changement au code existant est le calcul du CRC associé à une transaction en lecture de la MMC vers le processeur. En effet la MMC nous fournit toujours un CRC valide, même en mode SPI où le calcul du CRC n'est pas requis. Ainsi, tester le CRC en fin de CMD17 (lecture) n'implique aucune modification de notre code existant si ce n'est implémenter la version appropriée de CRC16 et comparer ce résultat à celui fourni par la MMC.

Malgré les excellentes présentations des CRC présentées par ailleurs dans ces pages pas Yann Guidon [10, 11], nous nous sommes contentés de trouver sur le web une implémentation compréhensible du code que nous recherchions [12]. La seule subtilité en terme d'implémentation portable à la fois sur processeur Coldfire et Intel est de tenir compte de l'endianness du résultat qui tient sur 2 octets. L'ajout de la fonction `htons()` après calcul du CRC se charge de convertir le résultat issu du processeur en un format compréhensible par la MMC.

6.2 Validation des écritures

Après avoir implémenté une méthode de contrôle des lectures, nous désirons valider les écritures de blocs de données. Nous verrons plus loin que l'activation de tous les contrôles par la MMC nécessite de valider d'autres méthodes de calcul (CRC7) : nous allons donc dans un premier temps nous contenter d'implémenter la méthode – relativement inefficace mais la plus simple – de relire tout bloc immédiatement après son écriture pour en vérifier la validité.

Cette correction n'a pas donné satisfaction car pour des raisons que nous n'expliquons pas, nous sommes capables d'écrire plusieurs MB sur une MMC connectée au bus QSPI du Coldfire sans qu'aucun bloc erroné ne soit détecté par comparaison d'une lecture après chaque écriture. Cependant, après avoir démonté (`umount`) et remonté le système de fichiers sur la MMC, le fichier est apparu corrompu. La MMC n'a pourant pas de mémoire cache en interne à notre connaissance.

6.3 Validation des commandes : CRC7

Ayant validé le code de calcul du CRC16 en constatant que notre calcul correspond toujours à celui de la MMC tel que fourni en fin de lecture d'un bloc, nous proposons finalement d'activer

totalement le contrôle de flux par la MMC. Ceci nécessite un calcul additionel : en plus de valider les transactions de blocs de données, la MMC valide aussi les commandes qu'elle reçoit au moyen d'un CRC plus cours, le CRC7.

Le CRC7 définit un code de vérification de petites quantités de données – dans notre cas 40 bits – au moyen d'un code sur 7 bits. Ce code est transmis à la carte sous forme de bits de poids fort d'un octet complétant la transmission dont le bit de poids le plus faible est défini à 1 [1, chap.5.7.2, p.55]. Ainsi, nous implémentons dans un premier temps le CRC7, puis devons multiplier par 2 le résultat et ajouter 1 avant d'avoir un octet validant la transaction. Nous avons repris un code simple de calcul de CRC7 (table 1) tel que disponible dans la littérature [13], bien que ce calcul courant dans les transmissions gagnerait probablement à être optimisé.

```
#include <stdio.h>
#include <stdlib.h>
/*
Name  Polynomial  Representations: Normal or Reverse (Normal of Reciprocal)
CRC-1             x + 1 (Used in hw, also known as parity bit) 0x1 or 0x1 (0x1)
CRC-5-USB   x5 + x2 + 1 (used in USB token packets)     0x05 or 0x14 (0x9)
CRC-7       x7 + x3 + 1 (used in some telecom systems)  0x09 or 0x48 (0x11)
CRC-8   x8 + x2 + x + 1                                 0x07 or 0xE0 (0xC1)
CRC-12  x12 + x11 + x3 + x2 + x + 1 (used in telecom)   0x80F or 0xF01 (0xE03)
CRC-16-CCITT     x16 + x12 + x5 + 1                  0x1021 or 0x8408 (0x0811)
CRC-16-IBM        x16 +x15 + x2 + 1                  0x8005 or 0xA001 (0x4003)
*/

/* Computation of CRC-7 polynom (x**7+x**3+1), used in MMC cmds and responses */
unsigned char crc7(unsigned char *pc)
{unsigned int i,ibit,c,crc=0x00,len=5;

 for (i=0; i<len; i++,pc++)
    {c = *pc;
     for (ibit = 0; ibit < 8; ibit++)
        {crc <<= 1;
         if ((c ^ crc) & 0x80) crc ^= 0x09;
         c <<= 1;
        }
     crc &= 0x7F;
 }
 return ((crc<<1)+1);
}


int main()
{char data[5]={0x40,0x00,0x00,0x00,0x00}; // CMD0
 printf("0x%x\n",crc7(data));
 return(0);
}
```

Tab. 1 – Implémentation naïve du calcul de CRC7 validant la transmission de commandes à la MMC.

Le seule façon de valider ce code est de calculer le CRC7 de la commande CMD0 – la seule pour laquelle nous connaissions le résultat. Nous constatons qu'appliquer le code présenté en table 1 aux octets {0x40 0x00 0x00 0x00 0x00} donne bien le résultat escompté, à savoir 0x95. Par ailleurs, nous constatons qu'en activant le contrôle de tous les flux (blocs et commandes) au niveau de la MMC, compléter les commandes avec ce CRC7 résulte en un acquittement de la commande par la carte.

L'activation de la vérification de tous les CRC lors des transactions avec la carte s'obtient en insérant la commande CMD59 (CRC_ON_OFF) juste après l'acquittement de CMD1 par la MMC. Désormais, tous les CRC que nous avions jusqu'ici ignoré devront être validés après chaque transaction.

7 Utilisation

L'utilisation du *block device* nécessite plusieurs étapes :

- la création du point d'entrée approprié dans le répertoire `/dev`. Nous avons choisi un block device de nombre majeur 254 nommé `mmca` (de nombre mineur 0) et les partition `mmca`i avec $i \in [1:4]$ le nombre mineur. Sur PC pour l'implémentation sur port parallèle du protocole SPI, le block device se crée par `mknod /dev/mmca b 254 0`,
- la disponibilité d'un outil de formattage. Alors que toutes les versions de `mkfs` sont disponibles sur PC, il nous faut cross-compiler une version de cet outil à uClinux sur processeur Coldfire. Par habitude et limités par les formats de fichiers reconnus par le noyau par défaut d'uClinux, nous avons sélectionné `mke2fs` pour formatter notre MMC en format ext2 [14],
- si l'ensemble de la MMC est utilisée sans partition, formatter la MMC complète sans y créer de table de partition : `mke2fs -I /dev/mmca` sous uClinux, ou sans l'option `-I` sur PC et répondre par l'affirmative à la confirmation de formater l'ensemble du périphérique,
- éventuellement créer des partitions sur `/dev/mmca` au moyen de `fdisk` et les formater individuellement en y accédant par `mke2fs /dev/mmca`i, $i \in [1:4]$,
- finalement, monter le nouveau périphérique en un point d'accès : le seul répertoire accessible en écriture sous uClinux tel que fourni installé sur le Coldfire 5282 du circuit SSV étant `/var`, et `/mnt` étant déjà occupé pour un accès NFS au PC de développement, nous avons choisi de `mkdir /var/mmc` suivi de `mount /dev/mmca /var/mmc` ou `mount /dev/mmca`i `/var/mmc` si la partition i a été initialisée. Notez qu'il arrive souvent que l'initisalisation de la MMC échoue à la première tentative : relancer la commande `mount` résoud ce problème dès le second essai.

Nous obtenons ainsi un nouveau répertoire dans notre arborescence donnant accès à l'espace mémoire de la MMC. Il est fondamental de penser à `umount /var/mmc` avant de retirer la carte si l'on veut conserver la cohérence du système de fichier de la MMC car nous constatons rapidement qu'une utilisation intensive du cache est effectuée. Cette utilisation de la mémoire est particulièrement appréciable compte tenu de la lenteur d'accès à la MMC.

À titre d'exemple, voici le fonctionnement de notre driver contrôlant un montage sur le port parallèle du Libretto 100CT, après avoir `insmod pport_spi.o` et `insmod mmc_qspi.o` :

1. nous commençons par partitionner notre carte afin de mettre en pratique le bout de code associé à la gestion de cette option du block device :

```
libretto:/home/jmfriedt# fdisk /dev/mmca
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that, of course, the previous
content won't be recoverable.


The number of cylinders for this disk is set to 3972.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-3972, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-3972, default 3972): +30M

Command (m for help): n
Command action
   e   extended
```

```
   p   primary partition (1-4)
p
Partition number (1-4): 2
First cylinder (962-3972, default 962):
Using default value 962
Last cylinder or +size or +sizeM or +sizeK (962-3972, default 3972): +40M

Command (m for help): p

Disk /dev/mmca: 4 heads, 16 sectors, 3972 cylinders
Units = cylinders of 64 * 512 bytes

    Device Boot    Start       End    Blocks   Id  System
/dev/mmca1             1       961     30744   83  Linux
/dev/mmca2           962      2242     40992   83  Linux

Command (m for help):
```

afin de créer deux partitions, l'une de 30 MB et l'autre de 40 MB, qui ne remplissent donc pas l'ensemble de notre carte de 128 MB. Le nombre de blocs a bien été reconnu, l'architecture de la carte est cohérente avec ce que nous nous attendons à trouver.

2. Il nous faut ensuite formater chacune de ces partitions. Si nous avions décidé de travailler sur l'ensemble de la carte sans réaliser de partition, une commande de type `mke2fs /dev/mmca` aurait convenu. Noter que le nombre d'octets par blocs – ici 1024 – est dépendant de la géométrie du block device et doit donc être alloué dynamiquement en fonction des requêtes du VFS :

```
libretto:/home/jmfriedt# mke2fs /dev/mmca1
mke2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
7712 inodes, 30744 blocks
1537 blocks (5.00%) reserved for the super user
First data block=1
4 block groups
8192 blocks per group, 8192 fragments per group
1928 inodes per group
Superblock backups stored on blocks:
        8193, 24577

Writing inode tables: done
Writing superblocks and filesystem accounting information: done
libretto:/home/jmfriedt# mount /dev/mmca1 /mnt/mmca1/
libretto:/home/jmfriedt# mount /dev/mmca2 /mnt/mmca2/
libretto:/home/jmfriedt# df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/libretto/rootfs    307184    265164     42020  87% /
/dev/libretto/var       102392     92928      9464  91% /var
/dev/libretto/usr      1261524    854044    407480  68% /usr
/dev/hda1               104792     38288     66504  37% /boot
/dev/mmca1               29765        13     28215   1% /mnt/mmca1
/dev/mmca2               39659        13     37597   1% /mnt/mmca2
```

3. ayant monté dans un point d'accès approprié notre nouveau bloc – et vérifié ainsi que le formatage a bien été reconnu puisque GNU/Linux teste l'intégrité du périphérique bloc au moment du montage, nous allons tester le débit en écriture :

```
libretto:/home/jmfriedt# cp /home/jmfriedt/latex/collagen/collagen.pdf /mnt/mmca1
libretto:/home/jmfriedt# time umount /mnt/mmca1

real    1m41.987s
user    0m0.000s
sys     1m41.380s
libretto:/home/jmfriedt# mount /dev/mmca1 /mnt/mmca1
```

```
libretto:/home/jmfriedt# ls -l /mnt/mmca1/
total 2265
-rwxr-xr-x    1 root     root      2296050 Mar 12 00:24 collagen.pdf
drwxr-xr-x    2 root     root        12288 Mar 12 00:00 lost+found
```

Nous avons donc mis 1 minute, 41 secondes pour écrire un fichier d'environ 2,3 MB.

Les performances ne sont pas pour le moment l'objectif principal de ce développement. L'implémentation logicielle du protocole SPI sur port parallèle prend un temps considérable : le temps de transfert est de l'ordre de 24 kB/s en écriture et 26 kB/s en lecture. Afin de rendre la main aux autres processus au cours des accès à la MMC sur le PC, nous insérons une fonction `schedule()` apès chaque accès à un octet échangé avec la MMC dans son implémentation sur le port parallèle (la version Coldfire utilisant les interruptions pour acquitter chaque transaction n'a pas ce problème) : nous avons constaté que les performances sur un système peu chargé ne sont pas affectées par la fréquence aux appels à `schedule()` ni par les moments auxquels cette fonction est appelée. Cependant, alors que les performances en terme de débit ne sont pas affectées sur un système peu chargé, l'interactivité pour l'utilisateur est nettement améliorée.

Enfin, nous utilisons ici l'algorithme d'allocation de blocs de ext2, qui vise à allouer des blocs d'inodes afin d'accélérer les accès aux fichiers. Cette notion n'a pas lieu d'être sur un support de type MMC où tous les emplacements mémoire sont accédés avec une lattence indépendante de leur localisation. Il est donc envisageable d'utiliser d'autres méthodes d'allocations d'inodes plus appropriées, notamment dans le but d'utiliser une fonctionnalité récente des MMC – qui a été ajoutée à la norme pour le mode de communication SPI – qu'est l'écriture de plusieurs blocs adjacents (CMD18 – `READ_MULTIPLE_BLOCK`, et CMD25 – `WRITE_MULTIPLE_BLOCK`). Il n'est cependant pas claire que la charge additionnelle de gestion des queues de données conservées en cache améliore réellement le débit au vu des gains limités qu'offre cette commande, ou qu'il existe un algorithme capable d'accélérer les accès à un périphérique de stockage de masse de type "mémoire non-volatile" tel que la MMC pour laquelle aucun élément mécanique ne limite les délais de lecture ou d'écriture.

8 Conclusion

Nous avons développé un montage électronique d'interfaçage de MultiMediaCard (MMC) et MMC+ avec le Coldfire fourni sur la carte SSV DNP5280 ainsi que sur le port parallèle d'un PC. Nous nous sommes familiarisés avec le protocole de communication (SPI) fourni comme périphérique matériel dans le premier cas, émulé de façon logicielle dans le second. Étant capable de lire et écrire des informations sur la MMC après en avoir extrait la géométrie, nous avons implémenté un block device donnant accès à toutes les fonctionnalités d'un système de fichier qui fait apparaître notre périphérique comme un nouveau répertoire de l'arborescence uClinux ou GNU/Linux.

Nous avons amélioré l'intégrité des transactions en implémentant l'ensemble des méthodes de validation des échanges par CRC.

Ce travail répond à un besoin lors de l'utilisation de circuits embarqués tel que le SSV DNP5280 à des fins de télémesure de pouvoir écrire une grande quantité de données sur mémoire non-volatile. Le block device ajoute une souplesse d'accès et d'intégrité des données qu'une écriture brute sur mémoire non-volatile ne propose pas. L'utilisation de MMC+ offre ainsi plus de 1 GB de mémoire avec des débits d'accès raisonnables (de plus de 40 kB/s sur processeur Coldfire à plus de 20 kB/s sur le port parallèle du PC).

Afin de parfaire les fonctionnalités de ce driver, nous avons implémenté le support des partitions. En effet, une des premières applications hors-embarqué d'un tel circuit qui vient à l'idée est la lecture de cartes utilisées dans des appareils photographiques numériques (APN). L'utilisation d'un format de type VFAT est pris en charge automatiquement par le VFS sous réserve que le noyau support ce format. Il apparaît à l'utilisation qu'une carte formatée par un APN tel que le Nikon Coolpix 3200 contient une unique partition en FAT16 s'étendant sur l'ensemble de la carte. Une solution possible que nous avons testé est de formater la carte *via* notre montage par

`mkfs.vfat /dev/mmca` (ce qui correspond à formater tout le périphérique bloc sans y créer de partition) et la carte résultante est reconnue par l'APN. Après ce formatage, l'APN se contente d'ajouter les répertoires dont il a besoin et les images stockées alors sont visibles sur un PC équipé de notre montage. Avec la gestion des partitions, une carte formatée par l'APN est directement accessible sous `/dev/mmca1` d'où nous avons pu lire et afficher les images qui y sont emmagaisnées.

De plus, nous devons encore valider la fiabilité des transactions à long terme et notamment la corruption possible de quelques blocs de 512 octets lors du stockage de fichiers volumineux (>7 MB).

Remerciements

Nous remercions l'entreprise Alcôve (www.alcove.fr) pour son soutien à la réalisation de ce projet. L'association de diffusion des logiciels libres sur la Franche-Comté – Sequanux (www.sequanux.org) est également remerciée pour son support logistique. Enfin, nous remercions Frédéric Tronel pour le don du Toshiba Libretto 100CT qui illustre cet article, grâce auquel même une application sur le port parallèle d'un PC garde une orientation "application embarquée".

Références

[1] *MultiMediaCard Product Manual*, rev. 3, Sandisk (07/2001), disponible à www.fuw.edu.pl/~pablo/s/opisy/mmcprodmn_r31.pdf

[2] une des premières webcams disponibles sur le marché, brièvement décrite à http://www.pcworld.com/news/article/0,aid,123950,pg,5,00.asp

[3] l'alimentation sur un clavier PS2 se prend entre les broches 3 et 4 telles que définies par exemple à http://www.burtonsys.com/PS2_keyboard_and_mouse_mini-DIN-6_connector_pinouts.html

[4] *MCF5282 Coldfire Microcontroller User's Manual*, MCF5282UM Rev. 2.3, 11/2004

[5] D. Bodor, *Réalisation de circuits*, GNU/Linux Magazine Hors Série **23**, Novembre/Décembre 2005, pp.18-23

[6] D. Bodor, *Programmation du port parallèle*, GNU/Linux Magazine Hors Série **23**, Novembre/Décembre 2005, pp.24-30

[7] J.-M. Friedt & É. Carry, *Enregistrement de trames GPS*, GNU/Linux Magazine France **81**, Mars 2006, pp.80-91.

[8] A. Rubini, J. Corbet & G. Kroah-Hartman, "Linux Device Drivers, 3nd ed.", O'Reilly Ed. (2001), disponible à http://lwn.net/Kernel/LDD3/

[9] *Conception d'OS : Pilotes de périphériques blocs*, GNU/Linux Magazine France **80**, Janvier 2006, pp. 74-90.

[10] Y. Guidon, *Réalisation du CRC pour le conteneur MDS*, GNU/Linux Magazine France **78**, Décembre 2005, pp. 70-86.

[11] Y. Guidon, *Comprendre les générateurs de nombres pseudo-aléatoires*, GNU/Linux Magazine France **81**, Mars 2006, pp. 64-76.

[12] G. Thomas, code source disponible à http://dsl.ee.unsw.edu.au/dsl-cdrom/unsw/projects/ecos/ecos/packages/services/crc/current/src/crc16.c (2002)

[13] http://en.wikipedia.org/wiki/Cyclic_redundancy_check

[14] R. Card, E. Dumas, F. Mével, *Programmation Linux 2.0*, Eyrolles (1997), pp.228-240