Introduction au Coldfire 5282

Jean-Michel Friedt, Simon Guinot,
friedt@free.fr, simon@sequanux.org
Association Projet Aurore, UFR-ST La Bouloie
16 route de Gray, 25044 Besançon
19 juillet 2005

1 Introduction

Nous utilisons une carte commercialisée par la société allemande SSV [1] : la DIL/Net 5280. Cette carte fournit un processeur cadencé à 66 MHz, avec 16 MB de RAM et 8 MB de mémoire flash, et de nombreux périphériques que nous allons présenter plus tard dans cet article, pour un prix unitaire de 150 euros (250 euros avec une carte d'évaluation permettant une mise en œuvre en quelques minutes, 99 euros pour 10 pièces). Nous présenterons ici la mise en œuvre de cette carte, les divers périphériques que nous y avons ajouté pour finalement conclure sur une application de capture d'images couleur retransmises par internet.

2 Présentation générale du matériel

La carte DIL/Net 5280 (fig. 1) fournie sur circuit d'évaluation se caractérise avant tout par sa simplicité de mise en œuvre : noyau préflashé avec de nombreuses commandes unix classiques, un client NFS se connectant directement sur un serveur tel que installé sur un PC sous Debian (i.e. NFS v.3), un bootloader (dBug [2]) très puissant permettant de récupérer un nouveau noyau rapidement par tftp et de modifier l'adresse IP de la carte sans avoir à reflasher un noyau complet. SSV a installé un shell souple d'utilisation avec rappel des dernières commandes.

En parallèle avec cette excellente qualité des logiciels sélectionnés, le circuit est simple à mettre en œuvre du point de vue matériel par l'utilisation d'un support DIL64 dont le brochage est très clairement documenté. Ainsi, contrairement à la carte uCdimm 5272 présentée auparavant [3], aucune compétence spéciale de soudure n'est requise pour faire fonctionner le DIL/Net 5280. Les principaux signaux utiles sont mis à disposition de l'utilisateur : bus de données du processeur, 4 lignes du bus d'adresse et un décodeur interne de plages d'adresses (Chip Select), divers bus de communication tels que I^2C, SPI, RS232, ethernet et CAN. Enfin, la mise sous tension de la carte 5282 ne lance pas nécessairement le bootloader mais permet de passer directement à l'exécution du noyau linux placé en mémoire flash en fonction de la position d'un jumper, rendant le système facilement autonome dès son allumage.

Malgré la ressemblance en terme de nomenclature avec le processeur Coldfire 5272 présenté antérieurement [3], une différence majeure du Coldfire 5282 est l'existence d'un mode protégé pour accéder aux registres matériels. Ainsi, contrairement au 5272 qui se programmait comme un micro-contrôleur en accédant à l'ensemble de la mémoire depuis le mode utilisateur, la programmation du 5282 ressemble beaucoup plus à ce dont nous avons l'habitude sur un PC, avec l'utilisation intensive de modules noyaux pour toutes les opérations nécessitant un accès aux périphériques matériels. La transition du 5272 au 5282 est rendue aisée par la mise à disposition par SSV à l'achat de leur carte d'évaluation d'un module noyau nommé `ssvhwa.o` (fourni avec ses sources) permettant un accès simple (en 8, 16 ou 32 bits, entrée ou sortie) à toutes les adresses mémoire du 5282, y compris les registres d'accès aux ports matériels. En l'absence de ce module chargé de traduire des requêtes vers les registres de commande du matériel depuis le mode utilisateur, il nous faudra systématiquement écrire un module noyau pour accéder aux périphériques. `ssvhwa` se résume en fait en un bon outil de débuggage mais dont le temps d'accès aux registres matériels est tellement long qu'il nous faudra en pratique toujours recourir à l'écriture de module noyau lorsque des performances doivent être obtenues tel que nous le verrons dans les exemples illustrant cet article.

En comparaison du 5272, la carte à base de 5282 garde les périphériques les plus communs :

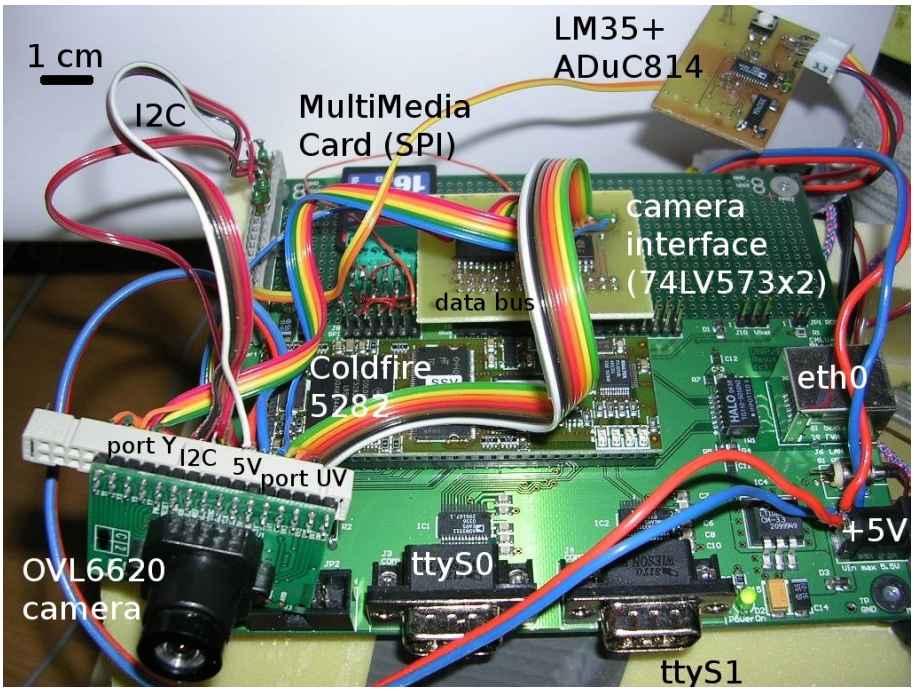

FIG. 1 – Photographie de la carte SSV DIL/Net 5280 sur son support de développement équipé des connecteurs et de l'électronique d'interfaçage d'une mémoire de stockage (MultiMediaCard, section 5.1) de masse et d'une caméra numérique CMOS couleur (section 5.3. L'ADuC814 est un microcontrôleur 8-bits compatible 8051 utilisé ici pour tester la lecture sur le port série (cf section 5.2

deux ports ethernet, deux ports série, port SPI, PWM. La carte SSV n'embarque pas les convertisseurs 3,3 V→ ±12 V et transformateurs ethernet mais reportent ces accessoires sur le circuit développé pour l'application spécifique (par exemple la carte d'évaluation SSV) : ainsi, cette carte est mieux appropriée pour communiquer avec des nombreux périphériques tels que microcontrôleurs et récepteurs GPS (qui requièrent généralement des connexions unipolaires sur leurs ports série) et évitent d'embarquer des transformateurs encombrants si la communication ethernet n'est pas nécessaire. Dans le cas de la carte d'évaluation SSV (DNP/EVA6), la conversion 3,3 V→ ±12 V est obtenue au moyen de ADM3311 et le transformateur ethernet est un HALO TG110-S050N2. Des périphériques peu utiles (esclave USB) sont avantageusement remplacés par des interfaces pratiques : I^2C et convertisseurs analogique-numérique par exemple. Nous allons ici décrire l'accès à ces divers périphériques.

3 Outils de développements logiciels

Les outils de compilation sont les mêmes que pour 5272 (`m68k-elf`) et la même arborescence de binaires peut être utilisée avec le 5282. La grande nouveauté est l'utilisation intensive des modules noyaux pour lesquels le Makefile diffère un peu de celui utilisé pour la compilation de programmes utilisateurs : un exemple est fourni ici pour présenter les diverses options spécifiques à la compilation d'un module.

```
PATH := $(PATH):/usr/bin:/usr/local/bin
CC = m68k-elf-gcc
OBJS = i2c_mod.o
MODUL = -nostdinc -D__KERNEL__ -Wall -Wstrict-prototypes -Wno-trigraphs -fno-strict-aliasing -fno-common \
-I/usr/local/lib/gcc-lib/m68k-elf/2.95.3/./include -pipe -DNO_MM -DNO_FPU -m5307 -Wa,-S -Wa,-m5307 -D__ELF__ \
```

```
-DMAGIC_ROM_PTR -DUTS_SYSNAME="uClinux" -D__linux__ -DMODULE -I $(UCPATH)/linux-2.4.x/lib/zlib_deflate \
-nostdinc -iwithprefix include -DKBUILD_BASENAME=deflate_syms  -DEXPORT_SYMTAB
UCPATH = /home/jmfriedt/uClinux-dist

all: $(OBJS)

i2c_mod.o: i2c_mod.c
$(CC) $(MODUL) -O3 -c i2c_mod.c -I$(UCPATH)/linux-2.4.x/include

# NE PAS METTRE -msep-data dans un module
```

4 Spécificités de la programmation sur système embarqué

Un certain nombre de différences existent entre le développement sur PC sous GNU/Linux et sur les systèmes embarqués fonctionnant sous uClinux. Nous allons présenter ici quelques aspects que nous avons abordés au cours de nos expériences, notamment concernant la gestion de la mémoire et des interruptions dans cet environnement particulier.

4.1 Rappels sur les modules

Les modules deviennent une part fondamentale de la programmation des périphériques matériels du Coldfire 5282 qui ne sont accessibles qu'en mode superviseur, *i.e.* mode noyau pour linux. La structure générale des modules sous uClinux est strictement identique à celle familière sous linux. La seule originalité de la programmation sur Coldfire 5282 dans cet environnement de module noyau est la présence de lignes telles que
`*((volatile u8 *)addr)=data ;` ou
`data = *((volatile u16 *)addr) ;`
qui correspondent respectivement à des écritures et lectures à des adresses prédéfinies telle que décrites dans la documentation du Coldfire 5282 [4] pour un accès aux périphériques matériels.

4.2 Les interruptions

Le Coldfire 5282 dispose de deux contrôleurs d'interruption. Chacun de ces contrôleurs comprend 63 vecteurs dont 56, pleinement configurables et programmables. La table des vecteurs d'interruptions comporte 256 entrées. Les 64 premiers vecteurs sont réservés au fonctionnement interne du processeur (gestion du reset, gestion des erreurs...). Le premier contrôleur permet de configurer les vecteurs 65 à 127 et le second, les vecteurs 128 à 190. Ces deux contrôleurs sont appelés respectivement INTC0 et INTC1. Ils peuvent être configurés par l'intermédiaire de registres mappés en mémoire. pour INTC0 par exemple, l'adresse de base est IPSBAR + 0xc00 et pour INTC1, elle est IPSBAR + 0xd00. Nous allons presenter les registres dont les fonctionnalités sont les plus intéressantes, puis nous illustrerons l'activation d'une interruption au travers d'un exemple.

4.2.1 Concurrence entre les interruptions

La concurrence entre les interruptions est un point critique des applications temps réel. Sur les processeurs de type Motorola Coldfire, il est possible de classer les vecteurs d'interruption en fonction de leur importance. Dans le cas du Coldfire 5282, le registre 8 bits ICR (Interrupt Control Register) va permettre d'assigner un niveau et une priorité à une interruption. Chaque vecteur d'interruption dispose de son propre registre ICR. Dans une situation de concurrence entre plusieurs interruptions, celle dont le niveau est le plus important sera exécutée en premier. Si plusieurs d'entre elles possèdent le même niveau, elles sont départagées en examinant leurs priorités. Les priorités et niveaux sont désignés par des valeurs comprises entre 0 et 7 (3 bits). Une interruption de niveau élevé sera également autorisée à préempter le gestionnaire associé à une interruption de niveau moindre. Pour éviter cela, les applications temps réels devront adjoindre

une forte priorité à leurs vecteurs d'interruption. À noter que le système de priorité n'existe pas sur le Coldfire 5272. Le registre PIWR (Programmable Interrupt Wakeup Register) est utilisé pour départager les interruptions de même niveau. Le vecteur représenté par le bit de poids le plus fort au sein de ce registre est acquitté le premier. Il est évident que ce type de configuration est bien moins souple a gérer pour le programmeur...

4.2.2 Masquer les interruptions

Chaque contrôleur d'interruption dispose de 2 registres de 32 bits IMRH et IMRL (Interrupt Mask Register Hight or Low). Chacun des bits de ces registres sont associés à un vecteur d'interruption du contrôleur. Positionner à 1 le bit correspondant à un vecteur d'interruption permet de masquer ce dernier. Il est également possible d'utiliser le registre global SR (Status Register) qui met 3 bits à disposition des interruptions. Ces 3 bits permettent de définir un niveau. Toutes les interruptions de niveau inférieur à la valeur stockée dans ce registre sont masquées. À noter que les fonctions assembleur `cli()` et `sti()`, permettant respectivement de masquer et démasquer toutes les interruptions, utilisent ce registre. L'idée est d'y inscrire la valeur 7, correspondant au niveau le plus important. Cela a pour effet de masquer toutes les interruptions, exceptées celles de niveau 7. Il n'existe aucune méthode pour masquer ces dernières. Le niveau 7 regroupe les interruptions internes considérées comme critiques pour le bon fonctionnement du processeur (gestion des différentes erreurs : opérations illégales, accès mémoire interdits... entre autres).

implémentation de la fonction `cli()` telle que trouvée dans le fichier */uClinux-dist/linux-2.4.x/include/asm/system.h* :

```
#define __cli() __asm__ __volatile__ ( \
        "move %/sr,%%d0\n\t" \
        "ori.l  #0x0700,%%d0\n\t" \
        "move %%d0,%/sr\n" \
        : /* no inputs */ \
        : \
        : "cc", "%d0", "memory")
#else
```

4.2.3 Les interruptions logicielles

Les registres 32 bits INTFRCH et INTFRCL (Interrupt Force Register Hight or Low) sont également disponibles sur chacun des contrôleurs. Chaque bit de ces registres est associé à un unique vecteur d'interruption. Positionné ce bit à 1 permet de générer de manière logicielle une interruption du vecteur correspondant. Ces registres sont particulièrement utile lors du développement et du debuggage de pilotes.

exemple de fonction permettant sur le Coldfire 5282 d'activer un vecteur d'interruption et de lui associer un gestionnaire :

```
struct irq_info
{
        unsigned int vector;
        unsigned char name[30];
        unsigned char level; /* level and priority requested */
        void (*handler)(int, void *, struct pt_regs *);
        void *dev_id;
};

static int enable_interrupt_vector (struct irq_info *irq_info)
{
        unsigned int retval = 0;
        unsigned int int_num;
        volatile unsigned char *icr;
        volatile unsigned long *imrh;
        volatile unsigned long *imrl;
```

```
        [...]
```

int_num, et les pointeurs icrp, imrh et imrl sont initialisés en fonction du numéro de vecteur irq_info→vector fournit en paramètre. Puis le gestionnaire désigné par irq_info→handler est associé à ce vecteur à l'aide de la fonction request_irq (). Cette phase d'enregistrement est identique à celle qui se pratique sur un système GNU/Linux classique.

```
        if ((retval = request_irq (irq_info->vector, irq_info->handler,
                                   SA_INTERRUPT, irq_info->name,
                                   irq_info->dev_id)))
        {
                dbg("unable to request irq %d", irq_info->vector);
                return retval;
        }
```

Le registre ICR correspondant au vecteur que l'on souhaite activer se voit affecter un niveau et une priorité. Cette information est également fournie en paramètre via irq_info→level.

```
        icr[int_num] = (irq_info->level > IRQ_LEVEL_MAX)?
                IRQ_LEVEL_DFLT : irq_info->level;
```

Et enfin on s'assure que le bit de masque correspondant a notre vecteur n'est pas positionner.

```
        *imrh &= ~(1 << (int_num - 32));
        *imrl &= ~1;
```

À partir d'ici, le vecteur d'interruption est actif et la fonction irq_info→handler lui est associée.

```
        [...]

        return (retval);
}
```

4.3 Gestion de la mémoire sur un système sans MMU (Memory Management Unit)

La MMU correspond à l'ensemble des mécanismes internes au processeur permettant de réaliser la conversion d'une adresse logique en une adresse physique. Ces mécanismes peuvent être la pagination, la segmentation ou encore la pagination de segments...

Une conséquence directe de l'absence de MMU est l'impossibilité pour uClinux d'implémenter un VM (Virtual Manager ou gestionnaire de mémoire virtuelle).

Le rôle du VM est de permettre aux processus de mapper des zones de mémoire physique non contiguës au sein de leur espace d'adressage linéaire.

C'est sur le VM que repose le partage des librairies. Cette technique permet à plusieurs processus d'utiliser une librairie commune en la mappant dans leur espace d'adressage virtuel. De cette façon, la librairie n'est chargée qu'une seule fois en mémoire physique. L'absence de VM rend impossible l'utilisation de librairies partagées. Toutes les applications à destination d'uClinux sont donc compilées statiquement et les librairies nécessaires à leur fonctionnement font parties intégrantes du binaire final. Lors du chargement en mémoire du programme, ces librairies seront copiées intégralement dans son espace d'adressage.

Il en résulte un problème de redondance. En effet les librairies les plus couramment utilisées, comme uClibc (version allégée et statique de la libc), seront présentes plusieurs fois en mémoire physique.

Le VM permet également de contrôler les accès mémoire et les permissions. Sur un système GNU/Linux classique, un accès mémoire illégal pourra engendrer une erreur de segmentation ou de pagination : le fameux “segmentation fault” bien connu des programmeurs. L'absence de tels mécanismes sur les processeurs sans MMU prive uClinux de contrôle sur les accès mémoire. Un processus pourra donc accéder et écrire à des adresses encore non allouées ou même extérieures à son

espace d'adressage. Prenons comme exemple un programme dont un tampon déborde légèrement et qui, la majeure partie du temps, s'exécute normalement. À l'issue d'un changement de contexte, on peut cette fois imaginer le tampon écraser des données importantes et causer une erreur. Ce genre de bug est particulièrement délicat à repérer et est le plus souvent entouré d'un épais mystère. C'est pourquoi les programmeurs devront être particulièrement vigilants vis à vis des accès mémoire sous uClinux.

Dans le cas du Coldfire 5272, l'absence de contrôle des accès mémoire présentait aussi des avantages. Les programmes utilisateurs étaient capables d'accéder à des registres normalement réservés au noyau. Il était ainsi possible d'activer un certain nombre de fonctionnalités matérielles depuis l'espace utilisateur. L'idée n'était pas de développer des pilotes en espace utilisateur mais plutôt de tester des solutions avant de se lancer dans le développement assez fastidieux d'un module noyau plus rapide et plus performant.

Le 5282 disposant d'un mode superviseur, justement destiné à protéger les registres critiques, ne nous offre plus ce petit avantage. Les registres protégés seront donc uniquement adressables en mode noyau.

4.3.1 Absence de fork

L'API uClinux est relativement compatible avec celle de GNU/linux... à quelques exceptions près. L'absence de l'appel système `fork()` en est une bien connue. Encore une fois, il s'agit d'une conséquence de l'absence de VM. Les implémentations modernes du `fork()` utilisent la stratégie du "copy on write". À l'issue de l'appel `fork()` un processus fils est créé et son espace d'adressage virtuel pointe sur le même espace mémoire physique que celui de son père. Les premiers accès en écriture vont engendrer la création de pages séparées pour le père et le fils. L'idée est de retarder le plus longtemps possible l'opération de duplication de l'espace d'adressage du processus père. Cette stratégie rend impossible le port de l'appel système `fork()` sur un système dépourvu de MMU.

Comme alternative uClinux offre le vieil appel système BSD `vfork()`. Historiquement, `vfork()` a été écrit comme une alternative aux premières implémentations de `fork()`. Ces dernières n'utilisaient justement pas le "copy on write" et allouaient directement à la création d'un processus fils un espace distinct et le contenu de celui du père y était dupliqué. Cette opération, relativement coûteuse en temps, s'avérait souvent inutile. Particulièrement lorsque après sa création, le processus appelait une fonction du type `execve()`, provoquant l'exécution d'un nouveau programme et toutes les allocations mémoire qui en découlent.

`vfork()` permet au père et au fils de partager l'intégralité de leur espace d'adressage, y compris la pile. Pour éviter tout conflit, le père voit son exécution suspendue jusqu'à ce que le fils appelle une fonction de type `execve()` ou `_exit()`.

Cet appel système est assez rapide mais ne permet pas, hélas, d'émuler plusieurs processus concurrents au sein d'une même application. Les restrictions sont donc relativement importantes et certaines applications ne sont pas trivialement portables sous uClinux.

4.3.2 Pile de taille fixe

Les applications fonctionnant sous uClinux disposent d'une pile de taille fixe et non extensible qui est allouée au chargement en mémoire du programme. Sur le Coldfire 5282, la taille par défaut de la pile est de 16 MB. Cette taille peut être fixée pendant la compilation en ajoutant la ligne `FLTFLAGS = -s 4096` au Makefile de l'application. Une autre méthode pour modifier la taille de la pile d'un binaire existant est d'utiliser l'outil `m68k-elf-flthdr` là encore avec l'option -s.

Si on cumule le fait que la pile n'est pas dynamiquement extensible et que les accès mémoire ne sont pas contrôlés, il devient évident que le programmeur devra être très attentif quant à la gestion de la pile.

Un programme dont la pile est en train de déborder ne générera pas d'erreur d'accès mémoire. Au mieux, elle débordera sur des zones mémoire encore non allouées et le programme s'exécutera

normalement. Dans le pire des cas, l'espace d'adressage d'un processus voisin sera altéré et ce dernier pourra en voir son fonctionnement modifié.

En règle générale, les fonctions récursives doivent être surveillées et les allocation dynamiques être préférées a celles sur la pile.

4.3.3 L'allocation dynamique de mémoire

Un problème associé à l'absence de gestionnaire de mémoire sur les processeurs exécutant le noyau uClinux est la limitation quant à l'allocation d'une grande quantité de mémoire telle que requise lors de l'acquisition d'un grand nombre de données.

Sur un système d'exploitation disposant d'un VM (gestionnaire de mémoire virtuelle), les processus allouent dynamiquement de la mémoire dans le tas (*heap*).

Ces allocations dynamiques se font à l'aide de fonctions ou d'appels systèmes mis à disposition par le système d'exploitation. Sous GNU/Linux, l'appel système `brk()` ou la fonction de plus haut niveau `malloc()` (elle même utilisant `brk()` en interne) peuvent être utilisés. Le tas correspond à un espace contigu d'adresses linéaires contenu dans l'espace d'adressage du processus. Le VM permet de mapper des adresses physiques non contiguës au sein de cet espace d'adresses linéaires. La taille du tas peut être dynamiquement augmentée si le besoin se fait sentir, en élargissant l'espace d'adresses linéaires lui étant attribué.

La plupart des systèmes d'exploitation sans VM implémente un tas de taille fixe et non extensible, résidant dans l'espace d'adressage du processus. Cette méthode a l'inconvénient de gaspiller inutilement de la mémoire si cet espace n'est pas utilisé par le processus et aussi de rendre impossible les allocations mémoire trop volumineuses (*i.e.* dépassant la taille totale prévue pour le tas). uClinux contourne cette difficulté en utilisant une stratégie originale : les processus ne disposent pas de tas individuel [12]. À la place, les allocations dynamiques sont satisfaites en piochant dans un pool de mémoire correspondant à l'ensemble de la mémoire libre du système. Grâce à cette stratégie, un processus peut, si il en a le besoin, s'attribuer l'ensemble de la mémoire disponible sur le système.

Un problème peut cependant survenir dans le cas de fortes allocations mémoire. En effet, la fragmentation de la mémoire physique va souvent rendre impossible les allocations importantes de mémoire contiguë. Dans l'exemple présenté fig. 2, schématisant 1 MB de mémoire physique, on peut remarquer que l'on dispose de 500 KB de mémoire disponible. Cependant, une allocation de 100 KB échouera. En raison de la fragmentation, une telle plage d'adresses mémoire contiguës n'est pas disponible.

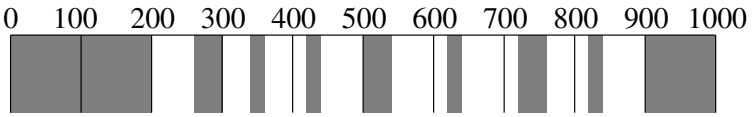

FIG. 2 – Exemple de fragmentation d'une mémoire de 1 MB : alors que 500 KB sont disponibles, toute tentative d'allocation de mémoire de 100 KB échoue.

Le problème de la fragmentation de la mémoire n'a pas vraiment de solutions satisfaisantes. Les allocations importantes devront être réalisées rapidement après le démarrage du système d'exploitation, au moment où la mémoire est la moins fragmentée.

Pour une application ne nécessitant pas impérativement que la mémoire allouée soit contiguë, on peut envisager d'utiliser une liste chaînée. Chaque élément de la liste pointe vers une zone mémoire allouée par le processus et aussi vers l'élément suivant. Ce dernier désignera une autre zone mémoire etc ...

```
struct buffer
{char *data;
 unsigned int size;
 struct buffer *next;
```

```
};

struct buffer * alloc_mem (unsigned int size, unsigned int *get_size);
```

Si l'on reprend l'exemple précédemment illustré fig. 2, où l'allocation de 100 KB était impossible ... elle pourra être réalisée à l'aide de cette liste, avec par exemple deux éléments, le premier pointant sur une zone de 80 KB et le suivant sur une zone 20 KB. Cette astuce ne palie pas à l'absence de mémoire contiguë, mais elle permet de stocker des zones de mémoire non-adjacentes dans une structure de données simple à manipuler.

5 Applications pratiques

Nous allons présenter ici quelques applications développées pour nous familiariser avec ce circuit, de complexité et de fonctionnalités croissantes :

1. écriture en entrée sortie sur un port numérique par le module `ssvhwa` puis pas notre propre module
2. le stockage de données sur mémoire non-volatile de type MultiMediaCard (MMC, http://www.sandisk.com/oem/mmc.asp)
3. la conversion analogique-numérique avec un échantillonage lent
4. l'acquisition d'images couleur *via* le bus de données et configuration de la caméra par bus I^2C Fig. 4.

5.1 Le port SPI et la MultiMediaCard (MMC)

De nombreuses méthodes de stockage de masse non-volatiles sont aujourd'hui disponibles à très faible coût du fait de leur utilisation dans l'électronique grand publique : cartes SmartMedia, Memory Stick, CompactFlash Multimedia (MultiMediaCard, MMC) pour ne citer que les plus courantes. Nous avons sélectionné le dernier type pour son faible nombre de broches (connecteur 7 broches séparées de 2.54 mm – un bout de connecteur ISA peut être utilisé pour s'interfacer), sa simplicité de programmation tel que nous allons le voir plus loin, son faible coût pour une zone de stockage importante (16 MB à 128 MB pour moins de 15 euros) et sa disponibilité en volume unitaire.

Nous allons développer en détail le mode de programmation de ces cartes car au-delà de uClinux tournant sur Coldfire, nous avons implémenté ce protocole sur de nombreuses autres architectures (microcontrôleurs 8 bits compatible 8051 Analog Devices ADuC814, Motoroloa 68HC908JB8, accès depuis le port parallèle d'un PC compatible IBM), ouvrant des possibilités de stockage de télémesure inimaginables sinon. Nous allons ici nous focaliser sur l'aspect QSPI du Coldfire 5282, une description des autres implémentations étant disponible sur la page web des auteurs ou dans la littérature [7, 8, 9].

La MultiMediaCard s'initialise par défaut dans un mode de communication synchrone spécifique à cette carte. Nous avons décidé de profiter de la disponibilité d'un port SPI sur le processeur Coldfire (comme sur la majorité des microcontrôleurs Motorola/Freescale) pour utiliser ce mode de communication que comprend la MMC après une phase d'initialisation appropriée.

Nous allons commencer par décrire l'accès au bus SPI du Coldfire en général, pour ensuite nous focaliser sur le cas particulier de la MMC [10].

5.1.1 Le port SPI

Le bus SPI supporte un protocole bidirectionnel synchrone ne nécessitant que 3 fils : une horloge, une communication du maître vers l'esclave (MOSI : Master Out Slave In, du Coldfire vers la MMC dans notre cas) et de l'esclave vers le maître (MISO : Master In Slave Out).

Le Coldfire 5282 est équipé d'une QSPI, *i.e.* une version évoluée du port SPI incluant une pile de 64 mots permettant au logiciel de gestion de la communication une plus grande latence entre

deux accès au bus. Nous allons dans notre cas généralement nous limiter à n'utiliser que le premier élément de cette pile, de façon très semblable au mode d'accès sur microcontrôleur.

Une première phase d'initialisation active le port SPI (registre `PQSPAR`=0x0f), définit le protocole de communication à 8 bits/donnée à une fréquence de l'ordre de 100 kHz (registres `QMR`=0xA1A2, `QDLYR`=0x0202 et `QIR`=0xD00F pour désactiver toute interruption associée au port SPI).

5.1.2 Cas particulier de la MMC

Le passage du mode de communication parallèle MMC au mode série SPI se fait en envoyant à l'allumage de la carte une trame de 80 créneaux sur le signal d'hologe tout en maintenant le niveau du signal CS# de la MMC *haut* (*i.e.* désactivé) [11]. Une pile de commandes est donc initialisée à cette fin en remplissant les 10 premiers éléments du registre `QDR` et en définissant le registre `QAR=0x20` pour indiquer que 2×10 commandes sont à exécuter. La suite des opérations est décrite de façon claire par le code source : envoi de la commande CMD0 (Reset : valeur 0x40 transmise du Coldfire vers la MMC avec CS# actif au niveau bas), puis attente de la transmission d'une valeur 1 par la MMC pour acquitter le passage au protocole SPI.

L'initialisation de la MMC s'obtient ensuite par transmission de la commande CMD1 (valeur 0x41 transmise du Coldfire vers la MMC avec CS# actif au niveau bas) avec une somme de contrôle (*checksum*) prédéfinie à 0x95, et acquittement de la MMC par une réponse de 0.

Noter que toutes les commandes CMDxx décrites dans les documentations de MMC correspondent à l'envoie de la valeur xx en *décimal* à la carte, masquée par un OU logique avec la valeur 0x40. Par exemple la transmission d'un ordre d'écriture de bloc (commande Write_Block CMD24) s'obtient par émission de l'octet 0x58 (OU logique entre 0x40 et 0x18 qui est l'expression de 0d24 en hexadécimal).

Une fois la MMC passée en mode SPI, le contrôle de checksum est désactivé et ce dernier sera toujours maintenu à 0xFF.

L'écriture en mode SPI sur la carte MMC se fait nécessairement pas blocs de taille déterminée : cette taille est définie par défaut à 512 octets et nous la laisserons à cette valeur par la suite (nous pourrions modifier cette taille de bloc par la commande Set_BlockLen, CMD16). Par conséquent toute adresse de bloc fournie comme point de départ d'une nouvelle lecture ou écriture est nécessairement multiple de 512, soit une adresse de la forme `0xab 0xcd 0x20*i 0x00` (i=0..7). La commande MMC d'écriture dans un block est 0x18 (commande CMD24 en décimal dans la nomenclature MMC) qui prend pour argument l'adresse d'écriture et se conclut après remplissage du bloc (transfert de 512 octets) par un acquittement confirmant l'écriture des données sur la carte mémoire.

Un protocole similaire est appliqué pour la lecture de bloc – opération 0x11 (CMD17 en décimal dans la nomenclature MMC : Read_Single_Block) avec là encore pour argument l'adresse du bloc de 512 octets à lire de la forme de celle vue auparavant.

Ce mode de stockage a été utilisé avec succès lors de l'acquisition de données issues des convertisseurs analogiques numériques ou pour la sauvegarde de données synthétisées par notre programme. Le passage par la carte MMC selon un protocole que nous implémentons par nos propres soins a notamment l'avantage, par rapport aux appels systèmes standards de type `write()` qui bloquent les interruptions, de n'interférer avec aucune interruption et par conséquent de permettre à une application de temps réel dur de s'exécuter en ne sauvant les données que lorsque le temps le permet.

5.2 Les convertisseurs analogique-numérique

Nous allons dans un premier temps illustrer une méthode relativement lente d'échantillonage des convertisseurs analogique-numériques – de l'ordre de quelques échantillons par seconde au mieux pour des applications telles que l'observation de l'environnement par exemple (température telle que présentée fig. 3, pression) – l'acquisition rapide à des fréquences audio faisant l'objet d'un autre article du fait des diverses difficultés rencontrées.

Comme dans le cas du bus SPI, les convertisseurs du Coldfire 5282 sont accessibles *via* une pile (QADC). Là encore nous chargerons dans un premier temps un certain nombre de commandes de conversion (mode de conversion et numéro de la voie à activer) pour ensuite lancer l'ensemble de ces commandes et enfin lire dans une seconde pile les résultats de conversion.

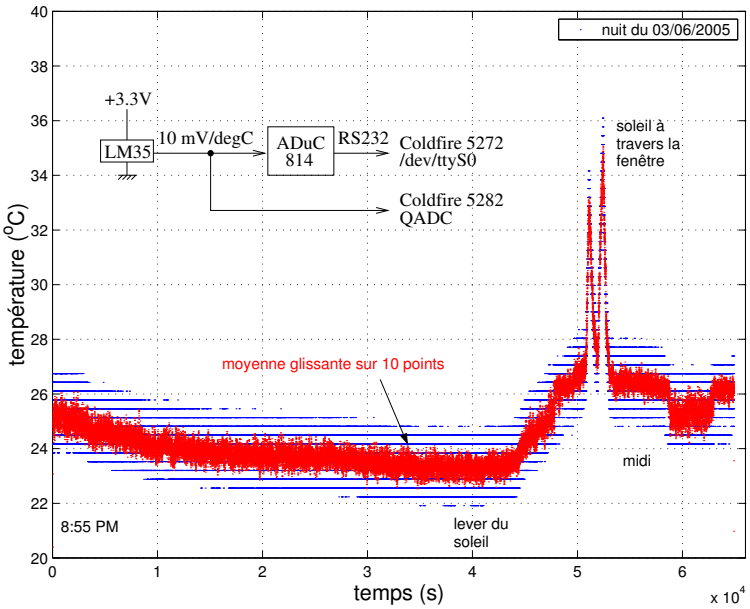

FIG. 3 – Exemple de mesure de température à 1 échantillon/s par le port QADC du 5282, issue sans traitement d'un capteur LM35 fournissant 10 mV/degrés C. Le rapport signal sur bruit est amélioré (et la résolution temporelle dégradée) par une moyenne glissante sur 10 points qui permet de faire disparaître la discrétisation par la conversion analogique-numérique.

Un exemple de conversion à un échantillon par seconde est présenté sur la fig. 3 tel que obtenu avec le programme ci-dessous :

```
#include <stdio.h>
#include <math.h>
#include "ssvhwa.h"

#define MCFBAR  0x40000000

#define QADCMCR  (MCFBAR + 0x00190000)  /* 16 bit */
#define DDRQA    (MCFBAR + 0x00190008)  /* 8 bit */
#define QACR0    (MCFBAR + 0x0019000A)  /* 16 bit */
#define QACR1    (MCFBAR + 0x0019000C)  /* 16 bit */
#define QASR0    (MCFBAR + 0x00190010)  /* 16 bit */
#define QASR1    (MCFBAR + 0x00190012)  /* 16 bit */
#define CCW      (MCFBAR + 0x00190200)  /* N bit */
#define RJURR    (MCFBAR + 0x00190280)  /* N bit */
#define LJSRR    (MCFBAR + 0x00190300)  /* N bit */
#define LJURR    (MCFBAR + 0x00190380)  /* N bit */

#define fichier // sauvegarde dans un fichier => pas d'affichage

#define nbscan 1 // 64

int main (void)
{unsigned char data_b;
 unsigned short data_w,i;
#ifdef fichier
 FILE *f;
 f=fopen("adc.dat","w");
#endif
    // check user identity
    if (geteuid() != 0) {printf("No root access rights !\n");exit(1);}
    if (ssvhwa_open() < 0) {perror("ssvhwa open");exit(-1);}

// set DIL/NetPC PIO Port A = output
 ssvhwa_write8(DDRQA, 0x00);        // Port A[0-3] output : 0x1B
 ssvhwa_write16(QADCMCR, 0x0000);   // enable ADCs
 ssvhwa_write16(QACR0,0x7f);        // default val=19, QACR0 pour definir QCK

// 52=AN52=PQA0, 53=AN53=PQA1, 62=(VH-VL)/2
for (i=0;i<nbscan*2;i+=2) ssvhwa_write16(CCW+i,0x00C0|52); // queue1 read, 16 ck
```

```
while (1) {
 ssvhwa_write16(QASR0,0x0000);
 ssvhwa_write16(QACR1,0x2100);  // queue 1 single scan mode, software trig 2100
 do {data_w=(ssvhwa_read16(QASR0));
#ifndef fichier
 printf("QASR0=%x ",data_w);
#endif
 } while ((data_w&0x8000)==0);
#ifndef fichier
 printf("\n");
#endif
 for (i=0;i<nbscan*2;i+=2)
   {data_w=ssvhwa_read16(RJURR+i);printf("%x=%d /10=degC",data_w,(int)((float)data_w*3.3/1.024));
#ifdef fichier
    fprintf(f,"%u\n",data_w);fflush(f);
#endif
  }
 printf("\n"); sleep(1); // 1 value/second
 }
 ssvhwa_close();return(0);
}
```

Ce programme se charge dans un premier temps d'activer les convertisseurs analogique-numérique (désactivés par défaut pour limiter la consommation électrique du Coldfire) en mettant à 0 le registre `QADCMCR`. Le principe des queues de conversion est qu'une table de 64 commandes est mise à disposition de l'utilisateur au moyen de la table de registres `CCW`. Nous y plaçons le mode de conversion (continu, unique, déclenché par un signal externe ...) et la voie de multiplexage activée. Dans notre exemple nous faisons une unique conversion immédiate sur la voie 52 (broche AN52 du Coldfire 5282). Après lecture du registre de statut (`QASR0`) qui nous annonce que la conversion a eu lieu, nous lisons le résultat de conversion dans un mot de 16 bits aligné à droite (i.e. dont seuls les 10 premiers bits sont significatifs) sous la forme du premier élément de la table de 64 mots nommée `RJURR`. Nous présenterons ultérieurement pour une application spécifique le cas de conversions multiples à fréquence d'échantillonage élevée. Nous passons ici sous silence la disponibilité d'une méthode de division des tables de 64 mots en sous-ensembles (nommés queue 1 et queue 2) dont l'intérêt nous échappe encore malgré des tentatives d'éclaircissement lors de communications avec les ingénieurs Freescale chargé de l'assistance des développeurs sur Coldfire.

La conversion des valeurs lues en valeurs significatives physiquement se fait par tension= valeur$\times 3,3/1024$ où 3,3 V est la tension de référence (tension d'alimentation du Coldfire) et 1024 la résolution des convertisseurs (10 bits). Dans l'exemple qui nous intéresse ici, le capteur de température est un LM35 fournissant 10 mV/degré celsius.

5.3 La caméra couleur : les bus du processeur et bus I^{2}C

Les capteurs optiques classiques de type CCD tendent aujourd'hui à être remplacés, pour de nombreuses applications où la sensibilité lumineuse n'est pas primordiale, par des capteurs CMOS. Ces derniers ont l'avantage d'intégrer dans un même composant le capteur optique analogique et l'électronique numérique de gestion de ce capteur pour finalement ne fournir à l'utilisateur qu'une interface totalement numérique, beaucoup plus simple à gérer que les nombreuses tensions et signaux d'horloge nécessaires pour une CCD [5]. Nous avons sélectionné, pour des raisons de coût et de disponibilité en petites quantités, le capteur CMOS OV6620 de Omnivision disponible monté sur circuit imprimé et équipé d'une optique chez Lextronic sous la dénomination CA88 [6].

Contrairement à la caméra Connectix Quickcam que nous avions décrit antérieurement [3], l'OV6620 n'est pas esclave des requêtes du processeur pour un nouveau pixel mais impose sa fréquence d'horloge. Ainsi, le composant est cadencé par une référence de fréquence fixe, divisée en interne par une valeur programmable, pour fournir en sortie à intervalle de temps régulier des pixels. La vitesse par défaut, 50 images de 352×288 pixels par seconde, est trop élevée pour un traitement logiciel par un processeur généraliste comme le Coldfire. Notre première tâche consiste donc à comprendre la communication par protocole I^{2}C pour programmer l'OV6620 et ainsi abaisser la vitesse de sortie des pixels.

5.3.1 Le protocole I^{2}C

Le bus I^{2}C supporte un protocole synchrone et ne nécessite que deux fils : une horloge (SC) et le transfert bidirectionnel de données (SD). Ce bus est utilisé – sous un nom (SCCB) qui

n'enfreint pas le copyright de Philips sur le nom de ce bus – par les caméras Omnivision pour leur configuration. D'un point de vue matériel, le port I^2C du Coldfire (3,3 V) se connecte au bus SCCB de l'OV6620 (5 V) en ajoutant des résistances de pull-up à 5 V ($\simeq$5,5 kΩ), ou directement au bus SCCB de l'OV3620 (elle aussi en 3,3 V) sans résistance de pull-up.

Ayant décrit en détail l'accès au bus SPI auparavant, nous n'allons pas ici entrer dans les détails de l'accès au bus I^2C qui est très similaire dans le principe. Un exemple de programme court se chargeant de l'initialisation de la caméra est fourni ici pour référence. Il utilise le module `ssvhwa` fourni par SSV pour un accès aux registres de contrôle matériel depuis l'espace utilisateur, mais pourra facilement être transféré dans un module noyau gérant l'initialisation de la caméra. Divers conditions permettent de sélectionner la vitesse de transfert des images, le choix du mode RGB ou YUV, et le transfert de données en mode 8 bits (port Y uniquement) ou 16 bits (port Y et UV) :

```
#include <stdio.h>
#include "ssvhwa.h"

#define I2C 0xc0                                        // CA88 base address
#define MCFBAR  0x40000000

#define I2ADR (MCFBAR + 0x300)
#define I2FDR (MCFBAR + 0x304)
#define I2CR (MCFBAR + 0x308)
#define I2SR (MCFBAR + 0x30C)
#define I2DR (MCFBAR + 0x310)

#define PASPAR (MCFBAR + 0x00100056)          // 16 bits, AS=mot de poids fort

void stop_i2c()
{ssvhwa_write8(I2CR, 0x90);}                                  // slave rcv mode

void start_i2c()
{unsigned char data_b;
 ssvhwa_write8(I2CR, 0xB0);                             // master transmit mode
 do {data_b=ssvhwa_read8(I2SR);printf("I2SR=%x - ",data_b);}
    while ((data_b&0x20)==0);                 // IBB is not set = not yet busy
}

char snd_i2c(unsigned char b)
{unsigned char data_b;
 ssvhwa_write8(I2DR,b);                                       // transmit data
 printf("%x: ",b);fflush(stdout);
 do {data_b=ssvhwa_read8(I2SR);printf("I2SR=%x\n",data_b);fflush(stdout);} while ((data_b&0x80)==0);  // sent ?
 ssvhwa_write8(I2SR,0x00);                          // clear "byte received" flag
 data_b=(ssvhwa_read8(I2SR)&0x01);
 return(data_b);                           // returns 0 if success, 1 if failed
}

void snd_cmd(unsigned char reg,unsigned char val)
{start_i2c();
 while (snd_i2c(I2C)!=0) {stop_i2c();start_i2c();}         // dest @ (0=write)
 snd_i2c(reg);snd_i2c(val);            // destination register -- value written
 stop_i2c();
}

int main (int argc, char **argv)
{unsigned char data_b;
 unsigned long data_l;
 int duree=0x3F;
 // check user identity
    if (geteuid() != 0) {printf("No root access rights !\n");exit(1);}
    if (ssvhwa_open() < 0) {perror("ssvhwa open");exit(-1);}

// ENABLE I2C
   data_l=ssvhwa_read16(PASPAR); data_l|=0xffff; ssvhwa_write16(PASPAR,data_l);
   printf("PASPAR=%x\n",ssvhwa_read16(PASPAR));

    ssvhwa_write8(I2FDR,0x15);                        // ck ~ 100 kHz
    ssvhwa_write8(I2ADR,0x42);                        // slave @: not used

    ssvhwa_write8(I2CR,0x00);                         // reset I2C
    ssvhwa_write8(I2CR,0xA0);
    data_b=ssvhwa_read8(I2DR);
    ssvhwa_write8(I2SR,0xA0);
    ssvhwa_write8(I2CR,0x00);

    ssvhwa_write8(I2CR, 0x90);                         // enable I2C as master
    printf("init done \n");
    if (argc>1) duree=atoi(argv[1]); if (duree>0x3f) duree=0x3f;

    snd_cmd(0x12,0xa4);                               // 0x80|0x24 = soft reset
    snd_cmd(0x11,duree);                  // select fps: 50=1 fps, max=0x3F=63
#ifdef color                                                  // select RGB
    snd_cmd(0x12,0x2C);
#endif
#ifdef huit                                              // select 8 bit mode
    snd_cmd(0x13,0x21);
#endif
    ssvhwa_close();return(0);
}
```

Notons cependant un certain nombre d'erreurs dans la datasheet de l'OV6620, notamment concernant le registre 0x11. Ce registre nous est utile pour abaisser la vitesse à laquelle la caméra

capture et transfert les images. Nos captures se faisant, dans un objectif de simplifier au maximum les aspects matériels de l'interfaçage, de façon totalement logicielle, il nous faut absolument abaisser cette vitesse pour atteindre au mieux 2 images par seconde. Pour ce faire il nous faut *écrire* dans le registre 0x11 (facteur de division de l'horloge interne), fonction qui est documentée comme impossible dans la datasheet (registre 0x11 en lecture uniquement). Nous avons pu vérifier, tel que décrit dans [13], que le registre 0x11 est bien accessible en écriture.

L'adresse de la caméra n'est pas non-plus nécessairement documentée, nécessitant une exploration systématique des 127 adresses paires possibles (le dernier bit de parité étant utilisé dans le protocole I^2C pour indiquer une lecture – valeur impaire – ou une écriture – valeur paire). Par exemple alors que le capteur CMOS OV6620 que nous utilisons est documenté correctement comme répondant aux accès aux adresses 0xC0 et 0xC1, une exploration des adresses est nécessaire pour découvrir que la caméra 3 Mpixels OV3620 répond à l'adresse 0x60 (adresse paire donc en écriture, l'assignation des registres intéressants étant la même que pour l'OV6620).

5.3.2 Capture d'une image

D'un point de vue matériel, la seule contrainte de connexion d'une caméra OV6620 au Coldfire 5282 est l'abaissement des signaux 5 V de la caméra en une logique 3,3 V pour le processeur, opération effectuée par des latchs 74LV573 alimentés en 3,3 V (fig. 4). Une contrainte supplémentaire provient de la rapidité des signaux lus : le transfert le plus rapide que peut supporter une lecture totalement logicielle des signaux issus de la caméra tel que décrit ici implique une période des signaux de pixels de 2,5 microsecondes (pour un taux de transfert de 2 images/seconde). Or nous avons constaté que la capacité électrique de certaines entrées – notamment les bits servant aussi de convertisseur analogique-numérique – est trop importante pour supporter des transitions aussi rapides. Nous précisons donc de connecter les latchs directement au bus de données du Coldfire 5282 (signaux D0-D7) et de n'imposer la valeur du bus (signal OE# des latchs) que lors de la lecture d'une adresse libre déclenchant un des signaux Chip Select (CSi# ; i=1..3 du Coldfire). Ce processeur a en effet l'avantage d'inclure en interne un décodeur d'adresse qui déclenche les signaux CSi# pour les conditions décrites dans le tableau 1.

Signal	Plage d'adresses	broche SSV
CS1#	0x1000.000-0x100F.FFFF	48
CS2#	0x1010.000-0x101F.FFFF	47
CS3#	0x1020.000-0x102F.FFFF	46

Tab. 1 – Tableau récapitulatif des adresses associées à chaque signal de Chip Select disponible en sortie du Coldfire 5282. Nous utilisons pour notre application CS2# et CS3#.

5.3.3 Capture d'une image couleur

Trois signaux sont analysés lors de la capture d'une image : VSYNC annonce par un passage au niveau bas le début d'une nouvelle image, HREF annonce tant qu'il est au niveau haut la validité des pixels le long d'une ligne (le passage au niveau bas de HREF est donc le signal de début d'une nouvelle ligne) et finalement PCLK annonce la validité (sur son front montant) d'une valeur de pixel. Les pixels sont soit transférés comme deux mots de 8 bits présentés simultanément sur les bus Y et UV de la caméra, ou comme un octet de 8 bits sur le port Y (UV étant alors inutilisé). Deux points importants sont à noter :

1. PCLK continue à osciller pendant que HREF est au niveau bas. Le test de ce second signal pour valider un pixel est donc fondamental (il n'est pas suffisant de tester uniquement PCLK et négliger HREF)
2. le rapport cyclique du signal issu de PCLK est 50% en mode 16 bits (tel que annoncé dans la datasheet), mais en mode 8 bits ce signal n'est qu'un fin pulse d'une centaine de nanosecondes

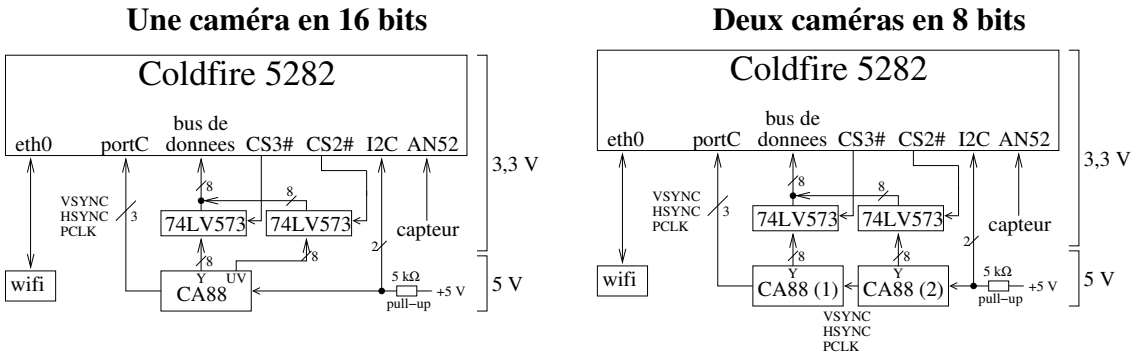

FIG. 4 – Schéma de principe de la connection des caméras CA88 au Coldfire 5282 avec les 74LV573 chargés de convertir la logique 5 V en 3,3 V requis par la carte embarquée, de la commande I^2C des caméras, et éventuellement d'une entrée analogique pour une mesure et transmission de paramètre.

même en abaissant la fréquence de l'horloge interne après écriture dans le registre 0x11. Afin de transformer le signal issu de PCLK en un pulse suffisamment long pour être détecté avec certitude de façon logicielle, un circuit additionnel est inséré entre la caméra et le Coldfire afin d'allonger PCLK à une valeur constante déterminée par la constante de temps du filtre RC présenté fig. 5. Nous avons choisi un circuit à base de trigger de Schmitt car ce composant était disponible, mais de nombreuses autres solutions (par exemple à base de timer 555 monté en monostable) sont possibles.

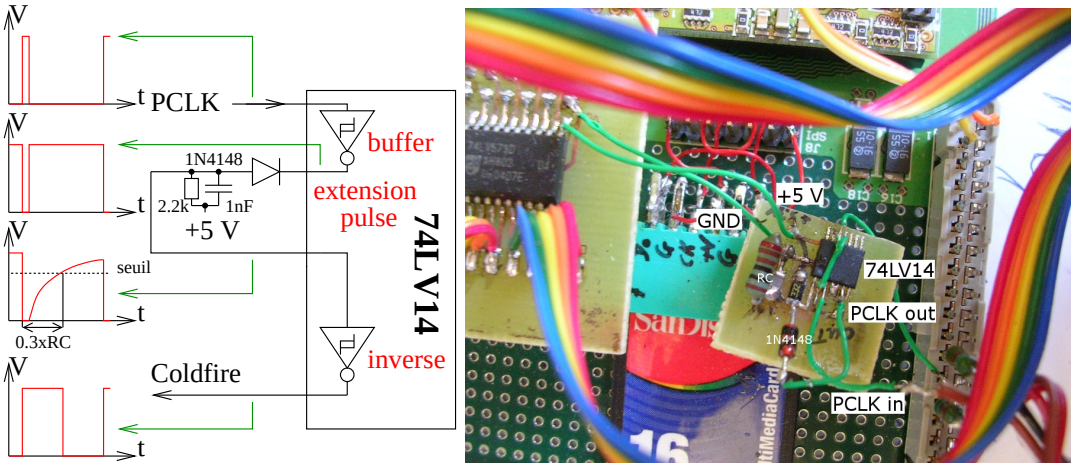

FIG. 5 – Circuit pour l'élargissement des impulsions PCLK lors de l'utilisation de la caméra en mode 8 bits. Le premier inverseur sert de tampon dont la sortie charge le circuit RC au travers d'une diode. Après la fin du pulse court (<100 ns) qui a servi à décharger rapidement le condensateur, la diode empêche la recharge du condensateur par la sortie du trigger de schmitt et la largeur de l'impulsion résultante τ telle que détectée par le Coldfire est déterminée par la constante de temps du circuit RC : $\tau \simeq 0.3RC$. Ce circuit est inutile en mode 16 bits où le rapport cyclique de PCLK est de 50% (tel que décrit dans la datasheet de l'OV6620) mais n'en affecte pas son fonctionnement : il peut donc rester connecté quel que soit le mode d'utilisation de la caméra.

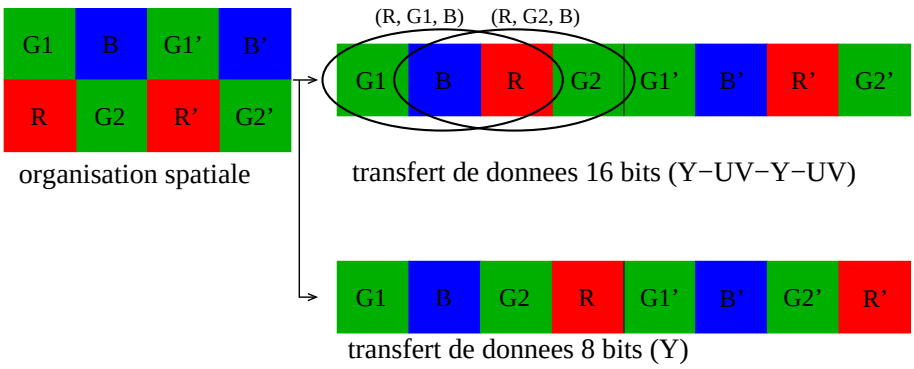

FIG. 6 – Organisation des données telles que reçues du capteur CMOS OVL6620 (G_1, B, R, G_2 soit deux pixels verts pour chaque pixel rouge et bleu), et la conversion que nous avons choisi d'utiliser pour générer deux pixels (R, G_1, B) et (R, G_2, B) pour chaque quartet de données acquis.

5.3.4 Organisation d'un capteur CMOS couleur

Une erreur de documentation concernant le mode 8 bits nous est apparu dans la datasheet de l'OV6620 que nous rectifions ici avec le schéma de la fig. 6. En effet contrairement à ce qui semble être indiqué dans la documentation, la sortie en mode 8 bits est une alternance de valeurs RG et BG (et non une ligne complète de BG suivi d'une ligne complète de RG tel que nous interprétons la fig. 4 (bas) de la datasheet de l'OV6620, version 1.4 datée du 13 Mai 2000). Nous vérifions la validité de notre analyse en convertissant les données brutes capturées sur les bus de la caméra (présentées ici comme une image en tons de gris) en des images couleur (format PPM) qui présentent bien ce qui est observé en réalité (notamment l'arc en ciel de la nappe de câble), tel que présenté sur la fig. 7.

Nous avons implémenté un couple client/serveur où :

1. un module noyau capture les images et les transfère en espace utilisateur lors de la requête `read()`. Ce module noyau désactive les interruptions au cours d'une capture d'image (pendant environ une demi seconde : fonctions `save_flags(flags)` ; dans un `unsigned long flags` ; suivi de `cli()` ; – les interruptions sont réactivées en fin de capture d'une image par `restore_flags(flags)` ; qui implicitement appelle `sti()` ;) faute de quoi un certain nombre de pixels est perdu au cours d'une acquisition.
2. le programme utilisateur est dans ce cas un serveur multithreadé (librairie `pthread` fournie par uClinux)
3. plusieurs clients fonctionnant sous unix se connectent au serveur, réceptionnent les images transférés par ethernet (TCP/IP), convertissent les données brutes lues des caméras en images RGB pour affichage par les fonctions X11 (fonctions `XCreateSimpleWindow()` et `XPutImage()`). En convertissant les données brutes en image RGB au niveau du client, nous gagnons un facteur 2/3 en taille des paquets transférés (4 octets donnent 2 pixels RGB). Nous avons implémenté une compression sans perte (telle que fournie par la `libz` de uClinux) mais les résultats sont décevants : la compression est longue et ne permet de gagner qu'environ 15% sur la taille des données à transférer pour une image typique.

Un problème majeur rencontré en situation où la caméra est mobile (par exemple embarquée dans un ballon captif) est qu'en abaissant l'horloge interne du capteur CMOS pour permette la lecture logicielle des valeurs des pixels, nous augmentons dans les mêmes proportions le temps d'exposition. La plupart des images obtenues sont alors déformées tel que illustré sur la figure 8 (droite).

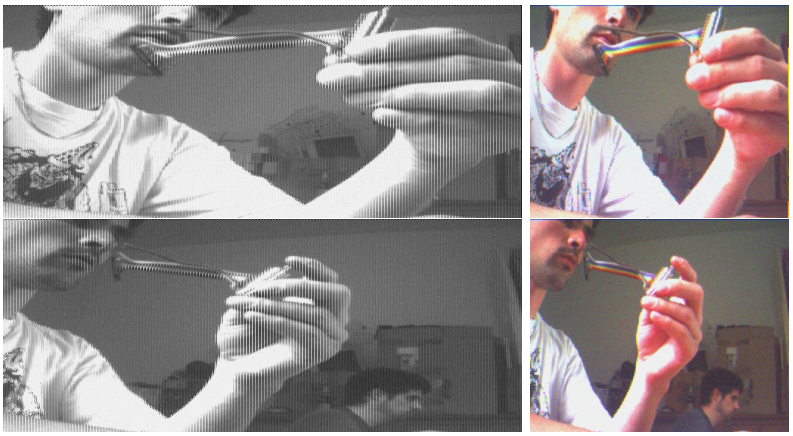

FIG. 7 – En haut : données brutes issues de la caméra présentées sous forme d'une image en tons de gris (mode 8 bits, RGB) et conversion de ces données brutes en une image couleur par réorganisation de l'ordre des octets en des séquences RGB tel que présenté à la fig. 6. Bas : idem pour le mode 16 bits.

FIG. 8 – Gauche : exemple de prise de vue aérienne au moyen d'une caméra CA88 connectée au Coldfire 5282 du parc de l'Observatoire de Besançon. Droite : image d'un bâtiment de l'Observatoire déformée par le mouvement du ballon captif auquel est fixé le circuit de prise de vues.

6 Conclusion

Nous avons présenté dans ce document une brève description d'une carte supportant le noyau uClinux tout en apportant une puissance de calcul importante, une quantité de mémoire non-volatile et volatile suffisante pour la plupart des applications embarquées, et une large panoplie de ports de communication, pour une prix de l'ordre de la centaine d'euros. Nous avons illustré le mode de programmation de la plupart des ces périphériques par la mise en œvre d'applications concrètes : stockage sur carte MultiMediaCard, interfaçage de capteurs sur les ports de conversion analogique-numérique, interfaçage d'une caméra couleur. Ces résultats ont nécessité un certain nombre de développements logiciels pour palier aux limitations d'uClinux et s'adapter à cet environnement de développement : modules noyaux pour accéder aux registres contrôlant le matériel fourni avec le processeur Coldfire 5282, méthodes efficaces de gestion et d'occupation de la mémoire.

Nous présenterons ultérieurement quelques applications concrètes d'instruments embarqués basées sur ce matériel.

Remerciements

Nous remercions F. Vernotte, directeur de l'Observatoire de Besançon, pour son hébergement gracieux de l'association étudiante Projet Aurore dans ses locaux, ainsi que Christian Ferrandez (FEMTO-ST/LPMO, Besançon) pour le prêt de la carte SSV et de l'OV3620. Vincent Giordanno et l'équipe "métrologie des oscillateurs" du laboratoire FEMTO-ST/LPMO nous ont fourni l'helium pour la ballon captif. L'association de diffusion des logiciels libres sur la Franche-Comté – Sequanux (www.sequanux.org) – est remerciée pour son support logistique.

Références

[1] http://www.dilnetpc.com/dnp0038.htm

[2] http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=0162468rH3YTLCFqnN

[3] J.-M Friedt, S. Guinot, *Introduction au Coldfire 5272*, GNU/Linux Magazine France **73** (Juin 2005) pp.26-33

[4] *MCF5282 ColdFire Microcontroller User's Manual* disponible à http://www.freescale.com/files/dsp/doc/ref_manual/MCF5282UM.pdf

[5] R. Berry, V. Kanto, J. Munger, *The CCD Camera Cookbook*, Willmann-Bell

[6] http://www.lextronic.fr/Capteurs/CAMERA.htm

[7] http://friedtj.free.fr/mmc_pport.pdf

[8] http://friedtj.free.fr/aduc816.pdf

[9] *SanDisk MultiMediaCard & Motorola 8 bit microcontroller Interface Design Reference Example* Rev. 2 (04/1999), Sandisk

[10] Datasheet *HB288016MM1 MultiMediaCard 16 MByte* Rev 0.1 (Nov. 24, 1999), Hitachi, ou *HB28E016MM2* disponible à http://www.ulrichradig.de/site/atmel/avr_mmcsd/pdf/hitachi_hb28b128mm2.pdf

[11] http://www.ulrichradig.de/site/atmel/avr_mmcsd/pdf/MMCSDTimming.pdf

[12] D. McCullough, Why is Malloc Different Under uClinux ?, disponible à http://www.linuxdevices.com/articles/AT7777470166.html

[13] I.N. Oiza, *Digital Camera Interface* (Mai 2004), disponible à http://www.robozes.com/inaki/dproject