La réception d'images météorologiques issues de satellites : utilisation d'un système embarqué

Simon Guinot[1,2], Jean-Michel Friedt[1]
[1] Association Projet Aurore, UFR-ST La Bouloie,
16, route de Gray
25030 Besançon Cedex
[2] Alcôve, 15 avenue de l'Agent Sarre
92700 Colombe
simon.guinot@alcove.com, friedtj@free.fr
21 janvier 2006

Ayant présenté les principes de base concernant la génération d'images météorologiques par les satellites en orbite polaire basse, leur mode de transmission et les divers éléments nécessaires à la réception de ces signaux, nous allons appliquer les concepts développés auparavant concernant l'acquisition et le traitement des données pour la réalisation d'un système embarqué capable de stocker et traiter les signaux définissant de telles images.

1 Introduction et objectifs

Ayant maîtrisé l'acquisitition de données issues de satellites depuis un ordinateur fonctionnant sous GNU/Linux et équipé d'une carte son, nous nous sommes fixés pour objectif de recevoir une telle image au moyen d'un montage basé sur le Coldfire 5282 tournant sous uClinux [1]. Les applications possibles d'un tel montage sont multiples, et nous identifions notamment la capacité à placer un système complet de réception d'images satellites en milieu hostile (désertique ou latitude élevée) puisque la consommation d'un Coldfire est considérablement plus faible que celle d'un ordinateur portable ou même de PCs aux formats industriels Biscuit PC ou PC104. Une autre possibilité est d'élever l'antenne réceptrice (qui doit être en vue directe du satellite pour capter un signal) en altitude au moyen d'un ballon captif afin de se dégager des obstacles limitant l'angle solide de réception du signal radio. Une telle application est envisagée dans l'avenir mais n'a pas encore été réalisée : bien que le diagramme de rayonnement de l'antenne ne soit pas nécessairement amélioré en augmentant l'altitude telle que nous l'avons vu précédemment, il est probable que nous gagnerons en zone de couverture en s'affranchissant des obstacles à l'horizon.

2 Problématique

L'architecture et les ressources matérielles du Coldfire 5282 sont très différentes de celles d'un ordinateur type PC. Les techniques utilisées jusqu'alors pour recueillir les données audio vont devoir être adaptées à notre système embarqué.
Par exemple sur un PC, l'acquisition d'un signal analogique via une carte son ne pose pas de problèmes particuliers puisqu'un tel périphérique dispose d'un codec et d'une mémoire tampon dédiés à cet effet. Sur le Coldfire 5282, l'acquisition devra être réalisée en utilisant le module généraliste de conversion analogique-numérique (QADC : *Queued Analogic-to-Digital Converter*) [2, chapitre 28]. Malheureusement, ce dernier n'est pas supporté par le noyau uClinux. Le développement d'un pilote pour le module QADC du Coldfire 5282 va donc être l'étape la plus importante et aussi la plus délicate dans la réalisation de ce projet.
De même, il nous est impossible d'embarquer une application telle que `wxtoimg` sur le Coldfire. Nous développerons donc également un programme utilisateur. Ce dernier aura pour charge de transférer en espace utilisateur les données recueillies par le pilote et également de stocker ces dernières en vue d'un traitement ultérieur. Là encore, nous sommes confrontés à un problème commun à la plupart des systèmes embarqués : la gestion de la mémoire. En effet, la carte SSV DNP5280 dispose de 16 Mo de RAM [4]. Une partie de cette mémoire, environ 2 Mo, est nécessaire au fonctionnement du système, ce qui nous laisse environ 14 Mo de mémoire disponible. La taille que va occuper une image météo en mémoire peut être évaluée à l'aide de la formule suivante :

$$taille\_image = durée\_acquisition \times freq\_échantillonage \times taille\_échantillon$$

– la durée d'acquisition correspond au délais d'exposition de notre antenne aux ondes transmises par le satellite (environ 15 minutes pour obtenir une image complète),

– la taille d'un échantillon est comprise entre 8 et 16 bits, suivant la qualité d'image souhaitée,
– La fréquence d'échantillonage minimale pour obtenir une image visible est de 5 kHz. Une fréquence de 12 kHz donne une image de qualité correcte.

Tous ces calculs, nous permettent d'estimer la taille d'une image complète à environ 20 Mo. Soit plus que la totalité de la mémoire présente sur le système. Des compromis sur la qualité ou la taille (durée de l'acquisition) de l'image sont donc indispensables pour rendre possible son stockage.

Un autre problème lié à la mémoire sur un processeur comme le Coldfire 5282, ne disposant pas de MMU, est la fragmentation. En effet, comme évoqué dans un précédent article [1], il est impossible pour le noyau uClinux d'utiliser un gestionnaire de mémoire virtuelle sur un tel processeur. Ce qui signifie, qu'une allocation d'une certaine quantité de mémoire ne pourra réussir que si le système dispose d'une même quantité de mémoire libre et *contiguë*. Il est donc très probable que la fragmentation de la mémoire nous empêche de réussir une allocation de 14 Mo. Nous verrons comment le programme utilisateur contourne cette limitation et rend possible le stockage d'une image.

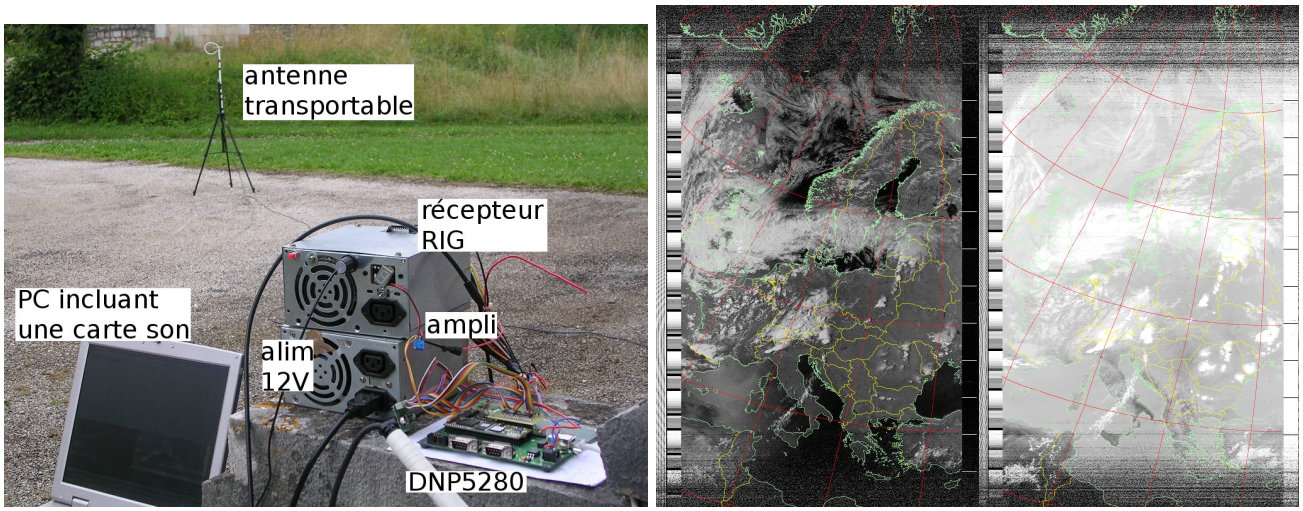

Fig. 1 – L'objectif final de cette présentation : remplacer le PC nécessaire à l'acquisition par sa carte son des données transmises par les satellites météorologiques en orbite basse par une carte SSV DNP5280 compatible avec une application embarquée transportable et de consommation réduite. Ce montage a été utilisé avec succès dans le parc de l'Observatoire de Besançon et lors de la conférence What The Hack en Hollande tel que illustré ici. Bas : exemple d'image obtenue depuis la Hollande au moyen de `wxtoimg` exécuté sur un PC portable équipé d'une carte son, illustrant l'intérêt de pouvoir placer un récepteur embarqué dans un lieu *a priori* difficile d'accès : cette image est la seule que nous ayons obtenue où un signal a été détecté alors que des terres au nord du cercle polaire arctique sont visibles depuis le satellite (Islande, Groenland et Sptizberg sont visibles sur les contours ajoutés par `wxtoimg`). La croix jaune indique l'emplacement de la station de réception, près de Eindhoven en Hollande.

3 Présentation du module QADC du Coldfire 5282

L'écriture d'un pilote pour le module QADC débute par la lecture des spécifications de ce dernier [2]. Cette opération est en effet un mal nécessaire pour se familiariser avec les registres matériels du module QADC. À noter que ce dernier, propose également une abondante variété de modes de fonctionnement. Malgré de nombreux échanges avec le support technique de Freescale, nous ne sommes pas parvenu à percer les mystères entourant certains d'entre eux.
Aussi cette section se limitera à décrire le mode de fonctionnement adéquat pour répondre aux contraintes imposées par notre application. C'est-à-dire : réaliser des conversions analogique-numériques de manière continue et ce à la fréquence la plus élevée possible.

3.1 Matériel et fonctionnalités

Le module QADC offre 8 ports de conversion analogique-numérique de 10 bits de résolution et un temps de conversion minimum de 7 μs [2, chapitre 33], extensible à 18 voies en utilisant un système de multiplexage externe. La tension lue est unipolaire, entre 0 et 3,3 V. Ces ports de conversion sont accessibles par l'intermédiaire des registres de données PORTQA et PORTQB. À noter qu'une configuration alternative permet de les utiliser comme des ports d'entrées-sorties numériques standards.
Le module QADC met à disposition de l'utilisateur une table de conversion et un système de deux queues. Il est ainsi possible de combiner des acquisitions en provenance de ports et donc de sources différentes. Il est également possible de jouer sur les fréquences d'acquisition des différentes entrées.
Les lignes d'interruptions 37 à 40 du contrôleur INTC0 [1] sont réservées au module QADC. Chacune des deux queues dispose de deux vecteurs servant à signaler l'achèvement ou la suspension d'une conversion.
Un système de *timer* rend configurable la fréquence des acquisitions.
Des *trigger* externes ou logiciels peuvent être employés pour démarrer ce que nous appellerons un cycle de conversion.

Voici citées pêle-mêles les principales fonctionnalités du module QADC. Nous allons maintenant nous attacher à les décrire plus en détail et voir comment elles peuvent être employées pour construire notre application.

3.2 Queues et table de conversion

La table de conversion est représentée par le registre CCW (*Conversion Command Word Table*). Cette table comporte 64 entrées de 16 bits dont 10 bits sont réellement utilisés. Chacune de ces entrées est programmable et doit correspondre à une commande. À quelques exception près, une commande consiste juste à désigner un des 8 ports du module QADC grâce la combinaison de bits adéquate.

Le module QADC du Coldfire 5282 implémente 2 queues de conversion. Une queue peut être comparée à un fil d'exécution de conversions. Lorsqu'une queue démarre un cycle de conversion, elle parcourt les entrées de la table CCW une à une. À chacune de ces entrées, elle exécute la commande indiquée. Ce qui revient à réaliser une acquisition sur le port ou canal désigné. Il est également possible de configurer une entrée, via une commande spéciale pour demander à une queue de stopper ou de suspendre son exécution. Il ne peut y avoir qu'une seule queue active à un instant donné. Par contre les queues peuvent être configurées pour s'interrompre entre elles et parcourir des portions distinctes de la table CCW. Il est également possible de définir une fréquence d'acquisition différente pour les deux queues. Ces mécanismes permettent au cours d'un même cycle de conversion de procéder à des acquisitions sur des canaux différents à des fréquences différentes.

3.3 Mode d'acquisition continu

N'étant intéressé que par l'acquisition d'un signal sur un unique port, toutes les entrées de la table CCW désigneront le même canal. De même, n'ayant pas besoin d'alterner les fréquences d'acquisition, nous n'utiliserons qu'une queue de conversion, la première. Afin de réaliser un échantillonage continu, cette dernière devra être configurée pour parcourir indéfiniment la table de conversion. Lorsque la dernière entrée sera atteinte, elle devra automatiquement se repositionner sur la première... ce qui correspond au mode d'exécution de la queue appelé : *continuous-scan mode*. À noter que nous n'utiliserons que la première entrée de la table CCW. En effet, le délais de repositionnement de la queue en début de table ne correspond pas au délais qui espace l'exécution de deux entrée successives ordinaires. Ce problème empêche d'obtenir l'acquisition continue et périodique nécessaire à notre application. La seule solution pour s'assurer de la régularité des intervalles de temps entre deux acquisitions est de programmer une conversion uniquement sur la première entrée et de définir la seconde entrée comme une “fin de queue”. En *continuous-scan mode*, cela a pour effet de relancer la queue sur la première entrée de la table de conversion. Ainsi entre deux conversions, nous conservons le même intervalle de temps.

3.4 Registres matériels

Dans cette section, nous décrirons les registres du module QADC et principalement leurs fonctionnalités en rapport avec notre application. Le lecteur désireux d'en apprendre plus est vivement encouragé à consulter les spécifications du Coldfire 5282 [2].

3.4.1 Registres globaux

- QADCMCR : (*QADC Module Configuration Register*)
  Ce registre permet de configurer le mode de fonctionnement du module QADC (normal, stop ou debug). À noter l'existence d'un bit superviseur (*SUPV*) qui permet également de changer les permissions d'accès aux registres du module QADC. Si ce bit est positionné, un processus ne pourra accéder aux registres du module QADC qu'en mode noyau. À l'inverse s'il n'est pas positionné, un processus en mode utilisateur pourra manipuler les registres. Ce réglage est relativement intéressant car il permet de se familiariser avec le fonctionnement des convertisseurs analogique-numériques depuis un programme utilisateur (sans avoir à utiliser le module noyau `ssvhwa` fourni par SSV avec son kit de développement).

- CCW : (*Conversion Command Word Table*)
  Comme expliqué précédemment, cette table comporte 64 entrées de 16 bits (10 bits utiles). Chacune de ces entrées peut contenir une commande. La queue de conversion active exécutera ces commandes dans l'ordre ou elle les rencontrera en parcourant la table de l'entrée 0 à l'entrée 63.
  Le sous-registre IST (*Imput Sample Timer - 2 bits*) permet de configurer le temps que le module QADC passera pour réaliser une conversion. Ce registre permet donc d'agir sur la durée d'une acquisition et également sur sa qualité.
  Le sous-registre CHAN contient la commande proprement dite. Si il est utilisé pour désigner un port ou un canal, alors une conversion sera réalisée en provenance de ce dernier. Ce sous-registre peut également contenir quelques commandes spéciales, comme par exemple `stop` ou `pause`. Lorsqu'une queue rencontre puis exécute une telle commande elle stoppe ou suspend son exécution.

3.4.2 Registres de contrôles

- QACR0 : (*QADC Control Register 0*)
  Le premier des registres de contrôle définit le sous-registre QPR (*Prescaler clock divider* - 7 bits). Le QPR peut être utiliser pour agir sur la fréquence d'horloge du module QADC.

- QACR1 : (*QADC Control Register 1*)
  Il définit plusieurs sous-registres utilisables pour configurer le mode de fonctionnement de la queue 1.
  Le sous-registre CIE1 (1 bit) permet d'associer ou non une interruption matérielle à l'achèvement de la queue 1.
  Le sous-registre SSE1 (1 bit) permet d'émettre un *trigger* logiciel à destination de la queue 1. Une fois le signal reçu, cette dernière démarre immédiatement son exécution.
  Le sous-registre MQ1 (4 bits) définit le mode opératoire de la queue 1. C'est ici que le *continuous-scan mode* peut être sélectionné. C'est également par l'intermédiaire de ce sous-registre qu'un timer pour espacer deux conversions peut être définit. Ce timer sera utilisé par notre pilote pour influer sur la fréquence d'acquisition, tel que illustré sur la Fig. 2.

- QACR2 : (*QADC Control Register 2*)
  Ce registre présente toutes les fonctionnalités du registre QACR1 mais cette fois, appliquées à la queue 2. La seule différence est la présence d'un sous-registre BQ2. Il permet de définir à partir de quelle entrée de la table CCW, la queue 2 devra démarrer. C'est en jouant avec BQ2 que l'on va pouvoir obtenir un entrelacement des queue 1 et 2. N'utilisant pas la queue 2, notre pilote n'a donc pas besoin de renseigner le registre QACR2.

3.4.3 Registres de status

- QASR0 : (*QADC Status Register 0*)
  Ce registre offre à tout moment une vue sur l'état des deux queues. Le seul sous-registre intéressant pour notre pilote est le CF1 (queue 1). Il correspond au bit de complétion de la queue 1. Chaque fois que cette dernière achève un cycle de conversion, ce bit est positionné. Si une interruption matérielle est associée à cette fin de cycle, il devra être remis à 0 pour acquitter l'interruption.

- QASR1 : (*QADC Status Register 1*)
  Ce registre que nous n'utiliseront pas permet à tout moment de connaître la dernière commande exécutée par

chacune des deux queues dans la table CCW.

3.4.4 Registres de données ou assimilés

- PORTQA et PORTQB : (*Port QA and QB Data Register*)
  Ces registres ne sont utiles que dans le cas où les Ports QA et QB sont utilisés pour réaliser des entrées-sorties numériques. Dans le cas de conversions analogique-numériques, les données sont manipulées par le biais de registres dédiés.

- DDRQA et DDRQB : (*Port QA and QB Data Direction Register*)
  Ces registres permettent de configurer la direction des échanges donnée avec les ports QA et QB (entrée ou sortie). En mode analogique, nous veillerons à bien utiliser le mode "input".

- RJUUR : (*Right-Justified Unsigned Result Register*)
  Tout comme CCW, ce registre est une table de 64 entrées de 16 bits chacunes. Encore une fois, seuls 10 bits sont utilisés. Chacunes de ces entrées contient le résultat de la conversion effectuée par la commande CCW correspondante. Par exemple, une fois un cycle de conversion achevé, RJUUR[12] contiendra la conversion issue de la commande CCW[12]. Les données se présentent sous un format justifié à droite.
- LJSRR (*Left-Justified Signed Result Register*) et LJURR (*Left-Justified Unsigned Result Register*) :
  Ces deux registres fonctionnent exactement comme RJUUR à l'exception près que les données ne sont pas présentées sous le même format.

4 Implémentation du pilote qadc

Maintenant familier avec le fonctionnement du module QADC, il ne nous reste plus qu'à implémenter le module noyau uClinux, chargé de le piloter. Notre pilote – disponible sur `http://www.sequanux.org` et `http://jmfriedt.free.fr` – fonctionnera pour la version 2.4 du noyau uClinux et fournira une interface type caractère (`char device`) [3] aux applications utilisateurs. Étant donné le mode de fonctionnement des convertisseurs analogique-numériques, une communication caractère par caractère est la plus adaptée. Le module exportera donc, via un fichier spécial `/dev/qadc`, des méthodes(`open()`, `close()`, `read()`, `ioctl()`) pour les programmes utilisateurs.

La structure globale du pilote nous est dictée par les contraintes auxquelles il sera soumis... à savoir l'acquisition en continue d'un signal analogique et la transmission des données à l'espace utilisateur.

4.1 Conversions analogiques continues et temps-réel

Une série de conversions analogique-numériques peut être assimilée à une succession d'événements périodiques. Dans ce cas particulier, réaliser un échantillonage en temps-réel revient à traiter ces événements dans un délais inférieur à la durée d'une période. Plus la fréquence d'échantillonage sera élevée, plus la période sera courte et donc plus le traitement de l'information devra être rapide. Deux techniques différentes peuvent être envisagées pour gérer l'événement correspondant à une fin de conversion :

1. scruter activement (*polling*) le bit correspondant à la complétion d'un cycle de conversion
2. associer une interruption matérielle à la fin d'un cycle

4.1.1 Mode *polling*

Cette méthode est la plus simple à implémenter. Lorsqu'un processus accède à la méthode `read()` du pilote, il suffit de le placer dans une boucle où il scrute le registre QASR0 en attente de la complétion de la queue de conversion. La boucle d'attente pourrait ressembler à :

```
while ((QASR0 & 0x8000) == 0)
{};
/* traitement des donnes */
```

En plus de sa simplicité, cette méthode est également celle qui offre le délai le plus court entre l'achèvement d'une conversion et la prise en charge de l'événement.
Le problème avec ce genre de boucle est une très forte consommation de CPU. En effet, notre module est destiné à fonctionner sur un noyau 2.4 non préemptif. Ce qui signifie qu'un processus pris dans une telle boucle en mode noyau va accaparer le processeur et sera ininterruptible. Si l'événement attendu ne se produit jamais, la seule option sera de redémarrer manuellement le système. On peut envisager d'introduire des appels à la fonction `schedule_timeout ()` pour casser la boucle et rendre possible l'ordonnancement d'autres processus. La boucle deviendrait :

```
while ((QASR0 & 0x8000) == 0)
{
        set_current_state(TASK_INTERRUPTIBLE);
        schedule_timeout(delay);
};
/* traitement des donnes */
```

La variable `delay` doit être initialisée à une valeur en jiffies inférieure à l'intervalle séparant deux conversions. Même si cette solution est plus élégante, elle n'est néanmoins pas satisfaisante. L'ordonnanceur de base du noyau 2.4 peut être amené à élire un processus avec un temps de latence assez important (surtout en cas de charge CPU). Rien ne nous garantit que le timer sera respecté et donc que le processus sera réordonnancé dans un délais satisfaisant. Peut-être que l'événement attendu se sera produit depuis longtemps lorsqu'il récupérera le processeur...

Qui plus est, l'acquisition doit être continue, ce qui exclut que le processus ait à itérer des appels successifs à `read()`. Nous serions donc obligés de maintenir le processus en mode noyau après le premier appel et attendre la fin de l'acquisition pour le libérer et lui retourner enfin une énorme quantité de donnée. Il est évident que cette solution rencontrerait d'insurmontables problèmes d'allocation mémoire.

Au vue de cette analyse, il semble clair que le mode polling ne constitue pas une solution acceptable à notre problème.

4.1.2 Mode interruption

Une des fonctionnalités du module QADC est d'autoriser une queue à générer une interruption matérielle lorsqu'elle achève un cycle de conversions. Ce mécanisme permet à un pilote d'acquérir des données en continu sans pour autant immobiliser un processus en mode noyau au travers de l'appel système `read()`.
Le *driver* peut alors organiser le traitement de cette interruption en lui associant un gestionnaire (handler). Cette association est réalisée à l'aide de la fonction `request_irq()`. En fait les choses sont légèrement plus complexes... L'activation d'une interruption sur un processeur d'architecture Coldfire nécessite la configuration d'un certain nombre de registres. La fonction `request_irq` réalise juste l'initialisation logicielle. La partie bas-niveau est laissée à la charge du pilote. Ce dernier utilise pour cela la fonction `enable_interrupt_vector()`. Le mécanisme des interruptions sur Coldfire 5282 ayant été largement traité au cours d'un précédent article [1], nous invitons le lecteur curieux à s'y reporter. Il y trouvera entre autres l'implémentation de la fonction `enable_interrupt_vector()`.

Le rôle du gestionnaire est donc de récupérer puis de stocker dans un buffer les données issues de la conversion. La rapidité d'exécution du gestionnaire est un point critique. En effet, plus la fréquence d'acquisition sera élevée, moins le gestionnaire disposera de temps pour traiter l'information. De plus le délais de traitement doit être bien inférieur à la période séparant deux interruptions. Il doit rester suffisamment de temps CPU pour que le noyau puisse accomplir les tâches vitales et ordonnancer les processus utilisateurs.

Un autre facteur à définir est la priorité de notre interruption. Le module d'interruption du Coldfire 5282 permet d'affecter un niveau et une priorité à une ligne d'interruption [1]. Une interruption peut donc en interrompre une autre si son couple (niveau + priorité) est plus important. Tout retard pris par notre gestionnaire dans le traitement de l'interruption introduira des irrégularités au niveau échantillonage. Pour se mettre le plus possible à l'abri de ce problème, les niveau et priorité de la ligne d'interruption du module QADC ont été positionnés aux plus fortes valeurs possibles (respectivement 6 et 7).

Avant de présenter l'implémentation du gestionnaire d'interruption, il est indispensable d'introduire une des structure de données utilisée par notre pilote. Il s'agit du type `struct buffer`. Cette structure va servir à stocker le résultat des conversions en attendant de pouvoir les transférer à un processus utilisateur.

```
struct buffer
{
        unsigned short data[BUFFER_SIZE];
        unsigned short *data_start;
        unsigned short size;
};
```

Cette structure, relativement simple, modélise un tampon rotatif. Le champ `data` désigne le *buffer* destiné à recevoir les données. `size` est un compteur indiquant le nombre d'éléments contenus dans le *buffer* et `data_start` est un pointeur sur l'emplacement des données utiles dans le tampon.
Dans sa structure de données privée `struct qadc`, le pilote définit deux (`NB_BUFFER`) éléments de type `struct buffer`.

```
struct qadc
{
        unsigned int open_count;
        struct buffer buff[NB_BUFFER];
        struct buffer *write_ptr;
        struct buffer *read_ptr;
        struct semaphore qadc_sem; /* to protect this struct */
        wait_queue_head_t qadc_queue;
        struct irq_info *qadc_irq_info;
        unsigned char qadc_timer;
        unsigned char qadc_ist; /* input sample time configuration */
        unsigned char qadc_qpr; /* prescaler clock divider */
};
```

On remarque également la présence de deux pointeurs `write_ptr` et `read_ptr` sur des éléments de type `struct buffer`. Leur rôle sera mis en évidence au fur et à mesure de la présentation du pilote.

À la lumière de toutes ces informations, voici l'implémentation que nous proposons pour le gestionnaire d'interruptions :

```
static void qadc_irqhandler (int irq, void *dev_id, struct pt_regs *regs)
{
```

Dans un premier temps le *handler*, en consultant le paramètre `dev_id`, récupère le pointeur sur la structure de données privée du pilote.

```
        struct qadc *dev = (struct qadc *) dev_id;
        volatile unsigned short *qasr0 = (volatile unsigned short *) QASR0;
```

Puis il extrait du registre RJURR les données résultantes de la conversion et les place dans un tampon de type `struct buffer`. Ce tampon est désigné par le pointeur `write_ptr`. Pour éviter au gestionnaire des opérations complexes de manipulations de buffer et donc une perte de temps, nous avons décidé que ce dernier utiliserait un pointeur unique. Une partie du travail de la méthode `read()`, présentée dans la section suivante, est justement de définir vers quel buffer pointe réellement `write_ptr`. On remarque que le gestionnaire gère le débordement du tampon de réception par un système de modulo. Si les données n'ont pas été consommées assez rapidement, elles seront écrasées par de plus récentes.

```
        dev->write_ptr->data[dev->write_ptr->size] =
                *((unsigned short *) (RJURR)) & 0x03ff;

        dev->write_ptr->size =
                (dev->write_ptr->size + 1)%BUFFER_SIZE;
```

À partir du moment où le tampon est à moitié plein, le gestionnaire envoie des signaux pour réveiller les processus en attente de données. Le signal n'est pas envoyé plus tôt pour essayer d'optimiser la taille des transferts vers l'espace utilisateur. L'objectif est d'éviter de multiplier les copies de petits tampons.

```
        if (dev->write_ptr->size > (BUFFER_SIZE / 2))
                wake_up_interruptible (&dev->qadc_queue);
```

Il ne reste plus qu'à effacer le drapeau d'interruption pour signaler au module QADC que l'interruption a bien été traitée. La queue de conversion est alors réactivée et reprend l'exécution d'un cycle.

```
        *qasr0 &= 0x7fff;

        return;
}
```

4.2 Méthodes du pilote

Après avoir détaillé la technique utilisée par le pilote pour lire et stocker le résultat des conversions, nous allons maintenant présenter les méthodes exportées par ce dernier.

4.2.1 Méthode `open()`

Cette méthode est appelée lorsqu'un programme utilisateur ouvre le fichier spécial `/dev/qadc` en utilisant l'appel système `open()`. Ses principales fonctions sont :

1. procéder à l'initialisation des structures de données privées du pilote (allocations de mémoire... etc),
2. activer l'interruption matérielle du module QADC et lui associer un gestionnaire,
3. initialiser le module QADC et démarrer les acquisitions.

Commençons par illustrer les deux premiers points en présentant la méthode `open()` :

```
static int qadc_open (struct inode *inode, struct file *file)
{
        struct qadc *dev = NULL;
        int retval = 0, i;
        struct irq_info *qadc_irq_info = NULL;

        dev = (struct qadc *) file->private_data;

        dbg("enter %s", __FUNCTION__);

        MOD_INC_USE_COUNT;
```

S'il s'agit de la première utilisation du périphérique QADC, la structure de données privée du pilote est allouée puis initialisée.

Les champs contenant les réglages du module QADC sont renseignés avec des valeurs par défaut.

On remarquera également l'affectation des pointeurs `read_ptr` et `write_ptr` sur les deux tampons de type `struct buffer`.

```
        /* if it is the first open, it is time to
         * initiate some structure */
        if (!dev)
        {
                dbg("%s - alloc device", __FUNCTION__);
                dev = (struct qadc *) kmalloc
                        (sizeof (struct qadc), GFP_KERNEL);
                qadc_irq_info = (struct irq_info *) kmalloc
                        (sizeof (struct irq_info), GFP_KERNEL);
                if ((dev == NULL) || (qadc_irq_info == NULL))
                {
                        retval = -ENOMEM;
                        goto exit;
                }
                memset (dev, 0x00, sizeof (struct qadc));
                for (i = 0; i < NB_BUFFER; i++)
```

```
                {
                        dev->buff[i].size = 0;
                }
                dev->write_ptr = &dev->buff[0];
                dev->read_ptr = &dev->buff[1];
                dev->write_ptr->data_start = &dev->write_ptr->data[0];
                dev->read_ptr->data_start = &dev->read_ptr->data[0];
                dev->qadc_timer = 0x14;
                dev->qadc_qpr = DFLT_QPR;
                dev->qadc_ist = MAX_IST;
                init_waitqueue_head (&dev->qadc_queue);
                sema_init (&dev->qadc_sem, 1);
                dev->open_count = 0;
                file->private_data = dev;
        }

        /* lock this device */
        if (down_interruptible (&dev->qadc_sem))
        {
                retval = -ERESTARTSYS;
                goto exit_free;
        }
```

La portion de code suivante permet l'activation des interruptions et procède à l'appel de la fonction `start_qadc()`. Cette dernière est chargée d'initialiser le module QADC et de démarrer les acquisitions.

```
        /* initiate the qadc and enable the corresponding interrupt */
        if (dev->open_count == 0)
        {
                stop_qadc ();
                /* initiate the structure to request our interrupt vector */
                sprintf (qadc_irq_info->name, "adc_int");
                qadc_irq_info->vector = COMPLETE_Q1_IRQ + INTC0_VECT_MIN;
                qadc_irq_info->level = 0x37; /* level 6 and pririty 7 */
                qadc_irq_info->handler = qadc_irqhandler;
                qadc_irq_info->dev_id = (void *) dev;
                if ((retval = enable_interrupt_vector (qadc_irq_info)))
                {
                        dbg("can't enable qadc interrupt vector");
                        goto exit_sem;
                }
                dev->qadc_irq_info = qadc_irq_info;
                start_qadc (dev);
        }

        /* increment our own counter */
        ++dev->open_count;

        dbg("%s - open_count = %d", __FUNCTION__, dev->open_count);

exit_sem :
        up (&dev->qadc_sem);

exit_free :
        if (retval)
                kfree (dev);

exit :
        if (retval)
```

```
                MOD_DEC_USE_COUNT;

        return (retval);
}
```

Examinons maintenant la fonction `start_qadc()` au travers de laquelle la méthode `open()` procède à l'initialisation et l'activation du module QADC :

```
static void start_qadc (struct qadc *dev)
{
        volatile unsigned short *ccw = (volatile unsigned short *) (CCW);

        dbg("enter %s", __FUNCTION__);
```

Le registre QADCMCR est renseigné pour placer le module QADC en mode acquisition analogique et le registre de direction DDRQA est configuré pour que les canaux du port QA soient utilisés en mode *input*.

```
        /* QADC register initialisation */
        *((volatile unsigned short *) QADCMCR) = 0x0000; /* control register */
        *((volatile unsigned short *) DDRQA) &= 0xff00; /* input mode for QA ports */
```

Le registre QACR0 est initialisé avec le diviseur d'horloge fourni par l'utilisateur. Plus il sera petit, plus la fréquence d'horloge du module QADC sera importante et donc plus les conversions seront rapides... En pratique nous ne toucherons pas à ce paramètre et utiliserons la valeur préconisée pour la fréquence d'horloge du Coldfire.

```
        if (dev->qadc_qpr > QPR_MAX )
        {
                dbg("bad prescaler clock diviser... fix it to default value");
                dev->qadc_qpr = DFLT_QPR;
        }

        *((volatile unsigned char *) QACR0) = dev->qadc_qpr; /* set the clock */

        if (dev->qadc_ist > MAX_IST)
        {
                dbg("bad input sample timer... fix it to default value");
                dev->qadc_ist = DFLT_IST;
        }
```

On construit maintenant la table de conversion (CCW). La première entrée est configurée pour désigner le port analogique AN52, sur lequel est branchée la sortie audio du récepteur radiofréquence. L'ISP (*Input Sample Time Register*) est également renseigné. En faisant varier ce paramètre, l'utilisateur pourra influer sur le temps de conversion du signal analogique et donc la qualité des acquisitions.
La seconde entrée de la table CCW est renseignée avec le code de fin de queue. En *continuous-scan mode*, cela aura pour effet de relancer la queue 1 sur la première entrée.

```
        /* construct the CCW table... we set PQA0 or AN52
         * as input channel */
        ccw[0] = (dev->qadc_ist << 6) | 52;
        ccw[1] = 63; /* end of queue */

        /* enable periodic timer continuous scan mode for queue 1 and
         * associate an interruption to the queue 1 end */

        if ((!(dev->qadc_timer <= TIMER_MAX)) &&
                        (!(dev->qadc_timer >= TIMER_MIN)))
        {
                dbg("bad timer value... use the no timer configuration");
                dev->qadc_timer = NO_TIMER;
        }
```

Le registre QACR1 est renseigné pour activer le *continuous-scan mode*. Le paramètre `qadc_timer` permet de définir le timer influant sur le délais espaçant deux conversions. Si cette variable est définie à `NO_TIMER` (`0x11`), le `timer` ne sera pas utilisé et les conversions seront effectuées à pleine vitesse.
En pratique, c'est en jouant avec ce timer que nous abaisserons ou augmenterons la fréquence des acquisitions.

```
        *((volatile unsigned short *) QACR1) = ((1 << 15) |
                        (dev->qadc_timer << 8));
```

Et finalement le bit *trigger* du registre QACR1 est positionné afin de démarrer les conversions.

```
        /* send a software trigger to start conversion */
        *((volatile unsigned short *) QACR1) |= 0x2000;

        return;
}
```

4.2.2 Méthode `read()`

Cette méthode va permettre de transférer les données dans l'espace utilisateur du processus appelant. Nous avons vu que le module noyau est parfaitement capable grâce à son gestionnaire d'interruption de lire et stocker les données numérisées. Cependant, un programme utilisateur devra régulièrement lire ces données *via* l'appel système `read()`. Si cette opération n'est pas effectuée suffisamment fréquemment, les informations les plus vieilles seront perdues.

Voici maintenant comment le pilote implémente la méthode `read()` :

```
static ssize_t qadc_read (struct file *file, char *buffer, size_t count, loff_t *ppos)
{
        struct qadc *dev = (struct qadc *) file->private_data;
        struct buffer *tmp;
        unsigned short size = 0;

        if (!dev)
                return (-ENODEV);
```

S'il reste encore des données dans le tampon d'envoi `read_ptr`, alors l'exécution se poursuit au label `exit_copy_to`. Les données seront recopiées dans l'espace utilisateur.

```
        if (dev->read_ptr->size)
        {
                goto exit_copy_to;
        }
```

Si le tampon d'envoi est vide, on le fait pointer vers le tampon de réception. Le processus est ensuite placé en sommeil en attente d'un signal. Ce dernier est émis par le gestionnaire d'interruption à partir du moment où le tampon de réception est à moitié plein.

```
        dev->read_ptr->data_start = dev->read_ptr->data;
        tmp = dev->read_ptr;
        dev->read_ptr = dev->write_ptr;

        /* sleep until a interruption send a wake up signal */
        if (wait_event_interruptible (dev->qadc_queue,
                                (dev->read_ptr->size > (BUFFER_SIZE / 2))))
                return (-ERESTARTSYS);
```

Le pointeur du tampon de réception est maintenant affecté au tampon vide (le précédent tampon d'envoi). L'opération est transparente pour le gestionnaire d'interruption qui continue à manipuler `write_ptr` indépendamment du tampon qu'il désigne.

```
        dev->write_ptr = tmp;
```

Il ne reste finalement plus qu'à recopier les données dans l'espace utilisateur.

```
exit_copy_to :

        size = ((dev->read_ptr->size * sizeof (unsigned short)) < count)?
                (dev->read_ptr->size * sizeof (unsigned short)) : count;

        __copy_to_user (buffer, (unsigned char *) dev->read_ptr->data_start, size);

        dev->read_ptr->data_start += (size / sizeof (unsigned short));
        dev->read_ptr->size -= (size / sizeof (unsigned short));

        return (size);
}
```

4.3 Validation des acquisitions

Nous avons dans un premier temps testé ce pilote sur un signal connu issu d'un générateur basse fréquence réglé pour founir une sinusoïde d'amplitude environ 0,8 V et de fréquence 240 Hz (pour tester la continuité des courbes acquises) ou 2400 Hz (pour identifier la fréquence d'échantillonage selon les paramètres d'acquisition).

Les résultats de ces acquisitions et leurs interprétations sont illustrées sur la Fig. 2. Le graphique inséré illustre la continuité entre les buffers successifs : ici une sinusoïde à 240 Hz est échantillonée et stockée dans des buffers de 200 données avant d'être transmise à l'espace utilisateur. Aucune discontinuité n'est visible entre les données issues de buffers successifs.

Au-delà de ces méthodes visuelles qualitatives et ne permettant d'appréhender que localement le comportement des algorithmes, nous nous sommes intéressés à la transformée de Fourier de signaux issus d'acquisitions longues (25000 points). La transformée de Fourier d'une sinusoïde devrait être un pic dans l'espace des fréquences d'autant plus fin que l'acquisition a été longue et régulière. Toute discontinuité dans les acquisitions (introduction d'une phase entre les buffers successifs) se traduira par un élargissement des pics, tandis que des fluctuations de la fréquence d'échantillonage se traduiront par des pics multiples. Nous travaillons ici à fréquence de signal constante et fréquence d'échantillonage f_{ech} inconnue : sachant qu'une transformée de Fourier sur N points distribue dans la plage $-N/2$ à $N/2$ (entiers) les fréquences $-f_{ech}/2$ à $f_{ech}/2$, la position du pic P caractérisant un signal de fréquence connue f_0 nous donne la fréquence d'échantillonage par $f_{ech} = \frac{N}{P} \times f_0$. Étant donné que nous ne nous intéressons qu'à des signaux réels, le spectre allant de $-f_{ech}/2$ à 0 est égal à celui de 0 à $f_{ech}/2$ et seule cette seconde moitié est tracée sur la Fig. 2.

Nous avons donc tracé ici la transformée de Fourier des données acquises sur une sinusoïde à 2400 Hz fournie par un générateur basse fréquence supposé idéal. Dans le cas de l'algorithme asservi sur une timer, nous constatons que le signal est pur d'un point de vue spectral (courbe rouge), et suivant le raisonnement formulé précédemment, nous déduisons que lorsque le pic se situe au point 197 la fréquence d'échantillonage est $2 \times \frac{2400 \times 512}{197} = 12475$ échantillons/s. En l'absence de timer (courbes verte et bleue), nous constatons que le signal original spectralement pur fournit parfois deux pics : l'algorithme d'acquisitions de données s'est ici exécuté selon deux fréquences d'échantillonage déterminées par la charge du processeur au moment de l'acquisition. Ces fréquences d'échantillonage sont plus élevées – 28249 et 22341 Hz – mais les fluctuations observées rendent le traitement ultérieur des données difficiles. Toutes ces courbes sont issues d'acquisitions sur 25000 points. Cette multiplication des pics n'est pas systématiquement observée (courbe bleue) et illustre l'incertitude liée à ce mode de fonctionnement de la QADC : nous favoriserons désormais le mode asservi sur un timer qui garantit un intervalle de temps fixe entre deux acquisitions.

5 Application utilisateur

Une fois le développement du pilote achevé, il ne reste plus qu'à écrire une application utilisateur pour lire le résultat des conversions analogique-numériques. Cet exercice est beaucoup plus simple et relève de la programmation système classique. Il reste cependant une dernière difficulté à gérer, le stockage des données retournées par le pilote. Comme il a été précisé dans l'introduction, le volume d'informations à stocker est relativement important (environ 12-14 Mo pour une image correcte). Cependant, dans notre cas particulier, le problème lié l'absence de mémoire contiguë et à la fragmentation peut être contourné. Il suffit juste de constituer un pool de blocs mémoire libre non contigus. Ces

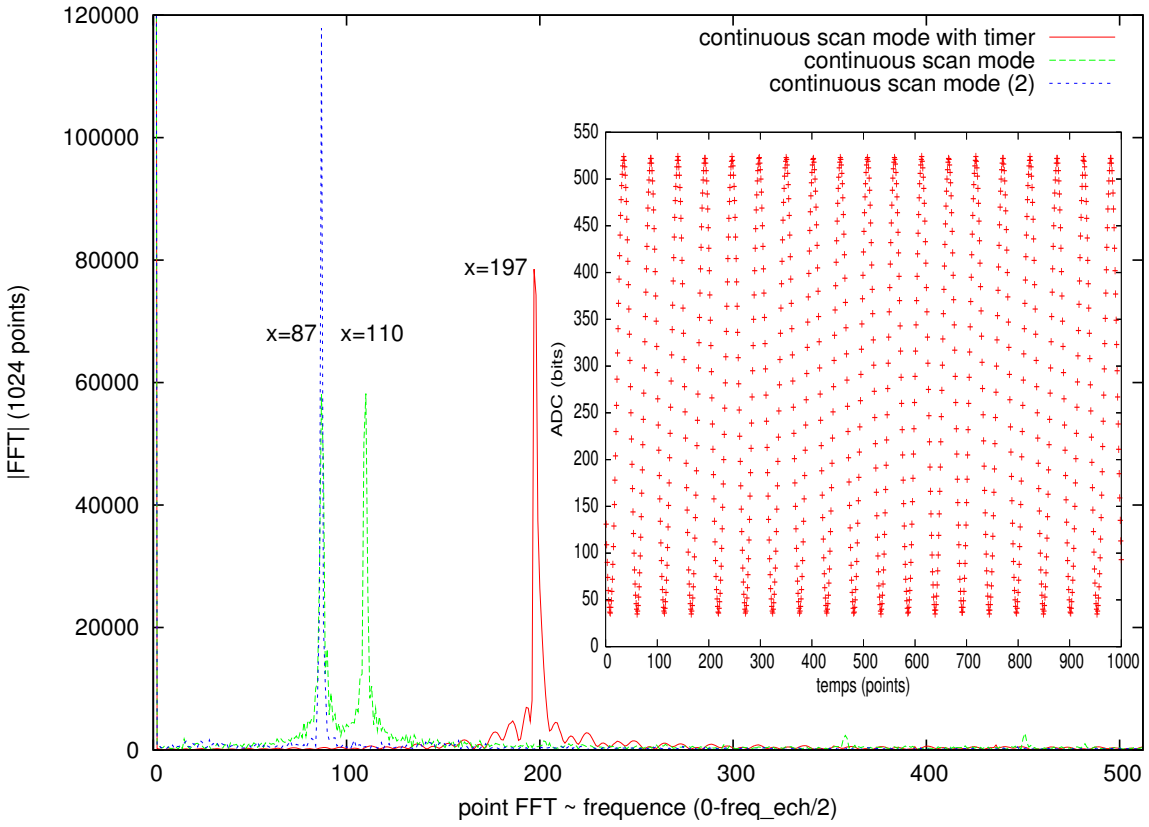

FIG. 2 – Acquisition selon divers algorithmes d'un signal analogique fourni par une générateur basse fréquence formé d'une sinusoïde. Les courbes insérées sont l'évolution temporelle du signal, tandis que le graphique principal présente les transformées de Fourier de certain de ces signaux.

blocs mémoires sont ensuite reliés les uns aux autres grâce à une liste chaînée. Chacun des éléments de la liste est une structure (`struct buffer`) contenant un pointeur sur l'adresse un bloc de mémoire utilisable et indiquant la taille de ce dernier.

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

L'allocation mémoire et la constitution de la liste chaînée est réalisée par un algorithme récursif. Ce dernier tente d'allouer un bloc avec le maximum de mémoire disponible. Une fois ce bloc alloué, un appel récursif pour un autre bloc est réalisé. Lorsque la quantité de mémoire demandée a été allouée, une liste chaînée de blocs est retournée.
Voici un aperçu de la fonction principale :

```
struct buffer * alloc_mem (unsigned int size, unsigned int *get_size)
{
        unsigned int elt_size = size;
        struct buffer *buff = NULL;

        if (size == 0)
                return (NULL);
```

```
        while ((buff = alloc_buffer (elt_size)) == NULL)
        {
                fprintf (stderr, "echec to allocate %d bytes : %d\n", elt_size);

                if (elt_size < MIN_SIZE)
                        return (buff);

                elt_size = (unsigned int) ((elt_size / 10) * 9);
        }
        fprintf (stderr, "succes to allocate %d bytes\n", elt_size);

        buff->next = alloc_mem (size - elt_size, get_size);
        *get_size += elt_size;

        return (buff);
}
```

L'application se contentera de communiquer avec le pilote via les méthodes exportées par ce dernier. Voici un aperçu de son implémentation :

```
int main (int argc, char **argv)
{
        int desc, nb_read, i, ask_size, get_size = 0;
        struct buffer *head, *ptr;

    [...]

        head = alloc_mem (ask_size, &get_size);
        fprintf (stderr, "allocate %d bytes for a %d bytes request\n",
                        get_size, ask_size);
        fflush (stderr);
        if (get_size < ask_size)
        {
                fprintf (stderr, "error... to small allocation\n");
                free_mem (head);
                return (-1);
        }
        fflush (stderr);
```

Le processus bascule en mode noyau par l'intermédiaire de la méthode `open()` du *driver* et procède à l'initialisation du module QADC puis au démarrage des acquisitions.

```
        if ((desc = open ("/dev/qadc", O_RDONLY)) == -1)
        {
                perror ("open ()");
                return (-1);
        }
```

La méthode `ioctl()` du pilote est utilisée pour espacer d'un *timer* chaque conversion. Un échantillonage à pleine vitesse ne nous permettrait pas de stocker une image complète. Aussi nous sommes forcés à quelques concessions au niveau de la qualité de l'image.
En effet, en diminuant la fréquence d'acquisition, on réduit le nombre de points à stocker en mémoire mais on réduit également la définition de l'image finale.

```
        if ((ioctl (desc, CMD_TIMER, 0x14)) < 0)
        {
                perror ("ioctl (CMD_TIMER)");
                return (-1);
        }
```

```
        ptr = head;
        while (ptr != NULL)
        {
                if ((get_qadc_result (desc, ptr)) == -1)
                        break;

                ptr = ptr->next;
        }
```

Une fois l'acquisition achevée et les données enregistrées en mémoire, il ne reste plus qu'à afficher ces dernières sur la sortie standard en appelant la fonction `show_result()`. La sortie standard peut par exemple être redirigée vers un fichier importé d'une machine distante par NFS, ou une mémoire de stockage de masse non volatile interfacée à un des nombreux ports d'entrée-sortie encore disponibles sur le Coldfire.

```
        while (ptr != NULL)
        {
                show_result (ptr);
                ptr = ptr->next;
        }
        free_mem (head);
        return (0);
}
```

Et voici enfin la fonction `get_qadc_result()` qui appelle la méthode `read()` du pilote et stocke les données dans la liste chaînée de tampons.

```
int get_qadc_result (int desc, struct buffer *buff)
{
        int size = buff->size;
        unsigned int nb_read;
        unsigned char *result = buff->data;

        while (size > 2)
        {
                if ((nb_read = read (desc,
                        (unsigned char *) result, size)) == -1)
                {
                        perror ("read ()");
                        return (-1);
                }
                size -= nb_read;
                fprintf (stderr, "size : %d - nbread : %d\n", size, nb_read);
                fflush (stderr);
                ((unsigned char *) result) += nb_read;
        }
        return (0);
}
```

6 Résultats

Le logiciel présenté ci-dessus a été mis en œuvre pour remplacer l'utilisation d'un ordinateur personnel équipé d'une carte son par un processeur Coldfire équipant une carte SSV DNP5280 lors de l'enregistrement et du traitement de données issues du récepteur radiofréquence du RIG décrit dans l'article précédent. Il est à noter qu'aussi bien le montage de réception radio que le montage à base de Coldfire sont alimentés via des régulateurs de tension par une batterie de tension variable de l'ordre de 12 V : la consommation totale est de l'ordre de 400 mA, soit une autonomie de plusieurs heures sur des batteries LiPo commercialement disponibles avec un encombrement et un poids total du montage compatible avec un transport en sac à dos. L'élément le plus encombrant – l'antenne – a dans notre cas été monté de façon à pouvoir retirer les rayons des dipôles de leur support, lui même fixé à un pied d'appareil photo, pour

finalement donner un montage transportable. La seule modification électronique apportée par rapport à la description faite dans l'article précédent est l'ajout d'un amplificateur opérationnel monté en amplificateur afin d'ajuster finement le niveau du signal sonore transmis au convertisseur analogique-numérique du Coldfire (qui s'attend à une tension entre 0 et 3,3 V, nettement supérieure aux quelques centaines de mV requis par une carte son).

Nous avons mis en pratique ce concept lors de la conférence What The Hack [5] au cours de laquelle plusieurs démonstrations ont été réalisées au moyen de ce montage, nous donnant l'opportunité d'obtenir d'excellentes images (absence de relief en Hollande où se tenait la conférence) depuis une position géographique inhabituelle pour nous.

Les images présentées en Figs. 3,4 illustrent plusieurs points :

- le bon fonctionnement des convertisseurs analogique-numériques dans toute leur gamme de sensibilité : la dynamique de l'image a pu être ajustée de façon à utiliser 9 des 10 bits de résolution disponibles sur ces convertisseurs
- la capacité à enregistrer un signal aux fréquences audio pendant plusieurs minutes sans discontinuité (absence de coupure dans les images résultantes)
- la capacité à stocker les données pour traitement ultérieur par un programme que nous avons développé et qui peut être implémenté sur le système embarqué pour une application finale totalement autonome (nous n'avons pas été capables d'ajouter un écran graphique pour l'affichage de l'image résultante bien que cette image soit disponible en mémoire).

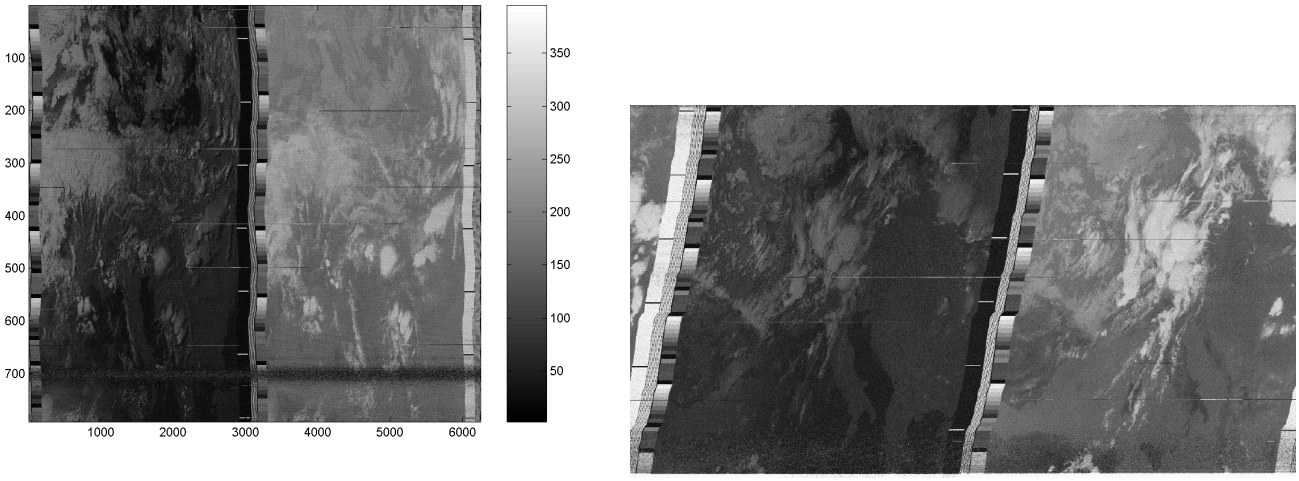

FIG. 3 – Exemple d'image issue de données acquises par un processeur Coldfire relié au récepteur radiofréquence du RIG. Les niveaux de gris sont en unité arbitraire et illustrent une utilisation de 9 des 10 bits disponibles sur la QADC : la dynamique du signal aurait pu être améliorée en augmentant le gain de l'amplificateur entre la sortie du récepteur radiofréquence et le Coldfire. Haut : image obtenue après filtrage par une fenêtre glissante moyennant 5 points successifs de l'enregistrement audio (fonction `conv` de Matlab/octave). La fenêtre glissante nous permet de conserver toute la résolution temporelle de l'acquisition initiale et ainsi d'ajuster avec précision la largeur de l'image pour minimiser la distorsion lors de la conversion audio→image. Nous déduisons de cette transformation que la fréquence d'échantillonage est de 12498 Hz au cours de l'acquisition – compatible avec le résultat de la Fig. 2. Bas : résultat d'une moyenne sur des ensembles successifs de 5 points adjacents. Cette méthode plus grossière fournit des images de taille plus raisonnable (1250 pixels de large au lieu de 6250) mais la perte de résolution temporelle se traduit par une distorsion de l'image. En effet prendre les points par paquet de 5 revient à imposer une fréquence d'échantillonage multiple de 10 (chaque ligne est transmise en 0,5 s), soit ici 6250 Hz qui se traduit par un décalage de 1 pixel par ligne. Nous verrons plus bas (Fig. 5) comment compenser cette distorsion par traitement d'image.

Il apparaît clairement sur les figures 3 et 4 que la fréquence d'échantillonage fluctue légèrement au cours de l'acquisition. La norme APT de transmission des images météorologiques issues des satellites inclut plusieurs signaux de synchronisation au début de chaque ligne, visibles sous forme de bandes verticales périodiques sur les images démodulées. Deux approches sont possibles afin d'aligner les images et éliminer les distorsions associées à ces fluctuations de fréquence d'échantillonage : identifier sur le signal audio ces signaux de synchronisation et en tenir compte

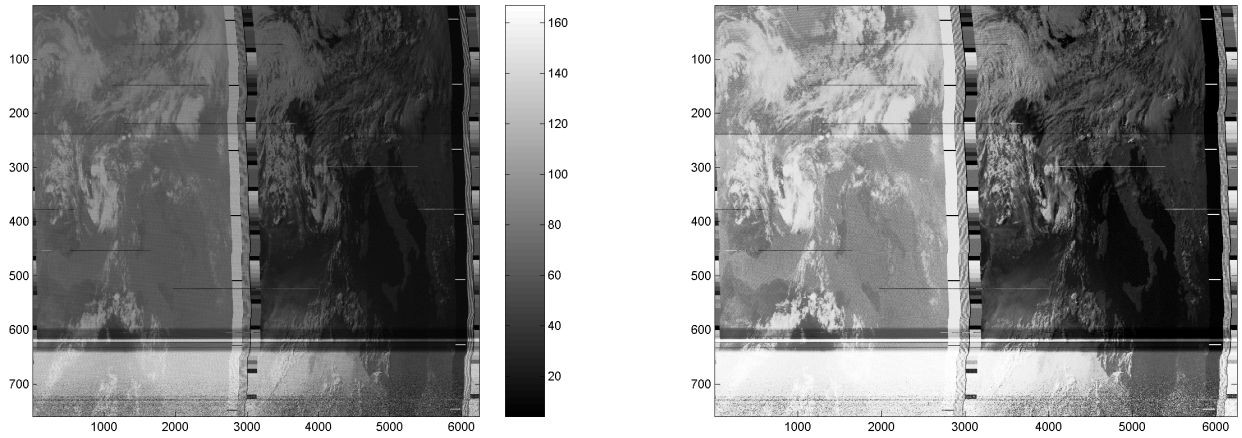

FIG. 4 – Exemple d'image issue de données acquises par un processeur Coldfire relié au récepteur radiofréquence du RIG après traitement sous Matlab/octave, suivi d'une application de l'algorithme d'égalisation d'histogramme présenté dans l'article précédent afin de mettre en relief les terres cachées dans la zone la plus sombre de l'image. Ce traitement a cependant aussi l'inconvénient de mettre en évidence les limitations de notre montage et notamment un signal périodique sur l'image probablement associé à une mauvaise stabilisation de l'alimentation commune à tous nos montages.

au cours du traitement visant à convertir le signal audio en image, ou après traitement simple des signaux audio tel que présenté jusqu'ici puis retravailler l'image pour en compenser les distorsions. Il semblerait que le logiciel `wxtoimg` travaille directement sur le signal audio afin de retrouver le retour chariot et ainsi aligne directement ses images pour un résultat irréprochable. Nous avons pour notre part été incapables d'extraire ces signaux de synchronisation de la porteuse audio à 2400 Hz (par démodulation I/Q logicielle, transformée de Fourier locale ou intercorrélation) et avons donc du nous contenter de la solution de traitement d'image *a posteriori* telle que présentée sur la Fig. 5.

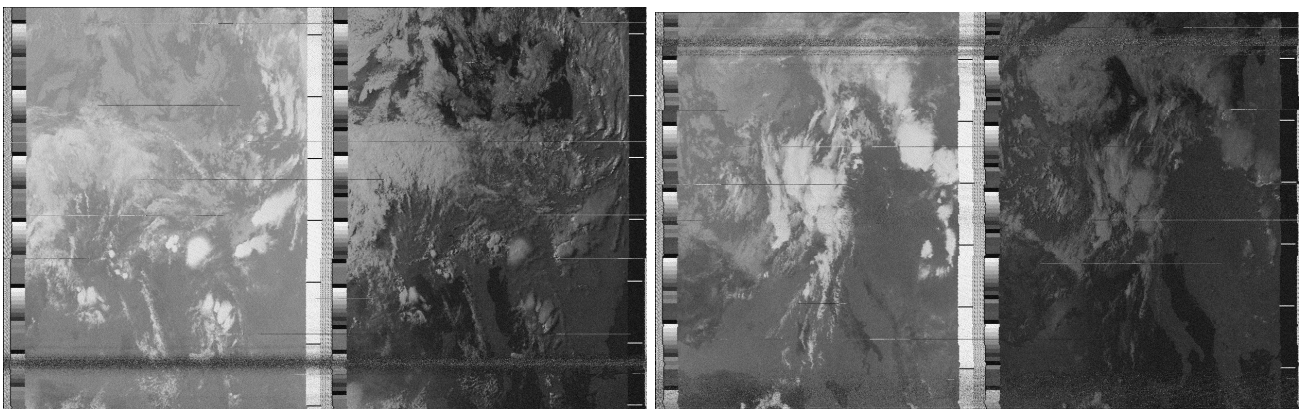

FIG. 5 – Les deux images présentées sur la Fig. 3, issues des acquisitions par `qadc` tournant sur Coldfire, décodées par filtrage passe-bas des signaux audio puis alignement sur le signal de synchronisation que sont les bandes noires et blanches présentes à gauche des images.

Nous constatons que l'extension en latitude des images acquises par le montage embarqué est inférieure à la plage observée lors de l'utilisation du PC : il s'agit là d'une conséquence du stockage en RAM des données audio. La quantité limitée de RAM disponible nous empêche d'acquérir le signal tout au long d'un passage et nous perdons la fin de l'image

faute de place.

7 Conclusions et perspectives

Nous avons présenté la mise en œuvre d'un circuit à base de Coldfire 5282 et l'utilisation des convertisseurs ananlogique-numériques pour une acquisition de signaux audio à une fréquence d'échantillonage de l'ordre de 12 kHz, et ce sans coupure pendant plusieurs minutes. Nous avons développé un module noyau garantissant une latence constante entre deux acquisitions et fournissant les méthodes pour transférer les données acquises en espace utilisateur pour un stockage en mémoire volatile. Nous avons pour cela dû palier à l'absence de gestionnaire de mémoire afin de pouvoir utiliser au mieux la totalité de la RAM physiquement disponible.

Nos développements actuels se focalisent sur le stockage directement en mémoire non-volatile de type MultiMediaCard (MMC) afin de s'affranchir de la limite de la dizaine de MB de RAM disponible sur la carte DNP5280. Le résultat escompté d'une telle amélioration sera la capacité à enregistrer le signal audio pendant la totalité d'un passage et donc de pouvoir restituer une image s'étendant en latitude sur toute la plage de réception du signal. D'autre part, nous ne serons plus forcés d'avoir un ordinateur pour rapatrier les signaux placés en RAM : les résultats de plusieurs passages de satellites successifs pourront alors être stockés dans des zones distinctes de la MMC pour un traitement ultérieur.

Remerciements

Nous remercions l'entreprise Alcôve (www.alcove.com) pour son support 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.

Références

[1] J.-M Friedt, S. Guinot, *Introduction au Coldfire 5282*, GNU/Linux Magazine France, **75** (Septembre 2005)

[2] *MCF5282 Coldfire Microcontroller User's Manual*, rev. 2.3 (11/2004), disponible à http://www.freescale.com/files/dsp/doc/ref_manual/MCF5282UM.pdf

[3] T. Petazzoni & D. Decotigny, *Conception d'OS : pilotes de périphériques caractère*, GNU/Linux Magazine France, **79** (Janvier 2006)

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

[5] http://wiki.whatthehack.org et en particulier http://wiki.whatthehack.org/index.php/Weather_satellite_image_reception_demo pour ce qui nous concerne.