Développement sur processeur à base de cœur ARM7 sous GNU/Linux

J.-M Friedt, 31 mars 2009

Le cœur de processeur ARM annonce les meilleures performances en termes de puissance de calcul (MIPS) par unité de puissance électrique consommée (watts). Nous proposons une présentation des outils pour développer sur cœur ARM7, et en particulier sur le microcontrôleur Analog Devices ADuC7026. Nous fournissons une toolchain libre fonctionnelle sous GNU/Linux, dont nous illustrerons l'exploitation sur quelques exemples concrets de communications avec divers périphériques pour le stockage et la communication sans fil de données, de calcul de racine de fonctions complexes, et de compression de données. Afin d'exploiter au mieux les fonctionnalités de ce processeur, nous proposons l'exploitation sous `wine` du logiciel gratuit de configuration d'une matrice de portes logiques reconfigurables : nous démontrerons ainsi que l'exploitation d'un processeur sous GNU/Linux – avec des outils mis à disposition par le fabriquant exclusivement sous forme de binaires pour MS-Windows – est possible sans en limiter les fonctionnalités.

1 Introduction

ARM (Advanced RISC Machines) propose une large gamme de processeurs adaptés aux applications embarquées gourmandes en ressources de calcul (téléphonie mobile, serveurs TCP/IP embarqués, consoles de jeu ...) par leur excellent rendement puissance de calcul/consommation électrique [1]. Afin de maîtriser cet outils que tout développeur de systèmes embarqués ne manquera pas de rencontrer dans ses activités, nous allons nous intéresser à une version microcontrôleur (contenant donc tous les périphériques nécessaires à son fonctionnement – notamment mémoires volatile et non-volatile pour le stockage du programme) incluant un cœur milieu de gamme de type ARM7TDMI. Le composant qui nous intéresse est simple à mettre en œuvre, ne nécessite aucun composant actif externe additionnel pour fonctionner, et est supporté par des outils de développement libres.

Le cœur ARM7TDMI [2] – TDMI signifiant la disponibilité du jeu d'instructions Thumb, d'un Debugger, de la Multiplication matérielle, ICE – propose un processeur 32 bits d'architecture RISC. Équipé de 16 registres, dont 13 disponibles pour le programmeur (les 3 autres étant réservés respectivement à la gestion de la pile, la mémorisation de l'adresse de retour lors d'un saut dans une procédure – fonction habituellement gérée sur la pile sur les autres architectures, et l'index d'exécution du programme [3, p.11]) il fournit une puissance de calcul considérable – exploitée par exemple dans l'iPod [4], Lego Mindstorm NXT [5] et la console Gameboy Advance – pour une surface occupée de l'ordre de 4 cm^2 et un coût unitaire de l'ordre de la dizaine d'euros [1]. Cette architecture est spécifiquement dédiée pour les applications embarquées avec la capacité d'exécuter un jeu d'instructions codés sur 16 bits, économisant ainsi de la mémoire de stockage sans réduire notablement les performances dans la majorité des cas : il s'agit du jeu d'instructions Thumb (le "T" de TDMI).

ARM ne fabrique pas de processeur mais vend l'architecture du cœur à des fondeurs : diverses sources sont donc disponibles et garantissent la pérennité du code développé sur ce type de processeur ainsi que sa déclinaison avec une large variété de périphériques. Mentionnons, sans prétention d'exhaustivité, les sources suivantes [6, 7] :

- la série NPX (anciennement Philips) LPC2000 et LH7
- la série Analog Devices ADuC7xxx que nous utiliserons ici [2]
- série ST STR7

[1] ADUC7026BSTZ62, référence 1162646 chez Farnell pour 17 euros, ou disponible pour 9,40 euros auprès de Avnet en septembre 2007

[2] http://www.analog.com/en/prod/0,,762_0_ADUC7026,00.html

- Freescale MAC7100
- Texas Instruments TMS470
- Atmel AT91SAM7A
- la série Cirrus Logic EP7
- Samsung et VLSI semblent avoir acquis le droit d'exploiter le cœur pour des applications dédiées
- des processeurs à base d'ARM7TDMI sont au cœur de la console de jeu GameBoy Advance et coprocesseur de la Nintendo DS (aux côtés d'un ARM9).

Puisque `gcc` est capable de générer du code pour processeur ARM, nous allons compiler une chaîne de développement libre pour ARM7TDMI, fonctionnelle sous GNU/Linux, et l'exploiter pour présenter quelques fonctionnalités de ce processeur. Le processeur que nous avons sélectionné ne possède pas assez de mémoire pour y loger un système d'exploitation : bien que Linux et uClinux soient fonctionnels sur ce type d'architecture [8], nous allons nous contenter de développer des applications dédiées, sans système d'exploitation sous-jacent.

2 Description du matériel

Le matériel nécessaire pour tester les concepts développés dans cet article consiste en une plate-forme de développement contenant un processeur de Analog Devices de la classe de ADuC702x. Tous nos exemples ont été testés avec un ADuC7026, initialement sur la carte de démonstration commercialisée par Analog Devices [3], puis rapidement sur des circuits spécialement développés pour nos tests (Fig. 1).

Les composants périphériques au microcontrôleur sont un résonateur à quartz à 32 kHz qui sera multiplié en interne pour fournir une horloge jusqu'à 41,78 MHz, une paire de condensateurs de découplage pour les tensions de référence des convertisseurs analogique-numérique et numérique-analogique, ainsi qu'un convertisseur de niveau pour la liaison RS232, ou un FT232RL si la liaison USB est préférable (cas rarement rencontré dans les applications embarquées). Une paire d'interrupteurs, sur les broches RESET (broche 37) et DLOAD (broche 20), permettront de réinitialiser le processeur et de le passer en mode programmation (DLOAD à la masse) en appuyant sur ces boutons simultanément.

Notons par ailleurs la disponibilité de platines de développements réalisées par Olimex (référence ADuC-MT7020, disponible pour 70$ ches Sparkfun) pour processeur ADuC7020 (Fig. 1).

3 Compilateur et outils associés

3.1 La chaîne de cross-compilation

Le cœur ARM7 est exploité dans de nombreuses applications commerciales. Par conséquent, une version de `gcc` a été portée à cette architecture, avec notamment le support de l'ensemble d'instructions Thumb (option `-mthumb`). Nous avons, dans notre cas, exploité la procédure de compilation de la toolchain décrite à `http//paul.chavent.free.fr/gba-crosscompil.html`, à l'exclusion de l'application des patchs spécifiques à la console Gameboy Advance et de l'installation de Insight dont nous n'aurons pas l'utilité. Il s'agit principalement d'exploiter les bonnes versions du trio gcc, binutils et newlib (4.0.0, 2.15 et 1.13.0 respectivement) configurées (`configure`) avec les options

```
--target=arm-thumb-elf --without-local-prefix --disable-shared  --enable-multilib
--disable-threads --with-cpu="arm7tdmi" --enable-interwork  --enable-languages="c,c++"
--disable-nls
```

sans oublier le répertoire d'installation avec `--prefix`.

Une alternative, pour les plus impatients, à la compilation manuelle de la toolchain est, si l'exploitation du code thumb n'est pas une priorité, d'utiliser la toolchain précompilée fournie

[3] Analog Devices EVAL-ADUC7026QSZ, disponible pour 91 euros chez Farnell

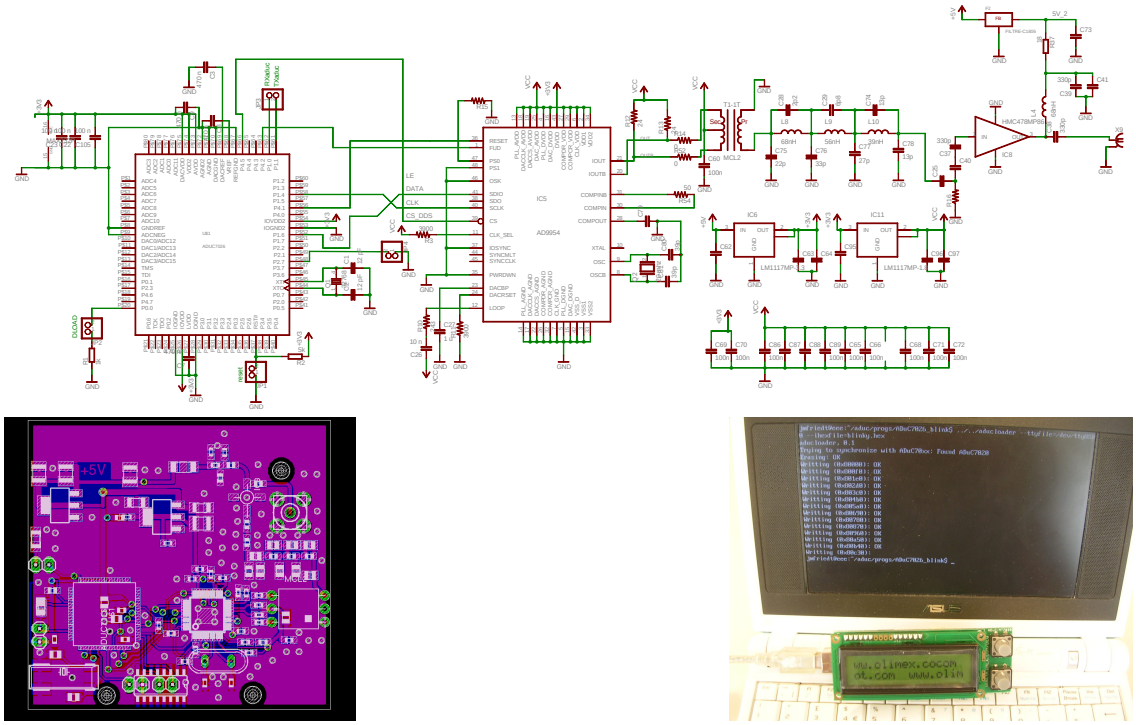


FIG. 1 – Haut : schéma du circuit de test qui sera utilisé pour implémenter la majorité de nos exemples : un ADuC7026 avec ses quelques composants passifs additionnels (condensateurs de découplage des tensions de référence et résonateur à quartz) est associé à un synthétiseur numérique de fréquence AD9954 pour en faire un émetteur radiofréquence contrôlé par logiciel. En bas à gauche : implémentation des composants, avec en bas à gauche le processeur et son résonateur à quartz, en haut à gauche les régulateurs linéaires de tension, et en bas à droite le synthétiseur numérique de fréquences. Le surface du circuit double-face est de l'ordre de 55×60 mm^2. En bas à droite : l'ADuC-MT7020 exécute un exemple compilé et transféré depuis un EeePC701, un environnement de développement mobile et autonome.

sous forme d'exécutables pour Intel x86 par le projet uClinux à http://www.uclinux.org/pub/uClinux/dist/. Cette toolchain fournit `glibc` au lieu de `newlib` : dans ce cas, toute tentative d'exploitation de `stdio` sur ADuC7026 se traduit par le dépassement de la capacité de mémoire non-volatile et l'impossibilité de compiler le code. Il faudra donc éviter d'utiliser tout appel aux fonctions de type `printf`, gestion des chaînes de caractères ou bibliothèques de fonctions mathématiques.

3.2 Compiler un programme pour ARM7TDMI

Ayant obtenu une toolchain, nous allons décrire les étapes pour générer un fichier prêt à être transmis au microcontrôleur : assemblage, linkage des objets et des librairies, puis extraction du contenu de la mémoire dans un format prêt au transfert par RS232. L'ensemble de ces opérations sera classiquement automatisé dans un `Makefile`.

Bien que nous programmions un processeur aussi puissant que l'ARM7 en C, il est toujours bon de pouvoir revenir à l'assembleur, qu'il s'agisse en vue d'optimisations de bouts de codes ou pour vérifier si `gcc` a généré un code convenable et réalisant les tâches attendues :
`arm-thumb-elf-objdump -h -S -C fichier.elf > fichier.lst`
produit la séquence de mnémoniques, leurs arguments et leur emplacement en mémoire.

La compilation s'obtient, pour du code assembleur, par

```
arm-thumb-elf-gcc -mcpu=arm7tdmi -x assembler-with-cpp -DROM_RUN fichier.S  \
-o fichier.elf
```

et pour le code C par

```
arm-thumb-elf-gcc -mcpu=arm7tdmi  -I. -DROM_RUN -Os -Wcast-align -Wimplicit \
-Wpointer-arith -Wswitch -Wredundant-decls -Wreturn-type -Wshadow -Wunused  \
-I./include -Wcast-qual  -Wnested-externs  -std=gnu99  -Wmissing-prototypes \
-Wmissing-declarations fichier.c -o fichier.elf
```

Nous pourrons éventuellement ajouter l'option `-mthumb` pour exploiter le jeu d'instructions 16 bits. L'intérêt de ce jeu d'instructions est de réduire la taille de l'exécutable en ne fournissant qu'un sous ensemble des fonctionnalités du jeu d'instructions 32 bits. Les gains ne sont pas aussi intéressants qu'il pourrait y paraître car il faut souvent plusieurs instructions Thumb pour effectuer la même opération qu'en jeu d'instructions ARM. Le gain est estimé à 20-30% en espace occupé par le code au lieu des 50% attendus [9, 2].

Le fichier au format ELF n'est pas reconnu en l'état par les outils de transfert du programme vers la mémoire flash du microcontrôleur : il faut générer, à partir du binaire, un fichier reproduisant la futur contenu de la mémoire. Ainsi, le code en C aura été converti en un fichier contenant au format hexadécimal intel la séquences des opcodes et de leurs arguments. Le passage du format ELF au format hexadécimal intel s'obtient par la version appropriée de `objcopy` :
`arm-thumb-elf-objcopy -O ihex fichier.elf fichier.hex`

Ayant identifié une toolchain de cross-compilation fonctionnelle et les éventuelles bibliothèques de fonctions nécessaires aux développements, il nous reste à configurer le processeur lors de sa mise en marche pour un fonctionnement correspondant à nos attentes. D'habitude, `gcc` cherche à linker un programme avec l'objet issu de la compilation de `crt0.S` qui effectue l'appel à la fonction `main()` par convention [4].

Dans les exemples qui vont suivre, nous linkerons systématiquement un fichier nommé `startup.S` proposé par Martin Thomas à http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/index_adi_arm.html#aduc_gpio, issu d'une traduction pour `gcc` du fichier de configuration proposé par le vendeur de compilateurs propriétaires Keil. Ce fichier est chargé de l'initialisation de registres fondamentaux au fonctionnement du processeur (multiplication d'horloge par PLL, piles, points de retour par défaut des interruptions) et des périphériques (ports d'entrée sortie notamment), ainsi que l'exécution sur le point d'entrée par défaut d'un programme C nommé `main()`. Par ailleurs, le linker exploite par l'option `-T` le script `ADuC7026-ROM.ld` de Martin Thomas [5] pour définir les emplacements des divers éléments du programme (pile, tas, RAM, mémoire flash). Nous profiterons de cette archive pour y trouver un exemple d'initialisation (`startup.S`) et de gestion des interruptions (`irq.c`) à linker à nos propres applications.

3.3 Transférer le programme au processeur

L'ADuC7026 est muni d'un bootloader capable de charger par le port série et de placer en mémoire non volatile (flash) un programme. Nous avons utilisé avec succès deux programmes permettant la communication avec le microcontrôleur depuis GNU/Linux :

- `armwsd`, court et très efficace, contient en dur le nom du port série utilisé, et lit le fichier au format hexadécimal intel sur stdin. L'unique programme C est disponible à http://www.koka-in.org/~kensyu/handicraft/misc/armwsd.c se compile trivialement et s'exécute par
  ```
  ./armwsd < newton.hex
  Warning: Type 3 is ignored (The start address = 0x00080000).
  ADuC7026   -62 I31
  ***************
  ```

[4] http://sca.uwaterloo.ca/coldfire/gcc-doc/docs/porting_3.html
[5] http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects/ADuC7026_blink_20060309b.zip

– `aducloader` [6] , plus complet, accepte des arguments pour définir le périphérique de communication série, le nom du programme à placer dans le microcontrôleur (au format hexadécimal intel), et la vitesse de communication : par exemple, `aducloader --ttyfile=/dev/ttyUSB0 --ihexfile=blinky.hex`

Dans tous les cas, le transfert du programme en mémoire flash du microcontrôleur s'obtient en appuyant sur DLOAD, puis en validant brièvement la broche de RESET, et enfin en lançant le programme de communication sur le PC (le microcontrôleur doit donc être en attente des ordres du logiciel de communication avant le lancement de ce dernier). Plusieurs essais sont parfois nécessaires afin de vider l'UART du PC des caractères résiduels, afin que le logiciel reçoive le bon identifiant de microcontrôleur en réponse à ses requêtes.

3.4 Gestion des ports numériques d'entrée-sortie

Le premier exemple pour découvrir tout nouveau microcontrôleur consiste à faire clignoter une diode. Au-delà de la validation de la chaîne de cross-compilation, de l'outil de transfert des programmes en mémoire non-volatile et du programme d'initialisation du microcontrôleur et d'appel à la fonction `main()`, il s'agit de la première découverte des spécificités matérielles du microcontroleur, dont le périphérique le plus simple est le port numérique d'entrée-sortie (*General Purpose Input-Output*, GPIO).

Dans le cas particulier de l'ADuC7026, chaque port est représenté par un registre de 32 bits (`GPxDAT`), dont les 8 bits de poids les plus forts représentent la direction de chaque bit du port (1 pour la sortie, 0 pour l'entrée), les 8 bits suivant définissent la valeur en sortie, les 8 bits suivant mémorisent l'état du port au moment de la réinitialisation, et finalement les 8 bits de poids les plus faibles donnent la lecture des valeurs des bits en entrée. Le fait qu'autant de fonctions soient réunies dans un même registre signifie que nous y ferons le plus souvent appel – après une affectation pour l'initialisation – par un masquage.

Les 5 ports sont accessibles par les registres GPiDAT, $i \in [0..4]$.

Pour, par exemple, faire clignoter une diode associée à une broche du port 4, une initialisation du registre `GP4DAT` pour définir les broches en sortie est suivie d'une boucle infinie sur un XOR pour changer l'état des diodes à chaque itération :

```
#include "ADuC7026.h"
int main(void)
{GP4DAT = 0xFF000000;    // P4  en sortie
 while (1) {
   GP4DAT ^= 0x00FF0000; // Complemente P4.x
   // delai
 }
}
```

3.5 Configuration et communication sur port série asynchrone

Les ports d'entrée-sortie numérique ont chacun une multitude de fonctions (au plus 4 par broche, définies sur 2 bits pour chaque broche dans le registre `GPxCON` pour le port x), soit comme bits contrôlés individuellement, soit en relation avec un périphérique plus complexe tel que par exemple un UART. Pour l'ADuC7026, cette configuration se fait par le registre `GP1CON` (configuration du port 1) qui, pour une exploitation des broches P1.0 et P1.1 en vue d'une communication asynchrone, s'obtient par `GP1CON = 0x011 ;`

Ayant configuré les broches, il nous faut définir la vitesse de communication en divisant par un facteur approprié l'horloge interne du processeur :

```
COMCON0 = 0x080;  // Setting DLAB
COMDIV0 = 0x088;  // Setting DIV0 and DIV1: 0x88=9600 bauds, 23=57600 bauds
COMDIV1 = 0x000;
COMCON0 = 0x007;  // Clearing DLAB
```

[6] http://cyclerecorder.org/aducloader/, version 0.1 à la date de rédaction de ce document

On adaptera `COMDIV0` au baudrate recherché, avec 23 (décimal) pour du 57600 bauds ou 0x22 pour 38400 bauds.

Finalement, la communication se fait en vérifiant l'état de l'UART et en lisant ou écrivant un octet : il n'existe pas de pile pour accumuler les caractères transmis et toute nouvelle donnée écrase irrémédiablement la précédente :

```
int uart0_getc()
{while(!(0x01==(COMSTA0 & 0x01))) {}
 return (COMRX);
}

int uart0_putc(int ch)
{while(!(0x020==(COMSTA0 & 0x020))) {}
 return (COMTX = ch);
}
```

La fonction `uart0_getc()` proposée ici est bloquante comme le requiert la norme de l'ANSI C : si le besoin d'interroger le port série sans bloquer l'exécution du programme jusqu'à la prochaine transmission de données, on pourra remplacer le `while` par un test `if` sur l'état du registre `COMSTA0` : `if((0x01==(COMSTA0 & 0x01))) return (COMRX); else return(0);`

3.6 Le problème de `newlib`

Nous désirions, lors de la mise en œuvre de ces outils de travail, ne pas être restreint par rapport aux outils fournis par Keil [7] pour MS-Windows avec le kit de démonstration de l'ADuC7026. Cela signifie notamment disposer d'une implémentation de `stdio` capable d'afficher des messages sur le port série par `printf()`, ainsi qu'une librairie mathématique capable d'émuler de façon logicielle du calcul sur des nombres flottants. Le choix par défaut de librairie libre supportant ces fonctionnalités est `newlib`, dont nous allons montrer qu'il est totalement inapproprié pour ces application fortement embarquées à faible empreinte mémoire. En effet, les performances de `newlib` sont dignes des besoins d'un système d'exploitation, avec la consommation de ressources associée. Par exemple, Keil a fait le choix de ne nécessiter que la redéfinition de `putchar()` et `getchar()` pour implémenter un sous ensemble des fonctions associées à `stdio`. Au contraire, `newlib` fournit au travers de ses *stubs* (code Tab. 1) une implémentation complète mais beaucoup plus gourmande en ressources de `stdio`, avec notamment l'exploitation de `malloc`.

Keil semble avoir développé ses propres implémentations de `printf` et autres fonctions d'affichage avec une empreinte mémoire réduite. Nous avons donc deux solutions :

- écrire nos propres routines d'affichage répondant aux besoins spécifiques de chaque programme, par exemple pour les cas les plus courant de debuggage, afficher en hexadécimal des entiers de différentes tailles. Cette solution est favorisée par les auteurs, mais nécessite un investissement initial supplémentaire qui peut rebuter le développeur lors de son choix entre une toolchain libre ou propriétaire ne nécessitant par de tels développements initiaux.
- proposer une implémentation complète de `stdio`, avec une empreinte mémoire aussi faible que possible, sans être nécessairement aussi performant que l'implémentation dédiée de Keil. C'est cette seconde solution que nous allons développer ici en présentant l'exploitation de `newlib` sur ARM7TDMI.

Dans tous les exemples qui suivent, nous linkerons nos programmes avec le code `syscalls.c` contenant ces stubs [8] (code Tab. 1 dérivé de la description fournie à http://www.embeddedrelated.com/groups/lpc2000/show/5873.php). L'objectif de ces points d'entrée est de fournir quelques méthodes standard afin d'informer `newlib` comment interagir avec l'utilisateur (lire et écrire un caractère) ainsi que sur la gestion de la mémoire. Les prototypes de fonctions sont imposés et nous nous sommes contentés d'ajouter les fonctions de lecture et d'écriture sur le port série.

Une fois les stubs définis dans `syscalls.c` – et notamment les fonctions `_read_r()` et `_write_r()` pour la lecture et l'écriture sur le port série, nous sommes en mesure d'utiliser les fonctions de

[7] http://www.keil.com/dd/chip/3694.htm
[8] http://wiki.osdev.org/Porting_Newlib

```
/*****************************************************************/
/*  SYSCALLS.C:  System Calls                                    */
/*  most of this is from newlib-lpc and a Keil-demo              */
/*  These are "reentrant functions" as needed by                 */
/*  the WinARM-newlib-config, see newlib-manual.                 */
/*  Collected and modified by Martin Thomas                      */
/*****************************************************************/
/* adapted for the SAM7 "serial.h" mthomas 10/2005 */
#include <stdlib.h>
#include <reent.h>
#include <sys/stat.h>
#include "ADuC7026.h"

int uart0_kbhit() {return(1);}

int uart0_getc()
{while(!(0x01==(COMSTA0 & 0x01))) {}
 return (COMRX);
}

int uart0_putc(int ch)
{while(!(0x020==(COMSTA0 & 0x020))) {}
 return (COMTX = ch);
}

_ssize_t _read_r(struct _reent *r, int file, void *ptr, size_t len)
{char c; int  i; unsigned char *p;

 p = (unsigned char*)ptr;
 for (i = 0; i < len; i++) {
   while ( !uart0_kbhit() ) ;
   c = (char) uart0_getc();
   if (c == 0x0D) {*p='\0';break;}
   *p++ = c; uart0_putc(c);
 }
 return len - i;
}

_ssize_t _write_r ( struct _reent *r, int file, const void *ptr, size_t len)
{
 int i;
 const unsigned char *p;

 p = (const unsigned char*) ptr;

 for (i = 0; i < len; i++) {
   if (*p == '\n' ) uart0_putc('\r');
   uart0_putc(*p++);
 }
 return len;
}


int _close_r(struct _reent *r, int file)
{return 0;}


_off_t _lseek_r(struct _reent *r,int file,_off_t ptr,int dir)
{return (_off_t)0;      /*  Always indicate we are at file beginning.  */
}


int _fstat_r(struct _reent *r,int file,struct stat *st)
{st->st_mode = S_IFCHR;return 0; }

/**** Locally used variables. ****/
// mt: "cleaner": extern char* end;
extern char end[];      /*  end is set in the linker command    */
                        /* file and is the end of statically    */
                        /* allocated data (thus start of heap). */

static char *heap_ptr;  /* Points to current end of the heap.   */

/************************ _sbrk_r *************************************
 * Support function. Adjusts end of heap to provide more memory to
 * memory allocator. Simple and dumb with no sanity checks.
 *  struct _reent *r -- re-entrancy structure, used by newlib to
 *                      support multiple threads of operation.
 *  ptrdiff_t nbytes -- number of bytes to add.
 *                      Returns pointer to start of new heap area.
 */
void * _sbrk_r(
    struct _reent *_s_r,
    ptrdiff_t nbytes)
{
        char  *base;            /*  errno should be set to  ENOMEM on error  */

        if (!heap_ptr) {        /*  Initialize if first time through.  */
                heap_ptr = end;
        }
        base = heap_ptr;        /*  Point to end of heap.  */
        heap_ptr += nbytes;     /*  Increase heap.  */
        return base;            /*  Return pointer to start of new heap area.*/
}
```

TAB. 1 – Définition des *stubs* – points d'entrée spécifiques à une architecture – pour exploiter la `newlib`. Nous y trouvons notamment des fonctions spécifiques au matériel avec l'émission et la réception de caractères par l'UART dans les fonctions `uart0_putc()` et `uart0_getc()` exploitées dans les fonctions standard `_write_r()` et `_read_r()`.

gestion de chaînes de caractères comme nous le ferions habituellement en ANSI C. Noter que la compréhension des méthodes d'interfaçage avec `newlib` est un investissement qui peut s'avérer utile puisqu'exploitable sur tout microcontrôleur suffisamment puissant pour supporter cette bibliothèque de fonctions.

L'utilisation de `printf` et du calcul flottant n'est pas à prendre à la légère : nous illustrons l'explosion de la taille du code lors de l'utilisation de `printf()` et d'une fonction de calcul flottant (`atan()`) dans la table 2.

affichage	instructions	taille (octets)	affichage & calcul (instructions thumb)	taille (octets)
avec stdio	thumb	80336	thumb, sans stdio, avec une division flottante	12950
sans stdio	thumb	4048	thumb, sans stdio, avec atan sur flottant	17090
avec stdio	arm	81641	thumb, avec stdio, avec une division flottante	80397
sans stdio	arm	4993	thumb, avec stdio, avec atan sur flottant	80397

TAB. 2 – La taille du fichier .hex intel prêt à flasher dans l'ADuC7026 dépend des fonctionnalités exploitées dans le code source (`printf` et calcul sur nombres flottants). Nous constatons que l'utilisation de stdio augmente la taille du code d'environ 80 KB, tandis que l'émulation logicielle du calcul sur les nombres flottants ajoute un modeste 10 à 15 KB.

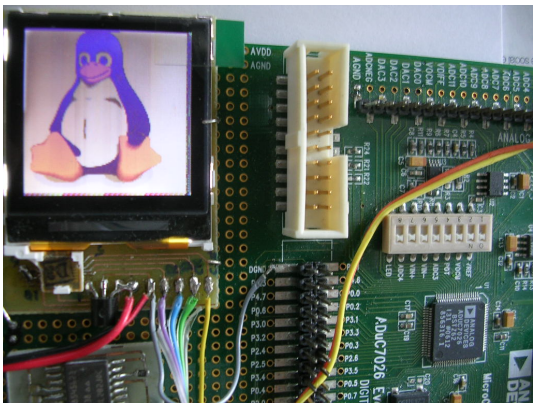


FIG. 2 – Exemple d'image affichée sur un écran LCD couleur Nokia obtenu à http://www.sparkfun.com/commerce/product_info.php?products_id=569. Un exemple de code disponible sur cette page web pour le NXP LPC2148 a pu facilement être porté à l'ADuC7026. La RAM de ce microcontrôleur est trop réduite pour contenir l'image complète, qui doit donc être transmise par le port série.

4 Quelques exemples d'applications

4.1 Communication synchrone : affichage sur écran LCD graphique

La communication synchrone – SPI pour des signaux unidirectionnels du maître vers l'esclave (MOSI) et de l'esclave vers le maître (MISO), et I^2C pour un signal de données bidirectionnel – partage une horloge entre les interlocuteurs et permet par conséquent un débit de communication plus élevé que la communication asynchrone. Ce protocole est par conséquent exploité dans de nombreux périphériques, dont nous allons présenter 3 exemples ici : afficheur graphique, carte mémoire SD et synthétiseur de fréquence. Un protocole synchrone est facile à implémenter de façon logicielle en manipulant les bits d'un port général d'entrée sortie, mais le résultat s'obtient au détriment des performances. L'ADuC propose une implémentation matérielle de la communication synchrone qui s'obtient par une configuration appropriée de l'UART *via* le registre `GP1CON` : comme les broches de communication synchrones (P1.4 à P1.7) et asynchrone (P1.0 et P1.1) sont distinctes, ces deux modes de communication peuvent être exploités simultanément : `GP1CON=0x22220011`.

Cependant, dans ce premier exemple, la communication des données avec un écran LCD graphique nécessite de communiquer une donnée sur 9 bits, format quelque peu inhabituel et qui n'est pas supporté de façon matérielle par l'ADuC7026 : une implémentation logicielle s'impose donc. Nous sommes partis de l'archive `NokiaLCD_LPC2148` [9] et nous nous sommes contentés d'adapter la gestion des ports d'entrée sortie du microcontrôleur NXP à l'ADuC7026 (Code. Tab. 3 et Fig. 2).

Les principales modifications portent sur la définition des signaux de communication : étant donné que la communication synchrone est émulée de façon logicielle, nous n'abordons pas ici la programmation du contrôleur SPI mais il suffit, lors du portage de la bibliothèque de fonctions de communication avec le LCD `LCD_driver.c`, de redéfinir les broches de communication. Nous avons, dans nos développements, exploité P1.4 et P1.6 respectivement comme broches d'horloge et de communication du maître (le microcontrôleur) vers l'esclave (le LCD) tandis que la broche d'activation du périphérique (Chip Select actif bas) est associée à P4.4. Le RESET du LCD s'obtient pas manipulation de la broche P4.2. Ces divers signaux sont manipulés dans les fonctions `LCD_init()`, `LCD_command()`, et `LCD_data()`. Le reste du code (allumage d'un pixel, définition de sa couleur, dessin de motifs plus complexes) est portable puisqu'il fait appel à ces fonctions qui sont les seules à accéder au matériel et donc à dépendre de la plateforme de développement.

[9] http://www.sparkfun.com/Code/Nokia_LCD_driver.zip

La durée d'affichage d'un motif sur l'ensemble des 132×132 pixels de l'écran est de l'ordre de 2,5 s, soit environ 75 μs pour définir la couleur d'un pixel. Selon l'utilisation du LCD (par exemple tracer une courbe ou un sprite sur un sous-ensemble de la surface active), le taux de rafraîchissement pourra aller du hertz à quelques centaines de Hz. Un exemple d'affichage sur toute la surface du LCD est proposé dans le code 3 qui exploite `LCD_driver.c`. Ce programme attend en entrée des couleurs `c` au format 8 bits du LCD, soit RRRGGGBB : ce format s'obtient, depuis une image au format PPM classique, en convertissant chaque pixel dont la couleur est définie sur 3 octets `r`, `g` et `b` par le masque `c=((r&0xE0)+((g&0xE0)>>3)+((b&0xC0)>>6));`.

```
#include <ADuC7026.h>
#include <LCD_driver.h>

int getchar (void)  { // lecture sur port serie *non-bloquant*
 if ((0x01==(COMSTA0 & 0x01))) return(COMRX);
 else return(0);
}

int main(void) {
  unsigned char c,x,y;

  GP1CON = 0x00000011;    // setup tx & rx pins on P1.0 and P1.1
  GP4DAT = 0xff180000;    // P4.[2-4] en sortie, P4.[3,4] = 1 & P4.2 = 0

 COMCON0 = 0x080;        // Setting DLAB
 COMDIV0 = 0x22;         // 38400 bauds
 COMDIV1 = 0x000;
 COMCON0 = 0x007;        // Clearing DLAB

 LCD_init();                 // [RRRGGGBB]

 for (x=0;x<131;x++) for (y=0;y<131;y++) {pset(x+y, x,  y);}
 for (x=30;x<50;x++) for (y=30;y<50;y++) pset(0x00, x,  y);
 while(1) {
       for (x=0;x<128;x++) for (y=0;y<128;y++) {
           do {c=(unsigned char) getchar();} while (c==0);
               pset(c,x,y);}
          }
}
```

Tab. 3 – Exemple de lecture des couleurs des pixels successifs en vue de l'affichage d'images sur l'écran LCD (Fig. 2). Le format des couleur suppose une définition de la couleur en 8 bits/pixel, selon le format RRRGGGBB. La conversion depuis une image au format PPM classique s'obtient par masquage tel que décrit dans le texte.

4.2 Communication synchrone : stockage au format FAT sur carte SD

Nous avons déjà présenté à plusieurs reprises, dans différents contextes, l'exploitation des cartes de stockage de masse non-volatiles de format Secure Digital (SD) pour fournir un mode de conservation quasi-illimité des informations acquises par un système embarqué. Dans la plupart des cas, nous nous sommes contentés de stocker les informations sur le support de stockage sans formattage : cette méthode, efficace en ressouces consommées, ne permet cependant pas de définir des notions telles que fichier et répertoire, et implique de restituer les informations par le même système que celui utilisé pour le stockage.

Afin d'étendre les perspectives du stockage de masse sur système embarqué, nous nous sommes proposés d'exploiter un support formaté selon un protocole reconnu par la majorité des systèmes d'exploitations modernes, et néanmoins compatible avec les performances d'un système fortement restreint tel que proposé par l'ADuC7026. Nous nous sommes par conséquent tournés vers le format FAT, suffisamment ancien pour être compatible avec une application embarquée et néanmoins toujours supporté par les systèmes récents, en portant une de ses multiples implémentations sur

ARM7. Nous avons sélectionné au hasard `efsl`, et en particulier sa version 0.2.8, comme librairie implémentant la plupart des méthodes supportant le format FAT (création de fichier, écriture et lecture).

`efsl` est fourni avec le support pour un certain nombre de plateformes, définies dans le répertoire `conf/` et par un `Makefile` approprié. Nous prendrons donc soin de renommer le fichier de configuration approprié dans `conf/` en `config.h` et d'adapter les chemins d'accès vers son compilateur favori dans le `Makefile`.

Un premier essai pour se familiariser avec la bibliothèque de fonctions `efsl` et les exemples associés peut se faire sous GNU/Linux au moyen d'un système de fichiers virtuel obtenu par
`dd if=/dev/zero of=test.dd bs=512 count=1024` pour créer un espace disque virtuel de 512 KB, suivi de
`mkfs -t vfat test.dd` pour le formattage. Ce système de fichier est accessible soit par la fonction `open("test.dd",...)` du C, soit par `mount -o loop test.dd /mnt` depuis le shell.

Un exemple d'accès à un système de fichier ainsi créé depuis `efsl` est proposé ci-dessous, pour une compilation et une exécution sous GNU/Linux par : `gcc -I inc/ -I conf -o linuxtest linuxtest.c -L. -lefsl`

```
#include <stdio.h>
#include <string.h>
#include <efs.h>

int main(void)
{
 EmbeddedFileSystem efs;
 EmbeddedFile file;
 unsigned short i,e;
 char buf[512];

 if(efs_init(&efs,"./test.dd")!=0)
    printf("Could not open filesystem.\n");
 else
   {if(file_fopen(&file,&efs.myFs,"TOTO",'r')!=0)
       printf("Could not open file.\n");
    else {while (e=file_read(&file,512,buf))
            for(i=0;i<e;i++) printf("%c",buf[i]);
          file_fclose(&file);
         }
   }

// il FAUT utiliser 'a' pour append, sinon erreur quand le fichier existe deja
// et qu'on utilise 'w'
   if (file_fopen(&file,&efs.myFs,"TOTO",'a')!=0)
      {printf("Couldn't open file for appending\n");
       if (file_fopen(&file,&efs.myFs,"TOTO",'w')!=0)
          {printf("Couldn't open file for writing\n");return(-3);}
      }
   sprintf(buf,"je suis la suite du texte\n");
   printf("\necriture : %d\n",file_write(&file,strlen(buf),buf));
   file_fclose(&file);

   fs_umount(&efs.myFs);return(0);
}
```

Au premier accès, le ficher recherché n'existe pas et il doit être créé :

```
jmfriedt@eee:~/fatfs/efsl/efsl-0.2.8$ ./linuxtest
Could not open file.
```

Au second accès, le fichier est créé et contient du texte qui est affiché :

```
jmfriedt@eee:~/fatfs/efsl/efsl-0.2.8$ ./linuxtest
je suis la suite du texte
```

Ceci continu à chaque exécution du programme de démonstration

```
jmfriedt@eee:~/fatfs/efsl/efsl-0.2.8$ ./linuxtest
je suis la suite du texte
je suis la suite du texte
```

Le même résultat est observable sur ADuC7026, exécutant un programme similaire cross-compilé et lié à la bibliothèque de fonctions `efsl` convenablement adaptée (Fig. 3). Un rapide test de performances, sans prétention d'exaustivité, montre que l'écriture de 100000 chaînes de 41 caractères (pour créer un ficheir de 4,1 MB) prend 138 secondes, soit un débit de l'ordre de 29 KB/s. Un fichier de même taille mais obtenu par écriture de 50000 chaînes de 84 caractères ne prend que 80 secondes, soit un débit à peu près double de 52 KB/s. Il semble intuitif que le bloc de données stocké sur SD étant de 512 B, les performances vont s'améliorer lorsque la donnée à stocker atteint cette taille, cas néanmoins peu représentatif du système embarqué chargé d'acquérir épisodiquement des données scalaires.

Comme dans le cas du LCD, l'adaptation du code de `efsl` à une nouvelle architecture consiste en la redéfinition des quelques fonctions de communication entre l'ADuC7026 et la carte SD, sans avoir à reprendre toutes les fonctions de plus haut niveau de gestion du format FAT. L'ensemble de ces fonctions se regroupe dans `src/interfaces` de `efsl-0.2.8` : initialisation de la carte SD (`sd_Init()`), communication de commandes à la SD avec gestion du signal d'activation Chip Select (`sd_Command()`) au moyen de fonctions de base du support SPI (`com_spi()`). Avant de prétendre inclure toutes ces fonctions dans un environnement de travail aussi complexe que `efsl`, il est judicieux de valider les fonctions de base d'initialisation de la carte, lecture d'un bloc de donnée en mémoire SD (fonction `MMCread_block()`) et écriture d'un bloc de donnée (fonction `MMCwrite_block()`). Nous ne décrirons pas ici le protocole de fonctionnement de ces fonctions, qui a déjà été développé auparavant [11] et dont l'implémentation est parfaitement lisible dans le code source.

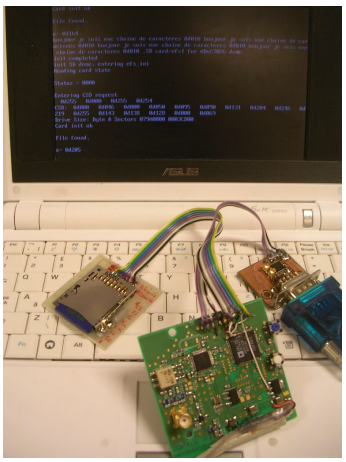


FIG. 3 – Circuit (cf Fig. 1 pour le schéma) incluant une liaison entre l'ADuC7026 et une carte SD, et affichage sur l'écran du PC de la sortie du programme d'exemple proposé dans le texte, démontrant l'ouverture d'un fichier stocké sur la carte SD formatée en FAT, la lecture du contenu et l'ajout d'une chaîne de caractères à la fin de ce fichier. La broche P1.2 sert de chip-select pour activer la communication avec la carte, les 3 autres broches de P1 (MOSI, MISO et CK) étant liées à l'implémentation matérielle du bus SPI dans l'ADuC7026, ici partagé avec le DDS AD9954.

4.3 Gestion des interruptions

Les interruptions sont un élément essentiel pour une programmation efficace d'un système embarqué chargé de réagir à son environnement. Les évènements peuvent être à l'expiration d'un intervalle de temps, une transition de signal sur une broche ou la transmission de données sur

un périphérique de communication. Deux points rendent la maîtrise des interruptions complexe dans l'environnement que nous décrivons : d'une part la disponibilité de deux encodages pour nos exécutables (16 bits en thumb ou 32 bits, le premier mode n'étant pas compatible avec une utilisation dans le gestionnaire d'interruptions – ISR), et d'autre part un bug susbsistant dans toutes les versions de `gcc` que nous avons utilisé (de 3.3 à 4.0.0), qui génère un code susceptible de corrompre des variables lors du retour de l'ISR. Nous avons constaté que *l'assignation* d'une valeur à une variable dans l'ISR fonctionne correctement, mais *incrémenter* ou manipuler une variable pour modifier son état antérieur ne fonctionne pas, laissant présager un problème avec la lecture de l'état de la variable lors de l'entrée dans l'ISR.

Deux ensembles d'interruptions sont particulièrement utiles lors du développement d'applications multitâches (code 4) :

- les interruptions *timer* qui permettent de séquencer les opérations
- une interruption associée à la communication série asynchrone qui permet d'arrêter l'exécution séquentielle du programme pour recevoir un message transmis par RS232, et notamment de remplir une queue de caractères afin de ne pas perdre de caractère puisque l'ADuC7026 ne possède qu'un unique registre pour stocker les caractères reçus. En cas d'activité au moment de la réception de ce caractère, un mode *polling* pourrait résulter en une perte de données si le traitement de la commande n'est pas achevé avant la transmission du caractère suivant.

4.4 Communication radiofréquence

La génération d'une interruption timer permet de déclencher un évènement à des intervalles de temps précis. Cela permet notamment de réaliser une communication asynchrone de façon logicielle et ainsi palier à la limitation d'un unique port série, mais surtout de réaliser un port série logiciel communiquant par une autre voie que la liaison filaire. En plus de simplement déclencher une transition de niveau en fonction de la valeur du bit à transmettre (fonction classique d'un UART), nous pouvons effectuer des tâches plus complexes tant que leur durée d'exécution est inférieure à l'intervalle de temps défini par le timer.

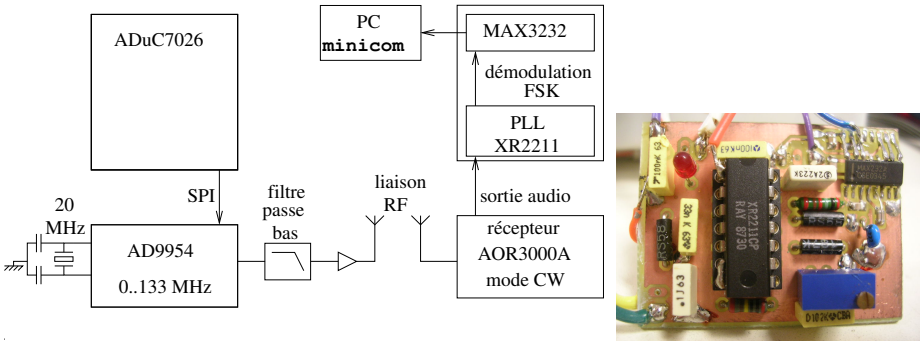


Fig. 4 – Gauche : liaison radiofréquence pour la transmission de données numériques ne nécessitant qu'un composant actif, le synhtétiseur de fréquence direct (DDS) AD9954. Droite : circuit de démodulation du signal codé en FSK à base de XR2211, basé sur le schéma de la Fig.11 de la datasheet de ce composant (version de Juin 1997). Ce dispositif prend en entrée la sortie audio d'un scanner radiofréquence (en mode CW, donc ayant converti le signal RF en signal BF par mélange avec un oscillateur local autour de 138 MHz) et fournit en sortie des trames compatibles en niveau avec le RS232.

Une application qui nous a intéressé est la transmission de données numériques par liaison radiofréquence, et ce avec le montage nécessitant le moins de composants possibles. Nous proposons ici un montage avec un unique composant numérique – le synthétiseur direct radiofréquence (DDS) AD9954. Toutes les communications entre le microcontrôleur et l'AD9954 se font en SPI, avec deux

```
// ATTENTION : irq.c ne DOIT PAS etre compile en thumb
// -fno-omit-frame-pointer
// bug de gcc : http://gcc.gnu.org/bugzilla/show_bug.cgi?id=16634
// temperature= (bits/4096*2500 -780)/1.3 +25 deg C

#include<ADuC7026.h>

volatile unsigned int tim0=0;

void delay(int del)
{
 while (del>0) del--;
}

int jmf_putchar(int ch)  {                  // Write character to Serial Port
  do {} while ((COMSTA0 & 0x020)!=0x20);
  COMTX = ch;
  return (1);
}

int jmf_getchar (void)  {  // lecture non bloquante du char : 0 signifie rien
  if((0x01==(COMSTA0 & 0x01))) return (COMRX);
  else return(0);
}

void writeASC(unsigned short *ptr,int len)
{int i;
 unsigned char b;
 for (i=0;i<len;i++)
 {b=((ptr[i]&0xf000)>>12);
  if (b<10) jmf_putchar(b+48); else jmf_putchar(b+55);
  b=((ptr[i]&0x0f00)>>8);
  if (b<10) jmf_putchar(b+48); else jmf_putchar(b+55);
  b=((ptr[i]&0x00f0)>>4);
  if (b<10) jmf_putchar(b+48); else jmf_putchar(b+55);
  b=((ptr[i]&0x000f));
  if (b<10) jmf_putchar(b+48); else jmf_putchar(b+55);
 }
}

// int write(unsigned char * ptr, int len) {
int jmf_write(unsigned char * ptr, int len) {
  int i;
  for (i = 0; i < len; i++) jmf_putchar (*ptr++);
  return len;
}

void ADCpoweron(int time)
{ADCCON = 0x20;
 while (time >=0) time--; // wait for ADC to be fully powered on
}

unsigned short temperature()
{unsigned short moy=0;
 unsigned char k;
 ADCCP=16;
 for (k=0;k<16;k++) {
   ADCCON=0x7E3;
   while (!ADCSTA) {}
   moy+=(unsigned short)((ADCDAT>>16)&0xfff);
 }
 ADCCP = 0;             // channel number
 return(moy);
}

// extern __attribute__ ((interrupt("IRQ"))) void IRQ_Handler(void)
void IRQ_Handler(void)
{int irqs=IRQSIG; // IRQSTA;

 if (irqs & GP_TIMER_BIT)  { // Timer1 Interrupt
     T1CLRI = 1;             // Clear Timer 1 interrupt
     tim0=1;      // !!! tim0++ renvoie n'importe quoi !!!
  }

 if (irqs & UART_BIT)  {     // UART Interrupt
     jmf_putchar(jmf_getchar()-0x20); // minuscule -> maj
    }

 if (irqs & RTOS_TIMER_BIT) {// Timer0 Interrupt
     jmf_putchar('t');
     T0CLRI= 0;
    }
}

int main(void) {
        int sec=0,tempe,heu=0,min=0;

GP1CON = 0x22220011; // config SPI on P1.4-P1.7, & uart P1.0 and P1.1
SPIDIV = 0x06;       // set SPI clock 40960000/(2x(1+SPIDIV))
SPICON = 0x1043;     // ena SPI master in continuous transfer mode

GP3DAT = 0xff000000;

    COMCON0 = 0x080;     // Setting DLAB
    COMDIV0 = 23;        // 23; pour 57600, 0x88=9600, 0x44=19200
    COMDIV1 = 0x000;     // *** RESET DE LA CARTE ***
    COMCON0 = 0x007;     // Clearing DLAB
// The baudrate is calculated with the following formula:
//
// DL = HCLK
//      -------
// Baudrate * 2 *16

     COMIEN0=1;

ADCpoweron(20000);   // power on ADC
REFCON= 0x01;      // internal 2.5V reference. 2.5V on Vref pin
     ADCCP = 0;           // channel number

       IRQ = IRQ_Handler;
  T1LD=32768;    // 41780000/256, 32 bits => 1 Hz
  T1CON=0x2C0;   // external xtal actif, pas de division 4=/16
  T0LD=16320;    // 41780000/256, 32 bits
  T0CON=0xC8;    // 163 kHz => 10 Hz a 16320
IRQEN = UART_BIT | GP_TIMER_BIT | RTOS_TIMER_BIT ;

tim0=0;
while (1)
          {
           if (tim0!=0)
      {jmf_putchar(tim0+'0');jmf_putchar(32);
       writeASC(&heu,1);jmf_putchar(':'); // heure:min:sec
       writeASC(&min,1);jmf_putchar(':');
       writeASC(&sec,1);jmf_putchar(32);
       tempe=temperature();
       writeASC(&tempe,1);jmf_putchar(10);jmf_putchar(13);
       tim0=0;
       sec++;
       if (sec==60) {min++;sec=0;}
       if (min==60) {heu++;min=0;}
      }

  }   // fin du while (1)
       return(1);
}
```

TAB. 4 – Exemple de gestion de deux timers (0 pour un décompte à 10 Hz et 1 pour une horloge temps réel avec affichage du temps toutes les secondes), et de la réception de caractères sur le port série. L'affichage de l'heure s'accompagne de l'affichage de la température, mesurée au moyen de la sonde interne de l'ADuC7026. La valeur lue sur le convertiseur se traduit en degrés celsius par $T_{degC} = \frac{bits/4096 \times 2500 - 780}{1{,}3} + 25$
.

signaux additionnels de validation des informations transmises, ici implémenté par un GPIO, que nous décrirons en détail plus loin.

La stratégie qui dirige nos choix technologiques est de maintenir le nombre de composants au minimum (gain en place et en consommation) et de définir le maximum d'opérations par logiciel afin de conserver un maximum de souplesse. La transmission se fera par modulation FSK (*Frequency Shift Keying*) : chaque valeur de bit à transmettre sera codée par une fréquence différente. Nous travaillerons ici sur 2 valeurs possibles de bits (une fréquence pour 1 et une fréquence pour 0). La stratégie que nous nous sommes proposés d'implémenter est la suivante :

– deux fréquences correspondent aux 1 et 0 à transmettre sont programmées par bus SPI dans le DDS : par exemple 138 MHz+1200 Hz et 138 MHz+2200 Hz.

– une interruption timer du microcontrôleur teste si un octet est en cours de transfert et, si c'est le cas, définit le bit à transmettre et donc la valeur de la fréquence à programmer dans le DDS (par exemple interruption toutes les 833 μs pour du 1200 bauds)
– du point de vue de la démodulation, un scanner AOR3000A est placé en mode CW pour une démodulation avec un oscillateur local à 138 MHz. Les deux tonalités issues du mélange sont alors à 1200 et 2200 Hz.
– une PLL de type XR2211 convertit les deux tons en signaux TTL, transmis au port série après passage dans un MAX232. Ce composant, obsolète, contient un exemple d'application qui correspond exactement à notre application : il mériterait cependant à être remplacé par un composant plus récent. Une alternative intéressante – que nous n'avons pas réussi à mettre en œuvre dans ce cas – est la démodulation par logicielle après acquisition par carte son du signal audio issu du récepteur radiofréquence (par exemple par `multimon`).

Cette stratégie s'est avérée à l'usage peu robuste car fortement dépendante de la fréquence de l'oscillateur local embarqué qui cadence le synthétiseur. Ce point est fondamental pour comprendre par ailleurs pourquoi une stratégie de type modulation (par exemple modulation de fréquence FM) est robuste, contraitement à notre stratégie

– dans une modulation de fréquence (Fig. 5, haut), une porteuse (par exemple 138 MHz) est modulée par la fréquence codant la valeur du bit à transmettre. Le spectre résultant contient *3 raies* : la porteuse, et les deux raies de modulation de part et d'autre de la porteuse. Lors de la démodulation, le récepteur peut se *caler* sur la porteuse afin de se recentrer pour convenablement identifier les raies de modulation. La référence (la porteuse) est donc contenue dans le signal, et toute dérive de l'oscillateur embarqué se traduit par une dérive de la porteuse, sur laquelle le récepteur peut se recaler tant que la dérive est inférieure à la bande passante du filtre de réception. Dans le schéma de démodulation que nous proposons, le signal d'erreur entre l'oscillateur local et la porteuse est directement disponible en sortie du filtre passe-bas LPF2 dont la fréquence de coupure est sous la fréquence de modulation audio (filtre très lent). Le filtre LPF1 a uniquement pour vocation de retirer les raies inutiles issues du mélangeur et a une fréquence de coupure au-dessus de la fréquence audio.
– dans notre stratégie, nous ne transmettons qu'une unique raie – l'équivalent de la porteuse dans le cas précédent – et faisons *l'hypothèse* de connaître la fréquence de référence que nous avons fixé par convention, et qui est la fréquence que nous programmons dans le récepteur radio. Cependant, toute dérive de l'oscillateur embarqué qui cadence le DDS par rapport à l'oscillateur local du récepteur au sol se traduit par une dérive de la sortie du mélange et donc une fréquence audio de sortie qui n'est pas une des valeurs désirées. Par ailleurs, tout biais initial d'un des deux oscillateurs se traduit par un décalage entre la fréquence théorique à programmer, et la fréquence que nous programmons en pratique dans le récepteur pour obtenir un signal audio exploitable. Nous avons constaté en pratique qu'une correction de quelques centaines de Hz est nécessaire, probablement en raison d'un décallage du résonateur faible coût embarqué aux côtés de l'AD9954 ou d'un ajustement peu précis des condensateurs de pieds de l'oscillateur.

Le synthétiseur de fréquences direct AD9954 se programme par protocole SPI et ne nécessite donc que 5 signaux de l'ADuC7026 : horloge, données du micronctrôleur vers le DDS, signal d'activation (*Chip Select*, actif bas), reset et un signal de validation de fin de transfert nommé `LE` (Latch Enable) ou `FQ_UD`.

```
#include<ADuC7026.h>

#define attend   15      // delai apres communication SPI et pour les CS# (20)

#define FSTOP    (0x58521DDA) // 138 MHz + 1200
#define FSTART   (0x585247CB) // 138 MHz + 2200, 400 MHz CK
#define FRIEN    (0x58000000) // 138 MHz + 2200, 400 MHz CK

// scanner 138.0182 CW, decodeur sur voie directe (PAS de NOT)

volatile int global=0,g2=0;
```

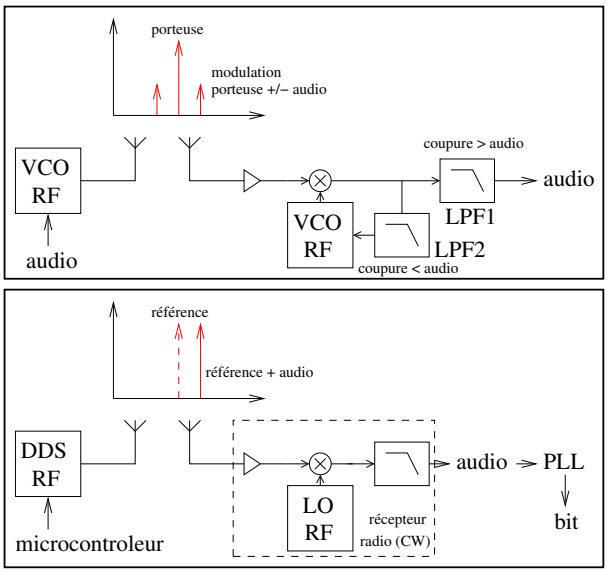


FIG. 5 – En haut : chaîne d'émission-réception avec une modulation en fréquence (VCO=*Voltage Controlled Oscillator*, LPF=*Low Pass Filter*), voir le texte pour les explications. En bas, la stratégie que nous proposons, nécessitant moins de composants mais moins robuste.

Après avoir chargé le fichier de définition des symboles associés aux périphériques de l'ADuC7026, nous déterminons les deux fréquences de communication qui correspondent aux transmissions de bit 1 et 0. Un troisième état d'absence de communication est prévu mais ne sera pas utilisé. Ces mots s'obtiennent en calcul $mot_{bits} = freq_{voulue}/400 \times 2^{32}$ avec 400 la fréquence d'horloge interne au DDS obtenue en multipliant par 20 l'oscillateur externe (résonateur 20 MHz).

```
// registres DDS AD9954
unsigned char CFR1[5] = {0x00, 0x00, 0x00, 0x00, 0x0C}; // CFR1 00 20 00 08
unsigned char CFR2[4] = {0x01, 0x18, 0x02, 0xA4};       // CFR2 $A0=x20 => 400 MHz
unsigned char ASF[3]  = {0x02, 0xC4, 0x50};             // ASF
unsigned char ARR[2]  = {0x03, 0xFC};                   // ARR
unsigned char FTW0[5] = {0x04, 0x18, 0x00, 0x00, 0x00}; // FTW0 2A 00 70 A3 = 32.814 MHz 2A FF 70 A3 = 33.592
unsigned char POW0[3] = {0x05, 0x00, 0x00};             // POW0
unsigned char FTW1[5] = {0x06, 0x2C, 0x8B, 0x43, 0x95}; // FTW1
unsigned char RSCW0[6] = {0x07, 0x32, 0x00, 0x1D, 0x7D, 0x8F};  // RSCW0
unsigned char RSCW1[6] = {0x08, 0x32, 0x00, 0x00, 0x21, 0x8D};  // RSCW1
```

La configuration des registres du DDS se résume en l'application des valeurs proposées dans la datasheet, avec une multiplication par 20 par PLL de la fréquence du résonateur externe. Le premier octet correspond à l'adresse de chaque registre dans le DDS.

```
void delay (int length) {while (length >=0) length--;}
```

La fonction de délai par boucle vide est une façon gourmande en énergie et en ressources de faire patienter le microcontrôleur, mais tellement simple à mettre en œuvre par rapport à une interruption timer que nous ne résisterons pas à l'utiliser. Cette fonction est d'autant plus une mauvaise idée qu'elle est programmée en C et donc sa durée dépend du compilateur, du niveau d'optimisation et donc du code assembleur généré (section 5).

```
void envoi_DDS1(char *entree,int n)
{unsigned char i;
 GP4DAT &= ~0x00200000; // CS# P4.5 = lo

 delay(attend);
 for (i=0;i<n;i++)
     {SPITX = entree[i];         // transmit command
      do {} while ((SPISTA & 0x01) == 0x01) ; // wait for data received status
     }
delay(attend);

GP4DAT |= 0x00200000;  // CS# P4.5 = hi

delay(attend);
```

```
 GP4DAT |= 0x00040000;  // FQ_UD  P4.2
 delay(attend);
 GP4DAT &= ~0x00040000; //
 delay(attend);
}

void programme(int i)
{unsigned int freq1;
 if (i==0) {     // bit 0 = bas
    GP4DAT&=~0x00010000; // UART logiciel
    freq1=FSTART;        // freq. DDS
    FTW0[1]=(freq1 & 0xFF000000)>>24;
    FTW0[2]=(freq1 & 0xFF0000)>>16;
    FTW0[3]=(freq1 & 0xFF00)>>8;
    FTW0[4]=(freq1 & 0xFF);
    envoi_DDS1(FTW0,5);
     }
 if (i==1) {     // bit 1 = haut
    GP4DAT|=0x00010000; // UART logiciel
    freq1=FSTOP;        // freq. DDS
    FTW0[1]=(freq1 & 0xFF000000)>>24;
    FTW0[2]=(freq1 & 0xFF0000)>>16;
    FTW0[3]=(freq1 & 0xFF00)>>8;
    FTW0[4]=(freq1 & 0xFF);
    envoi_DDS1(FTW0,5);
   }
```

```
}

void init_DDS(void){
 unsigned int i;

 GP4DAT |= 0x00020000;  // RESET P4.1
 delay(50);
 GP4DAT &= ~0x00020000;
 delay(50);

 for (i=0;i<10;i++) {   // init DDS 10 fois (un peu dur d'oreille ...)
   envoi_DDS1(CFR1,5);
   envoi_DDS1(CFR2,4);
   envoi_DDS1(ASF,3);
   envoi_DDS1(ARR,2);
   envoi_DDS1(FTW0,5);
   envoi_DDS1(POW0,3);
   envoi_DDS1(FTW1,5);
   envoi_DDS1(RSCW0,6);
   envoi_DDS1(RSCW1,6);
   //envoi_DDS(RSCW2,6);// inutile ici
   //envoi_DDS(RSCW3,6);// inutile ici
   delay(50);
 }
 programme(1);          // repos = haut
}
```

Les routines de communication avec le DDS sont simples et se contentent d'exploiter la communication par interface SPI fournie comme périphérique matériel par l'ADuC7026. Les signaux de contrôle de la communication avec le DDS sont générés manuellement en plus de la communication synchrone.

En plus de la communication par liaison radiofréquence, nous désirons maintenir la possibilité de debugger le programme par une liaison RS232 classique cadencée par UART, obtenue en définissant la valeur d'une broche de GPIO (P4.0 dans la fonction `programme()` de l'exemple proposé ci-dessus) en même temps que la programmation du DDS. Cette émulation logicielle d'un port série permet d'étendre le nombre de ports de communication asynchrone au-delà des UART fournis par le microcontrôleurs.

```
void InitTimer(void){
 T0LD=34588; // 4001=96 us, 8001=192
 T0CON=0xC0;
 IRQEN = RTOS_TIMER_BIT;          /* Configure Timer 0     */
}

void IRQ_Handler (void) {
  if (IRQSIG & RTOS_TIMER_BIT)  {       // Timer0 Interrupt
        T0CLRI=0;
        switch (g2)
         {
          case 10: {programme(0);g2--;break;}   // START bit
          case 9:
          case 8:
          case 7:
          case 6:
          case 5:
          case 4:
          case 3:
          case 2: {programme(global&0x01);global=global>>1;g2--;break;}
          case 1: {programme(1);g2--;break;}    // STOP bit
        //  case 0: {programme(-1);}
          default: break;
         }
        }
}
```

Finalement, le cœur de notre démonstration de synthèse d'UART logiciel – que ce soit en vue d'une liaison radiofréquence ou filaire – est centrée sur l'exploitation d'une interruption timer afin de changer l'état de la broche de communication à intervalles de temps réguliers correspondant au baud rate (833 μs pour 1200 bauds).

```
void InitPort(void) {
 GP1CON = 0x22220011;   // config SPI, setup tx & rx pins on P1.0 and P1.1
 SPIDIV = 0x0A;         // set SPI clk = 40960000/(2x(1+SPIDIV))
```

```
 SPICON = 0x1043;

 GP4DAT = 0x7f180000;   // P4.[2-4] en sortie, P4.[3,4] = 1 & P4.2 = 0
}

int main(void)
{unsigned char c;

 IRQ = IRQ_Handler; // Specify Interrupt Service Rountine

 InitPort();
 InitUart();
 init_DDS();
 InitTimer();

 while (1) for (c=32;c<127;c++) WriteChar(c);
}
```

Rappelons que dans cet exemple, l'AD9954 est cadencé par un résonateur à 20 MHz, multiplié en interne pour générer un signal d'horloge à 400 MHz – facteur de multiplication défini par logiciel dans le registre CFR2 du DDS – qui permet alors une synthèse de n'importe quelle fréquence entre 0 et 133 MHz. La fréquence qui nous intéresse dans le cadre de la communication dans un projet de Planète Sciences sur les fréquences allouées au CNES (autour de 138 MHz) sont suffisamment proches pour être exploitables tout en conservant un filtre passe bas en sortie de DDS pour filtrer efficacement l'horloge (400 MHz) et les raies parasites (400±138 MHz).

Le microcontrôleur reprogramme par le port SPI la nouvelle fréquence à émettre correspondant à chaque nouveau bit à transmettre. Du côté de la réception, un mélange avec un oscillateur à 138 MHz génère les signaux aux fréquences audio correspondant aux 1 et 0, qui sont alors fournis à un MAX232 pour mise à niveau et exploitation par l'UART du PC. Ce concept a été mis en œuvre dans diverses situations, allant d'une liaison sur plusieurs dizaines de mètres en vue directe entre émetteur et récepteur, à une communication pendant 11 h pour accumuler une statistique de taux d'erreur sur près de 500 KB transmis (Fig. 6). Dans ce second cas, il est intéressant de noter que le taux d'erreur évolue avec le temps du fait de la dérive avec la température soit de l'oscillateur du récepteur radio, mais plus probablement de la température du quartz qui cadence le DDS. Afin de tourner cette dérive à notre avantage, il serait envisageable d'observer les fréquences du signal audio reçu, en observer l'écart à la consigne (et ainsi corriger la fréquence du récepteur CW) tout en déduisant ainsi la température du résonateur à quartz fixé sur la carte du DDS. Avec un récepteur convenablement réglé, le taux d'erreur est inférieur au pourcent, nécessitant soit une redondance, soit un code correcteur d'erreurs pour être exploité dans une application autre que purement démonstrative.

Une perspective intéressante de ce travail serait d'exploiter la RAM interne à l'AD9954 qui permet de configurer une séquence quelconque de fréquences à émettre, et notamment la suite de fréquences correspondant à une modulation de fréquence. Dans ce cas, la commutation entre transmission de 1 ou de 0 en FSK serait simplement activée en modifiant la vitesse à laquelle la RAM est balayée pour mettre à jour la fréquence émise.

5 Comparaison de compilateurs

L'exemple qui suit n'a pas pour vocation de comparer les performances de divers compilateurs (gcc 4.0.0 et Keil 2.42) mais simplement d'insister sur le fait qu'un langage de haut niveau tel que le C peut résulter en des codes assembleur (et donc des occupations mémoire et temps d'exécution) très différents selon les compilateurs et options de compilation, et que seul le langage machine garantit un résultat optimal et reproductible pour définir les tâches les plus importants d'un code : nous compilons avec `arm-gcc-4.0` et la version 2.42 du compilateur propriétaire de Keil la boucle vide `void delay (int length) while (length >=0) length-- ;`. Ce type de code, bien que résultant en des délais dépendant du compilateur, se retrouve souvent dans les programmes qui nécessitent une temporisation fine (< 1 ms) de leur exécution :

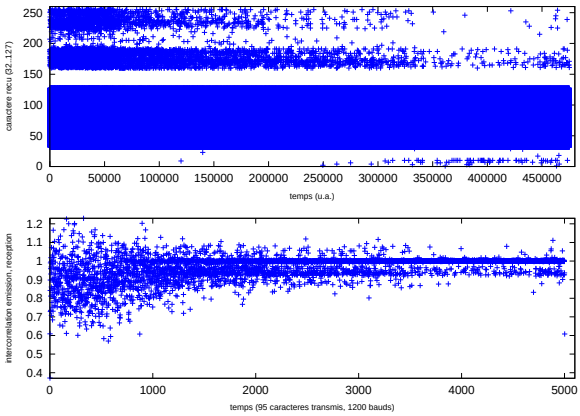


FIG. 6 – Analyse du taux d'erreur des informations transmises pendant 11 h par la liaison radiofréquence décrite dans le texte. Les caractères de 32 à 127 (ASCII affichable) sont continuellement émis au débit de 1200 bauds. Le graphique du haut présente l'évolution des caractères reçus en fonction du temps, avec un taux d'erreur clairement décroissant lorsque les oscillateurs de l'émetteur (DDS AD9954) et du récepteur s'approchent lorsque la température décroît au cours de la nuit. En bas : une analyse quantitative du taux d'erreur est obtenu par calcul du maximum d'intercorrélation entre la séquence reçue et la séquence émise, toutes deux normalisées pour présenter une puissance de 1 par unité de temps. La conversion entre puissance de l'intercorrélation et taux d'erreur de communication ne semble pas triviale : une analyse grossière des premiers paquets de données transmis semble indiquer un taux d'erreurs de l'ordre de 5% en début de séquence de transmission, pour se réduire en dessous du % en fin de transmission.

arm-thumb-elf-gcc (GCC) 4.0.0

```
196            delay:
197                    @ lr needed for prologue
198 0000 00E0          b       .L2
199            .L3:
200 0002 0138          sub     r0, r0, #1
201            .L2:
202 0004 0028          cmp     r0, #0
203 0006 FCDA          bge     .L3
204                    @ sp needed for prologue
205 0008 7047          bx      lr
```

Keil ARM Compiler V2.42

```
  47: void delay (int length) {       // delay(5000)=1 ms
00000000  B401  PUSH    {R0}
  48:     while (length >=0) length--;
00000002  E003  B       L_1  ; T=0x0000000C
00000004      L_3:
00000004  A800  ADD     R0,R13,#0x0
00000006  6801  LDR     R1,[R0,#0x0] ; length
00000008  3901  SUB     R1,#0x1
0000000A  6001  STR     R1,[R0,#0x0] ; length
0000000C      L_1:
0000000C  A800  ADD     R0,R13,#0x0
0000000E  6800  LDR     R0,[R0,#0x0] ; length
00000010  2800  CMP     R0,#0x0
00000012  DAF7  BGE     L_3  ; T=0x00000004
  49: }
00000014  B001  ADD     R13,#0x4
00000016  4770  BX      R14
00000018      ENDP ; 'delay?T'
```

Nous noterons que les deux compilateurs effectuent un passage de paramètre par le registre `r0`, mais là où `gcc` se contente de générer un code limpide limité à une soustraction et une comparaison, le compilateur Keil exploite la pile (`r13` sur laquelle est placée la valeur de `length` grâce au premier `push`) et le registre `r1` comme intermédiaire de ses calculs. La pile est dans ce cas remise en état en fin de boucle en ajoutant 4 au pointeur de pile `r13`. La conséquence est un temps d'exécution considérablement plus long – pour une même valeur de `length` – lorsque le code C est compilé par l'outil Keil que par `gcc`. Nous avons observé des manipulations de registres similaires par le compilateur Keil dans des bouts de code qui n'avaient pas pour vocation à être des attentes et dont la vitesse d'exécution devait être maximale. L'élimination manuelle de ces manipulations inutiles dans le code assembleur retire tout l'intérêt de la programmation en C sur microcontrôleur : seul un bon compilateur aux performances au moins égales à celles obtenues par la programmation manuelle en assembleur justifie l'utilisation d'un langage de haut niveau tel que le C. `gcc` génère un code tel qu'un humain l'écrirait naturellement (cette règle ne se généralise cependant pas lors de l'utilisation d'options d'optimisation qui peuvent amener `gcc` à générer du code performant mais illisible).

Une façon simple d'éliminer la dépendance du temps d'exécution de bouts de code critiques avec la version du compilateur ou les options d'optimisation est d'inclure dans le code C le bout d'assembleur qui a donné satisfaction, tel que proposé soit par l'option `-S` de `gcc` ou `-dSt` de `objdump`. Par exemple, pour inclure dans `gcc` le code assembleur issu de la compilation par Keil de la fonction `delai()` proposée ci-dessus, nous utiliserons l'assembleur de la façon suivante :

```
void delay (int length) { // length=45 => 2.43 us
//       while (length >=0) length--;
asm ( "    PUSH {R0}        ");
asm ( "    B    L_1         ");
asm ( "L_3:                 ");
asm ( "    ADD  R0,R13,#0x0 ");
asm ( "    LDR  R1,[R0,#0x0]");
asm ( "    SUB  R1,#0x1     ");
asm ( "    STR  R1,[R0,#0x0]");
asm ( "L_1:                 ");
asm ( "    ADD  R0,R13,#0x0 ");
asm ( "    LDR  R0,[R0,#0x0]");
asm ( "    CMP  R0,#0x0     ");
asm ( "    BGE  L_3         ");
asm ( "    ADD  R13,#0x4    ");
  }
```

après avoir constaté que le passage de paramètres à la fonction se fait de la même façon pour les deux compilateurs.

Une étude plus exaustive, et illustrant bien la complexité de l'analyse des benchmarks, est proposée à [10].

6 Un peu de calcul utile ...

Afin d'illustrer le calcul sur des flottants et d'en estimer le coût en temps de calcul (et donc en consommation électrique), nous proposons de recherche les racines d'une fonction complexe par la méthode de Newton.

Pour rappel, il s'agit d'une méthode itérative (Fig. 7) dans laquelle depuis une estimation grossière initiale de la position de la racine, l'algorithme recherche l'intersection de la tangente à la fonction avec l'axe des abscisses et itère le processus jusqu'à réduire l'erreur en-dessous d'un seuil donné. La dérivée de la fonction $f(z)$ au point z est $f'(z)$: la droite de pente $f'(z)$ et passant par le point $(z, f(z))$ a pour équation $y = f'(z) \times (x - z) + f(z)$, qui intersecte l'axe des ordonnées $y = 0$ en $x = z - f(z)/f'(z)$. La recherche de racine se réduit donc à étudier la convergence de la suite $z_{n+1} = z_n - f(z_n)/f'(z_n)$.

Cette méthode peut évidemment s'étendre aux fonctions complexes en prenant $z \in \mathbb{C}$: dans ce cas, nous nous proposons non seulement de trouver les racines de la fonction complexe f, mais aussi de savoir vers quelle racine converge la suite et à quelle vitesse en fonction de la condition initiale choisie z_0. Cette étude est à la base de la fractale dite de Newton : on démontre en effet que la frontière entre les bassins des diverses racines se ressemble à elle même quelque soit le grossissement effectué sur cette zone. Nous allons illustrer ce concept à la fonction polynomiale $f(z) = z^3 - 1$ avec $z \in \mathbb{C}$. Les trois racines de ce polynôme sont trivialement $e^{j2n\pi/3}$, $n = [0..2]$ et avec $j^2 = -1$.

La figure 8 présente graphiquement la racine vers laquelle converge la suite $z_{n+1} = z_n - \frac{z^3-1}{3\times z^2}$, avec le rouge, le vert et le bleu présentant les 3 racines. Nous pouvons aussi calculer le nombre d'itérations nécessaire pour atteindre cette racine à une distance inférieure à 0,1 (carré du module $< 0,01$).

Au delà de son aspect esthétique, ce calcul démontre que l'ARM7TDMI est suffisamment performant pour implémenter une méthode itérative de recherche d'une solution (ici racine de polynôme, mais cette stratégie pourrait s'étendre aux descentes de gradients pour identifier des paramètres optimaux de fonctions s'ajustant à des données expérimentales, ou à la phase d'apprentissage d'un réseau de neurones artificiel) et dépasser le stade du simple système embarqué de type automate capable uniquement de réagir de façon simple à un stimulus.

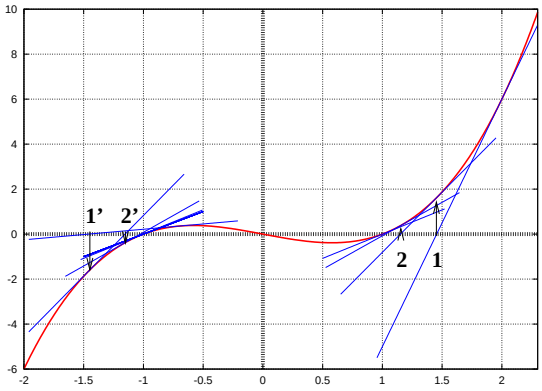


FIG. 7 – Exemple de recherche d'un zéro de fonction par la méthode de Newton pour le polynôme $z^3 - z$, $z \in \mathbb{R}$. La racine vers laquelle converge l'algorithme dépend du point de départ : ici la suite a été initialisée avec $z = -0,7$ et $z = 2$, induisant une limite respectivement vers la racine -1 et 1. Les premières itérations de chaque suite sont marquées 1', 2' et 1, 2 respectivement.

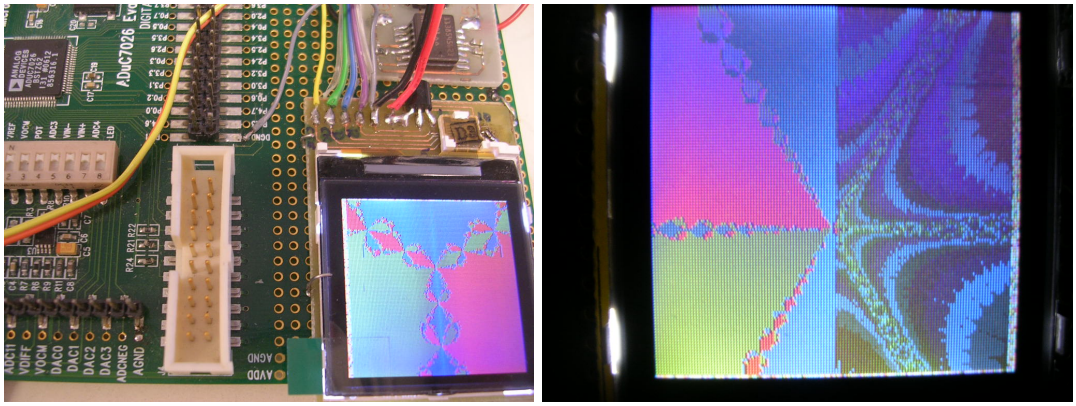


FIG. 8 – Exploitation de l'affichage graphique sur écran LCD couleur et du calcul sur flottants pour afficher la fractale de Newton (à gauche de $-3-2,5i$ à $2+2,5i$ et à droite lors d'une transition du nombre d'itérations nécessaire à la convergence vers la racine atteinte par l'algorithme, avec un grossissement autour de la région centrée sur $0+i\times0$.

Les temps d'exécution de 128×128 recherches de racines dans des intervalles commençant entre $2 + 2,5i$ et $-3 - 2,5i$ et dont les bornes sont divisées par 2 à chaque itération pour zoomer vers l'origine sont résumés dans le tableau 5.

Nous constatons donc que le gain d'un facteur 10 en précision sur le résultat n'induit par une augmentation significative du temps de calcul, qui augmente de 10 à 20% selon les facteurs de grossissement. Tous ces calculs se sont fait sur 128×128 itérations. Nous en déduisons un temps moyen de calcul et, connaissant la puissance moyenne consommée par le microcontrôleur d'après la datasheet (40 mA sous 3,3 V, soit 132 mW), nous en déduisons l'énergie consommée par une itération moyenne vers une racine du polynôme. La valeur obtenue varie entre 140 et 440 μJ, dépendant du nombre d'itérations qu'il a fallu réaliser pour atteindre la précision voulue. Pour rappel, une masse de 10 g dans le champ gravitationnel terrestre ($9,81$ kg.m.s^{-2}) acquiert une énergie potentielle de 980 μJ/cm qui, restituée sous forme de courant électrique par une dynamo avec un rendement de 10 % par exemple, se traduit par le fonctionnement du microcontrôleur pour une recherche de racine nécessitant une chute d'une bille de 10 g d'une hauteur de 5 mm environ.

Précision du calcul	Nombre itérations	Temps total (μs)	Temps moyen /itération (μs)	Énergie /itération (μJ)
0,01	109271	21073346±15	1286	170
0,01	104804	20177258±15	1232	163
0,01	127755	24509548±3998	1496	197
0,01	169397	32338043±3	1974	261
0,01	225585	42950737±4000	2622	346
0,01	281571	53531382±54	3267	431
0,1	90765	17593112±2	1074	142
0,1	86995	16825063±2	1027	136
0,1	109406	21061330±4002	1285	170
0,1	151231	28913824±4	1765	233
0,1	207337	39514498±2	2412	318
0,1	263290	50083163±4	3057	404

TAB. 5 – Temps de calcul mesurés sur un ADuC7026 pour itérer 128×128 recherches de racine, et estimation de l'énergie moyenne consommée par chacun de ces calculs. Le nombre d'itérations – qui est le nombre total de calculs du polynôme nécessaire au tracé de la fractale – a été calculé sur PC (Intel 32 bits).

7 Compression de données

La transmission de données communément utilisée sur systèmes embarqués se fait à bas débit : RS232, bluetooth, zigbee, radiomodems ... La mise sous tension d'éléments gourmands en énergie – modem et processeur chargé de la communication tel que nous l'avons vu en section 4.4 – nous incitent à réduire au mieux la quantité de données à transmettre.

Une stratégie pour minimiser la quantité d'informations transmise tient en l'optimisation du codage des données, ou en d'autres termes la compression sans pertes des échantillons acquis.

Le compromis qui va nous intéresser est le compromis entre la consommation de ressources associées à la compression (mémoire occupée et temps de calcul) et la réduction du temps de communication. Le taux de compression étant dépendant de la nature des échantillons, nous ne prétendrons par fournir une réponse définitive à cette question mais simplement effleurer le sujet par une analyse statistique sur quelques échantillons de données représentatives d'informations que nous serions susceptibles de collecter.

Les implémentations "classiques" d'algorithmes de compression sans pertes performants (Huffman, Lempel-Ziv) telles que fournies par exemple dans `libz` ne sont pas appropriées pour une exploitation sur systèmes embarqués : leur exploitation de `malloc` handicape leurs performances sur systèmes sans unité de gestion de mémoire (MMU) et la compilation pour ARM7 de la version 1.2.3 de zlib génère un code prêt a flasher de 83 KB environ, soit plus que toute la mémoire flash disponible sur l'ADuC7026.

C. Sadler a développé, pour un projet de réseau de capteurs mobiles, un version allégée de la compression sans pertes pour microcrontrôleurs MSP430 [12] : S-LZW est disponible à `http://cmsadler.googlepages.com/slzw.tar.gz`. Les ressources du processeur pour lequel ce code a été écrit étant à peu près identiques à celle de de l'ADuC7026, cette bibliothèque de fonctions est un bon candidat pour nos tests.

La configuration par défaut de l'exemple fourni exploite un tableau (variable globale `unsigned char* write_buffer`) en entrée de 500 octets. La compression fournit un nouveau tableau de dimensions *a priori* inférieur : `unsigned char *lzw_output_file_buffer`. La subtilité majeure dans l'exploitation de ce code, comme l'a indiqué son auteur Christopher Sadler, est le passage d'une architecture 16 bits pour laquelle les programmes ont été écrits (MSP430) vers des architectures 32 bits (x86 pour test sur PC, ou ARM7 qui va nous intéresser ici). Les corrections à apporter consistent à remplacer toutes les références aux `int` par des `short` et, dans `slzw.c`, définir `last_entry` comme un `short` signé au lieu de non-signé (faute de quoi la décompression ne se fera pas).

```
/* Sample program that shows how to use the S-LZW with Mini-Cache Functions.
   Its designed to crash if there is an error so that you can debug it.
*/

#include <stdio.h>
#include "slzw.h"
#include <string.h>

#ifdef arm7jmf
#include "ADuC7026.h"

void initUART()
{ GP1CON = 0x00000011;  // configure SPI, setup tx & rx pins on P1.0 and P1.1
  GP4DAT = 0xff180000;  // P4.[2-4] en sortie, P4.[3,4] = 1 & P4.2 = 0
  GP2DAT = 0xff100000;  // P2.0-7 en sortie, initialises a 0;
  GP0DAT = 0xffff0000;  // P0.0-3 et P0.4-7 controlent les deux attenuateurs

  COMCON0 = 0x080;   // Setting DLAB
  COMDIV0 = 0x88;    // Setting DIV0 and DIV1 to DL 88=9600, 44=19200
  COMDIV1 = 0x000;   // *** NE PAS LANCER PAR RUN MAIS RESET DE LA CARTE ***
  COMCON0 = 0x007;   // Clearing DLAB
}
#endif

int main(void) {
 unsigned short i, j, count = 0, compressed_size;
#ifdef arm7jmf
        initUART();
#endif

for (i=0; i<500; i++)
#ifdef arm7jmf
   write_buffer[count++]=uart0_getc();
#else
   write_buffer[count++]=getc(stdin);
#endif

 count=0;for (i=0; i<500; i++) printf("%c",write_buffer[count++]);

 printf("start compress\n");
 // Use a 32 entry mini-cache.
 // input = write_buffer, output = lzw_output_file_buffer
 compressed_size = slzw_compress(500, 32);
 printf("stop compress %d\n",compressed_size);

 // Lets wipe out the original data so that when we decompress, we
 // know that the data was really generated by the decompressor.
 memset(write_buffer, 0x00, 500);

 printf("start decompress\n");
 // input = lzw_output_file_buffer, output = write_buffer
 printf("stop decompress %d\n",slzw_decompress(compressed_size));

 count = 0;
 for (i=0; i<500; i++)
#ifdef arm7jmf
   uart0_putc(write_buffer[count++]);
#else
   putc(write_buffer[count++],stdout);
#endif

 printf("\nSuccess!\n");
 for (;;);
 return 0;
}
```

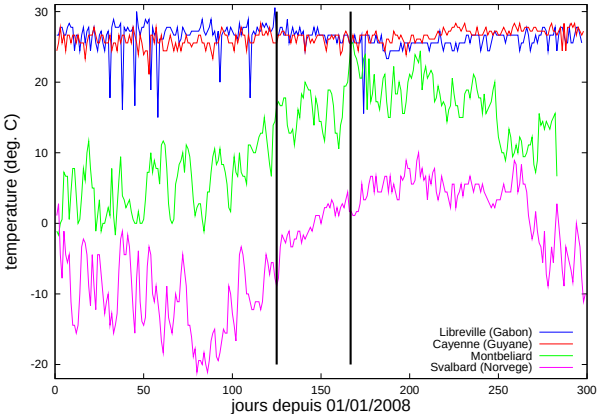
Nous avons simulé des séquences de mesures de températures en convertissant les archives quotidiennes de Weather Undergroud (http://english.wunderground.com/) en valeurs hexadécimales sur 12 bits telles que les auraient mesuré l'ADuC7026 (Fig. 9). Dans tous les cas le tableau d'entrée est rempli de 500 octets au total, soit de données seules codées sur 3 octets, soit de 3 octets suivis d'un retour chariot. Il apparaît clairement qu'en fonction des amplitudes des fluctuations annuelles de températures, le taux de compression peut être plus ou moins favorable (Fig. 9, droite). Dans les 3 exemples sélectionnés – Libreville sur l'équateur, Cayenne en Guyane, Montbéliard en France et le Svalbard dans le cercle arctique, nous avons constaté que l'efficacité de la compression décroît avec la latitude.

Ces résultats ne sont que partiellement satisfaisant car sachant que seuls 12 bits (tailles des convertisseurs analogique-numérique) sont utiles sur les 3 octets (représentation ASCII de la valeur de la température), les 3 octets pourraient être codés de façon optimale sur 3 quartets et donc n'occuper que 250 octets au total : nous constatons que l'exploitation d'un algorithme de compression général tel que LZW n'est pas nécessairement optimum dans ce cas particulier, même si ici 2 applications (Libreville et Cayenne) fournissent un meilleur résultat par LZW que par concaténation des données. Nous avons démontré par ailleurs que des algorithmes plus simples que Lempel-Ziv – par exemple RLE – sont plus appropriés pour les applications fortmenent embarquées [13].

Étant donné que le tableau de sortie de la compression n'est pas le même que celui contenant les données en entrée, nous pouvons itérer plusieurs fois la phase de compression et de décompression afin d'estimer avec précision le temps de ces calculs. Nous constatons que le temps d'exécution d'une compression de 500 octets contenant des données représentatives des problèmes qui nous intéressent se situe entre 6,8 et 8,5 ms. Sachant que chaque octet transmis à 9600 bauds prend environ 1 ms, nous pouvons aisément estimer quand la compression est avantageuse ou coûteuse en énergie. L'énergie E consommée par l'ensemble des opérations est égale à la puissance requise par chaque opération ($P_{transmission}$ et $P_{compression}$) multipliée par le temps que prend chacune de ces étapes ($t_{transmission}$ et $t_{compression}$),

$$E = P_{transmission} \times t_{transmission} + P_{compression} \times t_{compression}$$

avec $t_{transmission}$ d'autant plus petit que la compression est efficace. Notez que dans la plupart des cas, $P_{transmission} \gg P_{compression}$, donc sous réserve d'une gestion efficace de l'énergie consommée par la liaison de communication (coupure des amplificateurs radiofréquences lorsqu'ils ne sont pas utilisés), la compression sera généralement utile compte tenu des débits de communication faibles généralement rencontrés (RS232, Zigbee, Bluetooth).




```
b71
b7c
b91
b7c
b86
b71     b71b7cb91b7cb86b71b86b86b7cb86 ...
b86
b86
b7c
b86
...
```

Ville	latitude (degrés)	résultat compression avec retour	résultat compression sans retour	durée pour 1000 compressions	durée pour 1000 décompressions
'a' ×500		39 octets	26 octets	(μs)	(μs)
Libreville	$0{,}5^o$	167 octets	187 octets	6768146	367783
Cayenne	$4{,}9^o$	149 octets	155 octets	6487978	384989
Montbéliard	$47{,}5^o$	234 octets	283 octets	7600066	408378
Svalbard	79^o	264 octets	312 octets	8224089	399822

FIG. 9 – Archives de températures en 3 lieux allant de latitudes proches de l'équateur au cercle polaire arctique. Les variations de températures sont d'autant plus importantes que nous nous éloignons de l'équateur, résultant dans des taux de compression des données de températures de plus en plus mauvais. Les traits verticaux noirs indiquent quelle fraction des données a été exploitée : chaque mesure est codée sur 3 ou 4 octets (selon la présence éventuelle d'un retour chariot) et le tableau de données en entrée accepte un maximum de 500 éléments. Le temps d'exécution (en microsecondes) de 1000 compressions et décompressions est mesuré sur les fichiers avec retour chariot en fin de chaque ligne, et estimé par la fonction `gettimeofday()`

8 Exploitation de la PLA

Le cœur ARM7TDMI est exploité par de nombreux constructeurs, qui chacun l'agrémentent de périphériques répondant à leurs exigences. Dans le cas de l'ADuC7026, le périphérique le plus original, qui justifie le choix de ce processeur plutôt qu'un autre, est le *Programmable Logic Array* (PLA). Il s'agit d'un bout de logique reprogrammable de 16 blocs capables chacun d'appliquer n'importe quelle fonction logique de base, avec ou sans mémorisation sur une bascule en sortie de la porte logique, le tout cadencé avec une horloge capable de fonctionner aussi vite que le permet la PLL du processeur qui multiplie la fréquence du quartz horloger (41,78 MHz).

À notre connaissance, seul le logiciel gratuit mais propriétaire de Analog Devices permet de configurer la PLA : il s'agit d'une interface graphique chargée de convertir un dessin de la séquence de portes logiques en un code C (ou assembleur) prêt à inclure dans un programme à destination de l'ADuC7026. La table de conversion entre fonction logique et liaison entre les différents blocs est documentée dans la datasheet de l'ADuC7026 (pp.72-74 de la révision B) mais est quasiment inutilisable en l'état : nous allons présenter comment exécuter le logiciel propriétaire de Analog Devices nommé `PLATool` sous GNU/Linux (au moyen de `wine`) :

1. obtenir le logiciel gratuit de configuration de la PLA chez Analog Devices à http://www.analog.com/Analog_Root/static/technology/dataConverters/microConverter/PLATool_v2.2_setup.zip, désarchiver dans un répertoire au moyen de `unzip`
2. au moyen d'un `wine` récent (nous avons utilisé la version `wine-1.0-rc2`, installer les outils .NET 1.1 de Microsoft disponibles depuis http://msdn.microsoft.com/en-us/netframework/

`default.aspx` (obtenir le fichier `dotnetfx.exe` contenant le *Microsoft .NET Framework Version 1.1 Redistributable Package*)

3. installer ces outils par `wine dotnetfx.exe`
4. installer l'outil de Analog Devices par

   `msiexec /i PLATool_V2.2_Setup.msi`

5. compléter les librairies manquantes par `winetricks gdiplus` après avoir acquis le script `winetricks` par `wget http ://www.kegel.com/wine/winetricks`
6. l'outil Analog Devices est désormais fonctionnel (Fig. 10) : l'utilisateur peut l'exécuter par

   `wine ~/.wine/drive_c/Program\ Files/Analog\ Devices\ Inc/ADuC\ PLATool\ v2.2/PLATool.exe`

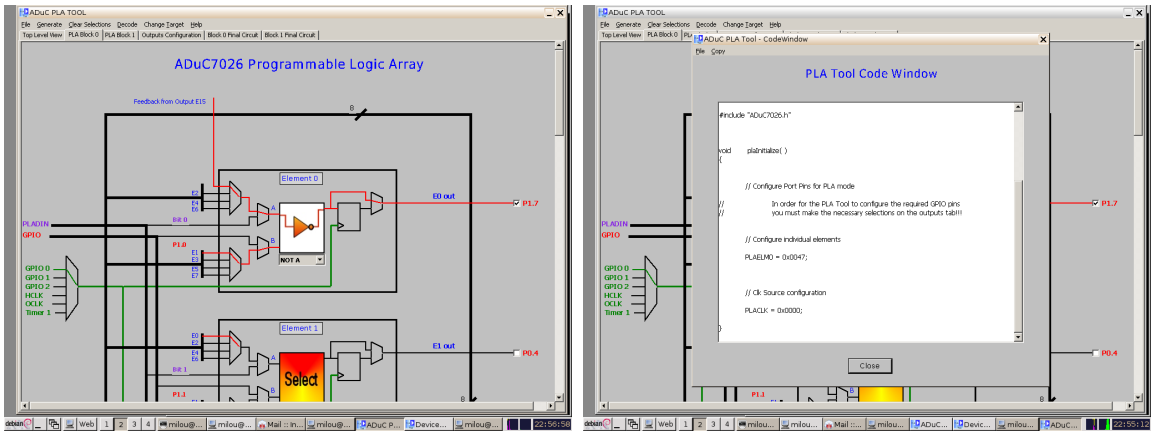


FIG. 10 – Exécution de l'outil de configuration de la PLA de l'ADuC7026 dans `wine` exécuté sous GNU/Linux. À gauche, le cheminement des signaux au travers des portes logiques se définit graphiquement. À droite, le code assembleur ou (dans ce cas) C résultant du dessin précédent.

La capacité à fournir l'horloge du processeur comme signal d'horloge des bascules de la PLA signifie que nous sommes capables de générer des signaux complexes cadencés à 41,78 MHz, ce qui serait impossible de façon logicielle puisque chaque instruction prend plus d'un cycle d'horloge. Ainsi, l'intérêt d'une matrice de portes logiques aux côtés d'un processeur généraliste est la rapidité d'exécution de fonctions logiques simples de façon synchrone ou non avec l'horloge du processeur.

9 Réalisation d'un live-CD dédié à l'enseignement

Dans le cadre des travaux pratiques d'enseignement sur le thème "Systèmes embarqués", nous avons désiré fournir aux étudiants un live-CD [14] afin qu'ils puissent conserver les outils exploités en cours. Ce live-CD, disponible avec les archives associées à cet article sur `http://jmfriedt.free.fr`, possède un environnement graphique, les compilateurs nécessaires aux développements sur systèmes embarqués de consommation réduite (MSP430 et ARM7), ainsi que les éditeurs de texte et outils de traitement des donnés (`scite` pour l'édition de texte, GNU/Octave pour le traitement de données). `wine` a été ajouté pour fournir un environnement libre de développement sur la PLA (section 8), ainsi que LaTeX qui n'a finalement pas pu être utilisé.

Pour générer ce live-CD, nous avons utilisé l'outil développé par la distribution Debian `live-helper`, décrit en détail dans [14]. Pour résumer, nous avons réalisé trois modifications principales :

- ajout de nos toolchains en les copiant dans `./config/chroot_local-includes/usr/local/bin`
- ajout d'un script permettant d'ajouter l'utilisateur aux groupes`lp` et `dialout` pour pouvoir utiliser le jtag sur port parallèle pour le MSP430, et le port série pour l'ADuC7026. Pour cela, nous créons le script dans

`./config/chroot_local-includes/usr/share/initramfs-tools/scripts/live-bottom/90group-user`
et nous le rendons exécutable avec
`chmod +x ./config/chroot_local-includes/usr/share/initramfs-tools/scripts/live-bottom/90grou`
`#!/bin/sh`
`chroot /root adduser -q user lp`
`chroot /root adduser -q user dialout`

– ajout du fichier `./config/chroot_local-includes/etc/profile` avec ajout dans ce fichier de l'exportation des variables d'environnement nécessaires
(`export PATH=$PATH :/usr/local/aduc/bin :/usr/local/cdk4msp/bin` et
`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH :/usr/local/cdk4msp/lib`)

De plus, il s'agit de ne pas oublier dans la liste de paquets à installer dans le live CD à l'aide de l'option `--packages` le méta-paquet `build-essential` pour avoir tous les outils nécessaires à la compilation.

Une subtilité que nous avons observé avec l'exploitation de live CDs comme support de travaux pratiques est l'*incapacité* d'exporter par NFS version 2 le système de fichier AUFS utilisé sur le CD. Ce problème majeur – surtout dans le cas des développement sous uClinux sur plateformes de types ARM9 ou Coldfire – est contourné par la création d'un petit (≃10 MB) loopback device [10] formaté en `ext3` qui peut lui être exporté par NFS. Nous copierons donc les fichiers à exécuter sur le système embarqué dans le répertoire de montage de ce loopback device.

10 Conclusion

Nous avons présenté une chaîne de compilation pour architecture ARM, et l'avons appliquée spécifiquement à un ARM7TDMI implémenté par Analog Devices aux cotés de divers périphériques tels que convertisseurs numériques-analogiques et analogiques-numériques rapides et une matrice de portes logiques reconfigurables. Nous avons exploité les ports de communication pour interfacer des supports de stockage de masse compatibles avec une récupération des informations sur ordinateur personnel (format FAT), écran graphique et synthétiseur de signaux radiofréquence. Nous avons rapidement survolé quelques limites de la toolchain libre (implémentation lourde de stdlib, problème de gestion des interruptions) et estimé la puissance de calculs requise par quelques algorithmes moins triviaux de compression de données ou de recherche de racines de fonctions complexes.

Ces outils fournissent, avec la famille des ARM7, une série de processeurs puissants répondant probablement aux besoins de nombreux utilisateurs cherchant à automatiser des tâches répétitives en répondant aux exigences de dimensions réduites, consommation modeste et puissance de calcul importante.

11 Remerciements

Nous remercion Jonathan Bennès (CSEM, Neuchatel, Suisse) pour avoir acquis une carte d'évaluation d'ADuC7026 et nous avoir démontré la puissance de ce processeur pour les applications embarquées. Analog Devices a gracieusement fourni un certain nombre d'échantillons pour réaliser les cartes de développements exploitées dans cette présentation. Olivier Sterenberg a fourni le point de départ de la plupart des routines fournies comme exemples dans ce document en développant le code pour un CanSat (http://www.planete-sciences.org/espace/spip.php?rubrique24), projet dirigé dans la cadre de Planète Sciences par Emmanuel Jolly.

[10] l'espace est créé par `dd if=/dev/zero of=/home/espace.dd bs=512 count=2000` pour un espace de 10 MB, puis formaté par `mkfs -t ext3 /home/espace.dd`. Finalement, cet espace de travail est accessible dans le répertoire `/home/espace` par `mount -o loop /home/espace.dd /home/espace` en supposant que le répertoire `/home/espace` existe. L'ensemble de ces tâches est exécuté automatiquement au boot comme décrit dans [14].

Références

[1] http://en.wikipedia.org/wiki/ARM_architecture propose un résumé des divers cœurs développés par ARM dans un contexte historique, ainsi que de nombreuses références

[2] S. Segars, K. Clarke & L. Goudge, *Embedded Control Problems, Thumb and the ARM7TDMI*, IEEE Micro **15** (5) (1995), pp. 22-30, disponible à http://www.cooper.edu/~sable2/courses/spring2004/ee453/docs/embedded_ctrl_prblm.pdf, et la description commerciale du cœur à http://www.arm.com/products/CPUs/ARM7TDMI.html

[3] T. Martin, *The insider's guide to the Philips ARM7-based microcontrollers*, Hitex (2005)

[4] D. Bodor, *Passez votre iPod sous uClinux*, GNU/Linux Magazine France n.80, Février 2006, pp. 92-98

[5] http://mindstorms.lego.com/Overview/The_NXT.aspx ou, plus sérieusement, http://bricxcc.sourceforge.net/nbc/

[6] http://martin.hinner.info/ARM-Microcontroller-HOWTO/ARM-Microcontroller-HOWTO.htm

[7] http://www.gnuarm.com/ArmDevices_frame.html

[8] E. Bernard, *Linux sur ARM : introduction, compilation et configuration*, GNU/Linux Magazine France, Hors Série **25** (Avril/Mai 2006), pp.20-33

[9] Embedded Systems Academy, *ARM7 Performance : Comparisons & Benchmarks*, transparents disponibles à www.esacademy.com (Mars 2007)

[10] quelque discussions concernant les benchmarks de divers compilateurs pour ARM7TDMI : S. Gomes Augusto & L. Orry, *C Compilers for ARM : Benchmark*, Raisonance (2006) disponible à http://www.mcu-raisonance.com/~str7-arm7tdmi-stmicroelectronics__microcontrollers__support~sfp__T016:4cnyx6krfmzf.html, et http://www.compuphase.com/dhrystone.htm

[11] S. Guinot, J.-M Friedt, *Stockage de masse non-volatile : un block device pour MultiMediaCard*, GNU/Linux Magazine France, Hors Série 25 (Avril 2006), disponible à http://jmfriedt.free.fr/lm_hs25.pdf

[12] C. Sadler and M. Martonosi, *Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks*, Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys), Novembre 2006, disponible à http://parapet.ee.princeton.edu/papers/csadler-sensys2006.pdf

[13] E. Pamba Capo-Chichi, H. Guyennet, J.-M Friedt, *K-RLE : A new Data Compression Algorithm for Wireless Sensor Network*, SENSORCOMM 2009, Athènes/Vouliagmeni, Grèce (Juin 2009)

[14] D. Bodor, *Créez votre live CD Debian 5.0 Lenny*, GNU/Linux Magazine France **115**, Avril 2009, pp.28-32

J.-M Friedt est membre de l'association Projet Aurore qui vise à la diffusion de la culture scientifique et technique sur le campus bisontin de l'université de Franche-Comté. Il utilise l'ADuC7026 dans le cadre de son activité professionnelle de développement d'instruments dans la société SENSeOR. Il a participé à un enseignement "systèmes embarqués" pour l'École Nationale Supérieure de Mécanique et Microtechiques.

É. Carry est président de l'association Sequanux pour la promotion du logiciel libre en Franche Comté. Maître de conférence à l'université de Franche Comté à Besançon, il participe à l'enseignement sur des thèmes liés à l'électronique numérique et systèmes embarqués.