Interfaces matérielles et OS libres pour Nintendo DS : DSLinux et RTEMS

J.-M Friedt, G. Goavec-Merou
23 mai 2009

> A good lesson when you fly COTS stuff - make sure you know how it works.
> Glenn Reeves, Mars Pathfinder Flight Software Cognizant Engineer

Alors que les protocoles de communication parallèles (port imprimante, bus ISA, bus IDE, bus des processeurs) tendent à disparaître au profit des modes de communication série (USB, SATA, I^2C ou SPI) – moins encombrants et plus rapides mais plus complexes à appréhender – il devient de plus en plus difficile, à l'amateur désireux d'expérimenter, de trouver des plateformes sur lesquelles découvrir le fonctionnement d'un processeur. La console de jeu portable de Nintendo Dual Screen (DS) est encore suffisamment ancienne tout en ayant eut assez de succès pour être encore largement disponible pour fournir une plateforme d'expérimentation intéressante. Chaque génération de console portable Nintendo conserve une compatibilité avec sa prédécesseur, et ainsi la DS fournit un port de communication compatible avec le Gameboy Advance. Ce port se caractérise par l'accès à un bus 16 bits similaire à celui qui équipait le microprocesseur 8086 [1, p.156], avec accès à une interruption, aux bus de données et d'adresse, ainsi qu'aux signaux de contrôle associés.
Nous proposons ici d'exploiter la Nintendo DS en exécutant un système GNU/Linux pour se familiariser avec quelques méthodes d'instrumentation, d'interfaçage et de contrôle de périphériques. Ayant constaté que les ressources requises par uClinux sont excessives devant celles mises à disposition par la console, nous nous tournerons vers un environnement de développement d'applications à faible empreinte mémoire, RTEMS. Nous verrons que la console de jeu, malgré ses ressources réduites, propose un environnement de développement propice aux découvertes tant du point de vue matériel que logiciel, avec au final sa conversion en système d'acquisitions de données et de transmission par wifi.

1 Matériel disponible

La console de jeu Nintendo DS – et son évolution plus récente DS Lite que nous utiliserons ici – est un système embarqué contenant deux processeurs ARM : un ARM9 (67 MHz) comme processeur principal et un ARM7 (33 MHz) [2] comme co-processeur chargé de la gestion des périphériques tels que son, graphisme 2D, écran tactile. Deux ports permettent de charger des exécutables pour les processeurs : le slot1 supporte des cartouches exclusivement dédiées à la DS, avec protocole de communication série en partie crypté, tandis que le slot2 est compatible avec les cartouches de GameBoy Advance, donnant accès à un bus de communication parallèle très bien documenté. Les 4 MB de RAM et les deux écrans de 256×192 pixel rendent le développement plus agréable et ludique qu'une carte nue dont les seuls ports de communication sont ethernet ou série. Éviter la version la plus récente de la console – DSi – qui a perdu le slot2 au profit d'une quantité de RAM plus importante.

Notre objectif est d'exploiter divers outils libres disponibles pour ces processeurs bien connus des développeurs de systèmes embarqués – et notamment la version pour ARM de `gcc` – pour exploiter au mieux cette console. La quantité de mémoire proposée est suffisante pour supporter

un système d'exploitation : nous allons donc développer des applications au-dessus d'un environnement qui nous protège des couches les plus basses entre logiciel et matériel, pour n'utiliser que des méthodes génériques, exploitables dans d'autres contextes que sur la console de jeu. En commençant par le port pour Nintendo DS de uClinux, nommé DSLinux, nous nous accrochons le plus longtemps possible à l'environnement de travail des lecteurs de ce journal, avant de devoir l'abandonner faute de ressources lorsque nous voudrons utiliser des périphériques plus gourmands tels que le wifi.

2 DSLinux : toolchain et fonctionnalités

L'exécution de nos propres logiciels, et en particulier de la distribution d'uClinux dédiée à la Nintendo DS nommée DSLinux [1], nécessite une cartouche permettant de contourner certaines protections censées interdire l'utilisation de logiciels autres que ceux validés par Nintendo. En effet, les jeux sont transférés de la cartouche dans le Slot 1 selon un protocole synchrone série dans un format crypté lors de l'allumage de la console. L'exécution de logiciels libres sur cette console nécessite de fournir de telles données cryptées : ces fonctionnalités sont par exemple fournies par la cartouche M3DS Real [2]. Nous avons utilisé cet outil, acquis pour environ 25 euros auprès de `www.consoleup.com`. Cette cartouche fournit un medium de développement confortable puisqu'il accepte des cartes mémoires au format MicroSD (1 GB pour environ 5 euros), facilement lisible sur PC, avec son formatage original VFAT. Un ensemble de logiciels [3] dédiés doit être désarchivé dans la racine de cette carte pour permettre le chargement des applications stockées dans le répertoire `nds`. Nous compilerons donc nos logiciels au moyen d'une toolchain de crosscompilation sur PC, transférerons le logiciel sur carte MicroSD en vue de son exécution sur la console de jeu.

Le site web de DSLinux propose une toolchain au format binaire [4] : exceptionnellement, nous allons nous autoriser à ne pas recompiler à la main nos outils mais allons nous contenter de décompresser et désarchiver ces outils dans un répertoire dédié que nous ajouterons à notre `$PATH`.

Une fois la toolchain disponible, il nous faut installer l'ensemble de l'arborescence de DSLinux : même si nous ne désirons pas aborder le développement de modules noyau (qui nécessitent les sources du noyau pour compiler), l'arborescence de DSLinux contient quelques utilitaires nécessaires à la compilation de programmes en espace utilisateur. Là encore, nous nous contentons de décompresser et désarchiver le fichier fourni à `http://stsp.spline.de/dslinux/dslinux-snapshot.tar.gz`.

Un clavier virtuel dessiné sur l'écran tactile permet de facilement interagir avec ses programmes en mode console, avec notamment la tabulation qui rend l'interaction avec le shell efficace. Les touches en forme de croix à gauche de la console font office de flèches.

Dans un premier temps, nous allons développer quelques petits programmes en espace utilisateur : après avoir inclus `dslinux-toolchain-2008-01-24-i686/bin/` dans son `$PATH`, nous entrons dans la racine de l'arborescence DSLinux et la commande

```
make xsh
```

initialise les variables nécessaires à la compilation de programmes. Ainsi, pour compiler un programme nommé `rumble.c`, nous utilisons `$CC $CFLAGS $LDFLAGS rumble.c -o rumble`. Le résultat de la compilation est, comme toujours sous uClinux, un binaire au format Binary Flat (BFLT). Noter que pour pouvoir exécuter `make xsh` avec succès, il *faut* avoir effectué un `make menuconfig`, dont nous accepterons toutes les options par défaut, au moins une fois afin de générer les fichiers de configuration nécessaires à la mise en place de l'environnement de cross-compilation.

Avant de recompiler son propre noyau ciselé à ses besoins, nous nous contentons d'exploiter une archive linux pré-compilée et fonctionnelle, disponible à `http://kineox.free.fr/DS/dslinux-dldi.tgz`. L'arborescence du système est placée à la racine de la carte MicroSD tandis que les deux fichiers aux extensions `.nds` vont dans le répertoire `nds` (un des fichiers est un

[1] `http://dslinux.org`
[2] `www.m3adapter.com`
[3] `http://www.megaupload.com/?d=J3T9UVPE`
[4] `http://stsp.spline.de/dslinux/toolchain`, version 2008-01-24 à la date de rédaction de cet article

noyau avec support de la communication avec la carte SD – DLDI (Dynamically Linked Device Interface) – et l'autre fichier est un noyau supportant en plus les extensions mémoire. En l'absence d'un tel périphérique, le lecteur n'utilisera que le logiciel `dslinux.nds`). Ainsi, GNU/Linux apparaît comme un jeu pour la console, dont l'arborescence est placée sur la carte SD pour une modification aisée : les fichiers fondamentaux au démarrage de GNU/Linux sont dans le `rootfs` avec notamment un répertoire `bin` contenant les outils nécessaires à l'initialisation du système, tandis que tous les fichiers de configuration, utilisateur etc ... se trouvent dans le répertoire `linux` en haut de l'arborescence sur la carte MicroSD, monté dans `/media` au démarrage.

```
jmfriedt@ns39351:~/dslinux/romfs$ ls -l | cut -c 53-100
 bin
 boot
 dev
 etc -> media/linux/etc
 home -> media/linux/home
 lib -> media/linux/lib
 media
 opt
 proc
 sbin
 tmp
 usr -> media/linux/usr
 var -> media/linux/var
```

La compilation manuelle d'une image s'obtient par le classique `make menuconfig` suivi de `make` : le résultat est d'une part une arborescence disponible dans le sous répertoire `rootfs`, et d'autre part l'archive compressée `images/dslinux-dldi.tgz` qui contient le "jeu" `dslinux.nds` pour lancer uClinux sur la DS, et l'arborescence `linux` que nous placerons dans le répertoire racine de la carte MicroSD. Mentionnons dès maintnenant que l'ajout de fonctionnalités se fait dans `vendors/Nintendo/DLDI` : par exemple l'ajout d'une entrée dans `/dev` (major 32, minor 0) se fait dans `vendors/Nintendo/DLDI/Makefile` en complétant la liste des `DEVICES` par `skeleton,c,32,0 \`, ou l'accès automatique à des systèmes de fichiers dans `inittab`. Toute modification dans un autre emplacement de l'arborescence `romfs` sera perdue à la prochaine compilation.

Ayant obtenu un système d'exploitation fonctionnel sur DS, nous allons proposer d'en exploiter les fonctionnalités accessibles compte tenu de la quantité réduite de mémoire disponible (4 MB initialement, quelques centaines de KB après chargement du système) : notre objectif va être d'interagir avec le monde extérieur en commandant des systèmes numériques (LEDs, selon des méthodes appliquables à toute commande binaire) et en acquérant une valeur analogique. Comme DSLinux est basé sur uClinux, le processeur ARM9 de la DS n'étant pas équipé de gestionnaire de mémoire, le contrôle de ces périphériques pourra se faire soit depuis l'espace utilisateur (comme nous le ferions sur un microcontrôleur monotâche sans système d'exploitation), soit depuis l'espace noyau pour tenir compte des couches d'abstraction isolant le développeur sous GNU/Linux des couches matérielles.

3 Hello World

Nous avons vu qu'un programme se compile en passant dans l'environnement de cross-compilation proposé dans la toolchain dslinux : `make xsh` suivi, pour la compilation elle-même, de `$CC $CFLAGS $LDFLAGS rumble.c -o rumble` dans le répertoire de développement. Nous pouvons ainsi valider notre capacité à générer un binaire au bon format pour être exécuté sous DSLinux depuis un emplacement de l'arborescence contenue dans la MicroSD, de communiquer avec l'utilisateur en mode texte et d'effectuer des opérations simples sur les entiers et les réels (code Tab. 1).

Notons que ces fonctionnalités simples qui ne font pas accès au matériel peuvent rapidement être validées sur l'émulateur `desmume` (Fig. 1), disponible comme paquet Debian au moins avec la version Lenny. Il semblerait que les versions plus récentes incluses dans Squeeze et Sid corrigent un

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

int main(int argc,char **argv)
{printf("demo rumble : 1/3=%f\n",1./3.);
 if (argc>1) {
   *(unsigned short*)(0x8000000)=(unsigned short)atoi(argv[1]);
   sleep(1);                          // active le moteur sur argv[1]=2
   *(unsigned short*)(0x8000000)=0;
 }
 return(0);
}
```

Tab. 1 – Programme de démonstration d'un affichage en mode console et l'accès aux périphériques du Slot 2 depuis l'espace utilisateur (section 5). Si le rumble pack est inséré dans ce slot, le moteur s'activera pendant une seconde si l'argument est un nombre pair.

certain nombre d'erreurs. Cet émulateur permet soit de valider un algorithme sur PC en évitant de copier son programme sur MicroSD et booter uClinux à chaque nouvel essai, soit au lecteur ne possédant pas (encore) de console de jeu de tester quelques unes des idées présentées dans ce document. Par ailleurs, une fonctionnalité intéressante de l'émulateur est la capacité à afficher l'état de certains registres et, pour la version MS-Windows, de désassembler le contenu de la mémoire, en faisant donc un outil de debuggage relativement pratique.

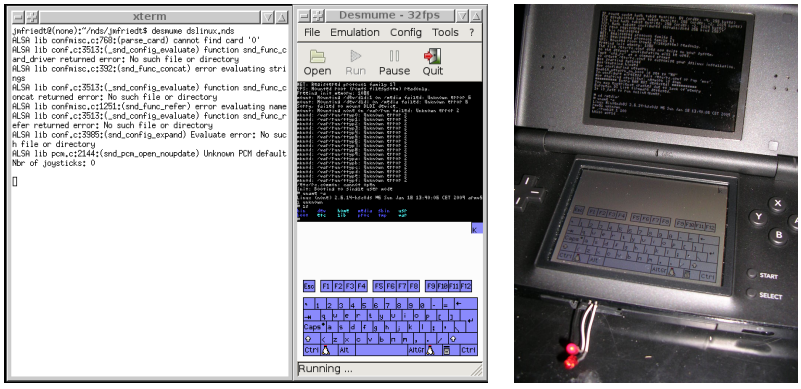


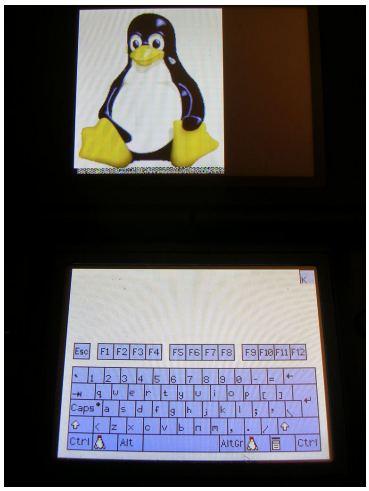
Fig. 1 – Gauche : DSLinux fonctionne parfaitement sur l'émulateur `desmume`, facilitant ainsi le développement sur PC avant l'exécution sur la console de jeu elle-même. Droite : résultat similaire sur la console, déjà équipée de LEDs (section 5).

4 Accès au *framebuffer* et programmation sous GNU/Linux

Notre objectif étant de tracer des courbes présentant des points expérimentaux acquis selon les méthodes présentées par la suite (section 5.2), nous désirons maîtriser une méthode d'affichage de graphiques sur les écrans de la console. Nos 4 MB de mémoire, en grande partie occupés par le système d'exploitation, sont largement insuffisant pour faire tourner une application compilée avec Qtopia, et même la bibliothèque SDL semble trop gourmande pour fonctionner avec les ressources

disponibles. Il ne reste donc que la solution la plus efficace, la plus rapide et requérant le moins de mémoire volatile : l'accès au *framebuffer* [3].

Afin de nous familiariser avec la compilation de programmes pour DSLinux, et en particulier pour accéder à ce périphérique qui n'est autre qu'un segment de la mémoire représentant la couleur de chaque pixel à l'écran, nous nous fixons pour objectif d'afficher une image couleur au format PNM (Fig. 2).



```
struct fb_var_screeninfo sinfo;
unsigned short * s_ptr;

inline void draw_pixel(int x, int y, int color)
{unsigned short *loc = s_ptr + \
    ((y+sinfo.yoffset)*sinfo.xres)+x+sinfo.xoffset;
 *loc = color;     // 5R, 5G, 5B
 *loc |= 1 << 15; // transparence ?
}

int main(int argc, char *argv[])
{char c;
 screen_fd = open("/dev/fb0", O_RDWR);
 ioctl(screen_fd, FBIOGET_VSCREENINFO, &sinfo);

 s_ptr = mmap(0,screen_height*screen_width/8,PROT_READ→
     ↪|PROT_WRITE,MAP_SHARED,screen_fd, 0);
 [...]
}
```

FIG. 2 – Exemple d'affichage sur l'écran du haut accessible par `/dev/fb0` : l'affichage d'une image en couleur démontre la bonne interprétation de l'organisation de la mémoire et de l'affectation des couleurs au sein de chaque mot.

Le programme est très classique, avec ouverture du périphérique (`open("/dev/fb0", O_RDWR);`), récupération des paramètres de l'écran par `ioctl()`, pour finalement modifier la couleur des pixels composant l'image selon les consignes fournies dans le fichier PNM. La principale subtilité a été d'identifier l'encodage des couleurs dans un *framebuffer* configuré pour définir la couleur de chaque pixel sur 16 bits : les couleurs sont chacune codée sur 5 bits, avec le bit de poids le plus fort systématiquement à 1. Cela signifie que nous masquons les 3 octets représentant les 3 couleurs lues dans le fichier PNM pour générer un mot de 16 bits contenant la concaténation de 3 couleurs sur 5 bits chacune. Ainsi, nous utilisons

```
for (y=0;y<sy;y++)
  for (x=0;x<sx;x++) {
    fscanf(f,"%c",&r); fscanf(f,"%c",&g); fscanf(f,"%c",&b);
    bpp=(((int)(r&0xf8))>>3)+(((int)(g&0xf8))<<2)+(((int)(b&0xf8))<<8);
    draw_pixel(x, y, bpp);
   }
```

pour tracer les `sx` × `sy` pixels qui forment l'image. La fonction `draw_pixel` et le résultat sont tous deux proposés sur la Fig. 2. Insistons sur l'intérêt du système d'exploitation : aucune connaissance sur l'architecture matérielle ou l'emplacement de la mémoire vidéo de la NDS n'est nécessaire pour cet exemple, qui est par ailleurs directement portable sur un PC.

Noter qu'avant d'exécuter un tel programme qui nécessite un écran dont la couleur de chaque pixel est définie sur 16 bits, on pensera à lancer depuis le shell `fbset -depth 16 -n`

5 Interfaçage de périphériques : la cartouche du Slot 2 (Gameboy Advance)

Bien que la Nintendo DS propose une architecture considérablement moins complexe (et fournissant moins de puissance de calcul) que ses concurrentes, la compatibilité avec la GameBoy Advance, plus ancienne, est l'opportunité d'exploiter toute la documentation associée aux protocoles

de communication de cet ancêtre de la DS. Nous trouverons notamment facilement sur le web les plans du bus de communication parallèle accessible sur la cartouche (`www.reinerziegler.de/GBA`) qui va s'avérer un environnement de prototypage idéal, digne des bus ISA et autres ports parallèles malheureusement disparus de nos PCs modernes.

Nous pourrons dans un premier temps nous familiariser avec la cartouche du Rumble Pack : ce périphérique, commercialisé en même temps que la cartouche de programmation citée plus haut, est aussi simple d'un point de vue électronique qu'inutile d'un point de vue ludique. Nous n'aurons donc aucun remord à la sacrifier pour nos expérimentations.

Nous allons dans un premier temps identifier la plage d'adresses mémoire permettant l'accès au bus du Slot 2 de la DS, que nous démontrerons en activant et désactivant le moteur du Rumble Pack. Ce moteur sera ensuite retiré pour être remplacé par des périphériques plus intéressant.

5.1 Accès à une plage mémoire en sortie

Avant d'attaquer au fer à souder la cartouche Rumble Pack, nous pouvons déjà valider quelques informations sur la plage d'adresses permettant l'accès aux périphériques sur Slot 2 en activant et désactivant le moteur. L'observation (Fig. 3) de la cartouche [5] indique que seuls les signaux `WR#` et `AD1` sont cablés : l'activation se fait donc soit lors de l'écriture sur une adresse paire, soit en écrivant une valeur paire dans la plage d'adresses de communication avec la cartouche [6]. Le fait que `WR#` (actif lors de la phase de communication des données) et non `CS#` (actif pendant les phases de communication des adresses et des données, avec une transition d'état sur une adresse valide) soit utilisé laisse penser que la seconde solution est la bonne. En effet, les broches AD0 à AD15 fournissent deux fonctions : dans un premier temps le mot de poids faible de l'adresse (lorsque CS# est bas mais WR# encore haut), suivi dans un second temps de la donnée transmise (sur 16 bits, lorsque CS# et WR# sont tous deux bas). Le première phase est valide sur le front descendant de `CS#`, tandis que la seconde phase est active avec une donnée valide sur le front montant de `WR#`. Par conséquent, l'instruction `*(unsigned short*)0x8000000=2` active le moteur, et `*(unsigned short*)0x8000000=0` l'arrête (code Tab. 1).

Les signaux accessibles sur le connecteur du Slot 2 n'appartiennent pas à proprement parler à un bus même s'ils en fournissent ponctuellement les fonctions : le décodage d'adresse se fait en amont puisque par exemple le signal `WR#` ne s'active *que* lors de l'accès à la plage d'accès de la carte.

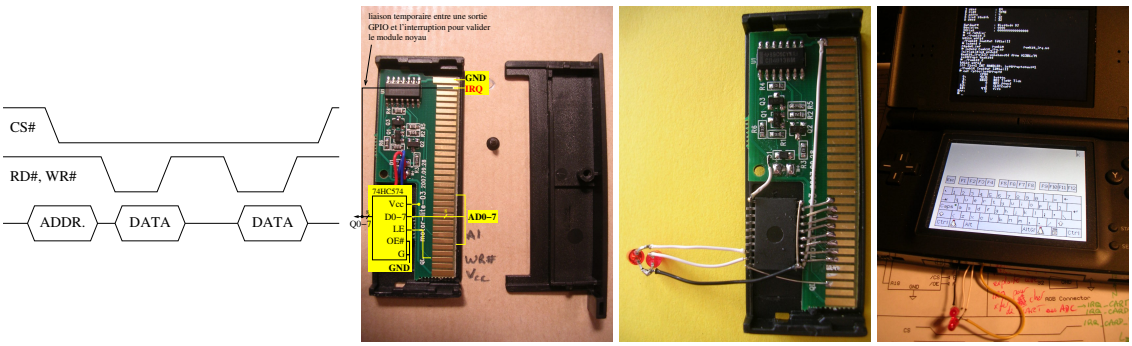


FIG. 3 – Chronogramme des signaux disponibles sur le bus du Slot 2 de la DS : un signal d'activation de la carte Chip Select, des signaux de contrôle indiquant la direction de la transaction (`RD#` et `WR#`) et 16 bits d'adresse dont le mot de poids faible est multiplexé avec le bus de données. Un circuit exploitant l'ensemble de la plage d'adresses doit donc mémoriser (*latch*) la valeur du mot de poids faible du bus d'adresses sur le front descendant de `CS#` avant d'exploiter les données fournies sur ces mêmes broches sur le front montant de `RD#` ou `WR#`.

[5] `http://nocash.emubase.de/gbatek.htm#dscartrumblepak`
[6] `http://nocash.emubase.de/gbatek.htm#dsmemorymaps`

L'ajout ou le remplacement de fonctionnalités dans la cartouche du Rumble Pack peut se faire de deux façons : tirer un câble en nappe en dehors de la cartouche pour prototyper sans contrainte d'encombrement, ou respecter la facteur de forme de la cartouche. La première solution semble trop fragile à l'usage : nous nous sommes efforcés de placer tous les composants dans la cartouche, ce qui impose d'exploiter des composants de petites dimensions prévus pour être montés en surface (CMS, Fig. 3 au milieu). L'utilisation de tels composants n'est pas difficile, mais nécessite de la la soudure fine, de la tresse à dessouder pour corriger ses erreurs, et surtout une binoculaire pour travailler avec un grossissement suffisant.

Cette méthode d'accès direct aux bus de communication entre le processeur et le matériel ne respecte cependant pas les principes d'abstraction mis en place par le système d'exploitation : nous verrons plus bas comment obtenir les mêmes fonctionnalités au travers d'un module noyau accessible *via* son entrée dans `/dev` : nous aborderons ce point plus bas avec l'écriture d'un module noyau.

5.2 Lecture d'une conversion analogique-numérique

Le bus de données est bidirectionnel, la direction de communication étant déterminée par les signaux de contrôle `WR#` (écriture) ou `RD#` (lecture). Un composant susceptible de fournir des informations au processeur *via* son bus de données doit présenter un état au repos en haute impédance dans lequel il n'impose pas l'état du bus (état généralement noté Z dans la description des composants), pour ne passer en basse impédance et imposer l'état du bus que sur ordre du processeur (transition au niveau bas du signal de contrôle `RD#`). Toute erreur de câblage en ce sens se traduira pas des courts circuits potentiels entre composants imposant simultanément leur niveau sur le bus, avec potentiellement destruction des composants les moins bien protégés contre la consommation excessive de courant.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define TAILLE 255

int main(int argc,char **argv)
{int f,taille=TAILLE;
 volatile int k;
 char *c;

 c=(char*)malloc(TAILLE); // demontre malloc en l'absence de MMU
 for (f=0;f<TAILLE;f++)
   {*(unsigned short*)(0x8000000)=(unsigned short)0;
    for (k=0;k<10;k++) {} // NE PAS compiler en -O2
    // usleep(7);         // l'appel a usleep est trop long !
    c[f]=*(unsigned short*)(0x8000000)&0xff;
   }
 for (f=0;f<TAILLE;f++) printf("%x ",c[f]);printf("\n");
 return(0);
}
```

Tab. 2 – Lecture d'une valeur fournie par un périphérique connecté au bus du Slot 2. Dans le cas qui nous intéresse ici de la conversion analogique-numérique, la conversion est amorcée par une écriture tandis que la lecture se fait après un temps prédéfini obtenu par une boucle vide pour une fréquence d'échantillonage maximale.

Nous allons exploiter un composant de conversion analogique-numérique, AD7492, comme exemple de système capable de fournir une information au processeur (Fig. 4). Le convertisseur

est un cas un peu plus complexe que la moyenne car il nécessite dans un premier temps un ordre en écriture pour annoncer le début de la conversion (impulsion au niveau bas sur la broche CONVST# du convertisseur), et ce n'est qu'une fois la conversion achevée que le composant est prêt à fournir le résultat de sa mesure (données sur 12 bits placés sur D0 à D11 lorsque les signaux d'activation des sorties CS# et RD# passent à l'état bas). Dans un premier temps nous nous contenterons d'attendre un délai prédéfini entre début et fin de conversion, pour aborder plus tard le cas du message (interruption) annonçant la fin de conversion (associée à une transition du signal BUSY).

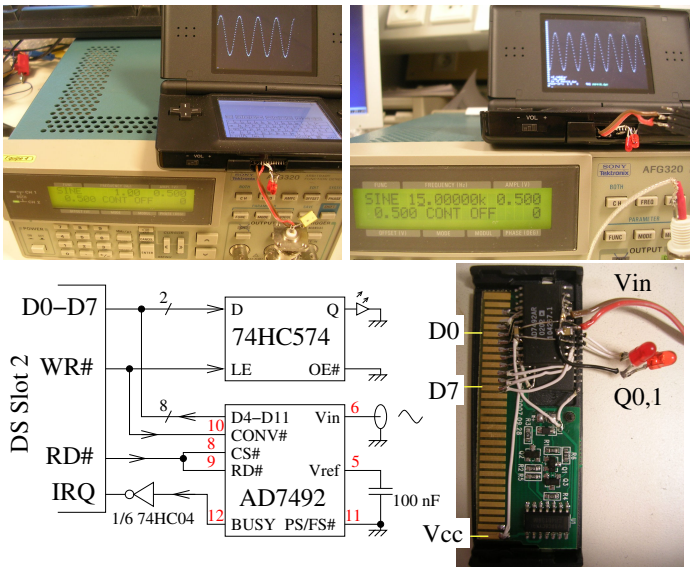


FIG. 4 – En haut : un synthétiseur de fréquences génére un signal sinusoïdal périodique d'amplitude 1 V et d'offset 0,5 V à des fréquences de 1 Hz (gauche, représentative d'une mesure d'une grandeur telle qu'une température) et 15 kHz (droite, pour évaluer la fréquence maximale d'échantillonage) en vue de la conversion par le circuit décrit en bas à gauche, connecté sur une cartouche de Rumble Pack (en bas à droite) insérée dans le Slot 2 de la console. Le programme de gestion de la conversion (Tab. 2) et d'affichage sur framebuffer tourne sous DSLinux. La soudure des composants CMS n'est pas difficile mais nécessite une loupe ou une binoculaire.

Une amélioration possible du concept est d'exploiter pleinement les capacités du bus de données de 16 bits de large pour câbler tous les signaux du convertisseur analogique-numérique 12 bits. Les modifications du matériel sont significatives (4 fils de plus, fig. 5) tandis que du point de vue logiciel, nous nous contentons de remplacer le type du tableau dans lequel les informations sont stockées de `unsigned char*` à `unsigned short*` qui comprendra les 12 bits de données significatives (code Tab. 2).

La vitesse de conversion peut se caractériser de plusieurs façons : nous avons pour notre part exploité la transformée de Fourier du signal numérisé afin d'établir la fréquence d'échantillonnage f_e. En effet, la transformée de Fourier (fonction `fft` de GNU/Octave) s'étale de $-f_e/2$ à $f_e/2$ en N points (par défaut, N est le nombre de points acquis). Connaissant la fréquence f du signal observé comme un maximum de puissance au point n, la fréquence d'échantillonnage est $f_e = \frac{N}{n} \times f$. L'incertitude sur la fréquence de conversion Δf_e, en supposant la fréquence mesurée parfaitement connue, est $\Delta f_e = \frac{f_e}{n}$: plus f est élevée, plus f_e est identifiée avec précision. Alternativement, le calcul dans le domaine temporel donne le même résultat : sur la conversion en 8 bits (bleue) de la Fig. 5, nous constatons qu'une période du signal à 5 kHz est échantillonné en 23 points, soit un échantillonnage à $f_e = 23 \times 5 = 115 \pm 5$ kHz (avec un délai de 10 itérations d'une boucle vide entre le début et la fin de conversion). La vitesse maximum théorique du convertisseur AD7492 est de 1 Méchantillons/seconde : un ajustement plus fin du délai devrait permettre d'atteindre de telles

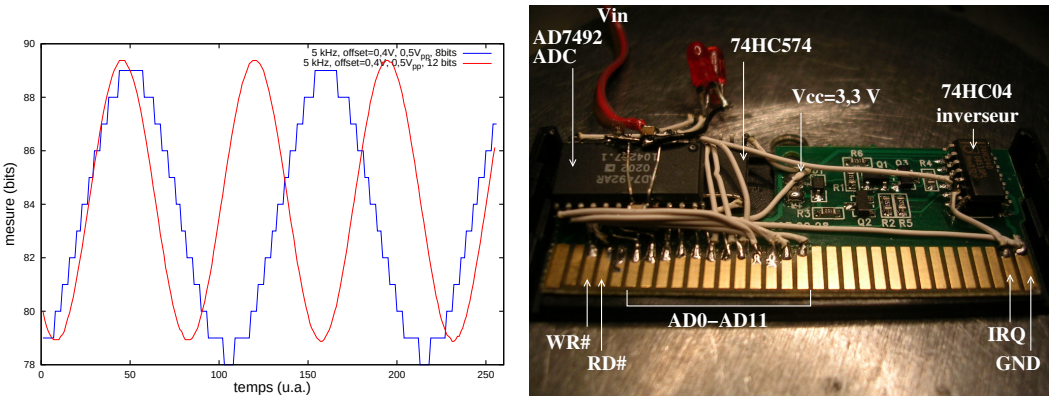


FIG. 5 – À gauche : comparaison entre la conversion analogique-numérique sur 8 bits (boucle vide de 10 itérations entre le début de conversion et lecture du résultat) et 12 bits (boucle vide de 5 itérations entre le début de conversion et lecture du résultat, résultat divisé par 16 pour fournir la même échelle). Dans les deux cas, le signal en entrée est une sinusoïde à 5 kHz, amplitude 0,05 V et offset 0,4 V. Le gain en résolution se fait néanmoins au détriment d'une carte plus encombrée (droite). Sur cette version de la cartouche, la broche BUSY du convertisseur est associée à l'interruption IRQ au travers d'un inverseur 74H04 monté au dessus de la bascule D (4013) fournie d'origine dans le Rumble Pack.

performances, mais ne présente que peu d'intérêt pour cette démonstration de fonctionnement.

5.3 Interruption

Il est habituellement admis que l'accès direct au matériel depuis l'espace utilisateur sur un système d'exploitation multitâches n'est pas une bonne façon de procéder. En effet, nous ne profitons pas des couches d'abstractions qui rendent le système portable, et nous somme sujets aux conflits d'accès si plusieurs programmes décident d'accéder au même périphérique. Par ailleurs, sur les plateformes proposant une MMU, les accès aux ports et à la mémoire sont en général interdits. Nous allons donc proposer une implémentation "propre" de l'accès aux données du convertisseur analogique-numérique, notamment pour profiter de la possibilité d'être informé par interruption de la fin de conversion. Théoriquement, nous devrions par cette méthode atteindre la vitesse de conversion la plus élevée possible : nous lançons une conversions en activant CONVST# en écrivant dans la plage d'adresses de la cartouche, nous vaquons à nos occupations jusqu'à être prévenu de la fin de conversion par interruption. À ce moment nous lisons le résultat que nous plaçons dans un tampon et relançons immédiatement une nouvelle conversion. Nous allons voir que ce beau scénario idéal va rencontrer quelques difficultés ...

Le module noyau donnant accès au périphérique de type caractère a été largement décrit dans ces pages et nous ne reviendrons pas sur les généralités. Nous partons de l'exemple de module avec interruption sur port parallèle de PC fourni à `http://www.captain.at/howto-linux-device-driver-template-sk` `php` (qui contient quelques erreurs, notamment sur la syntaxe des arguments à `request_irq()`).

Dans notre cas, nous n'avons besoin que d'implémenter les méthodes `open`, `read`, et `close` du module. Par ailleurs, nous exploiterons un `ioctl` pour déclencher la conversion et la mise en mémoire du résultat de la conversion afin de séparer la phase lecture des données (restitution des données acquises vers l'espace utilisateur dans la méthode `read`) et acquisition.

La structure de données contenant les méthodes implémentées se présente donc de la forme

```
// define which file operations are supported
struct file_operations skeleton_fops = {
        .owner   =       THIS_MODULE,
        .llseek =       NULL,
        .read    =       skeleton_read,
```

```
        .write  =       NULL,
        .readdir=       NULL,
        .poll   =       NULL,
        .ioctl  =       skeleton_ioctl,
        .mmap   =       NULL,
        .open   =       skeleton_open,
        .flush  =       NULL,
        .release=       skeleton_release,
        .fsync  =       NULL,
        .fasync =       NULL,
        .lock   =       NULL,
        .readv  =       NULL,
        .writev =       NULL,
};
```

Une broche nommée `IRQ` est disponible sur le connecteur du Slot 2 : la lecture de `linux/include/asm-asmnommu/arch-nds/irqs.h` nous indique qu'il s'agit de `IRQ_CART`, interruption numéro 13i (Fig. 6).

Les méthodes `open` et `release` ne présentent aucun intérêt : elles ne font qu'afficher un message validant l'opération. La méthode d'initialisation du module enregistre la communication entre espaces noyau et utilisateur au travers de `/dev/skeleton` de major number 32 et minor 0 :

```
int i = register_chrdev (32, "skeleton", &skeleton_fops);
```

ainsi que la gestion de l'interruption numéro 13 déclenchée par un front montant sur la broche IRQ du Slot 2 de la DS :

```
ret=request_irq(13, interrupt_handler, SA_INTERRUPT, "SLOT2cart", NULL);
enable_irq(13); printk("interrupt enabled\n");
```

Nous penserons simplement dans la méthode `release` du module à abandonner l'interruption lorsque le module est déchargé :

```
unregister_chrdev (32, "skeleton"); // /dev/skeleton 32 0
disable_irq(13);
free_irq(13, interrupt_handler);
```

Le gestionnaire d'interruption se contente d'effectuer une opération la plus brève possible, ici incrémenter un compteur indiquant qu'une interruption a eu lieu, et réveiller une fonction qui serait endormie en attente de l'interruption.

```
static irqreturn_t interrupt_handler(int irq, void *dummy)
{
        interruptcount++;
#ifdef jmf_debug
        printk(">>> Slot2 INT HANDLED: interruptcount=%d\n", interruptcount);
#endif
        wake_up_interruptible(&skeleton_wait);
        return IRQ_HANDLED;
}
```

Cette interruption a réveillé le processus endormi et en attente du signal associé à `skeleton_wait` : il s'agit dans notre cas de l'`ioctl` chargé de gérer la conversion :

```
static int skeleton_ioctl(struct inode *inode, struct file *file,
                unsigned int cmd, unsigned long arg) {
[...]
 wait_queue_t wait;
 switch ( cmd ) {
   case 1:
    copy_from_user(&count, (int *)arg, sizeof(int));
    init_waitqueue_entry(&wait, current);
    interruptcount=0,oldcount=0;
    while (interruptcount<count)
         {*(unsigned short*)(0x8000000)=(unsigned short)0;
//        interruptible_sleep_on(&skeleton_wait);           <-- race !
          add_wait_queue(&skeleton_wait, &wait);
          while (1) {
```

```
            set_current_state(TASK_INTERRUPTIBLE);
            if (interruptcount != oldcount )
               break;
               schedule();
               }
            set_current_state(TASK_RUNNING);
            remove_wait_queue(&skeleton_wait, &wait);
            oldcount=interruptcount;

          string[interruptcount-1]=*(char*)(0x8000000)&0xff;
         }
         len  = interruptcount;
         interruptcount=0;
         break;
   default: retval = -EINVAL;
 }
 return retval;
}
```

Dans ce gestionnaire d'`ioctl`, nous passons en argument à `skeleton_ioctl`, en plus de l'action à effectuer, le nombre de conversions à effectuer (variable `count`). Tant que le nombre d'acquisitions n'a pas atteint `count`, nous amorçons la conversion en écrivant dans la plage d'adresses du Slot 2, nous nous endormons le temps que l'interruption indique la fin de conversion, et nous lisons dans la plage d'adresses du Slot 2 le résultat de la mesure. Ces résultats successifs sont accumulés dans le tableau `static char string [256] ;` qui servira de tampon lors de la fonction `read`.

Une subtilité est apparue dans l'implémentation de ce code avec la méthode classique de sommeil et réveil par interruption `interruptible_sleep_on()` : la conversion est tellement rapide que le processeur n'a pas le temps d'exécuter toutes les instructions de cette fonction et appels associés avant que l'interruption ne soit déclenchée. Le résultat est que le module reste en veille au lieu d'acquérir les données successives. Ce problème est bien documenté dans [4], donc nous nous sommes contentés de prendre la solution qui consiste à explicitement séparer les étapes de mise en veille pour être capable de rapidement réagir à une interruption déclenchée trop tôt.

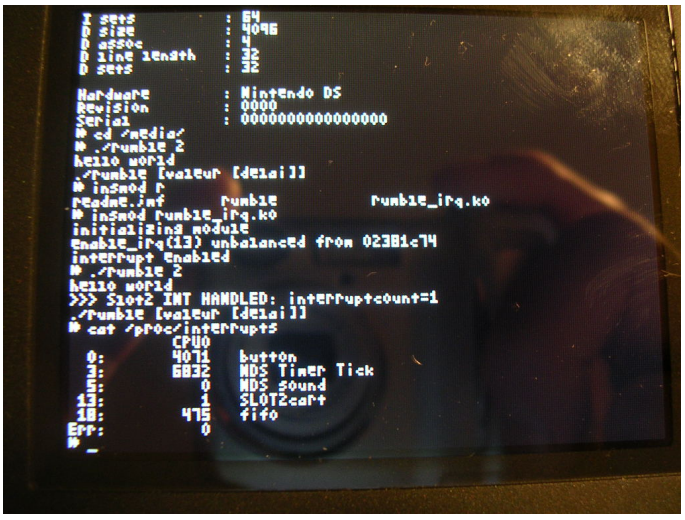


FIG. 6 – Capture d'écran présentant une séquence classique d'opérations : les programmes sont accessibles sur la carte MicroSD montée dans `/media`, le module noyau gérant l'interruption 13 – `rumble_irq.ko` – est lié au noyau : l'activation (transition du niveau bas de repos au niveau haut) d'une broche du 74HC574 dont la sortie est connectée au signal d'interruption déclenche un message du noyau validant la détection de l'évènement. La liste des interruptions gérées – dont la numéro 13 sous le nom SLOT2cart – est visible dans le pseudo-fichier `/proc/interrupts`.

5.4 Exploitation du module noyau

Afin de compiler ce module noyau, il nous faut générer une image GNU/Linux comprenant :

- un noyau 2.6.x supportant le chargement dynamique des modules (option `Loadable Module Support → Enable loadable module support`)
- une version de busybox comprenant les outils associés à `insmod`, et nettoyée des outils inutiles ou qui ne compilent pas afin d'alléger la taille de l'image (notamment `consoletools` dans Misc. Appls. et les jeux tels que `xrick`, Microwindows et `pixil`).

Une première compilation du noyau et de l'arborescence associée s'obtient par `make menuconfig` pour accéder à la configuration du noyau et de busybox : retirer les éléments qui ne compilent pas ou ne fonctionnent pas dans 4 MB de RAM tel que décrit plus haut. Une fois cette compilation convenablement achevée, nous ajoutons un répertoire qui contiendra notre propre programme, dans cet exemple nous allons créer `~/dslinux/linux-2.6.x/drivers/char/test_jm`. Ce répertoire va contenir les sources nécessaires à la compilation du module – en fait un unique programme en C nommé `rumble_irq.c`. Afin que Linux sache, lors de la compilation, qu'il faut fabriquer ce module, nous ajoutons au `Makefile` disponible dans `~/dslinux/linux-2.6.x/drivers/char` la ligne `obj-m += test_jm/` (`obj-m` contient en effet dans chaque sous-répertoire la liste des modules à compiler), et dans le répertoire `test_jm` nous incluons un Makefile réduit à sa plus simple expression :

```
~/dslinux/linux-2.6.x/drivers/char/test_jm$ cat Makefile
obj-m += rumble_irq.o
```

Afin de ne pas se contenter des messages aseptisés de la compilation du noyau Linux 2.6 mais retrouver le vrai sens des commandes de compilation, la compilation des modules se fait avec l'option `V=1` :

```
jmfriedt@ns39351:~/dslinux$ make modules V=1
[...]
make -f scripts/Makefile.build obj=drivers/char
make -f scripts/Makefile.build obj=drivers/char/test_jm
  arm-linux-elf-gcc -Wp,-MD,drivers/char/test_jm/.rumble_irq.o.d  -nostdinc -isystem \
/home/jmfriedt/dslinux-toolchain-2008-01-24-i686/bin/../lib/gcc/arm-linux-elf/4.0.4/include \
-D__KERNEL__ -Iinclude  -mlittle-endian -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
-fno-strict-aliasing -fno-common -ffreestanding -O2     -fomit-frame-pointer  -mno-thumb-interwork \
-D__LINUX_ARM_ARCH__=5 -march=armv5te   -msoft-float -Uarm -mswp-byte-writes -Wdeclaration-after-statement \
-Wno-pointer-sign   -DMODULE -DKBUILD_BASENAME=rumble_irq -DKBUILD_MODNAME=rumble_irq \
-c -o drivers/char/test_jm/rumble_irq.o drivers/char/test_jm/rumble_irq.c
[...]
  scripts/mod/modpost   -o /home/jmfriedt/dslinux/linux-2.6.x/Module.symvers vmlinux \
drivers/char/test_jm/rumble_irq.o
  arm-linux-elf-gcc -Wp,-MD,drivers/char/test_jm/.rumble_irq.mod.o.d  -nostdinc -isystem \
/home/jmfriedt/dslinux-toolchain-2008-01-24-i686/bin/../lib/gcc/arm-linux-elf/4.0.4/include \
-D__KERNEL__ -Iinclude  -mlittle-endian -Wall -Wundef -Wstrict-prototypes -Wno-trigraphs \
-fno-strict-aliasing -fno-common -ffreestanding -O2     -fomit-frame-pointer  -mno-thumb-interwork \
-D__LINUX_ARM_ARCH__=5 -march=armv5te   -msoft-float -Uarm -mswp-byte-writes -Wdeclaration-after-statement \
-Wno-pointer-sign    -DKBUILD_BASENAME=rumble_irq -DKBUILD_MODNAME=rumble_irq -DMODULE \
-c -o drivers/char/test_jm/rumble_irq.mod.o drivers/char/test_jm/rumble_irq.mod.c
  arm-linux-elf-ld -EL  -r -o drivers/char/test_jm/rumble_irq.ko \
drivers/char/test_jm/rumble_irq.o drivers/char/test_jm/rumble_irq.mod.o
```

Nous constatons que la génération d'un module pour le noyau 2.6 n'est plus l'affaire d'une unique commande de compilation.

La dernière ligne s'est conclue par la génération de `rumble_irq.ko`, module prêt à être lié (`insmod`) au noyau pour y ajouter nos fonctionnalités (Fig. 6).

Le résultat est décevant en terme de performances : nous avons vu que depuis l'espace utilisateur, avec un délai sous forme de boucle vide de 5 à 10 itérations, la conversion se fait correctement avec une fréquence d'échantillonnage de l'ordre de la centaine de kHz (535±10 kHz avec un délai de 5 itérations d'une boucle vide). Lorsque les données sont acquises par le module noyau, avec réveil par interruption associée à la fin de conversion, accumulation dans le tableau `string` pour finalement être restitué à l'utilisateur lors du `read`, nous n'atteignons que quelques kHz (44 kHz en exploitant le code de la table Tab. 3 associé au module noyau décrit ici). Ce résultat est néanmoins prévisible puisque la mise en veille et toutes les méthodes associées sont considérablement plus

gourmandes en ressources de calcul que quelques itérations d'une boucle vide. La solution "sale" de la conversion en espace utilisateur, aux performances très dépendantes de la charge du processeur et des opérations demandées au système d'exploitation, fournit néanmoins les meilleures performances lorsqu'il s'agit de la seule tâche effectuée par le processeur. Rappelons que cette seconde méthode de programmation n'est permise que sous uClinux et interdite sur les systèmes possédant un gestionnaire de mémoire, sur lesquels nous trouverons Linux.

```
// ~/dslinux/vendors/Nintendo/DLDI$ ajouter skeleton,c,32,0 \ dans DEVICES

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>

#define TAILLE 255

int main(int argc,char **argv)
{int f,taille=TAILLE;
 volatile int k;
 char *c;

 printf("ADC go\n");
 c=(char*)malloc(TAILLE);;
 f=open("/dev/skeleton",O_RDWR);
 if (f!=-1) printf("sekeleton open OK");
 ioctl(f,1,&taille);                          // lance la conversion
 read(f,c,TAILLE);                              // lit le resultat
 close(f);  // conclusion : avec ioctl et interruption, on a 44 kHz
 for (f=0;f<TAILLE;f++) printf("%x ",c[f]);printf("\n");
 return(0);
}
```

Tab. 3 – Programme proposant l'accès aux périphériques du Slot 2 au travers du point d'accès `/dev/skeleton`.

Nous avons donc démontré jusqu'ici que la Nintendo DS fournit une plateforme capable de supporter un environnement de type GNU/Linux, avec un shell et des périphériques accessibles selon les méthodes classiques de programmation des modules noyau. L'interface matérielle de la majorité des périphériques supportés par le noyau est cachée au développeur, tel que ce fut le cas pour le framebuffer ou la gestion de l'interruption. Nous avons par ailleurs vu comment ajouter nos propres interfaces *via* le bus accessible sur le Slot2. Cependant, le développeur se voit rapidement limité par la mémoire disponible, avec la majorité des 4 MB occupés par DSLinux. L'extension mémoire de 32 MB sur Slot 2 permet d'exécuter des programmes plus ambitieux – et notamment faire fonctionner la communication wifi avec les habituels `iwconfig` et `ifconfig` – mais au détriment de nos périphériques. Nous nous étions fixés comme objectif au cours de ces développements de transmettre les données acquises par le convertisseur analogique-numérique par wifi : DSLinux n'est simplement pas approprié pour cette tâche car trop gourmand, il nous faut trouver un autre environnement de travail mieux adapté aux 4 MB disponibles.

6 Un OS temps réel : RTEMS

Exploiter GNU/Linux, ou sa version pour systèmes sans MMU et à faible empreinte mémoire uClinux, est certe satisfaisant mais surtout un exercice académique compte tenu de la quantité de mémoire restante une fois le système exécuté. Par ailleurs, nous avons vu que pour de l'instrumentation, d'une part un shell interactif n'a que peu d'intérêt, et d'autre part nous n'avons aucune prétention de garantir une latence bornée sous uClinux (l'intervalle de temps entre deux

acquisitions dépend de la charge du processeur et de la bonne volonté du scheduler à laisser la main suffisamment longtemps à notre application pour obtenir tous les points requis).

Nous allons par conséquent proposer d'expérimenter avec un système d'exploitation temps-réel à faible empreinte mémoire qui semble plus approprié pour exploiter pleinement les fonctionnalités de la DS : RTEMS [7]. RTEMS (*Real Time Executive for M* Systems*, avec *M** signifiant actuellement *Multiprocessor*) fournit un certain nombre d'outils tels que pile réseau, système de fichier, gestion de plusieurs threads et éventuellement un shell qui en font un vrai système d'exploitation. Néanmoins, RTEMS ne gère pas la multiplicité de processus : son modèle de programmation est un processus unique multithreadé, sans protection de la mémoire. Ainsi, une image RTEMS se présente comme un binaire monolithique contenant l'ensemble des instructions à exécuter, sans chargemement dynamique d'exécutables ou de bibliothèques, d'où le nom d'*Executive* au lieu de OS : le lecteur prendra soin de corriger le titre de cette section de façon appropriée.

Un BSP (*board support package*) a récemment été inclus, suite au travail de M. Bucchianeri, B. Ratier, R. Voltz et C. Gestes, dans RTEMS pour la console DS : son utilisation est décrite à `http://www.rtems.com/ftp/pub/rtems/current_contrib/nds-bsp/manual.html`

Afin de permettre la compilation de RTEMS pour DS, il nous faut obtenir une nouvelle toolchain : celle obtenue auparavant pour DSLinux ne fonctionne pas. Toutes les étapes sont parfaitement décrites dans `http://www.rtems.com/onlinedocs//doc-current/share/rtems/pdf/started.pdf`[8] pour le lecteur suffisamment patient pour en lire l'ensemble du contenu. En résumé, il suffit de récupérer les dernières versions de `gcc`, `binutils` et `newlib` sur `ftp://ftp.rtems.com/pub/rtems/SOURCES/4.9/` et de les configurer pour la cible ARM par `--target=arm-rtems4.9`. Noter que compiler *dans les archives* pose un certain nombre de problèmes [9] : mieux vaut s'éviter ces ennuis et compiler les outils de la toolchain dans des répertoires autres que ceux de l'archive. Une fois la toolchain complète placée dans un répertoire accessible et inclus dans le PATH (espace disque : environ 100 MB après compilation), il reste à compiler RTEMS.

Ici encore nous prendrons soin de compiler dans un répertoire autre que celui contenant l'archive. Partant d'une arborescence propre de `rtems-4.9.1`, la configuration pour le BSP de la console d'obtient par
`../rtems-4.9.1/configure -target=arm-rtems4.9 -enable-rtemsbsp=nds`. La compilation elle même s'obtient par `make`, toujours en prenant soin d'avoir les outils de la toolchain (`arm-rtems4.9*` dans son PATH.

Les résultats de la compilation exploitables se trouvent dans `jmf/arm-rtems4.9/c/nds/testsuites/samples` en supposant que la compilation de RTEMS-4.9.1 ait été effectuée dans le répertoire `jmf`.

Afin de compiler ses propres applications ou, pour l'occasion, les exemples de la distribution "officielle" RTEMS 4.9.1 [10], nous définissons les variables d'environnement

```
export PATH=/home/jmfriedt/rtems/bin/:$PATH
export RTEMS_ROOT=/opt/rtems-4.9
export RTEMS_MAKEFILE_PATH=/opt/rtems-4.9/arm-rtems4.9/nds
```

(évidemment adaptées à son propre environnement) avant de récupérer `http://www.rtems.com/ftp/pub/rtems/4.9.1/examples-4.9.1.tar.bz2` dont le contenu se compile simplement par `make` (Figs. 7 et 8, programme Tab. 4). Les Makefiles ainsi obtenus serviront d'exemple pour nos programmes.

La console DS fournit donc un support idéal pour se former à cet environnement de développement d'applications temps-réel fournissant notamment une API compatible POSIX, qui ne dépayse donc pas fondamentalement le développeur sous GNU/Linux. Par ailleurs, un grand nombre de fonctionnalités restent à implémenter et le développeur amateur peut encore prétendre fournir une contribution significative dans ce cas (exploitation de la communication wifi, accès à la carte mémoire microSD).

[7] `http://www.rtems.com/`

[8] 4.1.4.2 page 22

[9] dont une solution, fonctionnelle à défaut d'être élégante, est founie à `http://www.mail-archive.com/gcc-bugs@gcc.gnu.org/msg191702.html`

[10] disponibles à `http://rtems.org/wiki/index.php/RTEMS_CVS_Repository`

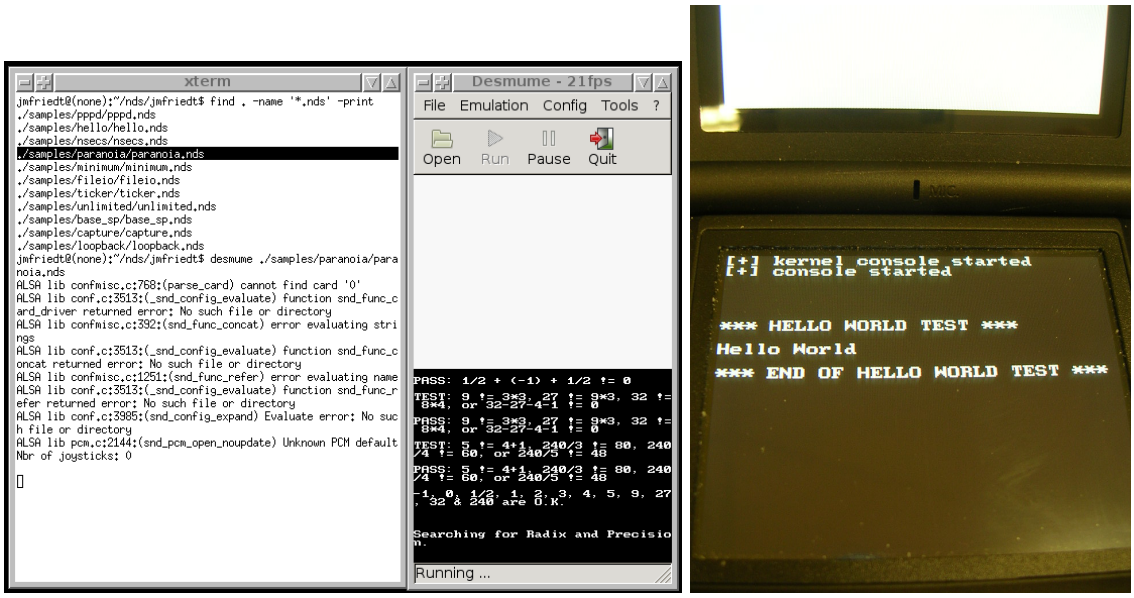


FIG. 7 – Gauche : RTEMS exécuté dans l'émulateur `desmume`. Droite : un des exemples le plus simple de RTEMS, exécuté sur la console.

FIG. 8 – Exploitation du framebuffer selon le même principe que proposé dans le cas de uClinux, mais cette fois sous RTEMS. Cet exemple est issu d'un programme de démonstration fourni par M. Bucchianeri. Ne sachant pas accéder depuis RTEMS au support de stockage de masse uSD, l'image est stockée dans le programme.

Tab. 4 – Programme d'affichage d'une image au format PPM sur le framebuffer, issu d'un programme de démonstration fourni par M. Bucchianeri.

Nous ne nous étendrons par sur les exemples de base fournis sur le site de RTEMS, tous fonctionnels. Mentionnons néanmoins quelques points qui peuvent surprendre à première vue

- RTEMS fournit un environnement de développement s'apparentant à un système d'exploitation, mais le résultat est un unique binaire avec notamment l'absence de notion de chargement dynamique d'exécutables. L'exécution du programme se décompose en tâches appelées selon les évènements qui influent sur le comportement du *scheduler* :
- dans ce contexte de système autonome à faible empreinte mémoire, le shell interactif avec l'utilisateur est une option gourmande en ressources qui n'est pas activée par défaut
- bien qu'écrite en C, une application RTEMS a une structure différente des programmes C habituellements écrits sous GNU/Linux. Une couche de compatibilité POSIX rend néanmoins le passage à RTEMS rapide pour le développeur sous Unix. Comme souvent un certain nombre de `#include` en entête de fichier annoncent quels fichiers de configuration charger, mais surtout un grand nombre de `#define` définissent quelles fonctionnalités du BSP activer. Chaque fonctionnalité a un coût en terme de ressources et on prendra soin de se limiter au strict minimum. Réciproquement, l'oubli d'une ressource peut induire un dysfonctionnement de l'application qui n'est pas toujours facile à debugger : par exemple, l'appel à la fonction `sleep` sans activer de timer induit nécessaire une attente infinie telle que décrite à `http://www.rtems.com/wiki/index.php/DebuggingHints`. L'*ordre* de ces `#define` est important car des variables définies dans un premier fichier peuvent être exploitées dans une définition ultérieure.

Nous allons dépasser le cadre de la simple application logicielle pour nous tourner, comme dans l'exemple d'utilisation de DSLinux, vers l'exploitation de périphériques matériels. Afin de faire clignoter (Code 5) les diodes connectées au latch installé à la place du moteur dans le Rumble Pack (bus du Slot 2), il faut affecter ce bus au processeur ARM9 (et non à son coprocesseur ARM7) par la commande `sysSetCartOwner(BUS_OWNER_ARM9) ;` : sans cette fonction, le processeur qui exécute RTEMS n'a pas le contrôle du bus du Slot 2 et les diodes ne changent pas d'état. La macro de cette commande, définie dans `rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/source/arm9/rumble.c`, a été développée pour faciliter la compilation : il s'agit de la ligne modifiant la valeur de l'emplacement mémoire `0x04000204`. Noter que cette affectation du bus à l'ARM9 était déjà effectuée sous DSLinux.

Étant capable de valider l'accès au bus du Slot2, il est trivial de convertir les divers exemples proposés sur DSLinux – et notamment la conversion analogique numérique et l'affichage sur framebuffer – sur RTEMS et obtenir les mêmes résultats. Nous voulons cependant aller plus loin et atteindre notre objectif de communication des données acquises par wifi : dans un premier temps nous allons voir comment activer un shell interactif sous RTEMS (afin de se rapprocher de l'environnement de travail habituel sous GNU/Linux et illustrer quelques unes des instructions originales proposées par RTEMS pour debugger des applications embarquées), avant de nous lancer dans l'aventure du fonctionnement du wifi.

6.1 Un shell interactif

Bien que son utilité dans un système fortement embarqué à faible empreinte mémoire soit discutable – l'interactivité avec le contrôleur d'une machine à laver est rarement utile à l'utilisateur en dehors des phases de debuggage – nous nous sommes proposés d'exploiter le shell de RTEMS et d'y ajouter nos propres fonctionnalités. Le shell de RTEMS est avant tout orienté, dans sa configuration par défaut, vers le diagnostic de l'état du système en cours d'exécution : occupation du processeur par chacune des tâches (`cpuuse`), mémoire occupée et taille de la pile allouée à chaque tâche (`stackuse`, Fig. 9) et contenu de cette mémoire (`mdump`). Quelques fonctions d'accès à un éventuel système de fichiers et interface de communication sont disponibles en option.

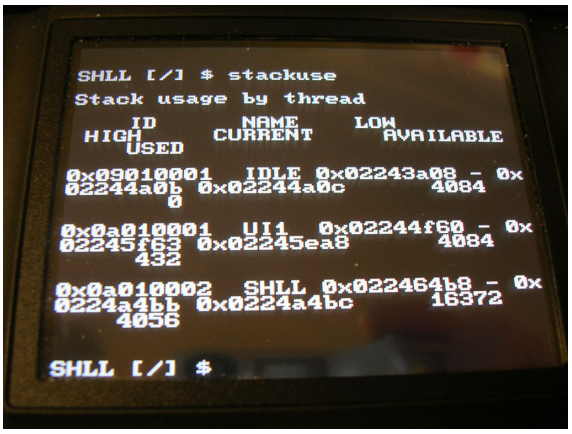


FIG. 9 – Une des fonctions originales de RTEMS et des plus utiles pour le développement de systèmes embarqués : l'occupation de la pile associée à chaque thread.

L'interaction avec un shell nécessite évidemment d'entrer des caractères formant les commandes des actions à communiquer au système. Le mode de communication proposé par le BSP NDS de RTEMS est la reconnaissance de l'écriture manuelle des lettres sur l'écran tactile suivant un algorithme implémenté dans la PALib : la façon d'écrire les lettres anglo-saxone doit être suivie scrupuleusement pour espérer entrer le bon caractère, selon la méthode décrite à `http://www.palib.info/wiki/doku.php?id=day3#keyboard` et reproduite sur la Fig. 10. Noter que l'activation de l'interface `graffiti` nécessite d'appuyer sur la touche L de la console tout en écrivant les lettres sur l'écran tactile.

FIG. 10 – Ordre de calligraphie des caractères afin d'être reconnus par la PALib exploitée par RTEMS pour recevoir des caractères de l'utilisateur. Noter que la reconnaissance des caractères est activée par l'appui du bouton L de la console et en écrivant simultanément sur l'écran tactile de la DS.

Afin de valider notre capacité à ajouter une fonction qui utilise un maximum de ressources, nous avons implémenté la fonction `newt` qui affiche la fractale de Newton [2] pour le polynôme

$z^3 - 1$, $z \in \mathbb{C}$: cette application démontre la fonctionnalité de l'émulation logicielle du calcul flottant, l'accès à une interface graphique sous forme de *framebuffer*, et le passage de paramètres.

Le prototype de la fonction à ajouter au shell s'apparente au `main(int, char**)` classique du C. Ainsi, une commande `newtn` qui exécute le contenu de la fonction `mon_main` qui reçoit ses arguments de la même façon que le `main` du C s'obtient par :

```
rtems_shell_cmd_t Shell_USERCMD_Command = {
  "newtn",                    /* name */
  "newt [mag.]",              /* usage */
  "user",                     /* topic */
  mon_main,                   /* command */
  NULL,                       /* alias */
  NULL                        /* next */
};
```

Comme par ailleurs `graffiti` est relativement pénible à utiliser et que les chances de taper une commande de 5 lettres sans erreur sont quasiment nulles, nous créons un alias sur la commande par

```
rtems_shell_alias_t Shell_USERECHO_Alias={
"newtn", /* command*/
"n"     /* alias */
};
```

Ces commandes sont ajoutées au shell [11] par

```
#define CONFIGURE_SHELL_USER_COMMANDS &Shell_USERCMD_Command
#define CONFIGURE_SHELL_USER_ALIASES  &Shell_USERECHO_Alias
#define CONFIGURE_SHELL_COMMANDS_INIT
#define CONFIGURE_SHELL_COMMANDS_ALL
#include <rtems/shellconfig.h>  // doit etre APRES les #define
```

Dans cet exemple, la commande `newtn` (ou `n`) dessine la fractale de Newton avec un facteur de grossissement défini par l'argument (Fig. 11).

6.2 Estimation des latences

RTEMS s'annonce comme un environnement temps réel. Nous avons donc tenté de caractériser les latences associées aux systèmes DSLinux et RTEMS lorsque trois threads sont actifs, deux threads pour faire clignoter des diodes avec des intervalles de temps définis par une attente (`usleep()`) et un troisième thread, activé manuellement par l'utilisateur, chargé d'effectuer un calcul long (le calcul et l'affichage simultané de la fractale de Newton sur le polynôme $z^3 - 1$) (Fig 12). Ces tests ont pour objectif

- de valider le fonctionnement de la bibliothèque `pthreads` classiquement utilisée sous GNU/-Linux, et en particulier son exploitation sous uClinux dans le cadre de DSLinux
- de comparer les méthodes d'implémentation des threads entre DSLinux [12] et RTEMS (Codes 6 et 7)
- de comparer les latences [5] en mesurant à l'osciloscope la période de commutation des diodes – période définie dans deux des trois threads – et la robustesse de l'application selon que le troisième thread chargé d'un calcul lourd (calcul et tracé de la fractale de Newton calculée pour tous les points de l'écran, soit une dizaine de secondes de calculs) soit actif ou non.

Les résultats de ce test grossier montre que les lattences sous RTEMS (Fig. 12, gauche) sont plus stables lors de la charge que sous DSLinux, mais la différence est moins impressionante qu'on pourrait l'attendre. Un aspect visible à l'oscilloscope mais qui ne peut pas être retranscrit sur ces images statiques est que la variation de période sous DSLinux intervient au moment de *lancer*

[11] `http://www.rtems.com/onlinedocs//doc-current/share/rtems/html/shell/shell00008.html`

[12] la compilation d'une application exploitant la bibliothèque des POSIX threads s'obtient dans l'environnement issu de `make xsh` par $CC $CFLAGS $LDFLAGS −L/home/jmfriedt/dslinux/uClibc/lib/ fb_thread.c −o → ↪fb_thread −lpthread

FIG. 11 – Une nouvelle commande pour le shell RTEMS : le calcul de la fractale de Newton (illustrant au passage l'émulation du calcul sur nombre à virgule flottante et l'affichage sur framebuffer) par `newtn`.

le calcul de la fractale, et non *pendant* le calcul : la variation de charge induit plus de lattence que la charge de calcul elle même. Nous n'avons pas cherché à identifier ce phénomène, qui serait probablement visible en analysant les sources du *scheduler* de chaque système d'exploitation. Noter que tous les threads sous DSLinux ont la même priorité tandis que sous RTEMS, nous affectons la plus grande priorité au thread contenant le shell.

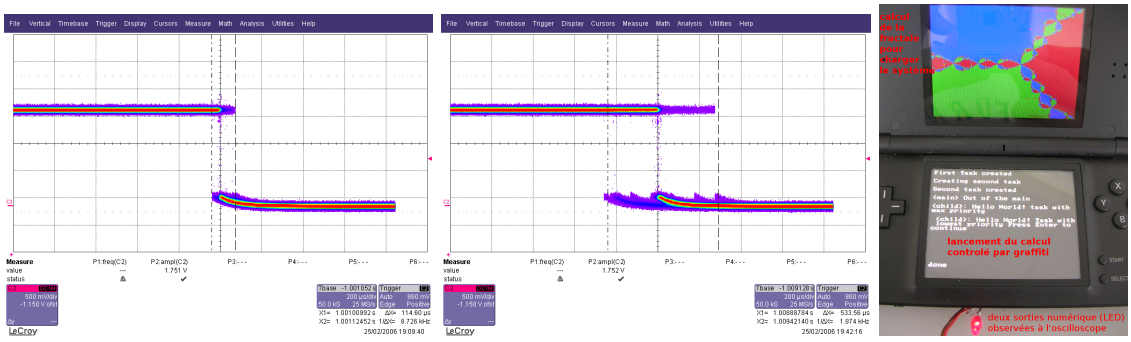


FIG. 12 – Un oscilloscope numérique déclenche sa mesure sur le front montant d'un crénau généré de façon logicielle en programmant le latch connecté à la carte du slot2. Nous agrandissons la mesure autour du front descendant pour estimer la variance de la longueur du crénau en fonction de la charge du système : à gauche le résultat de la mesure pour RTEMS dans le cas défavorable d'un shell de priorité maximale et de threads pour le clignotement de priorité faible, au milieu pour DSLinux. À droite : montage expérimental sous RTEMS, avec le déclenchement du calcul (induisant les lattences lors de l'ajout de cette charge de calcul additionnelle) par un retour chariot sous graffiti (bouton L et un trait sur l'écran tactile de en haut a droite vers en bas à gauche).

6.3 Communication Wifi

RTEMS ayant une empreinte mémoire plus faible de DSLinux, celui-ci permet l'utilisation du wifi de la DS sans devoir ajouter *l'expansion pack*.

Le wifi, tel que fourni par Nintendo, offre une fonctionnalité inhabituelle. En effet la NDS est capable de stocker les paramètres de configuration pour la connexion à trois points d'accès[13], afin que les mêmes informations soient disponibles pour n'importe quel jeu requérant le wifi. Bien que DSLinux prouve qu'il est possible d'utiliser le chipset d'une manière classique (à l'aide des applications `iwconfig` ou `wpa_supplicant`), tout comme le logiciel de démonstration wifi_lib_test [14], RTEMS exploite les données stockées dans la console comme le ferait un jeu. Il n'est d'ailleurs pas possible, à l'heure actuelle, de configurer la connexion à un `Access Point` depuis RTEMS, imposant, de ce fait, l'utilisation d'un jeu commercial supportant le wifi. Le driver fourni par RTEMS n'étant pas complètement fonctionnel, nous nous sommes efforcés de corriger quelques incompatibilités entre la couche réseau de RTEMS et le driver, afin de rendre la communication wifi sur NDS opérationnelle. Le patch proposé est disponible à `http://jmfriedt.free.fr/rtems_nds_wireless.patch` pour les versions 4.9.1 et 4.9.2 de RTEMS.

La présentation du wifi avec RTEMS se fera en trois temps :

1. dans un premier temps, pour valider le bon fonctionnement de la communication, nous allons utiliser `gethostbyname()`,
2. puis nous mettrons en œuvre un serveur `telnetd` disponible dans les librairies de RTEMS afin d'obtenir un shell sans devoir passer par `graffiti`,
3. et pour finir, une application qui fera parvenir à un ordinateur des valeurs issues d'une conversion analogique-numérique pour une application de télémesure.

6.4 `gethostbyname`

Comme la couche réseau de RTEMS est basée sur celle des BSD et compatible POSIX, cet exemple pourrait fonctionner directement sur GNU/Linux ou un Unix.

Son intérêt est d'une part de mettre en avant les configurations nécessaires pour faire fonctionner le réseau et d'autre part, d'illustrer le bon fonctionnement du BSP NDS (RTEMS 4.9.1, 4.9.2) après application de notre patch. Cet exemple simple permettra de présenter une solution de debug et de développer comment la solution a été identifiée puis mise en œuvre.

L'utilisation du réseau à partir de RTEMS consiste en la configuration de deux structures :

```
/* Default network interface */
static struct rtems_bsdnet_ifconfig netdriver_config = {
  RTEMS_BSP_NETWORK_DRIVER_NAME,
  RTEMS_BSP_NETWORK_DRIVER_ATTACH,
  NULL,                     /* No more interfaces */
  "10.0.1.20",              /* IP address */
  "255.255.255.0",              /* IP net mask */
  NULL,                     /* Driver supplies hardware address */
};
```

Cette première structure, équivalente à *ifconfig*, configure l'interface réseau. *RTEMS_BSP_NETWORK_DRIVER_N* donne le nom de l'interface et *RTEMS_BSP_NETWORK_DRIVER_ATTACH* fournit la méthode d'initialisation.

```
/* Network configuration */
struct rtems_bsdnet_config rtems_bsdnet_config = {
  &netdriver_config,
  NULL,                 /* do not use bootp */
  0,                    /* Default network task priority */
  0,                    /* Default mbuf capacity */
  0,                    /* Default mbuf cluster capacity */
  "rtems",              /* Host name */
  "trabucayre.com",       /* Domain name */
```

[13] Liste des *Access points* compatibles disponible sur `http://www.nintendo.com/consumer/wfc/en_na/ds/routerInfo.jsp`

[14] http ://www.akkit.org/dswifi/

```
  "10.0.1.1",          /* Gateway */
  "10.0.1.13",          /* Log host */
  {"10.0.1.13" },       /* Name server(s) */
  {"10.0.1.13" },       /* NTP server(s) */
};
```

Cette seconde structure, à l'instar de la commande *route* et du fichier *resolv.conf*, permet de fournir les paramètres globaux de configuration du réseau.

Ensuite, au lancement de l'application (*i.e.* fonction `rtems_task Init(rtems_task_argument ignored)`), l'application doit faire appel à `rtems_bsdnet_initialize_network();` qui se charge de l'initialisation de l'interface, de la connexion au point d'accès et de la configuration de l'ensemble des paramètres du réseau.

Ce premier exemple, une fois compilé et installé sur la μSD, affichera l'adresse IP d'une machine dont le nom a été fourni à `gethostbyname()`.

En l'absence de l'interlocuteur, ou avant application du patch (*i.e.* avec un RTEMS obtenu sur l'archive officielle), après un peu d'attente, un message d'erreur sera affiché en lieu et place de l'IP. Pour comprendre la cause de cette erreur, nous allons voir une solution pour debugger le réseau , celle-ci se déroulant en deux étapes :

La première partie concerne l'analyse du réseau afin de pouvoir cibler le problème [6]. À l'instar du debug d'un logiciel, ce type de travail nécessite l'utilisation d'un outil adéquat, tel que le sniffer réseau *ethereal*.

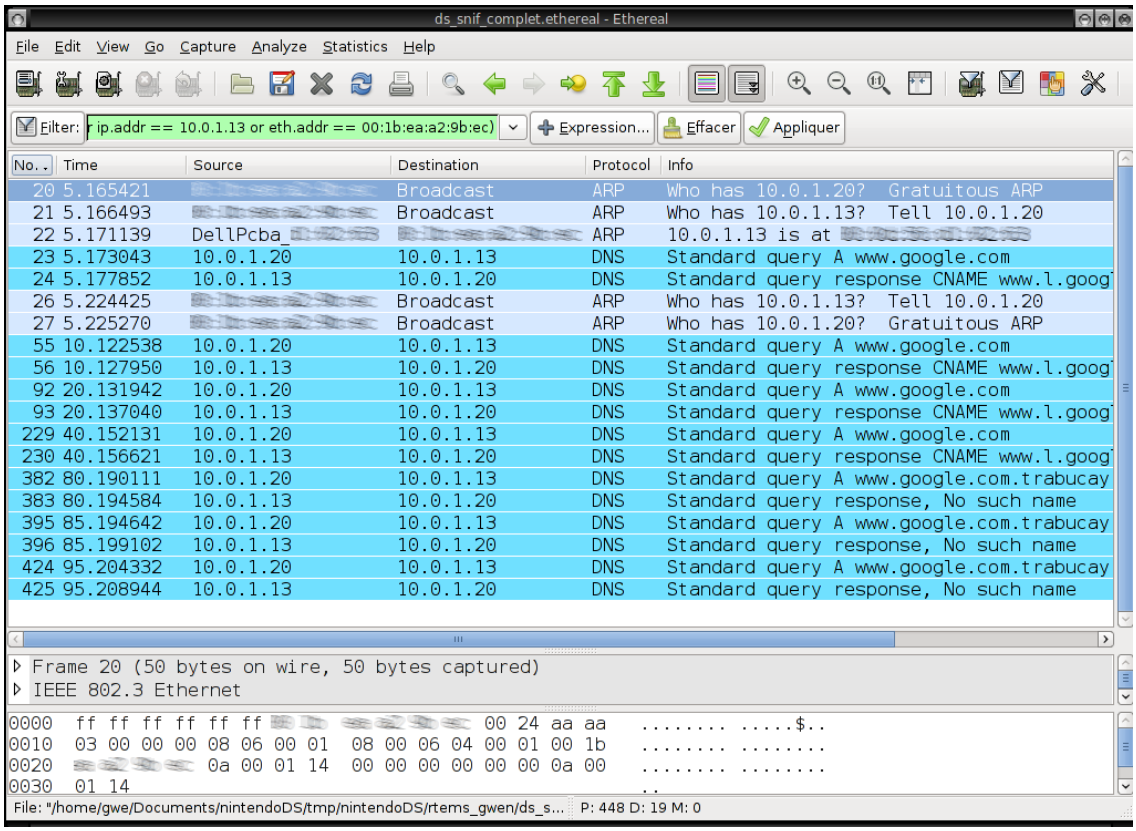


FIG. 13 – Capture réseau à l'aide de *ethereal*

En examinant les trames affichées dans *ethereal* (Fig. 13), nous pouvons suivre l'ensemble du dialogue réseau avec la NDS. La première partie est une requête ARP (trames 21,22) où la DS demande l'adresse MAC du serveur DNS, la machine concernée lui retourne l'information. Ce premier dialogue permet de déduire que les requêtes ARP sont correctement gérées car (trame 23) la DS fait une requête DNS auprès du serveur. Ce dernier lui répond (trame 24) mais la DS émet

à nouveau le même message comme le montrent les trames qui suivent : les paquets sont donc correctement émis et reçus, mais un point de la chaîne de traitement ne fonctionne pas.

Nous avons fait globalement le tour des points importants de l'analyse et nous pouvons déterminer que la DS est capable d'émettre et de recevoir des paquets vers et depuis le réseau, et que les paramètres de configuration sont correctement pris en compte. Dans le cas de l'ARP les paquets sont correctement traités, par contre dans le cas du DNS (UDP) l'information semble se perdre.

Maintenant que nous savons à quel niveau se trouve le problème, nous arrivons sur la seconde étape de l'analyse. Nous devons étudier le cheminement d'un paquet IP à travers les couches réseau pour trouver le point de "rupture".

Pour cela il n'est pas nécessaire de partir des couches les plus basses car il semble bien que le pilote reçoive les paquets correctement. Il faut donc trouver où est faite la distinction entre paquet ARP et paquet IP.

En partant donc des sources du BSP NDS au niveau de la fonction qui transfère le paquet à la bibliothèque RTEMS (`wifi/wifi.c` : `wifi_rxd`), et en suivant les appels de fonctions, la distinction entre le type du paquet se fait dans `net/if_ethersubr.c` : `ether_input`. En affichant dans cette fonction des messages avec `printk` pour des paquets de type *ETHERTYPE_IP*, nous pouvons voir que celui-ci existe toujours. Cette fonction fait indirectement appel pour la suite du transfert à *netinet/ip_input.c* : `ip_input`. En suivant l'exécution de cette fonction nous arrivons sur un `return`, appelé lorsque le paquet reçu n'est pas destiné à la console. Pourtant le paquet lui est bien destiné ...

Pour comprendre la raison de ce comportement, il suffit d'afficher l'adresse du destinataire contenue dans la `struct ip`. L'IP de l'émetteur se trouve à la place de celle du destinataire, ceci expliquant le rejet car le système s'attend à y trouver sa propre IP.

Il est donc possible de conclure que la conversion faite entre la `struct mbuf` et la `struct ip` n'aligne pas correctement les informations et qu'*à priori* les données arrivant du réseau dans le cas d'un paquet IP ne sont pas correctement stockées.

Notre solution simple, mais fonctionnelle, est l'utilisation de `m_pullup(...)` qui permet d'assurer que le contenu de la structure est bien contigu. Il ne reste donc plus qu'à déterminer dans le pilote du BSP ce qui entraîne ce problème, corrigé, actuellement, par réagencement des informations de la structures de données.

6.5 `telnetd`

Une fois la liaison wifi fonctionnelle, nous pouvons aborder des applications plus ambitieuses. RTEMS offre une implémentation d'un serveur `telnet`, permettant de pouvoir se connecter par le réseau sur la DS depuis n'importe quel terminal, d'une manière simple.

Cette implémentation n'implique pas forcément l'ouverture d'un shell. Dans le code présenté ci-dessous, le login n'est dû qu'à l'appel à `rtems_shell_main_loop()` qui lance un nouveau shell, qui sera disponible ensuite pour l'utilisateur distant.

En plus de permettre de valider la bonne intégration dans RTEMS, l'exemple suivant offre une solution pratique pour s'éviter l'utilisation (laborieuse) de `graffiti` (Fig. 14). Son utilisation est triviale car reposant uniquement sur l'appel à une fonction, après l'initialisation du réseau, de la manière suivante :

```
rtems_telnetd_initialize(
  rtemsShell, /* "shell" function */
  NULL,   /* no context necessary for echoShell */
  false,          /* listen on sockets */
  RTEMS_MINIMUM_STACK_SIZE * 20,  /* shell needs a large stack */
  1,    /* priority */
  false           /* telnetd does NOT ask for password */
);
```

le premier paramètre est le nom de la fonction qui sera appelée à chaque connexion :

```
void rtemsShell(char *pty_name, void *cmd_arg) {
  printk("========= Starting Shell =========\n");
  rtems_shell_main_loop( NULL );
```

```
  printk("========== Exiting Shell ==========\n");
}
```

Son utilisation nécessite également l'inclusion du fichier `rtems/telnetd.h` dans les sources, ainsi que l'ajout de la ligne `LD_LIBS += -ltelnetd` dans le Makefile de l'application.

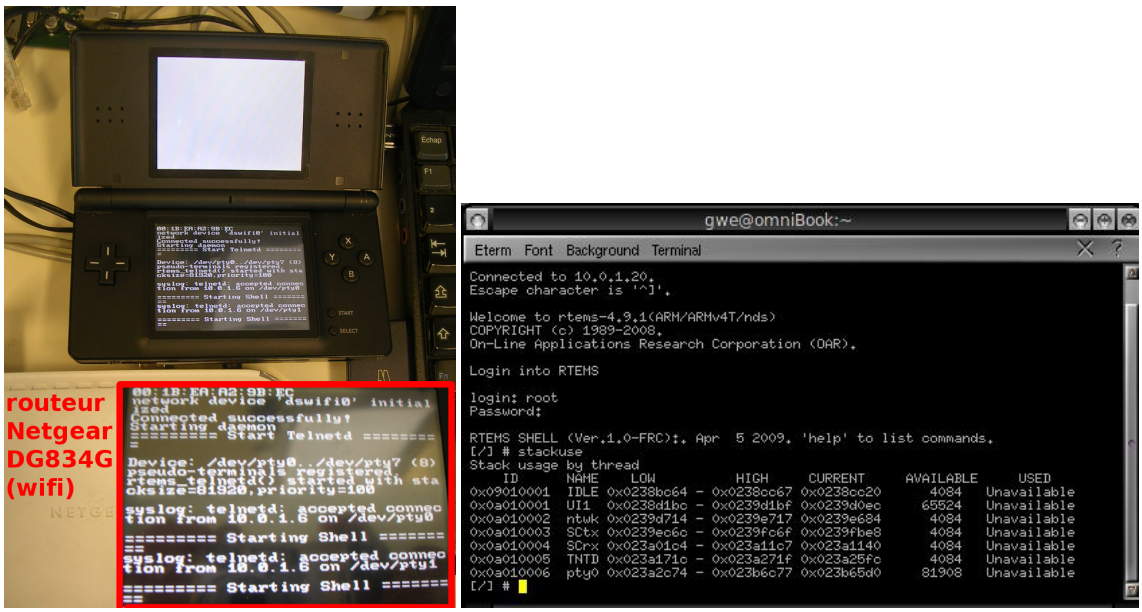


FIG. 14 – À gauche messages affichés sur la DS lors d'une connexion. À droite, l'affichage de la commande `stackuse` dans un terminal connecté par `telnet`.

La connexion de l'utilisateur se fera grâce à la commande `telnet ip_de_la_DS`.

6.6 Transfert de données issues du convertisseur analogique-numérique

Le dernier exemple repose sur une reprise de l'application de conversion analogique-numérique, presentée plus haut pour DSLinux. Dans ce cas, les valeurs ne seront plus affichées sur la DS mais vont être transférées à un ordinateur personnel à travers la connexion wifi *via* un point d'accès.

Ce code d'exemple a été allégé de l'initialisation du réseau qui se fait exactement comme dans le premier exemple (section 6.4) :

Comme présenté précédemment, la bibliothèque `telnetd` ne fait qu'appeler une fonction (`telnetADC`) lors de la connexion d'un utilisateur. Au lieu d'un shell, il est possible de lancer la conversion, tel que présenté Fig. 15. Les descripteurs d'entrée et de sortie standard (stdin, stdout) pointant sur la socket à laquelle est connectée l'application cliente côté PC, les valeurs obtenues, affichées avec `printf`, sont automatiquement transférées par le réseau. Du côté client, la solution la plus simple est l'utilitaire `telnet` puisque nous affichons en hexadécimal et en ASCII le résultat des mesures. Les valeurs sont alors transmises à l'utilisateur distant, qui peut les enregistrer par redirection dans un fichier avec une commande telle que : `telnet ip > acquiss.ds`.

Un traitement ultérieur avec GNU/Octave a permis d'obtenir la courbe présentée figure 16. Nous y observons une sinusoïde – en accord avec le signal injecté depuis un synthétiseur de fonctions – de fréquence variable puisque nous avons successivement programmé une fréquence de 500 et 1000 Hz. Les discontinuités entre les séries successives de 1024 points résultent de l'absence d'un second tampon chargé des mesures pendant le transfert par wifi des données accumulées. En comptant le nombre de points de mesure par période, nous déduisons une fréquence d'échantillonage de l'ordre de 300 kHz.

Nous avons donc atteint notre objectif de transmission de données au moyen d'un protocole réseau TCP/IP sur support wifi au moyen de la DS. Cette application a fourni l'opportunité d'ana-

```
/*  ADC to network */

#include <stdlib.h>
#include <stdio.h>
#include <bsp.h>
#include <rtems/telnetd.h>
#include <nds/memory.h>
#include <rtems/rtems_bsdnet.h>

// [...] CONFIGURATION DU RESEAU

#define TAILLE 1024
/* callback for telnet */
void telnetADC( char *pty_name, void *cmd_arg) {
  char *c;
  int f;

  printk( "Connected to %s with argument %p \n",
    pty_name, cmd_arg );

  c=(char*)malloc(TAILLE);
  while (1) {
    for (f=0;f<TAILLE;f++) {
      *(unsigned short*)(0x8000000)=(unsigned short)0;
      c[f]=*(unsigned short*)(0x8000000)&0xff;
    }
    for (f=0;f<TAILLE;f++) printf("%x ",c[f]);
    printf("\n");
  }
}

/* Init task */
rtems_task Init(rtems_task_argument argument){
  fprintf(stderr, "\n\n*** Telnetd Server Test ***\n\r" );
  fprintf(stderr, "========== Initializing Network ==========\n");
  rtems_bsdnet_initialize_network ();
  fprintf(stderr, "========== Start Telnetd ==========\n");
  (*(volatile uint16_t*)0x04000204) = ((*(volatile uint16_t*)0x04000204) & ~→
      ↪ARM7_OWNS_ROM);
  rtems_telnetd_initialize(
      telnetADC,        /* callback function */
    NULL,             /* no context necessary for echoShell */
    false,            /* false == listen on sockets */
    RTEMS_MINIMUM_STACK_SIZE * 20,  /* shell needs a large stack */
    1,                /* priority .. we feel important today */
    false             /* telnetd does NOT ask for password */
  );
  while(1);
}

#include "../rtems_common.h"
```

Fig. 15 – Code complet de l'application de conversion analogique-numérique vers le réseau

lyser en détail le cheminement d'un paquet IP dans RTEMS. Néanmoins, un point d'amélioration subsiste avec cette nécessité de fournir la configuration du point d'accès distant au moyen d'un jeu commercial. Bien que cette solution soit satisfaisante pour une application ludique, il pourrait être intéressant d'ajouter la possibilité de pouvoir soit utiliser les données stockées dans la DS, soit de fixer les paramètres à travers RTEMS et le rendre indépendant d'une application ou d'un jeu tiers.

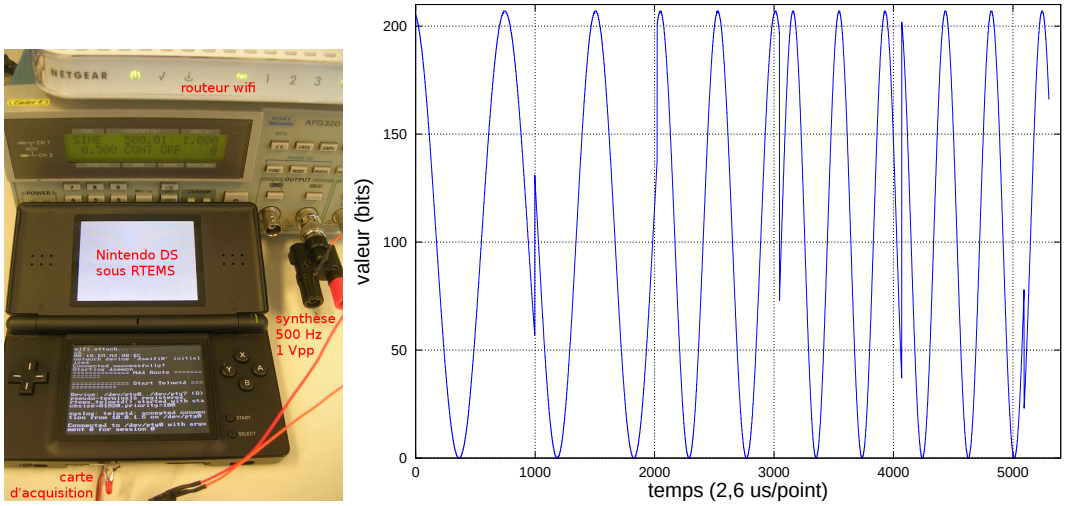


FIG. 16 – Gauche : montage pour l'acquisition de signaux par conversion analogique-numérique sous RTEMS et transfert par paquets de 1024 mots des données par Wifi au travers d'un routeur. Droite : signaux acquis, dans un premier temps une sinusoïde à 500 Hz puis dans un second temps à 1 kHz.

7 Conclusion et perspectives

Nous avons donc proposé dans cette présentation d'exploiter notre système d'exploitation favori pour tirer le meilleur parti d'une console de jeu, et étendu nos compétences en exécutant quelques applications simples d'un environnement de développement d'applications temps-réel. Ces activités auront été le prétexte d'appréhender ce qui est possible (et ce qui ne l'est pas) avec des ressources réduites en terme de puissance de calcul et surtout de mémoire – réduites certes mais malgré tout importantes par rapport à ce que proposent de nombreux systèmes embarqués. Nous nous sommes efforcés de compléter les fonctionnalités de la console de jeu par des périphériques dédiés plus appropriés pour des applications de contrôle d'expériences ou d'acquisitions de données.

Nous avons étendu notre champ d'investigation à un environnement de travail plus approprié pour les ressources de la DS, en exploitant l'environnement exécutif RTEMS. Après avoir reproduit un certain nombre des fonctionnalités exploitées sous DSLinux, nous avons rendu fonctionnel la liaison wifi pour exploiter pleinement les périphériques de la console.

Une fois que nous maîtrisons les concepts de communication bidirectionnelle sur le bus de données, de gestion des signaux de contrôle et de l'interruption fournie par `IRQ`, nous sommes capables d'interfacer à peu près n'importe quel composant numérique à la DS *via* son Slot2. Un exemple intéressant pour approfondir ces connaissances serait d'ajouter un UART pour la communication asynchrone (type RS232), périphérique qui manque cruellement à cette console de jeu [15] pour une liaison avec microcontrôleurs ou PC sans passer par une couche aussi lourde que la liaison sans fil.

Ces briques de base ouvrent des perspectives intéressantes d'exploitation d'une console, activité probablement plus ludiques que la majorité des jeux disponibles.

Remerciements

Pierre Kestener (CEA/IRFU, Saclay) nous a informé de la disponibilité du BSP pour Nintendo DS de RTEMS. Matthieu Bucchianeri a patiemment répondu à nos questions alors que nous

[15] par exemple le NXP – anciennement Philips – SCC2691 présente les fonctionnalités recherchées avec un nombre raisonnable de pattes pour un coût unitaire inférieur à 7 euros chez Farnell

découvrions RTEMS.

Références

[1] M. Rafiquzzaman, *Microprocessors and microcomputer-based system design, 2nd Ed.*, CRC Press (1995)

[2] J.-M Friedt & É. Carry, *Développement sur processeur à base de cœur ARM7 sous GNU/Linux*, GNU/Linux Magazine France **117** (Juin 2009), pp.40-59 GNU/Linux Magazine France (Juin 2009)

[3] Y. Guidon, *Programmation graphique : vers le framebuffer et au-delà*, GNU/Linux Magazine France 112, 82-94, (Janvier 2009)

[4] A. Rubini & J.Corbet, *Linux Device Drivers, 2nd Ed.*, O'Reilly (2001), disponible à `http://www.xml.com/ldd/chapter/book/ch09.html#t8`

[5] P. Ficheux & P. Kadionik, *Temps réel sous Linux (reloaded)*, GNU/Linux Magazine France, Hors Série 24 (Fév/Mars 2006), disponible à `http://www.unixgarden.com/index.php/comprendre/temps-reel-sous-linux-reloaded`

[6] D. Bodor, *Inspectez le trafic réseau en utilisant la libpcap*, GNU/Linux Magazine France, Hors Série 42 (Juin-Juillet 2009), pp.52-58

Gwenhaël Goavec-Merou finit son master en informatique, et est actuellement en stage chez ARMadeus Systems. Il est propriétaire d'une DS... Citroën.

Jean-Michel Friedt est membre de l'association Projet Aurore pour la diffusion de la culture scientifique et technique sur le campus bisontin de l'université de Franche Comté. Il s'intéresse aux développements de systèmes embarqués, dont les consoles de jeu portables fournissent un excellent support. Quand il ne code pas du logiciel libre, il est ingénieur dans la société Senseor.

```
// compiler avec -I/home/jmfriedt/rtems/jmf/arm-rtems4.9/nds/lib/include/libnds/

#include <bsp.h>
#include <stdlib.h>
#include <stdio.h>
#include <nds/memory.h>

rtems_id timer_id;
uint16_t l=0;

void callback()
{ printk("Callback %x\n",l);
  (*(volatile uint16_t*)0x08000000)=l;
  l=0xffff-l;
  rtems_timer_fire_after(timer_id, 100, callback, NULL);
}

rtems_task Init(rtems_task_argument ignored)
{ rtems_status_code status;
  rtems_name timer_name = rtems_build_name('C','P','U','T');

  printk( "\n\n*** HELLO WORLD TEST ***\n" );
// cf rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/source/arm9/rumble.c
// sysSetCartOwner(BUS_OWNER_ARM9);
// defini dans rtems-4.9.1/c/src/lib/libbsp/arm/nds/libnds/include/nds/memory.h
  (*(vuint16*)0x04000204) = ((*(vuint16*)0x04000204) & ~ARM7_OWNS_ROM);

  status = rtems_timer_create(timer_name,&timer_id);
  rtems_timer_fire_after(timer_id, 1, callback, NULL);
  rtems_stack_checker_report_usage(); // requires #define CONFIGURE_INIT

  printk( "*** END OF HELLO WORLD TEST ***\n" );
  while(1)
    ;
  exit( 0 );
}

/* configuration information */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

/* configuration information */
#define CONFIGURE_MAXIMUM_DEVICES                     40
#define CONFIGURE_MAXIMUM_TASKS                       100
#define CONFIGURE_MAXIMUM_TIMERS                      32
#define CONFIGURE_MAXIMUM_SEMAPHORES                  100
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES              20
#define CONFIGURE_MAXIMUM_PARTITIONS                  100
#define CONFIGURE_MAXIMUM_REGIONS                     100

/* This seetings overwrite the ones defined in confdefs.h */
#define CONFIGURE_MAXIMUM_POSIX_MUTEXES               32
#define CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES   32
#define CONFIGURE_MAXIMUM_POSIX_KEYS                  32
#define CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS        10
#define CONFIGURE_MAXIMUM_POSIX_THREADS               128
#define CONFIGURE_MAXIMUM_POSIX_TIMERS                10
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS      200

#define STACK_CHECKER_ON
#define CONFIGURE_INIT

#include <rtems/confdefs.h>

/* end of file */
```

TAB. 5 – Programme pour faire clignoter les diodes connectées au 74HC574 inséré dans le Rumble Pack à la place du moteur.

Tab. 6 – Exemple de pthread sous DSLinux.

Tab. 7 – Exemple de pthread sous RTEMS. L'initialisation du framebuffer (code Tab. 4) et l'affichage de la fractale censée ralentir les divers threads sont omis pour fournir un code plus concis.