LEPTON, UN SYSTÈME D'EXPLOITATION TEMPS RÉEL POUR LES SYSTÈMES ENFOUIS

par Jean-Jacques PITROLLE

Je vous propose de découvrir la mise en œuvre d'un petit système d'exploitation POSIX qui comme d'habitude a ses petites spécificités. Grâce à sa taille, il permet de maîtriser tous les aspects logiciel embarqué : du démarrage de la carte à l'exécution de l'application, en passant par les appels système et la gestion des interruptions. Néanmoins, cet article n'a pas vocation à présenter ces aspects. Il va se concentrer sur la mise en œuvre de Lepton sur la machine hôte et sur une carte à base de cœur Cortex-M4, la Freescale Kinetis K60.

Dans le sillage de la course à la puissance et à la performance quantitatives que se livrent les fondeurs des mondes PC et *smartphone*, les microcontrôleurs 32 bits à faible empreinte mémoire intègrent de plus en plus de fonctionnalités et de mémoire. Ce phénomène se caractérise par :

- une convergence fonctionnelle entre les domaines critiques et grand public ;
- une augmentation continue du coût du logiciel embarqué par rapport au coût total du produit final.

Le rapport ministériel de D. Potier **[POT]** met en avant ce point crucial en employant le terme « softwareisation ». En définitive, ce document affirme qu'il devient fondamental dans le domaine des systèmes embarqués de disposer de briques logicielles génériques réutilisables, adaptables, pérennes et mutualisables.

Le noyau Linux **[KERN]** est un exemple qui correspond parfaitement aux caractéristiques que nous venons d'énoncer. En effet, ce noyau a fédéré un ensemble de partenaires tels que les fondeurs, industriels, universitaires, fondations, sociétés de services, indépendants dont les intérêts divergents et antagonistes ont pu converger pour atteindre ce degré de maturité.

Le noyau Linux et les différentes formes de distribution occupent une place de plus en plus importante dans le domaine des systèmes embarqués. Cette percée se manifeste depuis quelques années au travers des produits grand public et industriels (smartphones, boîtiers ADSL, ...).

Par ailleurs, pour les systèmes à faible empreinte mémoire (quelques dizaines de Mio de mémoire RAM et FLASH interne/externe) et ne disposant pas de module matériel nécessaire (MMU principalement), uClinux est une alternative crédible. De récentes initiatives (portage sur STM32 **[STM32]** et K70 **[K70]**) montrent que ce sous-ensemble de GNU/Linux bénéficie encore du regard intéressé des fondeurs.

Néanmoins, pour des systèmes à très faible empreinte mémoire ou enfouis (mémoire de l'ordre du Mio), un uClinux fonctionnel et permettant l'exécution d'une application complète (acquisition, réseau, mesures, ...) n'est envisageable qu'en ajoutant de la mémoire externe au système.

Quelques systèmes d'exploitation commerciaux et *open source*/libres tentent de fournir des solutions pour répondre à cette contrainte de compacité mémoire. Bien d'autres exigences comme la fiabilité temporelle (temps réel), la « maintenabilité », les fonctionnalités disponibles, la maturité peuvent être fournies par ces systèmes.

1 Présentation de Lepton

Cette partie décrit de manière synthétique le système d'exploitation temps réel POSIX Lepton. Elle expose les fonctionnalités et l'architecture générale.

1.1 Buts

Lepton est un système d'exploitation destiné aux systèmes embarqués enfouis. En grec, Lepton signifie léger. Il propose une approche modulaire et « applicative » pour les systèmes à ressources limitées (quelques centaines de Ko de RAM et de FLASH). L'utilisation de ce système d'exploitation tend à pousser la réutilisation de briques logicielles au maximum tout en fournissant un environnement suffisamment souple pour répondre aux problématiques complexes et variées rencontrées dans le domaine embarqué.

Lepton est un RTOS modulaire, fiable, compréhensible et maintenable.

Afin de disposer de la multitude et de la richesse des applications et des bibliothèques du monde UNIX, Lepton permet de porter et de créer des programmes et des bibliothèques utilisant la norme POSIX **[POSIX]**. Toutes les fonctions de la norme ne sont pas disponibles (fort heureusement pour l'empreinte mémoire), mais les plus communes sont présentées pour faciliter et garder l'approche « applicative ». Il est ainsi possible de créer des processus (ou des pseudo-processus) grâce aux appels système `vfork` et `exec`, contrairement à d'autres systèmes similaires comme RTEMS.

Lepton fournit au développeur de systèmes enfouis le moyen de gérer la complexité croissante des applications embarquées. À l'aide de la norme POSIX, Lepton propose principalement :

- une abstraction permettant de s'affranchir de la nature de chaque module ;
- une standardisation pour interagir de manière uniforme entre chaque module.

D'une manière plus générale, les objectifs principaux de Lepton dans les systèmes enfouis sont :

- le développement de briques logicielles ;
- l'accueil de briques logicielles ;
- la réutilisation et l'amélioration de ces briques logicielles ;
- la mutualisation des ressources afin de fournir ces briques logicielles.

Concrètement, ce sont plus de 150 fonctions de la norme POSIX qui sont à la disposition du développeur pour concevoir, structurer et développer son application embarquée. En adoptant une approche plus fonctionnelle, Lepton propose :

- la création et la synchronisation des processus et des *threads* ;
- l'accès aux périphériques de manière simple et standard ;
- la notion de *streams* afin de maximiser la réutilisation logicielle ;
- le support du réseau et de l'API BSD *socket* (LWIP) ;
- des systèmes de fichiers ;
- des bibliothèques graphiques ;
- des services réseau tels qu'un serveur web et un serveur FTP.

Lepton est le système d'exploitation de produits commerciaux tels que :

- le contrôleur d'installation 6116 de Chauvin-Arnoux **[CA6116]** ;
- l'oscilloscope portable 5022 de Métrix **[MT5022]**.

1.2 Architecture

Lepton est un système d'exploitation POSIX (1003.1 a/c) articulé autour d'un micro-noyau enrichi temps réel créé par M. LE BOULANGER. Il a longtemps été un logiciel maintenu en interne, puis en accord avec la direction de Chauvin-Arnoux, il a été publié en licence open source MPL 1.0 en décembre 2011.

1.2.1 Architecture générale

Initialement, le système d'exploitation Lepton reposait sur le noyau temps réel propriétaire édité par Seggeer embOS **[EMBOS]**. Le noyau de Lepton fournissait ses services au travers d'un thread Segger. Le noyau temps réel, permettait (et permet toujours) de disposer de mécanismes de création et de synchronisation de tâches et d'un accès basique au matériel. Cette architecture prenait la forme d'un micro noyau enrichi.

Le coût prohibitif des licences de développement du noyau temps réel embOS et un support parfois difficile ont naturellement conduit M. LE BOULANGER à étudier des solutions de remplacement. Son choix se porta sur le noyau du système d'exploitation eCos.

eCos est un système d'exploitation temps réel à part entière édité par la société ecoscentric (anciennement cygnus). Il est disponible sous 2 formes :

- une version communautaire dont le code source est librement accessible à l'adresse http://ecos.sourceware.org ;
- une version « pro » maintenue par ecoscentric dont le support est payant **[ECOSPRO]**.

Depuis 1999, l'année de son apparition, il a démontré des gages de qualité, de fiabilité et de maturité qui ont permis la réalisation de nombreux projets allant des équipements réseau aux systèmes aéronautiques **[ECOSEX]**.

La licence appliquée aux sources de ce système d'exploitation est proche de GPL. La différence notable se situe au niveau de l'édition de lien. En effet, une application eCos se présente sous forme de micrologiciel (ou *firmware*), c'est-à-dire

que le code applicatif et le noyau ne forment qu'un seul et même binaire.

La mutation de Lepton en système d'exploitation open source s'est opérée en donnant la possibilité d'utiliser comme noyau temps réel la version communautaire de eCos.

1.2.2 Fonctionnement général

Les systèmes d'exploitation « classiques » utilisent le mécanisme de mémoire virtuelle pour cloisonner les actions que peuvent effectuer différentes parties du système. De ce fait, les problèmes que peuvent causer les applications sont généralement limités : chaque action touchant au matériel ou à d'autres parties essentielles du système doit faire l'objet d'une demande au noyau.

Le noyau gère les demandes de façon synchrone ou asynchrone suivant les interfaces présentées à la partie applicative.

Les systèmes pour lesquels Lepton est le plus adapté ne possèdent pas de contrôleur gérant la mémoire virtuelle (MMU) même si certains types de contrôleurs remplissent des fonctions élémentaires de protection mémoire (MPU).

En attendant de pouvoir exploiter cette dernière possibilité, Lepton conserve cette séparation conceptuelle entre la partie dite applicative et la partie noyau. Chaque processus ou thread souhaitant dialoguer avec les périphériques ou créer un IPC doit effectuer une requête au noyau.

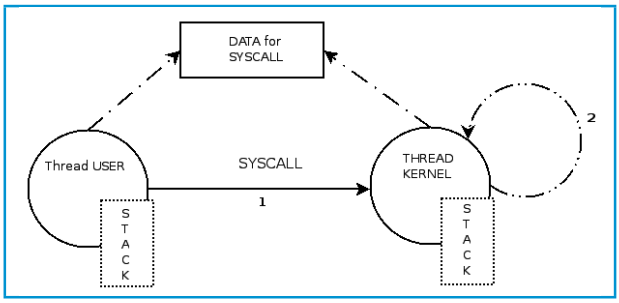

Illustration 1 : Dans cette configuration, un thread noyau répond aux requêtes des threads utilisateurs. Chaque thread ne possède qu'une seule pile.

Le couple Lepton-embOS utilise un thread noyau dont le but est de satisfaire les requêtes des threads ou processus applicatifs.

Chaque thread possède une pile « applicative » et les données sont disponibles pour le thread noyau à travers un pointeur de la structure du thread utilisateur. Le thread appelant émet un signal (étape 1) pour demander un service au thread noyau. Celui-ci récupère les informations nécessaires pour remplir la demande et l'exécute (étape 2). Le thread noyau stocke le résultat de la requête dans la structure du thread appelant et lui rend la main.

Ce mécanisme d'appel système est relativement simple à mettre en œuvre. Il permet de sérialiser les demandes au noyau et limite la taille de pile nécessaire pour chaque thread. Néanmoins, seul un thread utilisateur peut effectuer un appel système ce qui, dans certains cas, conduit à une exécution moins fluide du système.

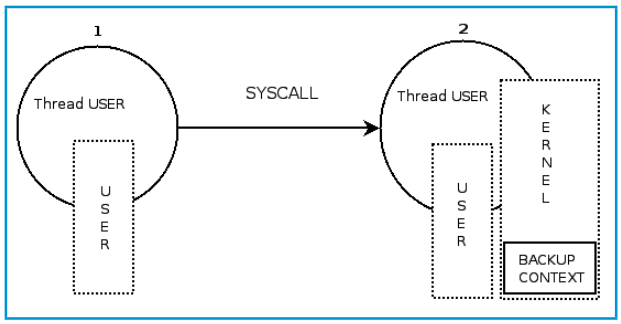

Illustration 2 : Chaque thread possède sa propre pile noyau. Il serait alors possible d'avoir plusieurs appels système concurrents.

Le couple Lepton-eCos (Tauon) définit une pile noyau supplémentaire pour chaque thread. Lorsqu'un thread effectue un appel système, il sauvegarde son contexte courant dans sa pile utilisateur et réalise la requête noyau sur sa propre pile noyau. L'épilogue de l'appel système restaure le contexte sauvegardé et déroute à nouveau le flux d'exécution, cette fois pour reprendre à l'instruction suivant l'appel système.

Cette manière de réaliser des appels système permettrait un meilleur entrelacement des chemins possibles et, à terme, l'exécution de plusieurs appels système simultanés. La contrepartie est évidemment une augmentation de la consommation mémoire : il faut désormais une pile noyau par thread. Pour l'instant, la version Lepton-eCos ne permet pas de réaliser plusieurs appels système simultanément.

De par sa conception, le système d'exploitation Lepton est portable. En effet, si l'architecture cible a été portée sur le noyau temps réel intégré à Lepton, le portage peut être rapide. Avec le noyau temps réel embOS de Segger et un pilote de périphérique basique pour UART, les portages M16 vers ARM7 et ARM7 vers ARM9 ont pris moins d'une journée.

Avec eCos, si le paquet d'une plate-forme similaire existe déjà, le portage peut se réaliser assez simplement aussi. Il faudra toutefois compter un peu plus de temps si l'architecture ou le processeur ne sont pas supportés par eCos.

De plus, ce système peut être adaptable à n'importe quel noyau embarqué à faible empreinte mémoire. On peut tout à fait envisager de remplacer les noyaux embOS et eCos par un noyau dont les mécanismes internes peuvent être plus familiers, comme RTEMS ou FreeRTOS.

2 Pratique

Nous verrons à travers les lignes qui suivent les étapes menant vers une installation Lepton fonctionnelle. Dès la fin de cette partie, nous disposerons d'une application Lepton débogable sur notre machine hôte.

2.1 Installation

La procédure d'installation a été validée sur une Debian Squeeze 32 bits disposant d'un serveur X fonctionnel. Elle devrait fonctionner correctement sur les clones Debian (ubuntu, knoppix, ...).

Les différentes commandes seront exécutées sous l'utilisateur kvm.

2.1.1 Installation de eCos

La première étape consiste à installer toutes les dépendances nécessaires à la construction et à l'utilisation des outils eCos.

```
#aptitude install build-essential libpng-dev libjpeg-dev libtiff-dev
tcl8.4-dev tk8.4-dev
```

Nous créons ensuite le répertoire **ecos-v3.0** qui contiendra tous les éléments temporaires et permanents de eCos. Ainsi, les répertoires temporaires **archives** et **ecos-build** permettent respectivement de stocker les fichiers téléchargés et de construire l'utilitaire de configuration de eCos. Ces derniers peuvent aussi être créés dans un répertoire de construction de son choix.

```
$ mkdir archives ecos-tools ecos-build
```

Deux choix sont envisageables pour récupérer le code source de eCos :

- l'utilisation d'une archive, générée à partir de la version courante, située à l'adresse http://hg-pub.ecoscentric.com/ecos ;
- le clonage du dépôt mercurial. Dans ce cas, l'installation préalable de mercurial est nécessaire.

Clonons le dépôt à l'aide la commande suivante :

```
$ hg clone http://hg-pub.ecoscentric.com/ecos/ ecos_hg
$ ln -s ecos_hg ecos
```

Utiliser un lien symbolique dans ce cas permet à terme de disposer de plusieurs dépôts eCos en provenance de différentes sources (versions 3.0 ou 2.0 « tagguées »).

Il faut maintenant compiler l'outil **ecosconfig** qui permet de créer, d'adapter et de modifier des configurations à partir de la ligne de commandes. Pour certains utilisateurs, il peut s'avérer utile ou bien plus pratique que l'outil graphique.

```
$ cd ecos-build
$ ../ecos/host/configure --prefix=/home/kvm/dev/ecos-v3.0/ecos-tools
--with-tcl=/usr --with-tcl-version=8.4
$ make ; make install
```

Une installation correcte fournira entre autres le binaire **ecosconfig** dans le répertoire **~/dev/ecos-v3.0/ecos-tools/bin**.

L'utilisation de l'outil graphique **configtool** simplifie la création, la modification et la génération de configurations pour une plate-forme supportée par eCos. L'installation manuelle est décrite à l'adresse http://www.ecoscentric.com/devzone/configtool.shtml.

Afin de disposer rapidement de cette interface, nous pouvons la télécharger et la décompresser grâce aux lignes suivantes :

```
$ cd ~/dev/ecos-v3.0/archives
$ wget ftp://ftp.mirrorservice.org/sites/sources.redhat.com/pub/ecos/
anoncvs/ecos-tools-bin-110209.i386linux.tar.bz2
$ tar jxf archives/ecos-tools-bin-110209.i386linux.tar.bz2
```

Nous pouvons supprimer tous les fichiers, sauf le fichier **configtool** que nous devons déplacer dans le répertoire **ecos-tools/bin**.

```
$ rm -rf ecosconfig ser_filter platforms.tar
$ mv configtool ../ecos-tools/bin
```

Nous pouvons désormais lancer l'application **configtool** et compléter la boîte de dialogue avec le chemin de notre dépôt eCos.

```
$ cd ~/dev/ecos-v3.0/ecos-tools/bin
$ ./configtool &
```

Nous remplissons la boîte de dialogue pour qu'elle pointe vers le chemin d'accès **/home/kvm/dev/ecos-v3.0/ecos/packages**.

Pour fonctionner correctement, les différents outils de manipulation de dépôt eCos ont besoin de la variable d'environnement **ECOS_REPOSITORY** : éditons le fichier **.bashrc** afin de l'exporter.

```
export ECOS_REPOSITORY="/home/kvm/dev/ecos-v3.0/ecos/packages"
```

Le script **ecosadmin.tcl** sera nécessaire lors de l'installation des paquets eCos fournis par Lepton. Initialement, ce fichier ne dispose pas des droits d'exécution, donc nous modifions cet attribut :

```
$ cd ..
$ chmod +x ecos/packages/ecosadmin.tcl
```

Si nous souhaitons lancer les outils de configuration de eCos à partir de n'importe quel endroit du système de

fichiers, nous ajoutons le chemin d'accès à la variable d'environnement **PATH**.

Modifions notre **.bashrc** comme suit :

```
export PATH="$PATH:/home/kvm/dev/ecos-v3.0/ecos-tools/bin"
```

Les outils **ecosconfig** et **configtool** permettent de créer et de modifier une configuration eCos pour une plateforme particulière. À partir de ce fichier de configuration, une bibliothèque statique spécifique est générée pour la plateforme souhaitée. La génération de cette bibliothèque nécessite une chaîne de compilation croisée pour l'architecture visée. Plusieurs choix s'offrent à nous :

- générer une chaîne de compilation « from scratch » ;
- utiliser une chaîne de compilation pré-compilée ;
- utiliser un outil de génération de chaîne de compilation.

Dans notre contexte, le but est de disposer d'une chaîne de compilation croisée le plus rapidement possible afin de pouvoir compiler et tester une application minimale.

eCos fournit des chaînes de compilation croisées pré-compilées pour différentes architectures dont celles qui nous intéressent **arm-eabi** et **i386-elf**. Il s'agit de la version 4.3.2 de gcc.

Exécutons les commandes suivantes :

```
$ mkdir -p ~/dev/toolchains
$ cd ~/dev/toolchains
$ wget --passive-ftp ftp://ecos.sourceware.org/pub/ecos/ecos-install.tcl
$ tclsh ecos-install.tcl -t
```

Choisissons le site souhaité pour le téléchargement et entrons **/home/kvm/dev/toolchains** comme chemin de stockage pour les chaînes de compilation croisées pré-compilées.

Tapons 1 (arm-eabi) et 3 (i386-elf) puis q.

Une fois cette étape terminée, nous disposons désormais d'un répertoire **gnutools** contenant les répertoires pour les deux architectures sélectionnées.

Ajoutons les chemins des binaires des chaînes de compilation croisées à notre **.bashrc**.

```
export PATH="$PATH:/home/kvm/dev/toolchains/gnutools/arm-eabi/bin"
export PATH="$PATH:/home/kvm/dev/toolchains/gnutools/i386-elf/bin"
```

Nous sommes maintenant prêts à installer Lepton.

2.1.2 Installation de Lepton

Les paquets que nous devons installer sont les suivants : **scons**, **libexpat-dev**, **libgtkgl2.0-dev**, **minicom**, **gdb**.

Créons un répertoire qui stockera une copie du dépôt Lepton et clonons-le.

```
$ mkdir ~/dev/tauon
$ cd ~/dev/tauon
$ hg clone https://code.google.com/p/lepton/
$ cd lepton
```

La commande suivante crée un lien symbolique dans notre répertoire personnel :

```
$ sh tauon_ln.sh
```

Nous allons procéder à une installation automatique de Lepton. Ce processus sera décrit plus en détail dans la partie « Création d'une application minimale », mais brièvement, il exécute les étapes suivantes :

- compilation de l'outil de configuration de Lepton **mklepton** ;
- compilation de l'outil pour la simulation **virtual_cpu** ;
- intégration des paquets correspondant aux cibles AT91SAM9261-EK, TWR-K60N512 et SYNTHETIC à la base eCos ;
- génération des bibliothèques pour les cibles correspondantes ;
- génération des fichiers de configuration Lepton à l'aide de **mklepton** puis compilation d'un firmware de test Lepton pour les trois cibles précédemment citées.

Nous pouvons maintenant lancer la compilation des trois firmwares (un par cible) à l'aide des commandes suivantes :

```
$ cd ~/tauon/tools/config
$ scons BUILD_MKLEPTON=True BUILD_VIRTUALCPU=True BUILD_SAMPLEAPP=True
```

Au bout de quelques instants, trois binaires seront disponibles dans le répertoire **~/tauon/sys/user/tauon_sampleapp/bin** : **tauon_synthetic.elf**, **tauon_at91sam9261.elf** et **tauon_k60n512.elf**.

2.1.3 Test en simulation

Le binaire généré correspond au fichier **tauon_synthetic.elf**. Le firmware contient quelques applications dont un *shell* accessible grâce à un terminal série comme minicom.

Copions le fichier de configuration fourni dans le répertoire de configuration de minicom :

```
# cp /home/shiby/tauon/sys/root/prj/config/minicom/minirc.tauonpt /etc/minicom/
```

Lançons le script **~/tauon/tools/host/debian/scripts/build_fifo.sh**. Il crée des tubes et un segment de mémoire partagée permettant la communication entre le simulateur **virtual_cpu** et le firmware simulé. Les interruptions sont simulées au moyen de signaux classiques.

Ouvrons deux terminaux, l'un pour l'exécution du firmware simulé, l'autre pour disposer d'un minicom.

```
(Lepton_firmware) $ cd ~/tauon/sys/user/tauon_sampleapp/bin
(Lepton_firmware) $ ./tauon_synthetic.elf
(minicom)$ minicom -o tauonpt
```

Dans la console minicom, un shell dont l'invite de commandes se nomme **lepton#2$** devrait apparaître.

Nous pouvons aller dans les répertoires **/usr/bin** et **/usr/sbin** pour consulter les binaires disponibles en utilisant la commande **ls -l**.

```
$Revision: 1.3 $ $Date: 2009-06-18 13:43:22 $

version 4.0.0.0 kernel compilation date: Aug 21 2012 - 16:27:02
Sun Oct 14 14:59:48 2012

type ctrl-x: .init script not run or any key to continue
lepton start!

       lepton shell

lepton#2$ ls -l /usr/bin
ls /usr/bin
-rwxrwxrwx      8  Aug 21 14:30 test
drwxrwxrwx     64  Aug 21 14:30 net

total 2
lepton#2$
```

Illustration 3 : Ceci est la console Lepton obtenue en appelant minicom. La commande suivante liste une partie des binaires accessibles.

2.2 Création d'une application minimale pour TWR-K60N512 et simulation

Cette partie tentera de fournir un maximum d'éléments pour créer une application utilisant le système d'exploitation Lepton. L'application fonctionnera aussi bien en simulation que sur une cible TWR-K60N512 (Cortex-m4). Elle ne disposera pas d'interface graphique et fera appel à l'API socket BSD.

2.2.1 Structure « sur disque » d'une application

Le répertoire **tauon_sampleapp** servira de modèle pour créer notre application minimale. Les répertoires suivants apparaissent lors de l'accès à ce dossier :

- **bin** : il contient les firmwares générés quelle que soit la cible sélectionnée. De plus, il stocke les fichiers « disque » (sdcard, mémoires flash, eeprom, ...) utilisés par le firmware simulé.
- **data** : il possède les données qui seront intégrées au micrologiciel ; dans le cas de notre application test, les pages HTML et les images qu'elles contiennent.
- **etc** : tous les fichiers de configuration se situent dans ce répertoire. Ils seront détaillés dans la partie mklepton.
- **etc/scripts** : ces fichiers permettent de déboguer le firmware généré à l'aide de gdb et de OpenOCD pour une cible physique.
- **hal** : il contient les fichiers spécifiques liés à une architecture matérielle particulière dont le support peut être partiellement proposé par eCos. Dans le cas de la carte Freescale (**board_freescale_twrk60n512**), une partie de l'initialisation a été modifiée pour fournir une exécution en RAM externe (MRAM). Pour supporter la carte Atmel (AT91SAM9261-EK), un travail plus important a dû être réalisé en s'inspirant du port non officiel de l'AT91SAM9263-EK.
- **obj** : les fichiers objets de la partie applicative seront enregistrés dans ce répertoire. Ils dépendent de l'architecture cible choisie pour le micrologiciel généré.
- **prj/scons** : tous les fichiers nécessaires à la construction d'une application sont présents dans ce dossier. Le fichier SConscript contient les éléments propres à l'application (fichiers à compiler, drapeaux particuliers par fichier, ...). Le lien symbolique SConstruct est commun à toutes les applications. Il définit les paramètres de construction généraux en fonction des cibles supportées. Les fichiers **.py** sont propres à chaque cible et contiennent les options modifiables accessibles à l'utilisateur.
- **src** : tous les fichiers source de votre application seront situés dans ce dossier. Les fichiers source peuvent être aussi bien des pilotes de périphériques (**src/dev**) que des fichiers applicatifs (**src/test.c**) ou même des scripts Lepton (**src/sh**).

En résumé, cette organisation n'est ni obligatoire et ni définitive. Pour l'instant, elle fournit une vision suffisamment adaptée pour la création de firmware à l'aide de Lepton. Tout développeur sera libre de choisir la structure qui lui paraît la plus appropriée.

2.2.2 Outils disponibles

2.2.2.1 mklepton

Cet outil permet de générer des fichiers de configuration nécessaires à la compilation du micrologiciel. Ces fichiers sont stockés à un emplacement fixe dépendant de l'architecture.

Les options de configuration qui peuvent être paramétrées à l'aide de mklepton sont nombreuses :

- nombre maximum de processus ;
- nombre maximum de fichiers ouverts ;
- pilotes de périphérique disponibles ;
- pseudo-binaires disponibles ;
- points de montage ;
- binaires à lancer au démarrage ;
- fichiers à stocker.

2.2.2.2 virtualcpu

eCos donne la possibilité de simuler une application à l'aide de la cible synthetic **[SYNTH]**. Un processus utilisateur GNU/Linux simule les entrées/sorties et dialogue, à l'aide de tubes et de signaux, avec l'application eCos simulée. Il est possible d'étendre cette cible (ajout de pilotes) à l'aide du couple TCL/tk.

Pour mieux comprendre ce processus de simulation et l'adapter à nos besoins, nous avons développé un petit simulateur nommé virtualcpu couplé à une cible synthetic modifiée. Cet ensemble permet de créer des pilotes de périphérique simulés et des boîtiers simulés uniquement à l'aide du langage C. De plus, la partie propre à eCos est bien circonscrite et peut être adaptée, à terme, pour la simulation d'une application fonctionnant sur un autre système d'exploitation temps réel.

2.2.3 Mise en place

Afin de faciliter les mises à jour noyau, le dossier contenant l'application utilisateur est incorporé à l'arbre des sources Lepton à l'aide d'un lien symbolique ; ainsi, ce répertoire applicatif peut se situer sur n'importe quel point de montage accessible à notre système.

Si nous souhaitons développer une application dont le répertoire racine est **tauon_myapp**, nous créons dans un dossier de notre choix le répertoire **tauon_myapp** et les sous-répertoires « standards ».

```
$ cd ~/dev
$ mkdir -p ~/dev/tauon_apps/tauon_myapp
$ cd ~/dev/tauon_apps/tauon_myapp
$ mkdir -p bin etc/scripts hal obj prj/scons src
```

Pour inclure le répertoire **tauon_myapp** dans l'arbre des sources de Lepton, il faut exécuter le script **bin_ln.sh** situé dans **sys/user**.

```
$ cd ~/tauon/sys/user
$ sh bin_ln.sh ~/dev/tauon_apps/tauon_myapp/
```

Les lecteurs attentifs remarqueront l'utilisation du répertoire prj/scons. Le système de construction utilisé n'est pas GNU make mais **SCons**. Ce logiciel open source permet de construire des applications à l'aide du langage haut niveau Python. Lepton fournit un fichier de construction global que chaque projet d'application Lepton doit utiliser. Il met à disposition aussi bien les options de compilation générales (optimisation générale du noyau, ...) que les activations de modules (compilation de la pile IP ou de la pile graphique).

```
$ cd ~/tauon/sys/user/tauon_myapp/prj/scons
$ ln -s /home/shiby/tauon/sys/user/build/SConstruct
```

L'utilisateur éditera un fichier **SConscript** contenant les spécificités de son projet (sources à compiler, optimisations à appliquer, ...). Un module python permet de simplifier l'ajout d'un fichier source, la création d'une bibliothèque ou la compilation finale de l'application. Pour en disposer, il faut créer ou modifier la variable d'environnement **PYTHONPATH**.

```
$  export PYTHONPATH="$PYTHONPATH:$HOME/tauon/sys/root/prj/scons/
common/module/"
```

Il ne faut pas oublier d'insérer cette variable au fichier **.bashrc** pour en disposer constamment.

Pour vérifier le bon fonctionnement de cette étape, on peut taper :

```
$ scons -h
```

Les options disponibles devraient apparaître.

2.2.3.1 Cible simulation

Il faut désormais intégrer le fichier de construction spécifique à la partie utilisateur. Pour cela, il faut copier les fichiers **SConscript** et **synthetic_opts.py** du projet **tauon_sampleapp** :

```
$ cp /home/kvm/tauon/sys/user/tauon_sampleapp/prj/scons/
{SConscript,synthetic_opts.py} .
```

La carte d'évaluation AT91SAM9261-EK ne sera pas supportée par ce projet ; on peut donc supprimer toute référence à cette cible dans le fichier SConscript. Néanmoins, nous gardons les fichiers liés à la kinetis (TWR-K60N512).

L'application que nous construisons n'utilisera ni le *toolkit* FLTK (FLNX pour être plus précis), ni le mini-serveur X nanoX. Les lignes y faisant référence peuvent être effacées du SConscript. De plus, nous pouvons supprimer le pseudo-binaire **tstsd** qui ne sera pas embarqué dans notre micrologiciel.

Pour gagner en clarté, renommons toutes les sous-chaînes **tauon_sampleapp** par **tauon_myapp**.

Récupérons le fichier de configuration **mkconf_tauon_sampleapp_gnu_simple_k60.xml** et enregistrons-le dans **tauon_myapp/etc**.

```
$ cd ../../etc
$ cp /home/shiby/tauon/sys/user/tauon_sampleapp/etc/ {mkconf_tauon_
sampleapp_gnu_k60.xml, .init_with_net_k60,.mount} .
```

Le fichier XML décrit le contenu du firmware. Décrivons un peu son architecture :

- Tout fichier de configuration commence par la balise **<mklepton>**.
- Les deux premières balises **<target>** définissent les architectures pour lesquelles un firmware sera généré. La balise **<target>** placée avant ou après certaines balises bien spécifiques permet de spécialiser le comportement.

- **<device>** fournit les pilotes supportés par toutes les architectures. Précédée (ou suivie) de **<target>**, elle inclura les pilotes spécifiquement pour l'architecture souhaitée. Ajoutons par exemple un périphérique série utilisant un pseudo-terminal de l'hôte plutôt qu'une liaison physique et se nommant **/dev/ttypt**.

```
<target name="gnu32_lepton">
   <devices>
      ...
      <!-- /dev/sdcard0 -->
      <dev name="dev_linux_sdcard_map" use="on"/>
      <!-- /dev/ttypt -->
      <dev name="dev_linux_compt_map" use="on"/>
   </devices>
</target>
```

- **<mount>** définit le type et le point de montage de périphériques bloc. Changeons le chemin du fichier **.mount** qui sera utilisé.

```
<mount dest_path="$(HOME)/tauon/sys/user/tauon_myapp/etc">
```

- **<boot>** précise le premier binaire qui sera exécuté après l'initialisation du noyau Lepton. Pour la cible simulation, modifions cette section comme suit :

```
<target name="gnu32_lepton">
   <boot dest_path="$(HOME)/tauon/sys/user/tauon_myapp/etc">
      <command arg="initd -i /dev/ttypt -o /dev/ttypt" />
   </boot>
</target>
```

Ainsi, nous utiliserons une liaison série simulée à l'aide d'un pseudo-terminal plutôt qu'une liaison physique.

- **<binaries>** fournit les pseudo-binaires embarqués dans le micrologiciel. On peut préciser le chemin à travers les paramètres de la balise.

- **<files>** permet d'embarquer des fichiers texte dans le rootfs. 3 fichiers sont réellement importants en dehors de ceux que l'utilisateur peut fournir.

- **.init** est un fichier qui sera exécuté par le shell lepton avant l'invite de commandes. On peut par exemple initialiser une carte réseau à l'aide de **ifconfig** ou lancer un service tel que **telnetd**. Pour l'instant, commentons les lignes du fichier **.init_with_net_k60**.

```
#net/ifconfig addif 192.168.1.2 192.168.1.1
#mount /dev/sd /dev/sdhc /dev/hd/sdhc0
```

- **.boot** est un fichier qui reflète la balise **<boot>**. Il sera appelé au démarrage du noyau.

- **.mount** définit les points de montage des périphériques de stockage.

N'oublions pas de modifier les chemins dans ces balises pour que les fichiers générés appartiennent au répertoire **tauon_myapp**.

Copions le fichier de débogage spécifique à la cible synthetic. Il permet d'empêcher l'arrêt du débogage sur le signal SIGUSR1.

```
$ cd scripts
$ cp /home/shiby/tauon/sys/user/tauon_sampleapp/etc/scripts/gdb_script_
synth.gdb  .
```

Récupérons un fichier source fourni par le projet **tauon_sampleapp**, par exemple **test.c** :

```
$ cd ../../src
$ cp /home/shiby/tauon/sys/user/tauon_sampleapp/src/test.c
```

2.2.4 Compilation et débogage

On peut désormais compiler la bibliothèque eCos pour notre cible synthetic modifiée. Cette bibliothèque sera liée statiquement à notre code applicatif.

```
$ cd ~/tauon/sys/root/lib/arch/synthetic/x86
$ sh script_ecos_synth.sh
```

La bibliothèque générée se trouve dans le répertoire **install/lib** et se nomme **libtarget.a**. Nous pouvons consulter la totalité des symboles présents dans cette bibliothèque à l'aide de la commande **nm**. Le fichier **target.ld** permet de maîtriser totalement l'édition de lien. Dans le cas de la cible synthetic modifiée par nos soins, nous autorisons 16 Mio de mémoire ROM (lecture seule) et 16 Mio de mémoire RAM (lecture/écriture).

Le répertoire **tauon/sys/root/lib/arch** contient pour l'instant toutes les familles de processeurs supportées par Lepton. Les bibliothèques statiques produites pour chaque carte spécifique seront stockées en suivant le même modèle que pour la cible synthetic.

On peut désormais extraire les informations de notre fichier de configuration XML et les intégrer à la version de notre noyau courant.

```
$ cd ~/tauon/tools/bin
$ ./mklepton_gnu.sh -t gnu32_lepton ~/tauon/sys/user/tauon_myapp/etc/
mkconf_tauon_sampleapp_gnu_k60.xml
```

L'option **-t** précise la famille de processeurs pour laquelle les fichiers de configuration sont générés. Plusieurs fichiers sont inclus dans le répertoire **~/tauon/sys/root/src/kernel/core/arch/synthetic/x86**. Décrivons un peu ces fichiers :

- **bin_mkconf.c** : contient le prototype des pseudo-binaires embarqués dans notre firmware. Il fournit aussi un tableau dont chaque élément décrit les informations présentes dans la balise **<binaries>** du fichier de configuration XML.

- **dev_dskimg.c** : stocke le système de fichiers racine sous forme de tableau. Il utilise comme système de fichiers un clone de FFS (ou de UFS à voir).
- **dev_mkconf.c** : liste tous les pilotes de périphérique présents dans le micrologiciel. Ces éléments correspondent à la section **<devices>**.
- **kernel_mkconf.h** : expose certaines définitions (nombre maximum de fichiers ouverts, ...) encadrées par la balise **<kernel>**.

Nous pouvons désormais compiler notre application en prenant soin de modifier les options qui nous concernent en supprimant celle liée à nanoX (et par la même FLNX) dans le fichier **synthetic_opts.py**.

```
#variable for user project
COMPILER_CC='i386-elf-gcc'
COMPILER_CXX='i386-elf-g++'
ARCHIVER='i386-elf-ar'
#ARCHIVE_INDEXER='randlib'
ARCH='CPU_GNU32'
PLATEFORME='synthetic/x86/'
TARGET_SUFFIX='synthetic'
OPTS_APP='-O0'
DEBUG_LEVEL='-gdwarf-2 -g3'
NANOX=0
FONTS_NANOX='FONT_VERA'
#FONTS_NANOX='FONT_MEDIUM,FONT_MEDIUM_JA,FONT_METRIX_SYMBOLS,FONT_
VERA,FONT_VERA_BD,FONT_ARIAL_UNICODE'
LWIP=1
OPTS_LWIP='-O2'
VERBOSE=0
STRIP=0
TINYGL=0
```

Pour supprimer tous les fichiers objets, les bibliothèques et les firmwares générés, nous utilisons les commandes suivantes :

```
$ cd ~/tauon/sys/user/tauon_myapp/prj/scons
$ scons -c
```

Pour compiler et stocker notre futur micrologiciel fonctionnant sur notre poste de développement dans le répertoire bin, nous utilisons la commande :

```
$ scons -Q bin
```

En listant le contenu du répertoire courant, nous pouvons observer :

- le binaire **tauon_synthetic.elf** qui est la copie originale du même fichier stocké dans le répertoire bin ;
- les bibliothèques avec lesquelles notre application est liée. Dans notre exemple, seules les bibliothèques statiques du noyau et de lwip sont présentes (respectivement **libkernel.a** et **liblwip.a**).

Le programme virtualcpu a besoin de tubes nommés pour dialoguer avec l'application eCos simulée. Ils sont créés à l'aide du script **build_fifo.sh** :

```
$ sh ~/tauon/tools/host/debian/scripts/build_fifo.sh
```

Nous pouvons ouvrir nos deux terminaux, l'un pour la console minicom et l'autre pour gdb.

```
(Lepton_firmware) $ cd ~/tauon/sys/user/tauon_myapp/bin
(Lepton_firmware) $ gdb -x ../etc/scripts/gdb_script_synth.gdb tauon_
synthetic.elf
(minicom)$ minicom -o tauonpt
```

La commande **minicom** doit être exécutée ***après*** le démarrage du programme dans la console gdb. Nous pouvons alors déboguer notre micrologiciel comme un processus GNU/Linux classique.

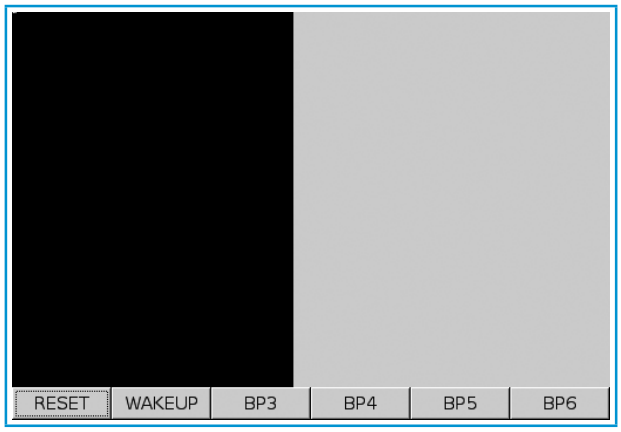

Illustration 4 : Cette interface représente fonctionnellement la carte d'évaluation AT91SAM9261-EK. Un écran 320x240 et 6 boutons sont disponibles. On peut ainsi simuler un large panel d'applications.

Il est aussi envisageable de changer l'interface du simulateur. En effet, celle-ci correspond fonctionnellement à la carte d'évaluation AT91SAM9261-EK. Les lecteurs intéressés peuvent consulter le répertoire **~/tauon/tools/virtual_cpu/ui**.

2.2.5 Ajout du réseau

2.2.5.1 Côté embarqué

La mise au point en simulation est réellement utile pour tester les éléments fonctionnels d'une application. Les communications réseau TCP/IP peuvent aussi être testées pour une application Lepton.

En premier lieu, il est nécessaire de fournir les pseudo-binaires embarqués **ifconfig** et **telnetd**. Le premier configure une carte réseau en lui attribuant une adresse IP, le second fournit un démon telnet. Éditons notre fichier de configuration XML (situé dans **tauon_myapp/etc**) en remplaçant :

```
<binaries src_path="bin/net" dest_path="bin/net">
      <bin name="ifconfig" stack="2048" priority="10" timeslice="1"/>
      <bin name="telnetd" stack="2048" priority="10" timeslice="1"/>
      <bin name="ftpd" stack="2048" priority="10" timeslice="1"/-->
</binaries>
```

par :

```
<binaries src_path="bin" dest_path="bin">
   <target name="gnu32_lepton">
        <!-- tools test-->
        <bin name="ifconfig" stack="8192" priority="10"
timeslice="1"/>
        <bin name="telnetd" stack="8192" priority="10"
timeslice="1"/>
   </target>
</binaries>
```

Les binaires sur l'architecture synthetic ont toujours une taille de pile minimale de 8 Kio car il semble y avoir des problèmes de sauvegarde de contexte avec une pile de taille inférieure.

L'idéal serait de pouvoir fixer une adresse IP dès le démarrage de notre application. Supprimons le commentaire de notre fichier d'init **.init_with_net_k60**.

```
net/ifconfig addif 192.168.1.2 192.168.1.1
#mount /dev/sd /dev/sdhc /dev/hd/sdhc0
```

Notre simulation aura pour adresse IP 192.168.1.2 et sa passerelle par défaut sera 192.168.1.1.

Générons à nouveau les fichiers de configuration à l'aide de **mklepton** :

```
$ cd ~/tauon/tools/bin
$ ./mklepton_gnu.sh -t gnu32_lepton ~/tauon/sys/user/tauon_myapp/etc/
mkconf_tauon_sampleapp_gnu_k60.xml
```

et reconstruisons notre application.

```
$ cd ~/tauon/sys/user/tauon_myapp/prj/scons
$ scons -c ; scons -Q bin
```

2.2.5.2 Côté hôte

Afin de disposer d'un maximum de souplesse pour les configurations réseau souhaitées (simulation sur le même réseau physique que l'hôte ou pas), nous allons utiliser les ponts réseau ou *bridge*. Vérifions que nous disposons des utilitaires **br** (paquet **bridge-utils** sous Debian) et **tunctl** (**uml-utilities**). Nous configurons un pont ne comprenant pas de carte physique dont l'adresse réseau sera 192.168.1.1. Une interface tap sans adresse IP sera ajoutée et fournira une interface directement accessible à l'application Lepton simulée.

```
# brctl addbr br0
# ifconfig br0 192.168.1.1 up
# tunctl -u root
```

La dernière ligne est un peu gênante : seul l'utilisateur root pourra accéder à cette interface. Concrètement, nous souhaitons pouvoir la manipuler à travers une socket RAW lue par le simulateur virtualcpu. Les paquets lus sont stockés dans une mémoire partagée qui sert de tampon de réception au périphérique réseau du micrologiciel Lepton simulé. Il serait intéressant d'utiliser les capacités pour permettre à un utilisateur classique de lire et d'écrire sur une socket RAW. Finissons notre configuration réseau en activant notre interface réseau virtuel et en l'ajoutant à notre pont :

```
# ifconfig tap0 0.0.0.0 promisc up
# brctl addif br0 tap0
```

Nous allons devoir exécuter notre firmware Lepton simulé ainsi que notre moniteur série minicom sous l'identité ROOT.

```
(Lepton_firmware) # cd ~/tauon/sys/user/tauon_myapp/bin
(Lepton_firmware) # gdb -x ../etc/scripts/gdb_script_synth.gdb tauon_
synthetic.elf
(minicom)# minicom -o tauonpt
```

Nous pouvons ouvrir une troisième console, une fois le débogage lancé et le moniteur série affichant l'invite Lepton, pour « pinguer » notre micrologiciel simulé.

```
(autre console) $ ping 192.168.1.2
```

Si nous voulons nous connecter en telnet sur notre firmware, exécutons :

```
lepton#2$ net/telnetd&
lepton#2$ ps
   PID PPID PGID     STIME   COMMAND
   1    0    1  14:41:10   initd
   2    1    1  14:41:10   /usr/sbin/lsh
   4    2    4  14:41:34   net/telnetd
   5    2    1  14:41:45   ps
```

puis :

```
(autre console) $ telnet 192.168.1.2 2000
lepton#6$ ps
ps
   PID PPID PGID     STIME   COMMAND
   1    0    1  14:41:10   initd
   2    1    1  14:41:10   /usr/sbin/lsh
   4    2    4  14:41:34   net/telnetd
   6    4    4  14:42:29   /usr/sbin/lsh
   7    6    4  14:42:46   ps
```

Nous pouvons taper **exit** pour fermer la connexion telnet.

2.2.6 Cible TWR-K60N512

2.2.6.1 Description

Nous utilisons la carte d'évaluation TWR-K60N512n **[TWR]** qui embarque un processeur ARM cortex-M4. Cette

carte appartient à la famille Kinetis et est commercialisée par Freescale. Brièvement, cette carte dispose :

- d'un processeur cadencé à 100 Mhz ;
- de 512 Kio de FLASH interne ;
- de 128 Kio de RAM interne ;
- d'un contrôleur Ethernet ;
- de contrôleurs de bus pour l'I2C, le SPI ;
- d'UART ;
- de convertisseurs analogique-numérique ;
- d'une carte externe disposant de 512 Mio de MRAM externe.

Illustration 5 : Carte d'évaluation Freescale. La carte supérieure est la plate-forme processeur. La carte inférieure propose un port série, un port Ethernet, des ports USB. La carte additionnelle MRAM n'est pas présente sur cette photo.

L'aspect le plus intéressant de ce processeur est la quantité de RAM interne. Avec 128 Kio de RAM interne, une application Lepton disposant d'une pile TCP/IP peut être envisagée. Pour le débogage, l'application sera téléchargée en MRAM ; pour le déploiement, l'application sera stockée et exécutée en FLASH interne. Un interpréteur de commandes sera disponible sur l'interface série.

2.2.6.2 Mise en œuvre

Commençons par récupérer le fichier d'options de construction propre à la cible physique :

```
$ cd ~/tauon/sys/user/tauon_myapp/prj/scons
$ cp ~/tauon/sys/user/tauon_sampleapp/prj/scons/k60n512_opts.py .
```

Son contenu est similaire au **synthetic_opts.py** mais quelques lignes subissent des aménagements :

- toutes les options liées aux outils de compilation sont modifiées ;
- l'architecture et la plate-forme sont ajustées.

```
#variable for user project
COMPILER_CC='arm-eabi-gcc'
COMPILER_CXX='arm-eabi-g++'
ARCHIVER='arm-eabi-ar'
ARCHIVE_INDEXER='arm-eabi-ranlib'
ARCH='CPU_CORTEXM'
PLATEFORME='cortexm/k60n512/'
TARGET_SUFFIX='k60n512'
OPTS_APP='-O0'
DEBUG_LEVEL='-gdwarf-2 -g3'
NANOX=0
FONTS_NANOX=''
LWIP=1
LWIP_OPTS='-Os'
VERBOSE=0
STRIP=0
TINYGL=0
FULL_STDIO=1
```

Une explication de chaque option est fournie à l'aide de la commande scons **-h**.

Il n'est pas nécessaire de récupérer un autre fichier de configuration XML. En effet, celui dont on dispose permet de configurer les deux cibles qui nous intéressent : la simulation et la physique.

Plusieurs techniques existent pour déboguer un logiciel embarqué :

- un moniteur ROM ;
- une sonde JTAG.

Pour déboguer sur notre cible, nous avons privilégié la technique du JTAG et l'utilisation du logiciel OpenOCD **[OOCD]**. Il peut fonctionner sur la machine hôte ou être embarqué sur une sonde. Il est compatible avec beaucoup de sondes JTAG et permet le débogage sur les architectures ARM. Ainsi, nous fournissons des fichiers de configuration pour les sondes Amontec et J-Link qui peuvent être adaptés pour permettre le débogage avec d'autres sondes JTAG.

Récupérons ces fichiers.

```
$ cd ~/tauon/sys/user/tauon_myapp
$ cp ~/tauon/sys/user/tauon_sampleapp/etc/scripts/gdb_script_k60_mram.gdb
etc/scripts
$ cp ~/tauon/sys/user/tauon_sampleapp/etc/scripts/gdb_k60_regs_map.gdb etc/
scripts
$ cp ~/tauon/sys/user/tauon_sampleapp/etc/scripts/jlink_k60* etc/scripts
$ cp ~/tauon/sys/user/tauon_sampleapp/etc/scripts/amontec_k60.cfg etc/scripts
```

Concrètement :

- **gdb_script_k60_mram.gdb** se connecte à OpenOCD et inclut le fichier **gdb_k60_regs_map.gdb**. Toute commande OpenOCD dans un script gdb est préfixée de « monitor ». Le fichier inclus permet de lister certains registres de la carte.
- **amontec_k60.cfg** est un fichier de configuration OpenOCD pour la sonde JTAG Amontec-Tiny Key (biblio : AMT). Il permet d'initialiser les horloges et la mémoire

externe lors d'une connexion par gdb. D'autres paramètres propres à la sonde (temps de *reset*, ...) peuvent être configurés.

- **jlink_k60.cfg** et **jlink_k60_flash.cfg** sont les fichiers de configuration OpenOCD pour les différentes versions de la sonde J-Link. Le premier fichier permet le débogage en RAM externe, le second fournit le débogage lors de l'exécution en FLASH.

Contrairement à la cible synthetic, il est nécessaire de récupérer le répertoire **board_freescale_twrk60n512** situé dans **~/tauon/sys/user/tauon_sampleapp/hal**. Il contient :

- **hal** qui contient toutes les parties spécifiques à l'initialisation de la carte. Il est ainsi possible de modifier l'initialisation des horloges, des vecteurs d'interruption et d'autres paramètres bas niveau pour les rendre spécifiques à l'application.
- **pkgadd.db** décrit le paquet qui sera intégré dans la base de données eCos. Si vous souhaitez la parcourir, vous pouvez ouvrir le fichier **$ECOS_REPOSITORY/ecos.db**.
- **lib/ecos.ecc** est le fichier de configuration eCos pour la cible. Il peut être ajusté en fonction des besoins spécifiques de l'application. On peut ainsi modifier le type d'exécution (MRAM ou ROM), le fréquence du *tick* système et bien d'autres options.

Il faut voir cette infrastructure comme un paquet eCos minimal permettant le démarrage de la carte et l'exécution de Lepton qui s'intégrera dans l'ensemble des paquets eCos disponibles. Par ailleurs, l'ajout d'une nouvelle cible passe par la création d'un paquet eCos comme celui-ci.

Copions ce répertoire.

```
$ cp -rf ~/tauon/sys/user/tauon_sampleapp/hal/board_freescale_twrk60n512 hal
```

Dans le fichier **Sconscript**, certains fichiers sont compilés en fonction de la cible choisie.

```
#build file per target
if tauon_build_envs.envs_map['DEFAULT']['TARGET_SUFFIX'].find('k60n512')>=0:
   tauon_myapp_src_list.extend([TauonSource(tauon_myapp_src_dev_dir+'board_
freescale_twrk60n512/dev_k60n512_i2c_x/dev_k60n512_i2c_x_m24xx.c'),
   TauonSource(tauon_myapp_src_dev_dir+'board_freescale_twrk60n512/dev_
k60n512_i2c_x/dev_twrk60n512_i2c_0_m24xx.c'),
   TauonSource(tauon_myapp_src_dev_dir+'board_freescale_twrk60n512/dev_
k60n512_i2c_x/dev_k60n512_i2c_x_mma7660.c'),
   TauonSource(tauon_myapp_src_dev_dir+'board_freescale_twrk60n512/dev_
k60n512_i2c_x/dev_twrk60n512_i2c_0_mma7660.c'),
   TauonSource(tauon_myapp_src_dir+'tsti2c.c'),
   TauonSource(tauon_myapp_src_dev_dir+'board_freescale_twrk60n512/dev_
k60n512_gpio_leds/dev_k60n512_gpio_leds.c'),
   TauonSource(tauon_myapp_src_dir+'tstleds.c'),
   ])
```

Ils sont à mettre en relation à des pilotes de périphérique et des binaires décrits dans le fichier de configuration **mkconf_tauon_sampleapp_gnu_k60.xml**.

```
<target name="cortexm_lepton">
   <devices>
  ...
      <!-- /dev/i2c0 -->
      <dev name="dev_twrk60n512_i2c_0_m24xx_map" use="off"/>
      <!-- /dev/hd/hdc -->
      <dev name="dev_eeprom_24xxx_0_map"         use="off"/>
      <!-- /dev/i2c1 on i2c0 use dev_twrk60n512_i2c_0_mma7660_map or
      dev_twrk60n512_i2c_0_m24xx_map + dev_eeprom_24xxx_0_map-->
      <dev name="dev_twrk60n512_i2c_0_mma7660_map" use="on"/>
      <!-- /dev/gleds -->
      <dev name="dev_twrk60n512_gpio_leds_map"     use="on"/>
   </devices>
</target>
...
<binaries src_path="bin" dest_path="bin">
   <target name="cortexm_lepton">
        <!-- tools test-->
  ...
        <bin name="tsti2c"      stack="4096"    priority="10" timeslice="1" />
        <bin name="tstleds"     stack="1024"    priority="10" timeslice="1" />
   </target>
</binaries>
```

Il faut fournir ces fichiers à notre nouvelle application.

```
$ mkdir src/dev
$ cp -rf ~/tauon/sys/user/tauon_sampleapp/src/dev/board_freescale_
twrk60n512/ src/dev/
$ cp ~/tauon/sys/user/tauon_sampleapp/src/tsti2c.c src/
$ cp ~/tauon/sys/user/tauon_sampleapp/src/tstleds.c src
```

Désormais, nous sommes en mesure de compiler notre application pour cette cible enfouie.

2.2.6.3 Compilation et débogage

La structure d'un paquet eCos est effectivement dans l'arborescence de notre nouvelle application sous le répertoire **hal/board_freescale_twrk60n512/hal/**. Néanmoins, si nous souhaitons effectuer des modifications au démarrage (ajuster les paramètres de cache pour la FLASH par exemple), l'ancien paquet présent dans la base eCos ne sera pas mis à jour. Un script permet de créer un paquet eCos à partir du répertoire **board_freescale_twrk60n512**.

```
$ cd tauon/sys/root/lib/arch
$  sh install-ecos-epk.sh ~/tauon/sys/user/tauon_myapp/hal/board_
freescale_twrk60n512 CYGPKG_HAL_CORTEXM_KINETIS_TWR_K60N512_MRAM
```

La chaîne **CYGPKG_HAL_CORTEXM_KINETIS_TWR_K60N512_MRAM** définit le nom du paquet dans la base de données eCos. Cette information provient directement du fichier **pkgadd.db**.

Suite au remplacement ou à l'ajout de notre nouveau paquet eCos, nous pouvons créer la bibliothèque **libtarget.a** qui sera utilisée par notre application.

```
$ sh script_ecos_lib.sh ~/tauon/sys/user/tauon_myapp/hal/board_
freescale_twrk60n512 ./cortexm/k60n512
```

Les fichiers générés sont stockés dans **~/tauon/sys/user/tauon_myapp/hal/lib/install**. Deux répertoires sont créés :

- **lib** qui contient la bibliothèque **libtarget.a** et le fichier d'édition de lien **target.ld** ;
- **include** qui fournit certains fichiers d'en-tête de bibliothèques standards.

Le chemin de recherche des fichiers d'en-tête et d'édition de liens étant fixe dans les fichiers de construction Scons, un lien symbolique est créé dans **tauon/sys/root/lib/arch/cortexm/k60n512**. La dernière chaîne (**./cortexm/k60n512**) définit justement un chemin relatif à **tauon/sys/root/lib/arch/**.

Nous pouvons à nouveau lancer le mklepton en changeant cette fois le type de cible en **cortexm_lepton**.

```
$ cd ~/tauon/tools/bin
$ ./mklepton_gnu.sh -t cortexm_lepton ~/tauon/sys/user/tauon_myapp/
etc/mkconf_tauon_sampleapp_gnu_k60.xml
```

Il ne reste plus qu'à construire notre application.

```
$ cd ~/tauon/sys/user/tauon_myapp/prj/scons
$ scons -c --targetfile=k60n512_opts.py ; scons -Q bin -targetfile=k60n512_opts.py
```

Le binaire **tauon_k60n512.elf** est généré et peut être désormais débogué sur une cible à l'aide d'une sonde JTAG, du moniteur **OpenOCD** (version 0.6.0) et de l'application **arm-eabi-gdb**.

Si nous souhaitons déboguer à l'aide de la sonde Amontec, nous tapons les commandes :

```
(OpenOCD)$ openocd -f ~/tauon/sys/user/tauon_myapp/etc/scripts/
amontec_k60.cfg
Open On-Chip Debugger 0.6.0 (2012-10-04-21:44)
Licensed under GNU GPL v2
For bug reports, read
       http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
srst_only separate srst_gates_jtag srst_open_drain
adapter speed: 4000 kHz
Info : J-Link initialization started / target CPU reset initiated
Info : J-Link ARM V6 compiled Feb  1 2011 14:28:14
Info : J-Link caps 0x99ff7bbf
Info : J-Link hw version 60000
Info : J-Link hw type J-Link
Info : J-Link max mem block 8864
Info : J-Link configuration
Info : USB-Address: 0x0
Info : Kickstart power on JTAG-pin 19: 0xffffffff
Info : Vref = 3.293 TCK = 1 TDI = 0 TDO = 0 TMS = 0 SRST = 0 TRST = 0
Info : J-Link JTAG Interface ready
Info : clock speed 4000 kHz
Info : JTAG tap: k60n512.cpu tap/device found: 0x4ba00477 (mfg:
0x23b, part: 0xba00, ver: 0x4)
Info : k60n512.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Dans une nouvelle console, nous exécutons le débogueur :

```
(gdb) $ cd ~/tauon/sys/user/tauon_myapp/bin
(gdb) $ gdb -x ../etc/scripts/gdb_script_k60_mram.gdb tauon_k60n512.
elf
GNU gdb (eCosCentric GNU tools 4.3.2-sw) 6.8.50.20080706
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-
eabi".
For bug reporting instructions, please see:
<http://bugs.ecos.sourceware.org/>...
0x00001232 in ?? ()
The target is assumed to be little endian
force soft breakpoints
Loading section .rom_vectors, size 0x8 lma 0x60000000
Loading section .ARM.exidx, size 0x10 lma 0x60000008
Loading section .text, size 0x503c4 lma 0x60000018
Loading section .rodata, size 0x32dc lma 0x600503e0
Loading section .data, size 0x9b0 lma 0x600536c8
Start address 0x60000019, load size 344168
Transfer rate: 58 KB/sec, 13766 bytes/write.
```

Le point d'entrée du noyau est **_start_kernel** ; plaçons un point d'arrêt sur cette fonction et exécutons le programme.

```
(gdb) b _start_kernel
(gdb) c
```

Si nous souhaitons déboguer le pseudo-binaire « test », nous ouvrons une console minicom liée au port série de la carte d'évaluation Kinetis. Un adaptateur série-USB est nécessaire en l'absence de port série natif sur la machine hôte. La configuration de ce port série est 38400 bauds/s, pas de parité, 8 bits de données.

```
(console minicom) $ minicom [port_com]
(gdb) b test_main
(gdb) c
```

La console série devrait afficher l'invite de commandes Lepton. Tapons « test » et « Entrée » pour activer le passage dans le débogueur et mettre au point notre programme.

Nous pouvons aussi faire fonctionner le réseau et le démon telnet comme pour la simulation.

2.2.6.4 Transformation pour flashage

Une fois la mise au point effectuée, le programme est écrit en mémoire flash et est exécuté dès la mise sous tension du micro-contrôleur. Comme on peut le deviner, l'adressage mémoire sera différent pour le binaire généré. Pour effectuer ce changement, il nous suffit de modifier le fichier de

configuration eCos afin que l'édition de lien se réalise avec l'espace d'adressage de la FLASH plutôt qu'avec celui de la MRAM externe.

Nous ouvrons le fichier

```
$ configtool ~/tauon/sys/user/tauon_myapp/hal/board_freescale_
twrk60n512/lib/ecos.ecc &
```

et nous recherchons l'*item* dans ***eCos HAL > Cortex-M Architecture > Freescale Kinetis Cortex-M4 Variant > Freescale Kinetis TWR-K60N512 Platform MRAM > Startup Type***. Nous validons l'option avec la valeur « ROM ». Nous dé-validons aussi l'option **« Utilize ".kinetis_misc" section for HAL »** car la taille de cette section devient trop petite pour loger le code compilé sans optimisation et avec l'option de débogage ; nous sauvegardons cette configuration.

Nous pouvons à nouveau construire la bibliothèque eCos et compiler notre micrologiciel.

```
$ cd tauon/sys/root/lib/arch/
$ sh script_ecos_lib.sh ~/tauon/sys/user/tauon_myapp/hal/board_
freescale_twrk60n512/ ./cortexm/k60n512/
$ cd ~/tauon/sys/user/tauon_sampleapp/prj/scons
$ scons -c --targetfile=k60n512_opts.py ; scons -Q bin
--targetfile=k60n512_opts.py
```

Avant de flasher le micrologiciel, il faut bien vérifier que le binaire final possède la section **flash_conf**. Sans la présence des 16 octets de cette section, la Kinetis ne démarrera plus !

```
$ arm-eabi-readelf -S tauon_k60n512.elf | grep "flash"
 [11] .flash_conf       PROGBITS        00000400 000400 000010 00   A  0   0  1
```

Nous transformons le fichier ELF au format bin à l'aide de objcopy :

```
$ cd ~/tauon/sys/user/tauon_sampleapp/bin
$ arm-eabi-objcopy -O binary tauon_k60n512.elf tauon_k60n512.bin
```

Le fichier est prêt à être flashé à l'aide de CodeWarrior ou de OpenOCD.

3 Conclusion et perspectives

Lepton est un système d'exploitation temps réel enfoui destiné pour l'instant aux architectures ARM. Il se présente sous la forme d'un micrologiciel qui embarque le noyau, les bibliothèques et le code applicatif dans un même espace d'adressage. Une cible simulation permet le développement d'une application sans disposer du matériel final.

Lepton s'exécute sur une cible Kinetis K60N512 disposant de 512 Kio de FLASH et de 128 Kio RAM. L'application de démonstration permet de faire fonctionner un serveur telnet et une invite de commandes. Lepton est déjà utilisé dans des applications industrielles.

Il reste néanmoins beaucoup de fonctionnalités qui pourraient être ajoutées à Lepton. Nous pouvons citer :

- l'amélioration et l'enrichissement des pilotes de périphérique pour la Kinetis ;
- le portage de nouvelles familles de processeurs Cortex-M3 et Cortex-M4 (STM32, Stellaris, ...) ;
- l'ajout d'un ELF *loader* minimal qui permettrait de charger séparément le noyau et les binaires ;
- la capacité de faire fonctionner plusieurs instances de la simulation simultanément ;
- l'ajout de protocoles industriels (EtherIP, Profibus) ;
- l'utilisation d'une chaîne de compilation plus récente (gcc 4.6 par exemple).

En définitive, Lepton s'inscrit comme une brique logicielle libre permettant de répondre aux exigences de complexité et de compacité croissantes des applications embarquées enfouies. ■

Bibliographie

[POT] Potier Dominique, Briques génériques du logiciel embarqué, 2010

[KERN] www.kernel.org

[STM32] http://www.st.com/internet/mcu/class/1734.jsp?WT.ac=?WT.ac=mcufa_bn_stm32_jul12

[K70] http://www.cnx-software.com/tag/uclinux/

[POSIX] http://www.opengroup.org/

[CA6116] http://www.chauvin-arnoux.fr/produit/famille_detail.asp?idFam=2116&idpole=1

[MT5022] http://www.handscope.chauvin-arnoux.com/fr/accueil.aspx

[EMBOS] http://www.segger.com/embos.html

[ECOSPRO] https://www.ecoscentric.com/

[ECOSEX] http://www.ecoscentric.com/ecos/examples.shtml

[SYNTH] http://ecos.sourceware.org/docs-latest/ref/synth.html

[TWR] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=TWR-K60N512

[OOCD] http://openocd.sourceforge.net/

KOBO GLO OU QUE FAIRE D'UTILE AVEC UNE LISEUSE ÉLECTRONIQUE ?

par Denis Bodor

Quel avenir y a-t-il pour le livre et le papier ? Cette question se pose depuis des années et certains constructeurs ont fait le pari de répondre à cette dernière via la technologie d'affichage e-paper. Ainsi sont apparues les liseuses, des tablettes disposant d'un affichage réflectif proche du rendu du papier. Cette technologie qui ne saurait satisfaire tous les utilisateurs constitue cependant un sous-domaine où l'embarqué open source peut s'épanouir et donc, implicitement, un terrain de jeu de choix pour l'amateur éclairé.

Noël dernier était encore une belle occasion pour les fabricants, distributeurs et détaillants de nous vendre le rêve nommé « technologie ». Les tablettes, téléviseurs 3D, ordinateurs, baladeurs multimédias... ou encore liseuses électroniques étaient sous de nombreux sapins, éveillant des émotions intenses et un sentiment d'appartenance à ce monde moderne et connecté. Sentiment pour certains et certaines rapidement changé en quelque chose de bien moins réjouissant mais tout aussi intense, le lendemain lorsque le ou la Homo se croyant alors très sapiens en vient à s'interroger sur son éventuelle proximité avec l'Homo neanderthalensis. On ne peut s'empêcher d'imaginer le nombre de réveils abruptes quelques jours après l'échange traditionnel de présents, des utilisateurs découvrant à leurs dépens qu'il existe bel et bien une différence entre le rêve « technologie » et la technologie elle-même.

Je vous rassure, le sujet de cet article, un Kobo Glo, l'une des liseuses officielles de l'enseigne Fnac, ne fut pas l'objet d'une telle déception. Non parce que le produit a parfaitement répondu à de vains espoirs de ma part, mais parce que je ne suis pas du tout adepte de cette commémoration de fin d'année et ne pratique pas, dans la mesure du possible, cet échange folklorique de cadeaux. Cependant, en prenant en main le matériel, j'ai eu une pensée pour tous ces gens ayant reçu un cadeau empoisonné qui dans les semaines ou mois à venir leur posera plus de problèmes qu'il ne donnera de plaisir.

Remarque : avant d'entamer la lecture de cet article, comprenez bien qu'il ne sera aucunement question ici de trouver un moyen de contourner une quelconque mesure de protection de contenu soumis à droit d'auteur. Il s'agit tout au plus d'explorer un matériel et de lui trouver une utilisation plus intéressante et raisonnable que la simple lecture de contenu numérique. Si vous cherchez à contourner un DRM, vous n'êtes pas en train de lire le bon magazine. Profitons-en pour ajouter ici l'avertissement d'usage : si vous tentez de reproduire les manipulations décrites dans le présent article et que votre matériel devient inutilisable, défectueux ou est transformé en presse-papier high-tech, ce n'est pas ma faute mais la vôtre et vous devez savoir ce que vous faites et ce que vous risquez.

1 Kobo Glo : ER-GO-NO-MIE !

Ergonomie ! Je le répète dans l'espoir de provoquer une brusque prise de conscience auprès du staff ayant partie prenante dans la conception de l'IHM de l'engin, du genre « Ah ouiiii ! L'ergonomie. Je me disais bien qu'il manquait des pages dans les specs ! ».

Soyons honnête. D'un point de vue matériel, le Kobo Glo est tout à fait acceptable. Il est léger, dispose d'un écran E Ink XGA Pearl de 6 pouces éclairé avec une résolution de 1024×758 et d'une finition exemplaire. La technologie e-paper étant ce qu'elle est actuellement, l'affichage est acceptable et on pourrait