Étude d'un système d'exploitation pour microcontrôleur faible consommation (TI MSP430) : pilote pour le stockage de masse au format FAT sur carte SD

Gwenhaël Goavec-Merou, J.-M Friedt

22 janvier 2010

1 Introduction

Deux grandes tendances se dessinent sur l'évolution de l'informatique : d'une part des plateformes de calcul puissantes, aux ressources quasiment illimitées, mais gourmandes en énergie. D'autre part, des applications aux puissances de calcul et mémoire réduites répondant aux besoins de l'utilisateur nomade qui veut pouvoir exploiter son système informatique en tous lieux et en toutes circonstances. La contrainte énergétique décrite précédemment dûe à l'autonomie de ces dispositifs implique l'utilisation de puissances de calcul réduites entraînant ainsi un développement de logiciels contraint par la capacité de la mémoire et par la vitesse d'exécution.

Les systèmes d'exploitation (*operating systems*, OS) développés pour les ordinateurs actuels requièrent trop de ressources mémoire et de calcul et ne constituent pas une solution pour l'adaptation à des plateformes ayant des ressources limitées. Par ailleurs, une programmation bas-niveau limite la portabilité d'algorithmes complexes sur des systèmes informatiques à caractéristiques différentes.

La mode actuelle est aux réseaux de capteur, dont chaque élément vise à une autonomie de plusieurs jours voir plusieurs années. Ces dispositifs sont déployés pour collecter de façon autonome des informations sur leur environnement, qu'il s'agisse d'études dans le cadre de comportement animaliers [1, 2], climatique [3] ou de contrôle industriel [4].

Les données acquises par ces systèmes doivent pouvoir être envoyées par une communication radio ou bien conservées en vue d'une exploitation ultérieure :

- la première approche fournit les informations en temps réel, mais n'est pas toujours adaptée, d'une part par manque de fiabilité de la liaison radiofréquence et d'autre part du fait de la consommation énergétique excessive de tous les modes de communication radiofréquence,
- la seconde approche offre une plus grande sécurité du point de vue de la conservation de l'information. Le volume de données n'est limité que par le support et permet à un utilisateur de récupérer rapidement l'intégralité des informations accumulées sous réserve de disposer d'un format de stockage compatible avec son ordinateur. Cependant, l'utilisateur doit venir physiquement sur site récupérer les informations accumulées et ne sait qu'en fin d'expérience si l'acquisition a fonctionné.

Nous nous sommes donc proposés d'évaluer une implémentation portable d'un système de fichiers en vue de la conservation d'informations sur support de stockage non volatil, en s'imposant de respecter les contraintes (mémoire, puissance de calcul, consommation) d'un système fortement embarqué.

2 Pourquoi un OS sur microcontrôleur ?

La méthode de développement classiquement utilisée sur système embarqué consiste généralement en la rédaction d'une application monolithique répondant aux besoins d'une tâche dédiée. Cette application peut éventuellement commuter entre divers états si la tâche le nécessite, commutation soit contrôlée par l'obtention d'un résultat (la planète est atteinte, il faut amorcer la séquence d'atterissage) ou de façon périodique sous le contrôle d'une horloge. Chaque nouvelle application réinvente donc son propre séquenceur de tâches (*scheduler*) répondant au mieux à ses besoins, en général sans changement de contexte afin d'économiser les ressources disponibles. L'abstraction est inexistante puisque l'application dédiée est fortement liée au matériel. Dans le meilleur des

cas, le développeur optimise les performances en programmant en assembleur, sinon il se contente du C ou d'un autre langage de plus haut niveau.

Sans aller jusqu'aux multiples couches d'abstraction des systèmes d'exploitation modernes – et des ressources consommées en conséquence – une tentative de portabilité des codes pour systèmes embarqués est visée avec les *environnements exécutifs* : sans fournir toutes les fonctionnalités d'un système d'exploitation moderne (chargement dynamique d'exécutables et de bibliothèques partagées, gestion de la mémoire, protection de l'espace noyau distinct de l'espace utilisateur, multiplicité d'utilisateurs, shell interactif), il s'agit d'un programme monolithique développé au moyen d'outil permettant au développeur de se croire sur un système d'exploitation. Le compilateur se charge alors d'assembler les divers éléments du programme afin de générer un unique exécutable contenant toutes les fonctionnalités de l'application. TinyOS – qui a abusivement pris l'extension d'OS – suit ce mode de fonctionnement, qui n'est pas sans rappeler la distinction entre `busybox` et l'ensemble des outil GNU disponibles dans `/bin` d'un système Un*x.

3 TinyOS

L'objectif des environnements exécutifs est donc de fournir une couche d'abstraction *au moment du développement*, avec la possibilité d'écrire du code portable et réutilisable, sans perdre de performance au *moment de l'exécution* puisque l'application monolithique exploite au mieux les ressources disponibles en étant statique.

TinyOS est un ensemble de routines mises à la disposition du programmeur en vue de simplifier le développement. Le code, écrit dans un langage dont la syntaxe s'inspire du C, nesC (section 5), est interprété pour générer un code C qui est compilé.

L'intérêt de TinyOS ne réside pas spécialement dans l'application installée sur le capteur mais au niveau de la simplification du développement. Il apporte la même notion d'abstraction qu'un système d'exploitation classique, sans imposer d'une part une trop forte dépendance entre le matériel et le logiciel, rendant dès lors possible une portabilité quasi transparente sur une multiplicité de plateformes différentes, et d'autre part sans que le programmeur n'ait un réel besoin de connaître les spécifications du matériel sur lequel il travail. Dans le cas d'un MSP430, l'ensemble de l'implémentation des caractéristiques d'accès au microcontrôleur est déjà réalisée. Il ne reste donc qu'à ce concentrer sur l'application ou le pilote à réaliser en utilisant les routines mises à la disposition de l'utilisateur.

TinyOS offre également un certain nombre de mécanismes permettant la communication d'une manière générique par rapport au médium sous-jacent, rendant ainsi la communication homogène sur un ensemble de nœuds d'origines et de caractéristiques différentes.

Afin de se familiariser au mieux avec toutes les notions de TinyOS, nous nous sommes proposés de travailler sur un microcontrôleur supporté par cet environnement (le MSP430F149) mais sur un circuit spécifiquement développé pour une application permettant l'acquisition de données d'un récepteur GPS afin d'être stockées sur une carte mémoire utilisant un système de fichier compatible avec n'importe quel ordinateur.

La plateforme (Fig. 1)

- est à base d'un MSP430F149 disposant de 60 KB de mémoire flash et de 2 KB de RAM,
- est équipée d'un récepteur GPS ET312 et de son antenne,
- offre une communication série asynchrone en écriture seule, convertie en USB par un FT232 pour être compatible avec la plupart des ordinateurs récents,
- fournit un emplacement pour une carte mémoire de type *secure digital* (SD) pour une communication par protocole série synchrone (SPI),
- est alimentée sur 4 accumulateurs NiMH pour une tension d'alimentation de 4,8 V et un régulateur linéaire LE33CZ vers 3,3 V, ou une paire d'accumulateurs NiMH et une pompe de charge MAX1674 élevant la tension à 3,3 V.

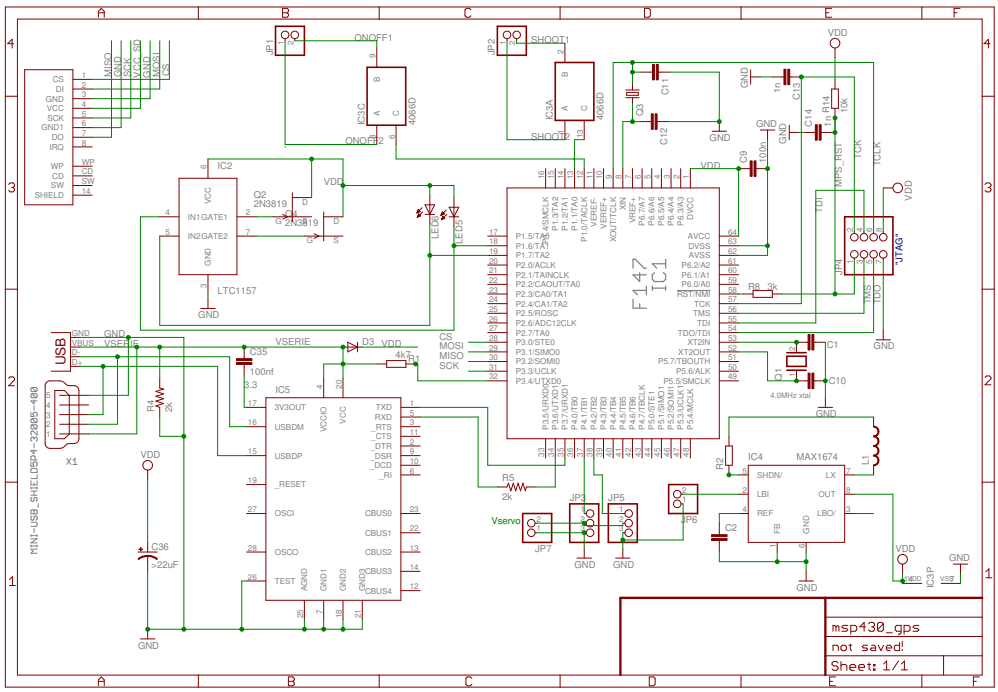

FIG. 1 – Schéma de la plateforme

4 Installation des outils de développement

Deux outils de développement seront nécessaires pour exploiter les programmes développés sous TinyOS-2.x : un cross-compilateur à destination de l'architecture cible – ici le MSP430 – et un outil de programmation pour tansférer le programme compilé au microcontrôleur.

Le MSP430 se programme au travers d'une interface de communication synchrone JTAG [1]. Une solution peu coûteuse est l'utilisation d'une interface sur port parallèle : le protocole est implémenté par l'outil `msp430-jtag` dont l'installation est développée ci-dessous.

L'archive des sources s'obtient par CVS par

```
export CVSROOT=:pserver:anonymous@mspgcc.cvs.sourceforge.net:/cvsroot/mspgcc
export CVS_RSH=ssh
cvs login
cvs checkout jtag
cvs checkout python
```

Un certain nombre de corrections aux sources est nécessaire avant de compiler msp430-jtag :

- dans **jtag/funclets/makefile** :
  remplacer msp430x2121 par msp430x149 afin d'adapter à la version du microcontrôleur exploité dans nos applications
- remplacer ASFLAGS = −mmcu=${CPU} par ASFLAGS = −mmcu=${CPU} −D_GNU_ASSEMBLER_→ ↪ dans **jtag/funclets/eraseFlashSegment.S**
- dans **lockSegmentA.S**, remplacer toutes les occurrences de LOCKA par 0x0040

Nous avons toujours travaillé sur MSP430F149 au moyen de `gcc`-3.3, compilé pour générer un binaire à destination du MSP430. Cependant, afin d'utiliser une version du microcontrôleur

[1] Nous n'avons pas expérimenté la programmation par communication série asynchrone au moyen du BSL décrit dans la note d'application SLAA089B de Texas Instruments.

compatible broche à broche mais fournissant plus de RAM, nous avons expérimenté l'utilisation du MSP430F1611. Ce nouveau microcontrôleur ne semble pas être supporté par gcc-3.3, et une version antérieure est nécessaire : mieux vaut donc tout de suite compiler `gcc-3.2` qui supporte toutes les versions de MSP430 exploitées dans ce document. La cross compilation de `gcc` à destination du MSP430 se fait en exécutant le script `build-gcc` fourni dans l'archive de tinyOS-1.x. Nous n'avons pas exploré l'utilisation de `gcc-4.x`.

5 Le nesC

Le nesC est un langage évènementiel, orienté objet. Contrairement aux autres langages objets la relation n'est pas du type "a contient b" mais "a est connecté à b à travers l'interface i".

La présentation qui suit n'a pas prétention à donner un cours sur le nesC. Les documents [5] et [6] donnent l'ensemble des explications, la suite ayant seulement pour but de permettre au lecteur de comprendre les codes présentés plus loin.

5.1 Interface

Ce premier type – `interface` – est un des points clefs du langage, il a deux buts :
- la communication entre composants par des appels de fonctions et des gestionnaires d'événements,
- l'abstraction logicielle. Comme les communications se font aux travers d'interfaces génériques, un composant A n'a pas besoin de savoir s'il fait un *write* (par exemple) sur un fichier, au travers d'une communication série ou bien par radio.

la Fig. 2 présente la relation qui peut exister entre deux composants.

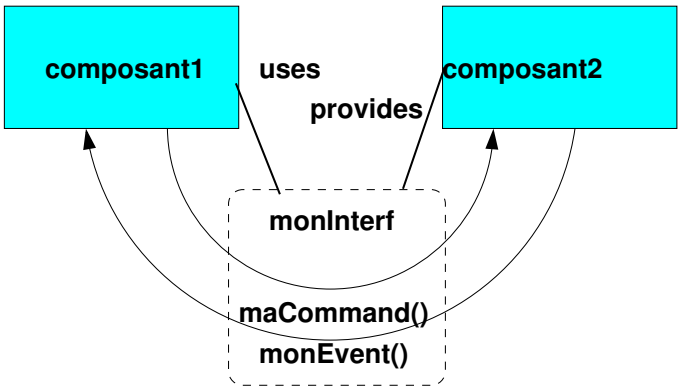

FIG. 2 – Exemple de relation entre composants

Le principe de l'interface est le même que dans d'autres langages tels que *C++* ou *Java*, c'est-à-dire imposer un ensemble de fonctions qu'un composant devra implémenter dès lors qu'il déclare sa relation avec l'interface. Toutefois, pour permettre l'aspect évènementiel, elle ajoute également une notion de bidirectionnel. Ceci est possible en proposant des fonctions allant de l'utilisateur vers le fournisseur et d'autres allant du fournisseur vers l'utilisateur.

La Fig.3 donne la représentation d'une interface simple.

```
interface interf1 {
  command void maCommand();
  event void monEvent();
}
```

FIG. 3 – exemple de définition d'une interface

Afin de pouvoir utiliser l'interface, dans un mode ou dans l'autre, le composant aura besoin de préciser ses intentions :

- avec *uses* pour en faire usage,
- avec *provides* pour en être fournisseur.

Comme présenté dans la Fig.3, il apparaît deux types de fonctions :

- celles commençant par *command* qui doivent être implémentées par le composant fournissant l'interface,
- celles commençant par *event* devant l'être par le composant utilisateur.

Les fonctions issues d'une interface donnée ne sont accessibles qu'en utilisant le nom pleinement défini de celle-ci :

```
nom_interface.nom_fonction(arguments);
```

C'est sur cette caractéristique que se base toute la notion d'abstraction de TinyOS.

Lors de l'utilisation et selon le type de la fonction, il faudra utiliser call nom_interface.nom_fonction→ ↪(arguments); pour une *command* et signal nom_interface.nom_fonction(arguments); pour utiliser un *event*.

Pour la définition des fonctions la même régle est à respecter. Ainsi la signature de la définition d'une fonction se fera par type_fonction type_retour nom_interface.nom_fonction(arguments) et l'appel (call ou signal) se fera par type_appel nom_interface.nom_fonction(arguments);

5.2 structure des composants

Structurellement, un fichier d'entité est composé de deux zones, comme dans le cas de l'*ADA* ou du *VHDL*. Une première définit la vue externe et une seconde, selon le type, donne l'implémentation du module ou bien une mise en relation de composants.

5.2.1 Vue externe

Pour ne pas risquer de confusion cette partie ne sera jamais appelée interface, même si son rôle est sensiblement le même que ce que l'on trouve dans un fichier .h en C par exemple.

Elle définit le type de l'entité ainsi que les entrées-sorties de celle-ci, au travers des interfaces.

Les entrées sorties se font en définissant quelles sont les interfaces qui sont utilisées et selon quelle relation.

```
type ent1 {
  provides {
    interface interf1;
  }
  uses interface interf2;
}
implementation {
    ...
}
```

FIG. 4 – Exemple de fichier d'entité

La Fig. 4 présente la définition de la vue externe pour *ent1*. L'interface *interf1* (Fig. 3) est fournie, ainsi un autre composant en relation avec celui-ci pourra faire des appels à *maCommand* mais devra également définir une implémentation pour *monEvent*.

Noter que les accolades après *provides* peuvent être omises si une seule interface est définie, comme montré pour *interf2*. Il est également possible de définir un alias pour une interface afin de la rendre plus claire ou d'éviter un conflit dans le cas où un composant ferait usage ou implémenterait plusieurs interfaces du même type.

5.3 type d'une entité

Il existe en nesC deux types d'entités :

- les modules dont la partie implementation contient la définition des *commands* ou des *events*,

– les configurations qui ne contiennent pas de codes mais permettent le branchement de modules entre eux.

Si l'on fait un parallèle avec un circuit physique, le module correspond à un composant et la configuration au circuit imprimé.

La norme dit que le fichier de configuration pour un module aura un nom finissant par un *C* (monModuleC.nc) et le fichier d'implémentation (type module) aura un nom finissant par un *P* (monModuleP.nc).

5.3.1 Type configuration

Il permet de brancher plusieurs modules entre eux. En prenant l'exemple d'un module pour un traitement quelconque, la configuration de celui-ci permettrait de pouvoir :

– fournir les ressources (interfaces) pour l'utilisation de ce module,
– réaliser les raccordements sur les couches inférieures d'une manière non visible pour les modules faisant usage de celui-ci.

```
configuration maConfig {
  provides interface interf1;
}
implementation {
  components PlatformPlop;
  components monDriver;
  interf1 = monDriver.interf1;
  monDriver.stdControl -> PlatformPlop.stdControl;
}
```

FIG. 5 – fichier de configuration d'un module

Dans la Fig. 5 deux types de relations sont présentées :

– celle liée par un = fait transiter l'information si elle a le même type (*uses* ou *provides*),
– celle liée par un $->$ pour lier l'interface d'un module utilisant celle-ci à l'interface d'un module la fournissant. La flêche va toujour de l'utilisateur vers le fournisseur.

Dans le cas présenté, un utilisateur de *maConfig* ne connaîtra que l'interface *interf1*, les relations entre *monDriver* et *PlatformPlop* étant cachées.

5.3.2 le type module

Il permet de fournir l'implémentation du module en lui même. Celui-ci va contenir :

– l'implémentation des *commands/events* définies dans les interfaces qu'il *uses/provides*,
– des variables locales au module,
– des fonctions elles aussi locales,
– des tasks, qui sont des fonctions appelées d'une manière asynchrone et dont la portée est limitée au module.

6 Plateforme

La création d'une plateforme n'est nécessaire que lorsque la définition du circuit cible n'est pas fournie d'origine par TinyOS. Cependant, étant donné que les applications se compilent en spécifiant la plateforme cible, il s'agit de l'étape obligatoire avant de pouvoir développer une application sur un nouvel environnement matériel.

La plateforme est une description du matériel fournissant par la suite l'abstraction entre les applications et la partie matérielle. L'explication de la création d'une plateforme se fera au travers de la réalisation de celle de l'architecture de la Fig. 1, et se nommera **projet**, en suivant le tutoriel disponible à [7].

L'ensemble des fichiers à créer/modifier/utiliser se trouvent dans le répertoire racine des sources de TinyOS, celui-ci dépend de la méthode d'installation. Le chemin est défini par la variable

TOSROOT.
Dans la suite de ce document les fichiers et répertoires seront toujours en chemin relatif vis à vis de cette racine.

6.1 Définition

Une plateforme permet une couche d'abstraction entre les applications et la carte utilisée. En d'autres termes, pour une communication selon un protocol quelconque, la hiérarchie du système se compose de la manière suivante :

1. l'application utilise le type correspondant sans toutefois connaître son implémentation par le système, ni le câblage sur la carte. Il n'a besoin que de savoir les actions possibles,
2. le module ne connaît pas non plus le câblage. Il se contente de faire les traitements liés au module et passe les "ordres" au fichier que le créateur de la plateforme a réalisé,
3. le fichier de la plateforme définit la liaison avec les ports du microcontrôleur, ainsi que les paramètres de configuration sans connaître les registres,
4. le pilote du microcontrôleur fait au final l'opération adéquate selon le besoin.

6.2 Structure minimale d'une plateforme

Celle-ci est définie en deux entités principales qui sont :

1. un fichier se trouvant dans *support/make*. Il est de la forme NomPlatform.target. Ce fichier informe la commande *make* de l'existence de la plateforme,
2. un répertoire, se trouvant dans *tos/platforms*, qui va contenir l'ensemble des fichiers de configuration de la plateforme. Chaque module s'attend à trouver un fichier portant un nom bien spécifique, permettant de simplifier le travail d'intégration.

6.3 Mise en place de la nouvelle plateforme

En tout premier lieu il est nécessaire de créer un répertoire projet dans tos/platforms avec la commande mkdir projet.

1. `projet.target` : dans le répertoire *support/make* nous créons un fichier *projet.target* qui va contenir les lignes présentes dans le listing 1 :

```
PLATFORM = projet
$(call TOSMake_include_platform,msp)

projet: $(BUILD_DEPS)
	@:
```

Listing 1 – projet.target

   La première ligne précise le nom de la plateforme.
   Les lignes suivantes donnent les informations dynamiques sur les dépendances de la plateforme ainsi que d'autres règles de compilation liées au microcontrôleur.

2. `.platform` : pour la suite tout se déroulera dans *tos/platforms/projet*.
   .platform concerne les spécifications sur les répertoires contenant les bibliothèques de fonctions et les options de compilation.
   Contrairement à un logiciel pour un système d'exploitation "classique" où les configurations et informations passées au compilateur sont fournies avec les sources de l'application, ces informations sont, dans TinyOS, au niveau de la plateforme.

```
push( @includes, qw(

%T/chips/msp430
 %T/chips/msp430/adc12
%T/chips/msp430/dma
 %T/chips/msp430/pins
```

```
%T/chips/msp430/timer
%T/chips/msp430/usart
%T/chips/msp430/sensors
%T/lib/timer
%T/lib/serial
%T/lib/power
));

push ( @opts, qw(
-gcc=msp430-gcc
-mmcu=msp430x149
-fnesc-target=msp430
-fnesc-no-debug
-fnesc-scheduler=TinySchedulerC,TinySchedulerC.TaskBasic,TaskBasic,TaskBasic→
    ↪,runTask,postTask
));
```

Listing 2 – .platform

Le listing 2 contient les configurations dans le cas de la plateforme *projet*. Il est écrit en PERL et contient les deux parties suivantes :

(a) les répertoires qui contiennent les bibliothèques,

(b) les options qui seront passées au compilateur.

3. `hardware.h` : ce fichier est inclus lors de la compilation d'une application. Il sert à définir des constantes et des entêtes (*headers*).

4. `PlatformC.nc`

```
#include "hardware.h"

configuration PlatformC {
  provides interface Init;
}
implementation {
  components PlatformP, Msp430ClockC;
  Init = PlatformP;
  PlatformP.Msp430ClockInit -> Msp430ClockC.Init;
}
```

Listing 3 – PlatformC

Ce fichier ne sert qu'à fournir une implémentation de l'*Init* qui sera appelée automatiquement lors du démarrage de la plateforme. Il permet de fournir un comportement automatique à ce moment, avant le lancement de l'application elle-même.

5. `PlatformP.nc` : ce fichier contient l'initialisation des divers composants qui seront utilisés dans cette plateforme.

```
#include "hardware.h"

module PlatformP {
  provides interface Init;
  uses interface Init as Msp430ClockInit;
  uses interface Init as LedsInit;
}
implementation {
  command error_t Init.init() {
    call Msp430ClockInit.init();
    call LedsInit.init();
    return SUCCESS;
  }
  default command error_t LedsInit.init() { return SUCCESS; }
}
```

Listing 4 – PlatformP

PlatformP implémente l'interface *Init* pour lancer l'initialisation de l'horloge interne du MSP430 ainsi que l'initialisation de LEDs.

6.4 Exemple pratique : contrôle des ports numériques

Les ports d'entrée-sortie numériques (*General Purpose Input-Output*, GPIO) sont probablement les périphériques les plus simples d'accès : nous allons les exploiter pour contrôler l'allumage et l'extinction de LEDs (Fig. 6), puis pour afficher du texte sur un afficheur LCD compatible HD44780 (Fig. 9).

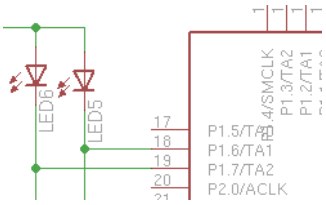

FIG. 6 – Câblage des LEDs

La figure 6 présente le câblage physique de deux LEDs de la carte aux bornes du microcontrôleur. Au niveau de leur implémentation dans TinyOS, il faut créer le fichier suivant, imposé par LedsC.nc :

```
#include "hardware.h"
configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation {
  components HplMsp430GeneralIOC as GeneralIOC,
    new Msp430GpioC() as Led0Impl,
      new Msp430GpioC() as Led1Impl;

  components new NoPinC() as Led2Impl;
  components PlatformP;

  Init = PlatformP.LedsInit; // Raccorde l'event init à celui de PlatformP

  Led0 = Led0Impl;
  Led0Impl -> GeneralIOC.Port16;
  Led1 = Led1Impl;
  Led1Impl -> GeneralIOC.Port17;
  Led2 = Led2Impl; // No led2 on board
}
```

Listing 5 – PlatformLedsC

Cette configuration permet de préciser la branchement matériel pour chaque LED définie par le pilote TinyOS. Les 5 dernières lignes définissent que led0 est raccordé au Port16 (noté P1.6 dans la documentation (*datasheet*) du MSP430) et led1 est raccordé au Port17 (noté P1.7). Le cas de led2 est particulier car le pilote Leds définit une troisième LED absente du montage physique. Celle-ci est connectée sur NoPinC signifiant son absence.

Un exemple d'utilisation des LEDs est l'application **apps/Blink** qui peut fonctionner sans avoir besoin de la moindre modification.

Note :
Dans le cas d'une plateforme exploitant moins de 3 LEDs, il faut utiliser NoPinC comme présenté plus haut pour désactiver les LEDs absentes. Dans le cas où la carte contiendrait plus de 3 LEDs, il faut copier les fichiers *tos/system/LedsC.nc* et *tos/system/LedsP.nc* afin d'ajouter les LEDs manquantes dans le pilote officiel.

6.5 Exploitation d'un LCD

Le protocole d'un afficheur LCD à base de contrôleur HD44780 est bien documenté [2]. Dans sa version la plus simple, l'implémentation du protocole ne nécessite que 4 bits pour les données à transmettre et 2 bits pour le contrôle de flux.

```
#include "hardware.h"
configuration PlatformLcdC {
  provides interface GeneralIO as LcdData0;
  provides interface GeneralIO as LcdData1;
  provides interface GeneralIO as LcdData2;
  provides interface GeneralIO as LcdData3;
  provides interface GeneralIO as LcdE;
  provides interface GeneralIO as LcdRS;
  uses interface Init;
}
implementation {
  components HplMsp430GeneralIOC as GeneralIOC,
    new Msp430GpioC() as LcdData0Impl,
    new Msp430GpioC() as LcdData1Impl,
    new Msp430GpioC() as LcdData2Impl,
    new Msp430GpioC() as LcdData3Impl,
    new Msp430GpioC() as LcdEImpl,
    new Msp430GpioC() as LcdRSImpl;

  components PlatformP;

  Init = PlatformP.LcdInit;

  LcdData0 = LcdData0Impl;
  LcdData0Impl -> GeneralIOC.Port24;

  LcdData1 = LcdData1Impl;
  LcdData1Impl -> GeneralIOC.Port25;

  LcdData2 = LcdData2Impl;
  LcdData2Impl -> GeneralIOC.Port26;

  LcdData3 = LcdData3Impl;
  LcdData3Impl -> GeneralIOC.Port27;

  LcdE = LcdEImpl;
  LcdEImpl -> GeneralIOC.Port23;

  LcdRS = LcdRSImpl;
  LcdRSImpl -> GeneralIOC.Port22;

}
```

Listing 6 – PlatformLcdC

Le listing 6 montre le fichier de configuration. Celui-ci est globalement équivalent à celui des LEDs, si ce n'est par le volume de broches définies. En effet, avec TinyOS il n'est pas possible d'accéder à un port complet mais uniquement aux broches individuelles. Il utilise le même type de ressources permettant ainsi d'émuler l'écriture des données sur le port.

De ce fait l'écriture sur le port se fait de façon un peu fastidieuse par le code proposé dans le listing 7.

Il n'existe pas d'application dans TinyOS pour tester le bon fonctionnement du module : il est donc nécessaire d'en ajouter une. En reprenant l'application *Blink*, il suffit de remplacer le contenu de Boot.booted par le contenu du listing 8.

La fonction *Lcd.write()*, définie dans l'interface *Lcd* va gérer tout le protocole du Lcd en vu de l'affichage et faire autant d'appel à *writeDB* qu'il y a de caractères à écrire.

[2] http://home.iae.nl/users/pouweha/lcd/lcd.html

```
// Ecrit un demi octet
void writeDB(uint8_t val) {
    val = val & 0x0f;
    if (val & LCD_DATA0)
      call LcdData0.set();
    else call LcdData0.clr();

    if (val & LCD_DATA1)
      call LcdData1.set();
    else call LcdData1.clr();

    if (val & LCD_DATA2)
      call LcdData2.set();
    else call LcdData2.clr();

    if (val & LCD_DATA3)
      call LcdData3.set();
    else call LcdData3.clr();
}
```

FIG. 7 – Fonction écrivant un demi octet

```
event void Boot.booted() {
    call Lcd.write("Hello World",11);
}
```

FIG. 8 – Écriture de Hello World sur le LCD

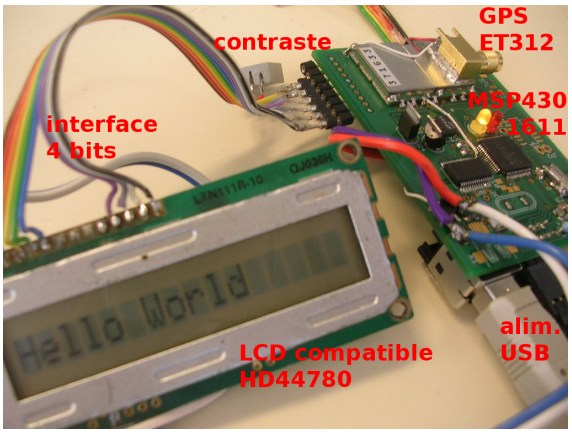

FIG. 9 – Circuit comprenant un MSP430F1611 auquel sont connectées deux LEDs, un écran LCD compatible HD44780 en mode de communication 4 bits, un récepteur GPS ET312 et un support de carte SD, le tout alimenté par le bus USB. Dans cet exemple, l'affichage du message de démonstration du bon fonctionnement de l'écran LCD qui servira au debuggage des diverses applications qui vont suivre.

7 Communications numériques

7.1 Communication asynchrone RS232

Ce protocole de communication asynchrone – normalisé sous la nomenclature RS232 – propose une communication bidirectionnelle nécessitant 2 fils (transmission et réception de données), supposant que les deux interlocuteurs possédent chacun une horloge de fréquence connue. Le seul moyen de synchroniser les deux interlocuteurs est d'une part un échange au préalable du protocole de communication (vitesse et nombre de bits/octet transmis), et au cours de la transmission deux

transitions nommées start et stop bits annonçant le début et fin de communication.

Ce protocole est le plus couramment utilisé de par sa simplicité : il est donc implémenté sous forme matérielle (*Universal asynchronous receiver/transmitter*, UART) dans la majorité des microcontrôleurs, et ce sera notre premier exemple de développement sous TinyOS.

TinyOS offre un mécanisme de communication intégré s'apparentant aux messages TCP/IP classiques : les paquets sont encapsulés dans une trame contenant un certain nombre d'informations. Cela permet leurs routage et la vérification du contenu, par somme de redondance cyclique. L'intérêt est d'offrir, d'une manière transparente, une interface simple cachant le mode de communication exploité (radio, zigbee, bluetooth, série), compatible quelque soit l'endianess du microcontrôleur.

Le modèle TinyOS possède clairement des avantages pour l'application visée – les réseaux de capteurs – car il permet d'homogénéiser la communication et d'en assurer la fiabilité. Mais cette encapsulation de la communication est un handicap pour les outils classiques tels que *minicom* ou avec un périphérique exploitant une connection série. Par exemple, dans le cas d'un récepteur GPS, qui fournit ses informations au moyen d'une chaîne ASCII, le modèle TinyOS n'est pas exploitable. L'encapsulation des paquets proposée par TinyOS doit donc être évitée dans le cadre du projet qui nous intéresse afin que TinyOS ne rejette pas des paquets ne se conformant pas à son modèle de communication.

Afin de pouvoir exploiter l'UART du microcontrôleur, *SerialActiveMessage* utilise deux fichiers spécifiques à la plateforme cible : *PlatformSerialC.nc* et *PlatformSerialP.nc*. Ces deux fichiers permettent :

- de relier le pilote et la couche matérielle,
- de spécifier la configuration du port.

Une exploitation brute de l'UART (sans encapsulation des données) va donc consister à définir un module pour la réception et l'envoie de données, directement raccordé sur la plateforme et n'utilisant pas la couche de communication fournie par tinyOS.

Le listing suivant donne l'interface publique raccordant le pilote et le microcontrôleur :

```
#include "hardware.h"
configuration PlatformSerialC {
  provides interface StdControl;
  provides interface UartStream;
  provides interface UartByte;
}
implementation {

  components new Msp430Uart1C() as UartC, projetSerialP;
  UartStream = UartC.UartStream;
  UartByte = UartC.UartByte;
  StdControl = projetSerialP.Control;
  projetSerialP.Msp430UartConfigure <- UartC.Msp430UartConfigure;
  projetSerialP.Resource -> UartC.Resource;
  projetSerialP.ResourceRequested -> UartC.ResourceRequested;
  components LedsC;
  projetSerialP.Leds -> LedsC;
}
```

FIG. 10 – PlatformSerialC

L'interface la plus importante à prendre en compte pour la suite est **UartStream** qui offre principalement :

- une commande permettant l'envoi d'une chaîne de caractères :
  async command error_t send(uint8_t* buf, uint16_t len);)
- un évènement qui permettra au programme appelant d'être averti de la fin de l'envoi ainsi que de son succès ou de son échec :
  async event void sendDone(uint8_t* buf, uint16_t len, error_t error);)

– un évènement qui est généré à chaque fois qu'un octet arrive sur l'UART du microcontrôleur : async event void receivedByte(uint8_t byte);

Le fichier **PlatformSerialP.nc** est utilisé principalement pour la gestion du lancement et de l'arrêt de l'UART. Il contient une structure définissant la configuration du port en terme de vitesse de transfert, de type et fréquence d'horloge, de parité, etc... :

```
msp430_uart_union_config_t msp430_uart_proj_config = { {
  ubr: UBR_32KHZ_4800,
  umctl: UMCTL_32KHZ_4800,
  ssel: 0x01,
  pena: 0,
  pev: 0,
  spb: 0,
  clen: 1,
  listen: 1,
  mm: 0,
  ckpl: 0,
  urxse: 0,
  urxeie: 1,
  urxwie: 0,
  urxe: 0,
  utxe: 1} };
```

FIG. 11 – Structure de configuration de la communication série.

Une fois ces fichiers à disposition et raccordés à l'application ou à un pilote, il est possible d'obtenir des données telles que des trames NMEA fournies par un récepteur GPS.

7.2 Acquisition des trames d'un récepteur GPS

Le récepteur GPS ET312 équipant la carte utilise le protocole RS232 au débit de 4800 bauds, protocole N81. Les mêmes broches sont exploitées pour la communication série synchrone et asynchrone (Tx pour l'émission de messages vers un ordinateur, Rx pour la réception des données du GPS). Le second port de communication disponible sur le MSP430 sera utilisé plus tard pour la liaison synchrone de type SPI.

Les trames GPS sont des chaînes de caractères en ASCII directement compréhensibles, commençant par un $ et se finissant par un *CRLN*. Comme TinyOS gère les interruptions, il est possible d'être averti de la réception d'un octet sur l'UART afin de le traiter et le stocker . Afin de ne pas risquer de pertes liées à l'aspect asynchrone, le module reçoit un tableau et sa taille. Il avertit l'application lorsque le tableau est complétement rempli.

Le démarrage de l'acquisition se fait grâce à l'interface *ReadStream*. *SplitControl* sert à la gestion de l'UART. Le fonctionnement du module est donc ainsi :

– lorsque *ReadStream.postBuffer(...)* n'a pas été appelé, l'état est GPS_IDLE. Dans cet état, le module ne s'occupe pas des octets qu'il reçoit,
– après un appel à *ReadStream.postBuffer* le module se met dans l'état GPS_READ. À partir de ce moment il va stocker tous les caractère reçus,
– lorsque le tableau est rempli, il se remet dans l'état *GPS_IDLE* et signale à l'application qu'il a fini sa lecture.

Afin de tester ce pilote, une première application simple a été mise en œuvre consistant à copier sur la sortie du port série (liée à un PC) les trames lues sur l'entrée du port série lié au récepteur GPS. Comme les deux communications ont la même vitesse, un tampon correspondant à la longueur d'une trame permet de rendre les deux actions indépendantes.

Le stockage des trames reçues sur le port série de l'ordinateur permet le calcul du débit d'informations.

```
$GPGGA,062546.000,4722.9018,N,00600.0719,E,1,07,2.6,219.5,M,47.9,M,,0000*53
2$GPGSA,A,3,27,29,09,12,30,02,26,,,,,,5.0,2.6,4.3*35
$GPGSV,3,1,12,30,77,322,41,05,60,287,,12,56,069,49,29,47,204,44*7F
```

```
4$GPGSV,3,2,12,14,39,254,,26,31,219,42,02,30,084,48,09,22,140,36*75
 $GPGSV,3,3,12,31,20,310,17,27,17,140,36,04,17,044,,25,12,312,*7B
6$GPRMC,062546.000,A,4722.9018,N,00600.0719,E,0.13,329.05,061209,,*09
```

En comptant le nombre d'octets pour un ensemble de trames avec la même estampille temporelle, il est possible de déterminer un débit de 402 octets par seconde.

7.3 Communication synchrone SPI : application à la carte mémoire SD

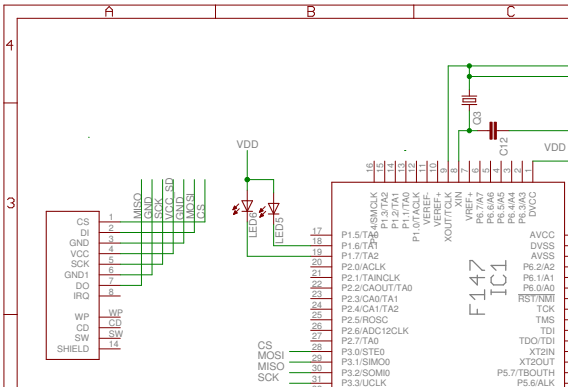

FIG. 12 – Câblage de la carte SD

La communication asynchrone fournit un débit de communication limité du fait de la difficulté à synchroniser des horloges, basées sur des résonateurs de qualité médiocre, cadençant les deux systèmes numériques communiquant. L'alternative est la communication synchrone qui partage l'horloge entre les deux dispositifs : des débits considérablement plus élevés peuvent ainsi être obtenus, au détriment de la distance de communication. Le bus SPI est une implémentation de bus de communication synchrone, asymétrique puisque distinguant la liaison du maître à l'esclave (MOSI, *Master Out Slave In*) et de l'esclave vers le maître (MISO). Un quatrième signal, Chip Select (CS), permet d'activer le périphérique du bus qui doit communiquer avec le maître (Fig. 12).

Le protocole SPI, rapide (quelques Mb/s) et couramment disponible sur microcontrôleurs, est implémenté de façon matérielle dans le MSP430F149 en partageant des ressources avec un port de communication asynchrone (UART). Ainsi, l'exploitation d'un bus SPI nous prive du second port de communication asynchrone.

L'implémentation matérielle du protocole [8, chap.14] SPI ne fournit qu'1 octet de tampon (*buffer*). Contrairement au bus asynchrone tel que le RS232, toute transaction sur SPI est initiée par le maître (microcontrôleur) : il ne peut donc pas y avoir perte de données en cas de délai dans la réponse à une transaction.

L'implémentation de la communication SPI pour le MSP430 ne sera pas nécessaire, car déjà disponible dans les routines d'exploitation du microcontrôleur. La seule tâche nécessaire est de définir, à l'instar de l'UART, les configurations (en terme de source et de vitesse d'horloge, de signe de la phase, etc...) ainsi que la broche correspondant au signal d'activation du périphérique.

Ce bus local est notamment utilisé pour communiquer avec les cartes mémoires SD (et leur ancêtre MultiMediaCard MMC), qui fournissent ainsi un support de stockage de masse non-volatile à faible coût, ne nécessitant que peu de connexions électriques pour leur mise en œuvre. Nous allons par conséquent utiliser comme prétexte le stockage des trames GPS sur un support non-volatile pour implémenter un pilote pour TinyOS permettant de communiquer avec les cartes SD.

7.4 Support de la carte SD pour TinyOS

Il n'existe pas à l'heure actuelle de pilote TinyOS-2.x destiné à l'exploitation d'une carte SD : la seule implémentation existante est destinée à la version 1.x. Cette solution ne répondant pas au modèle hiérarchique de TinyOS-2.x, il nous a été plus efficace de produire un nouveau pilote parfaitement intégré et homogène dans la version actuelle du système, permettant sa totale indépendance vis-à-vis du matériel.

Cette absence de support de carte SD dans TinyOS-2.x s'explique par la structure des plate-formes nativement supportées par TinyOS : celles-ci n'offrent du stockage non volatile que grâce à des composants de mémoire soudés sur la carte et ne répondant pas aux spécifications du protocole de la carte SD.

La carte SD permet de dialoguer soit à travers un protocole natif, soit à travers le protocole SPI. Les articles [9, 10] et le site [11] offrent la description et des exemples d'implémentation du protocole de communication par SPI. Il ne sera donc fait mention que des points qui sont spécifiques à l'implémentation d'un tel pilote pour TinyOS.

La carte SD offre les caractéristiques de lectures et d'écritures suivantes :

- lecture d'un nombre d'octets allant de 1 à 512 avec pour seule contrainte que l'adresse de début et l'adresse de fin soient contenues dans le même secteur. Le nombre d'octets étant par défaut 512, il est toutefois possible d'en fixer la valeur au moment de l'initialisation ou lors de l'écriture,
- écriture par blocs d'une taille fixe de 512 octets (1 secteur), l'adresse de début devant être un multiple de 512,
- effacement du contenu d'un nombre arbitraire de secteurs.

7.4.1 Connexion SPI

La première étape dans la création du pilote va consister, à l'instar du GPS, à créer les fichiers qui feront la liaison entre le pilote et le SPI du microcontrôleur. Globalement ces fichiers sont identiques à ceux utilisés pour le GPS ou la liaison RS232 :

```
#include "hardware.h"
configuration PlatformSdC {
  provides {
    interface SplitControl;
    interface SpiByte;
  }
}
implementation {
  components projetSdP;
  SplitControl = projetSdP.Control;

  components new Msp430Spi0C() as SpiC;
  projetSdP.Msp430SpiConfigure <- SpiC.Msp430SpiConfigure;
  projetSdP.Resource -> SpiC.Resource;
  SpiByte = SpiC;
}
```

Listing 7 – PlatformSdC

Le listing 7 présente le raccordement avec le module gérant le SPI sur le MSP430 (Msp430Spi0C). Contrairement au GPS, le dialogue avec le matériel se fait grâce à l'interface *SpiByte* qui n'offre qu'une seule fonction `command uint8_t write( uint8_t tx );`. Celle-ci envoie un octet et retourne la réponse du périphérique.

L'autre fichier comporte, comme pour le GPS, une structure pour la configuration de la communication (Fig. 13).

7.4.2 Interface d'exploitation

La seconde étape consiste à définir les interactions entre le pilote SD et les couches supérieures. L'utilisation d'un mécanisme tel que celui mis en œuvre pour le RS232 impose que le pilote contienne un tampon de 512 octets, *i.e.* le nombre d'octets d'un secteur.

```
msp430_spi_union_config_t msp430_spi_proj_config =
{{
  ubr: 0x0004,
  ssel: 11,
  clen: 1,
  listen: 0,
  mm: 1,
  ckph: 1,
  ckpl: 0,
  stc:1
  }};
```

FIG. 13 – Configuration de l'interface SPI

Cependant, la mémoire d'un microcontrôleur étant relativement faible, cet encombrement risquerait de réduire le volume de mémoire disponible pour une application utilisant ce pilote. Le module a donc été développé sur un modèle classique de fonctions ne rendant la main qu'une fois l'action finie. C'est l'application qui fournit le tampon contenant les données qui ne seront pas copiées au niveau du pilote. Ainsi, le choix du fonctionnement pour les couches exploitant la SD est laissé au soin du développeur.

L'interface se présente ainsi :

```
/**
 * SdIO
 * sert à l'exploitation de la carte sd
 * @author Gwenhaël GOAVEC-MEROU
 */

interface SdIO {
  /**
   * Commande pour la demande d'écriture d'une chaine
   * la commande est immédiate (au retour l'action est faite)
   *
   * @param addr : adresse de début d'écriture
   * @param buf tableau à envoyer
   *
   * @return SUCCESS Si la commande est acceptée
   */
  command error_t write(uint32_t addr, uint8_t*buf);

  /**
   * Commande pour la demande de lecture d'une chaine
   * la commande est immédiate (au retour l'action est faite)
   *
   * @param addr : position de début de lecture
   * @param buf : tableau dans lequel mettre l'information
   * @param count longueur du tableau
   *
   * @return SUCCESS Si la lecture est bonne
   */
  command error_t read(uint32_t addr, uint8_t*buf, uint16_t *count);
}
```

Listing 8 – interface d'utilisation du module SD

Les deux fonctions prennent l'offset (en octet) du début de la zone à écrire ou lire, le tampon contenant les données à écrire ou dans lequel seront mises les données lues, et la taille lue.

7.4.3 Mesure de débit

Afin de pouvoir obtenir une mesure de débit, une petite application simple a été réalisée. Celle-ci se contente d'écrire 2048 fois un buffer de 512 octets.

Le temps d'écriture de 1Mo est de 1min53s soit un débit d'environ 9Ko/s.

8 Stockage formaté en mémoire non volatile

L'objectif de l'implémentation d'un système de fichiers dans TinyOS est d'être en mesure, sur le système embarqué, de stocker les données acquises, pour ensuite pouvoir les restituer et les exploiter à l'aide d'un ordinateur et ceci sans avoir besoin d'utiliser une fonction dédiée du système embarqué chargée de la restitution des informations.

Les raisons d'allier l'utilisation d'une carte mémoire amovible à un système de fichiers sont :

1. de réduire le temps d'arrêt du capteur, puisque la carte peut être rapidement échangée par une autre. Cette opération est plus fiable que le transfert par RS232, notamment en environnement hostile,
2. la carte sans système de fichier peut être utilisée en RawWrite (écriture brute des données sans formatage). Cette solution impose :
   - un post traitement avant exploitation des données,
   - que l'utilisateur ait certaines compétences afin de récupérer les données,
   - une limite dans la possibilité de stocker des informations de diverses natures dans des emplacements séparés tels que le permet une multiplicité de fichiers.

Le choix d'un système de fichier répond à plusieurs critères :

1. à ressources réduites, système économe, ce qui exclut d'emblée tous les systèmes journalisés, qui bien que réduisant le risque de perte de données, entraînent l'augmentation du nombre de lectures, d'écritures et donc de traitements,
2. il faut qu'il soit multiplateforme afin que n'importe qui puisse nativement être en mesure de récupérer le contenu.

Le meilleur choix concerne un système de fichier originellement développé pour des ordinateurs disponibles il y a une vingtaine d'années, dont les performances étaient proches de celles des microcontrôleurs actuels, tels que Minix, CP/M ou FAT. Le seul survivant encore largement supporté est FAT, avec diverses déclinaisons dont la plus simple est encore compatible avec les systèmes d'exploitation les plus récents.

Remarque :
Tout support de stockage, pour être compatible et exploitable sur un ordinateur, doit comporter au tout début un MBR qui fournit des informations relatives au support en lui même et les informations sur les partitions du support.
Dans le cas de l'implémentation dans TinyOS, la seule information nécessaire se trouve être le numéro du premier secteur de la partition utilisée.
La formule pour trouver cette information est :
`debPartition = (*(uint32_t *)&buf[(446+8)+((numPart-1)*16)]);`
446 correspond à la position de la table de partition, 8 est le décalage pour obtenir le numéro du secteur de début de partition et 16 est la taille d'une entrée de partition.

Après une présentation globale de la structure d'un support de stockage et du système de fichier FAT16 [12], nous verrons les contraintes à prendre en compte lors de son implémentation afin de le rendre polyvalent, c'est-à-dire d'avoir de bonnes performances dans le cas d'acquisitions de données où le facteur temps est important, mais également économe en énergie dans le cas d'acquisitions de petits volumes d'informations sur un délai de plusieurs mois, voir années.

La troisième partie concernera l'implémentation à proprement parler dans laquelle nous présenterons le fonctionnement de chaque module, ainsi que les codes les plus pertinents.

8.1 Support physique et partitionnement

Un support de stockage physique, tel que la SD, n'est au final qu'un grand tableau, découpé en blocs plus petits, les secteurs, d'une taille de 512 octets pour la SD.

Afin d'utiliser un système de fichier, le support physique doit être partitionné (avec un outil comme *fdisk* par exemple). Ce partitionnement ajoute au tout début du support physique (adresse 0) une structure nommée MBR, ayant pour fonction de :

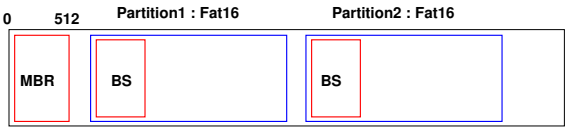

FIG. 14 – Exemple de partitionnement d'un support de stockage

– fournir des informations sur le support,
– stocker (si besoin) un exécutable permettant le boot depuis le support de stockage,
– fournir des informations telles que la taille et l'adresse des partitions contenues.

Nous allons nous intéresser surtout au dernier point car c'est grâce à cela qu'il est possible d'accéder à une partition.

```
000001B0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 01  ................
000001C0  01 00 06 0A  CA FF 0A 00  00 00 E4 A0  07 00 00 0A  ................
000001D0  CA FF 83 0A  CA FF EE A0  07 00 4A 42  0F 00 00 0A  ..........JB....
000001E0  CA FF 06 0A  CA FF 38 E3  16 00 78 CC  24 00 00 00  ......8...x.$...
000001F0  00 00 00 00  00 00 00 00  00 00 00 00  00 00 55 AA  ..............U.
```

FIG. 15 – Descripteur de partitions, la partition 4 (vert foncé) n'est pas utilisée.

Cette partie du MBR (Fig. 15) contient quatre entrées de 16 octets. Chacunes d'elles fournissant l'ensemble des informations sur une partition, toutefois la seule donnée importante pour la suite est la position du début de la partition (cadre rouge vif), se trouvant à l'offset 0x1C6 (dans le cas de la première partition).

Une fois la position de la partition trouvée, il est possible de se rendre à cette adresse afin de pouvoir exploiter celle-ci.

8.2 Structure

Avant de présenter le système de fichiers en lui-même, il est nécessaire de préciser à quoi correspond un cluster. Au sens de ce système de fichiers, c'est l'unité de base. Il correspond à un agglomérat de n secteurs. La taille est contenue dans la zone décrivant la partition.

Le système de fichiers FAT16 est structuré de la manière suivante :

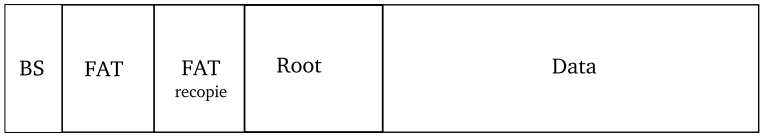

FIG. 16 – Structure d'une partition FAT16

8.2.1 Boot Sector

Au début de la partition se trouve le Boot Sector (BS). Il a le même rôle au niveau de la partition que le MBR pour le support de stockage entier. Il contient l'ensemble des informations nécessaires à l'exploitation de la partition.

Entre autre le BS contient la taille :
– de la partition,
– des diverses zones,
– d'un cluster,
– ...

Seul le BS possède une taille définie dans les spécifications, l'accès au reste des zones de la partition se fait grâce aux informations issues de cette zone.

8.2.2 FAT (File Allocation Table)

Dans la suite, afin d'éviter les confusions, le système de fichiers sera désigné par le mot *FAT16* tandis que les listes chaînées le seront par *FAT*.

Le système de fichiers contient deux *FATs*. La seconde sert de sauvegarde et devrait (si tout s'est bien passé) être la copie conforme de la première.

Cette structure est une liste simplement chaînée de clusters. Le contenu du fichier étant stocké par paquets de n secteurs non consécutifs, la relation entre chacun d'eux se fait grâce à la *FAT*. Le cluster débutant les données d'un fichier est indiqué par l'entrée de fichier dans la zone Root-DirSector.

Les premiers 16 bits de la *FAT* donne le type du médium, les seconds donne l'état de la partition. Le premier cluster exploitable de cette zone possède le numéro 2, le dernier dépend de la taille de la partition. Chaque index de clusters est codé sur 16 bits, la valeur d'un index pouvant-être de trois types :

- égale à 0x0000 pour signifier que le cluster n'est pas utilisé et peut donc être réservé,
- égale à 0xFFFF pour signifier que d'une part le cluster est déjà réservé mais qu'en plus il correspond au cluster de fin de fichier,
- avec une valeur comprise entre les deux précédentes pour donner l'index de son successeur dans la liste.

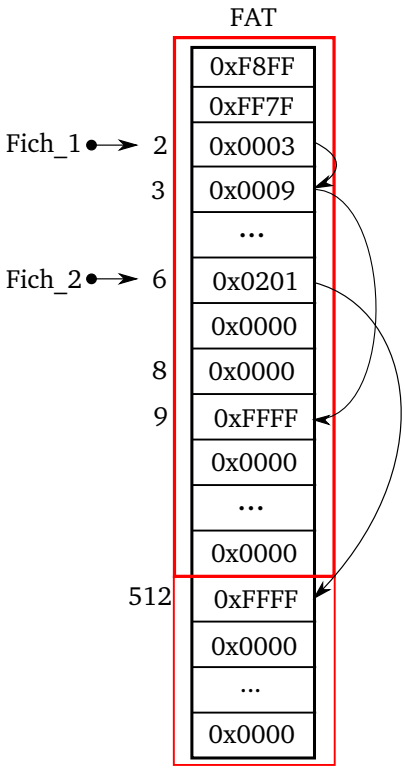

FIG. 17 – Exemple de FAT

La Fig.17 présente un exemple simple, dans lequel il existe deux fichiers. Le contenu du premier

- débute au cluster 2 (information obtenue depuis l'entrée du fichier,
- se continue au cluster 3 (valeurs du cluster 2),

– pour se finir au 9 (possède la valeur 0xffff).

Il en va de même pour le second fichier commençant au cluster 6.

8.2.3 Répertoire racine

Le répertoire racine de la partition (*RootDirSector*) contient l'ensemble des fichiers et répertoires se trouvant sous la racine, chaque entrée de fichier fournissant l'ensemble des données relatives à celui-ci. Les plus importantes étant l'index du premier cluster des données, ainsi que la taille du fichier.

```
0003E600   41 74 00 6F  00 74 00 6F  00 2E 00 0F  00 83 74 00  At.o.t.o......t.
0003E610   78 00 74 00  00 00 FF FF  FF FF 00 00  FF FF FF FF  x.t.............
0003E620   54 4F 54 4F  20 20 20 20  54 58 54 20  00 64 B8 5E  TOTO    TXT .d.^
0003E630   85 3B 85 3B  00 00 B8 5E  85 3B 02 00  00 00 00 00  .;.;...^.;......
0003E640   E5 74 00 65  00 6D 00 70  00 2E 00 0F  00 81 74 00  .t.e.m.p......t.
0003E650   78 00 74 00  00 00 FF FF  FF FF 00 00  FF FF FF FF  x.t.............
0003E660   E5 45 4D 50  20 20 20 20  54 58 54 20  00 64 B8 5E  .EMP    TXT .d.^
0003E670   85 3B 85 3B  00 00 B8 5E  85 3B 00 00  00 00 00 00  .;.;...^.;......
0003E680   42 2E 00 74  00 78 00 74  00 00 00 0F  00 8A FF FF  B..t.x.t........
0003E690   FF FF FF FF  FF FF FF FF  FF FF 00 00  FF FF FF FF  ................
0003E6A0   01 6E 00 6F  00 6D 00 5F  00 74 00 0F  00 8A 72 00  .n.o.m._.t....r.
0003E6B0   65 00 73 00  5F 00 6C 00  6F 00 00 00  6E 00 67 00  e.s._.l.o...n.g.
0003E6C0   4E 4F 4D 5F  54 52 7E 31  54 58 54 20  00 64 87 62  NOM_TR~1TXT .d.b
0003E6D0   85 3B 85 3B  00 00 87 62  85 3B 09 00  00 00 00 00  .;.;...b.;......
0003E6E0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
0003E6F0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
```

FIG. 18 – Répertoire racine d'une partition *FAT16*

Pour chaque entrée de fichier, le premier octet a une valeur particulière. Lorsqu'il vaut :

– **0x00**, le parcours est fini. Il n'y a plus de fichiers ensuite (comme l.15 de 18),

– **0xE5**, le fichier est effacé (l.5 et 7),

– **0x4n**, pour le début du nom long, le n donne le nombre de lignes utilisées pour stocker le nom long (n appartient à $[1;9]$) (l.9),

– dans le reste des cas, il correspond au premier caractère du nom du fichier (l.3 et l.13).

La recherche d'un fichier se fait donc en parcourant cette zone et en étudiant chaque entrée de fichier une à une.

Si l'on ne s'interésse qu'au nom DOS (nom court), le parcours se fait de la façon suivante : au départ du parcours, on peut s'attendre soit à un fichier supprimé, soit à un nom long.

– dans le premier cas, il suffit de passer 32 octets pour arriver sur l'entrée suivante.

– dans le second cas, il faut passer $n * 32 octets$ pour parvenir au nom DOS (format 8.3).

8.2.4 Zone de données

Le reste de la partition contient les données structurées sous la forme de cluster, eux-mêmes divisés en secteurs. L'accès n'est possible qu'en se servant de la *FAT* comme fil conducteur du parcours, tel que présenté précédemment.

8.3 Problématique

Un système de fichier, bien que créé pour des ordinateurs avec des ressources quasi équivalentes à celles d'un microcontrôleur moderne, entraîne des contraintes à prendre en compte. En plus de celles-ci, le support de stockage, en l'occurrence la SD, impose également son lot de contraintes. C'est ce que nous allons voir maintenant.

Tel que montré dans la section précédente, les FATs sont des listes simplement chaînées. Ainsi, lors de l'ouverture d'un fichier, ou lors du montage de la partition, il est nécessaire de retrouver le dernier cluster des données du fichier et le premier cluster libre. Ceci se fait par des "sauts de puce", dans le cas du fichier, en passant d'une entrée de cluster à sa suivante. Dans le cas de la recherche d'un cluster libre, toutes les entrées contenues dans un secteur sont balayées. Dans les deux cas, dès lors que le cluster suivant se trouve dans un autre secteur, il devient nécessaire de faire une

nouvelle lecture sur le support physique. Dans le cas d'une application se réveillant périodiquement pour faire un stockage, ceci peut donc entraîner la lecture d'un très gros volume de données et donc augmenter le temps nécessaire à l'initialisation, ainsi que la consommation d'énergie. Une solution serait de stocker ces informations quelque part tel que dans un fichier de configuration. Seulement cette solution ne serait valable que dans le cas d'un fichier, car imposant d'avoir déjà la partition initialisée, et par ailleurs, nécessiterait un parcours additionnel pour l'ouverture de ce fichier. Une autre solution serait de stocker ces informations dans une zone non utilisée du *BS*, mais ceci peut compromettre d'autres informations.

La solution mise en œuvre consiste à mettre en veille et à réveiller le descripteur de fichier et la partition. La seule différence, par rapport à un arrêt et un démarrage, réside dans la conservation des numéros de clusters en mémoire. Les gains de cette solution ne sont, certes, que partiels, limités au cas d'un réveil sur une application périodique, car lors du démarrage de l'application il est nécessaire de rechercher les clusters. Toutefois, et c'est également le cas pour la solution basée sur la lecture d'informations stockées quelque part sur la carte, il est préférable de considérer que la seule source d'information valable est issue de la *FAT16*.

Le second problème concerne le stockage des données acquises. En effet, la SD imposant l'écriture de 512 octets, il faut pouvoir concaténer les nouvelles données.

Dans le cas de l'écriture permanente de ce même volume d'informations, ceci ne pose pas de problèmes, mais si chaque écriture concerne un volume plus faible, il devient nécessaire de ne pas perdre ce qui est déjà écrit. Une solution, naïve, serait de dire que lors de chaque écriture une lecture du secteur serait faite afin de connaître à l'aide de *strlen()*, par exemple, à quel endroit ajouter les nouvelles informations.

Toutefois, cette solution pose de nouveaux problèmes :

- dans le cas de l'écriture de 512 octets, la phase de lecture est inutile voir même néfaste sur l'autonomie,
- pour pouvoir connaître le volume de données déjà stockées, il devient nécessaire d'assurer que de précédentes données ne viendraient pas parasiter le comptage, en d'autres termes, que le secteur soit "nettoyé" avant son utilisation. Pour cela, lors de la réservation d'un cluster, il faudrait remettre à "zéro" l'ensemble des secteurs contenus dans celui-ci, entraînant par la même occasion la multiplication par deux du volume écrit,
- pour finir avec les inconvénients de cette solution, dans le cas, par exemple, de l'utilisation de certains GPS dont les informations sont binaires, le comptage risque d'être faussé par la présence d'octets ayant pour valeur 0x00.

Ce problème a été résolu grâce à une variable se trouvant au niveau du descripteur de fichier : celle-ci, initialisée par défaut par bytesUsed = fileLength & 0x1FF;, permet de connaître le nombre d'octets utilisés dans le dernier secteur. Elle permet donc de savoir :

- s'il est possible d'écrire directement (si égale à 0),
- de passer au secteur et/ou cluster suivant (si égale à 512),
- ou bien s'il faut faire une lecture et ajouter les nouvelles données à la suite de celles existantes. Un simple `memcpy` utilisant cette variable permet cet ajout, sans faire l'hypothèse d'un contenu "propre".

Maintenant que les difficultés ont été présentées, nous allons voir le fonctionnement des modules ainsi que les points les plus importants en terme d'algorithmes.

8.4 Implémentation

L'ensemble des modules composant l'implémentation de la *FAT16* se présente ainsi :

Toute l'implémentation ne sera pas présentée, les documents [14] et [13] présentent la globalité de ce système de fichier. D'autre part les sources du pilote sont disponibles à l'adresse http://www.trabucayre.com/tinyos/fat.html.

L'implémentation réalisée ne permet que l'écriture de données. Il n'est pas possible depuis une application de lire le contenu d'un fichier. Ceci s'explique par le fait qu'un stockage en mémoire non volatile peut avoir plusieurs usages en ce qui concerne la lecture.

- le stockage temporaire des informations en vu d'une transmission radio (par exemple),

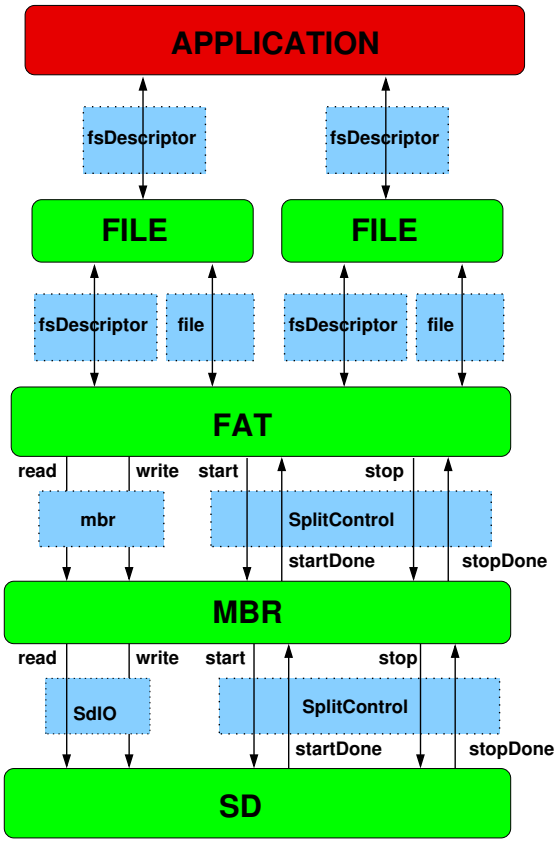

FIG. 19 – structure globale de l'implémentation

– le stockage de paramètres de configuration pour le nœud.

Dans ces deux cas il nous a semblé qu'un système de stockage avec lecture n'est pas adapté :

– dans le premier cas, la solution du système de stockage formaté est faite pour éviter l'aspect transmission et augmenterait la consommation globale,
– La seconde raison, vient du fait que TinyOS fournisse une solution utilisant les puces de flash de la plupart des cartes commerciales. Cette solution est mieux adaptée pour le stockage de configuration et de données temporaires en attente de transmission.

La suite de la présentation se fera de bas en haut sur le schéma, en commençant par le MBR.

8.4.1 MBR

La dernière couche avant la SD est un module gérant le MBR du support de stockage, son rôle se borne à :

– transmettre la commande de démarrage et d'arrêt au support de stockage,
– trouver la position du début de la partition lors de son initialisation :

```
[...]
if (call SdIO.read(0,buf,512) == SUCCESS){
  debPartition =  (*(uint32_t *)&buf[pos+8])<<9;
  mbrState = MBR_IDLE;
  error = SUCCESS;
}
[...]
```

– Ajouter cet offset lors de l'écriture (ou de la lecture s'il y a lieu) des données.

```
command error_t mbr.read(uint32_t offset, uint8_t *buffer) {
  [...]
```

```
    error = call SdIO.read(debPartition+offset,buffer,512);
    [...]
  }
  command error_t mbr.write(uint32_t offset, uint8_t *buffer) {
    [...]
    error = call SdIO.write(debPartition+offset,buffer,512);
    [...]
  }
```

8.4.2 FAT16

Ensuite vient le module FAT, qui réalise les plus gros traitements : lors du démarrage du système de fichier, il commence par lancer l'initialisation du *MBR* et analyse le *BS* afin d'extraire l'ensemble des informations nécessaires.

```
[...]
if (call mbr.read(0,buf) == SUCCESS) {
  BytsPerSec = (buf[12]<<8)+buf[11];
  SecPerClus=buf[13];
  RsvdSecCnt = (buf[15]<<8)+buf[14];
  NumFATs=buf[16];
  RootEntCnt = (buf[18]<<8)+buf[17];
  FATSz = (buf[23]<<8)+buf[22];
  FirstFatByts = RsvdSecCnt*BytsPerSec;
  // Repertoire racine
  RootDirByts = (RsvdSecCnt + NumFATs*FATSz)*BytsPerSec;
  // position premier secteur de donnees
  FirstDataSector = RootDirByts +
    fatAlignSup(RootEntCnt,BytsPerSec)*BytsPerSec;
  CountOfClusters = FATSz*((BytsPerSec/2)-2);
  if (fatState != FAT_SUSPEND)
    lastFreeCluster = 3;// Commence au début
[...]
```

Il se charge de la réservation des clusters : ceci ce passe exclusivement au niveau de la FAT du système de fichiers. La première étape consiste à obtenir la position, en terme d'offset et de secteur, de la dernière entrée de cluster précédemment réservée (ou de la première disponible). L'offset, d'une part donne la position par rapport au début du secteur, en cours d'utilisation, dans la FAT et d'autre part permet de déterminer si toutes les entrées de cluster, contenues dans ce secteur, ont étées parcourues.

```
[...]
getOffsetAndSector(lastFreeCluster,&fatSec,&off);
if (off < 512) {
  currentFatSec = (512*(fatSec-1))+FirstFatByts;
  if (call mbr.read(currentFatSec, buf) == FAIL)
    goto end;
}
```

Ensuite, le secteur est parcouru jusqu'à trouver un cluster libre. Lorsque le secteur en cours a été entièrement parcouru, la lecture du secteur suivant est faite.

```
do {
  if (off >= 512) { // Si on dépasse le secteur
    fatSec++; // passage secteur suivant
    currentFatSec = (512*(fatSec-1))+FirstFatByts;
    off = 0;  // raz de l'offset
    // Lecture d'un nouveau secteur de donnée
    if (call mbr.read(currentFatSec, buf) == FAIL)
      goto end;
  }
  // Secteur trouvé maintenant faut faire propre
  if ( (*(uint16_t*)&buf[off]) == 0x00){
    error = SUCCESS;
    break;
  }
  lastFreeCluster++; // Passage au cluster suivant
  off+=2; // Prochaine position à lire
} while(lastFreeCluster < CountOfClusters);
```

```
[...]
```

La boucle se fait tant qu'un cluster libre n'a pas été trouvé ou que la FAT n'a pas été complétement parcourue.

Une fois un cluster trouvé, il est marqué comme utilisé en tant que fin de fichier.

```
(*(uint16_t *)&buf[off])=0xffff;
[...]
```

Puis les deux FATs sont remises à jour afin que l'ancien dernier cluster prenne comme valeur l'index du nouveau, et que le nouveau soit noté comme utilisé en tant que cluster final de la chaîne des données du fichier. Pour la mise à jour, deux cas de figure sont possibles :

- les deux clusters se trouvent dans le même secteur de la FAT, dans ce cas une seule écriture par FAT est nécessaire,

```
if (*cluster != 0) // au cas ou pas de precedent a mettre a jour
  (*(uint16_t *)&(buf[offOr]))=lastFreeCluster;
if (call mbr.write(currentFatSec, buf) == FAIL ||
  call mbr.write(currentFatSec+(FATSz*512),buf)==FAIL)
  goto end;
}
[...]
```

- les deux clusters sont dans deux secteurs différents. Dans ce cas, il sera nécessaire de les remettre à jour en deux écritures successives.

```
if (fatSec != secOr) {
  if (call mbr.write(currentFatSec, buf)== FAIL ||
    call mbr.write(currentFatSec+(FATSz*512),buf)== FAIL)
    goto end;
  if (*cluster != 0) {
    tmp = (512*(secOr-1))+FirstFatByts;
    if (call mbr.read(tmp, buf)== FAIL) goto end;
    (*(uint16_t *)&(buf[offOr]))=lastFreeCluster;
    if (call mbr.write(tmp, buf) == FAIL ||
      call mbr.write(tmp+(FATSz*512),buf)==FAIL)
      goto end;
  }
```

Une fois les FATs mises à jour, les modifications sont répercutées sur les variables.

```
*cluster = lastFreeCluster;
lastFreeCluster++; // Passe au suivant vu que le courant est pris
```

La troisième et dernière tâche importante se passe lors de l'écriture de données : si le secteur qui va être utilisé dépasse le nombre de secteurs du cluster en cours, une réservation de cluster est faite.

```
[...]
if (*secteur >= SecPerClus || *cluster == 0) {
  //remise à jour de la valeur de cluster
  if (fatReserveCluster(buf, cluster) == FAIL)
    goto end;
  *bytesUsed = 0; // Nouveau cluster donc rien deja ecrit
  *secteur = 0;
}
```

Ensuite la position absolue (depuis le début de la SD) du secteur courant (en octet) est calculé.

```
// calcul de la position
sector = computeSecFromCluster(*cluster)+((*secteur)*512);
memset(buf,'\0',512);
```

Si le secteur en cours d'utilisation n'est pas vide, son contenu est récupéré afin de faire une concaténation.

```
/* lecture de cette position seulement si deja du contenu */
if (*bytesUsed != 0) {
  if (call mbr.read(sector, buf) == FAIL)
    goto end;
}
```

Il évalue ensuite la taille des données déjà présentes, plus la taille des données à écrire, afin de savoir si elles dépassent la taille du secteur.

```
if (*bytesUsed + s > 512){
  (*secteur)++; // La suite dans le prochain secteur
  size = 512 - *bytesUsed;
```

```
} else {
  if (*bytesUsed + s == 512)      (*secteur)++;
  size = s;
}
```

Et pour finir, remplit le secteur, puis fait une mise à jour du numéro du secteur, ainsi que du nombre d'octets écrits.

```
memcpy(buf+*bytesUsed,c,size);
if (call mbr.write(sector,buf) == FAIL)
  goto end;
s-=size;
*bytesUsed = (*bytesUsed+size) & 0x1FF;
c += size;
[...]
```

Et ceci tant que l'ensemble des informations ne sont pas complétement écrites.

8.4.3 file

Son principe est assez similaire au descripteur de fichiers dans un système d'exploitation. Il gére tous les aspects liés spécifiquement au fichier en lui-même.

A l'initialisation, l'entrée pour le fichier concerné est recherchée :

```
[...]
// recup du premier secteur du root
if (call fat.fatReadRoot(0,buf) == FAIL)
  return ret;
do {
  // parcours du secteur
  c = buf+offset;
  if (c[0] == 0x00) { // Fin de fichier
    break; // Rien a trouver
  // un nom de fichier long, a skipper
  } else if ((uint8_t)c[0] == 229 || c[11] == 15) {
  } else if (c[11] == 0x20) { // Trouve un fichier
    strncpy(fileName,c,7);
    fileName[8]='\0';
    if (!strncmp(fileName,name,strlen(name))) {
      ret = SUCCESS;
      break;
    }
  }
  // passage a l'entree suivante
  offset += 32;
  // si depassement du secteur faut lire le suivant
  if (offset >= 512) {
    sector++;
    offset -= 512;
    call fat.fatReadRoot(sector,buf);
  }
}while(c[0] != 0x00);
// renvoi des valeurs
*off = offset;
*sect = sector;
[...]
```

Une fois celui-ci trouvé, l'ensemble des variables est rempli. Dans le cas où le fichier serait vide (numéro de cluster égal à zéro), il est fait une demande de réservation. Si le fichier a été mis en veille (commande *suspend()*), la recherche du cluster de fin de fichier ne sera pas réalisée car la variable contient déjà cette information.

```
/* Enregistrement du nom de fichier */
c = (buf+offset);
// Position premier cluster du contenu
dataLocation = c[26];
// Longueur du fichier
fileLength = (*(uint32_t *)&(c+sec)[28]);
// Position dans le root dir sector de l'entrée
rootSector = sec;
```

```
  rootOffset = offset;
10// Cluster de fin du fichier
  // Si le fichier est vide
12if (dataLocation < 2) {
    // Réservation d'un cluster d'emblé
14  if (call fat.reserveCluster(&dataLocation,buf) == FAIL){
      return FAIL;
16  }

18  lastCluster = dataLocation;
    lastSecteur = 0;
20  bytesUsed = 0;
    call fat.fatReadRoot(rootSector,buf);
22  (*(uint16_t *)&(buf+rootOffset)[26]) = dataLocation;
    call fat.fatWriteRoot(rootSector,buf);
24} else {
    if (fileState == FILE_NOINIT) {
26    if ((lastCluster = call fat.fatLastCluster(dataLocation,buf))
                == -1)
28      return FAIL;
      if ((lastSecteur = call fat.fatLastSecteur(lastCluster,buf))
30              == -1)
        return FAIL;
32  }
    bytesUsed = fileLength & 0x1FF;
34}
```

Lors d'une demande d'écriture, le tableau contenant les données est passé au système de fichiers, ainsi que le nombre d'octets déjà écrits. À l'issue de l'écriture des données, la taille du fichier est mise à jour. Ce comportement, bien que nécessitant une écriture de plus, apporte une sûreté quant aux données écrites. En effet, si une coupure d'alimentation avait lieu, dans le cas contraire il ne serait pas possible de récupérer les données facilement.

8.5 Exemple d'utilisation

Pour montrer comment utiliser cette implémentation de la *FAT16*, nous allons présenter deux exemples. Le premier se contente de lancer le système de fichier, d'ouvrir le fichier *toto.txt*, dans lequel il écrit *"hello World"*, puis finit en fermant le fichier et le système de fichiers. Le second a pour but de montrer l'utilisation des méthodes de mise en sommeil et de réveil du système de fichier en lisant et stockant la température, à intervalles réguliers.

8.5.1 Utilisation de la FAT16

```
configuration fatTestAppC {}
2implementation {
  components fatTestC as App, MainC, fatC;
4 components new TimerMilliC() as Timer0;
  components new fileC("toto.txt");
6 App.Boot -> MainC.Boot;
  App.Timer0 -> Timer0;
8 App.fatControl -> fatC;
  App.file -> fileC;
10 App.fileControl -> fileC.fileControl;
  fileC.fat -> fatC.fat;
12}
```

Listing 9 – Fichier de configuration de l'application.

Le premier fichier (Fig. 9) câble l'ensemble des modules, avec dans le cas du fichier la spécification du nom de celui-ci.

```
module fatTestC {
2 uses {
    interface Boot;
4   interface Timer<TMilli> as Timer0;
    interface SplitControl as fatControl;
6   interface SplitControl as fileControl;
```

```
    interface fat;
8   interface file;
    }
10}
implementation {
12  event void Boot.booted()  {
    call Timer0.startOneShot(500);
14  }
    event void Timer0.fired() {
16    call fatControl.start();
    }
18  event void fatControl.startDone(error_t err){
    if (err == SUCCESS) call fileControl.start();
20  }
    event void fileControl.startDone(error_t err) {
22    if (err == SUCCESS)
        call file.write("hello world",11)
24  }
    event void file.writeDone(error_t err) {
26    if (err == SUCCESS) call fileControl.stop();
    }
28  event void fileControl.stopDone(error_t err) {
      if (err == SUCCESS) call fatControl.stop();
30    }
    event void fatControl.stopDone(error_t err){}
32  [...]
}
```

Listing 10 – Fichier de l'application.

Le second (listing 10) :

- monte la partition lorsque le timer arrive à expiration (l.15-17),
- si la partition est correctement montée, il lance l'ouverture du fichier (l.18-20),
- une fois le fichier ouvert, il fait une écriture de "hello world" dedans (l.21-24),
- une fois le contenu écrit ferme dans l'ordre inverse d'ouverture le fichier et la partition (l.25-31).

8.5.2 Stockage de la température et mise en veille

```
1configuration storeTempAppC {}
implementation {
3  components storeTempC as App, LedsC, MainC;
   App.Boot -> MainC.Boot;
5  App.Leds -> LedsC;
   components new TimerMilliC() as Timer0;
7  App.Timer0 -> Timer0;
   components fatC;
9  App.fatDescriptor -> fatC;
   components new fileC("temp.txt") as fileADC;
11 App.fileADC -> fileADC;
   App.fileADCDescriptor -> fileADC.fileDescriptor;
13 fileADC.fat -> fatC.fat;
   components new DemoSensorC() as Sensor;
15 App.readADC -> Sensor;
}
```

FIG. 20 – Fichier de configuration de l'application d'acquisition de température.

Le premier fichier (Fig. 20) contenant la configuration de l'application est globalement le même que celui de l'exemple précédent, hormis l'utilisation de *DemoSensorC* utilisé pour la température et le nom du fichier qui sera utilisé.

```
#include "Timer.h"
2#define TIMER_SLEEP 3600000
#define SHOW_ERROR do { \
4  call Leds.led1Off(); \
```

```
  call Leds.led0On(); } while(0)

module storeTempC {
  uses {
    interface Leds;
      interface Boot;
      interface Timer<TMilli> as Timer0;
    interface Read<uint16_t> as readADC;
      interface fat;
    interface fsDescriptor as fatDescriptor;
    interface file as fileADC;
    interface fsDescriptor as fileADCDescriptor;
  }
}
```

La vue externe est elle aussi globalement la même, avec seulement l'ajout de l'interface permettant l'exploitation de l'ADC12 du msp430.

La partie *implementation* peut être divisée en quatre grands blocs fonctionnels.

```
implementation {
  enum {
    APP_NOINIT,
    APP_STOP,
    APP_SLEEP,
    APP_ADC
  };
  uint8_t appState = APP_NOINIT;
  uint8_t *tampon=NULL;

  event void Boot.booted() {
    tampon    = (uint8_t *) malloc(8*sizeof(uint8_t *));
    appState  = APP_NOINIT;
    call Timer0.startPeriodic(TIMER_SLEEP);
  }
  event void Timer0.fired() {
    error_t error = FAIL;
    call Leds.led1Off();
    call Leds.led0Off();
    if (appState == APP_NOINIT || appState == APP_STOP)
      error = call fatDescriptor.open();
    else if (appState == APP_SLEEP)
      error = call fatDescriptor.resume();
    if (error == FAIL)
      SHOW_ERROR;
  }
```

La première concerne la définition de variables nécessaires pour conserver l'état de l'application au fil du temps et d'un tableau nécessaire à l'écriture sur la carte. Nous définissons plusieurs états (l.2-7) et une variable pour les stocker, afin que lors de l'expiration du timer (l.16-26) l'application soit en mesure d'ouvrir le système de fichier (appel à *call fatDescriptor.open()*) ou de le réveiller (appel de *call fatDescriptor.resume()*) .

Le reste concerne l'initialisation du timer et la gestion de l'expiration de celui-ci.

```
  event void fatDescriptor.openDone(error_t error){
    if (error == SUCCESS) error = call fileADCDescriptor.open();
    if (error == FAIL)    SHOW_ERROR;
  }
  event void fileADCDescriptor.openDone(error_t error) {
    if (error == SUCCESS){
      atomic { appState = APP_ADC;}
      error = call readADC.read();
    }
    if (error == FAIL) {
      call fatDescriptor.close();
      SHOW_ERROR;
    }
  }
  event void fatDescriptor.resumeDone(error_t error){
    if (error == SUCCESS) error = call fileADCDescriptor.resume();
```

```
    if (error == FAIL)     SHOW_ERROR;
  }
  event void fileADCDescriptor.resumeDone(error_t error) {
    if (error == SUCCESS){
      atomic { appState = APP_ADC;}
      error = call readADC.read();
    }
    if (error == FAIL) {
      call fileADCDescriptor.close();
      SHOW_ERROR;
    }
  }
```

Le second bloc concerne les aspects liés au démarrage/réveil du système de fichier. Comme nous pouvons le voir, il n'y a pas une grosse différence au niveau de l'application entre démarrage et réveil. Lors de la réussite de l'opération au niveau du système de fichier, le fichier lui-même est relancé et à la fin de l'opération, une acquisition est faite.

```
  event void fileADCDescriptor.closeDone(error_t error) {
    if (error == SUCCESS) error = call fatDescriptor.close();
    if (error==FAIL)     SHOW_ERROR;
  }
  event void fatDescriptor.closeDone(error_t error){
    if (error == FAIL)  SHOW_ERROR;
    else         atomic {appState = APP_STOP;}
  }
  event void fileADCDescriptor.suspendDone(error_t error) {
    if (error == SUCCESS) error = call fatDescriptor.suspend();
    if (error == FAIL)     SHOW_ERROR;
  }
  event void fatDescriptor.suspendDone(error_t error){
    call Leds.led1Off();
    if (error == FAIL)     SHOW_ERROR;
    else           atomic { appState = APP_SLEEP;}
  }
}
```

Le troisième bloc est le pendant du précédent et concerne la gestion de l'arrêt/mise en veille. Les opérations étant faites dans l'ordre contraire du démarrage.

```
  event void readADC.readDone(error_t result, uint16_t data) {
    uint16_t inter, i, z;
    float val = (((data/4096.0)*1.5)-0.986)/0.00355;
    memset(tampon,'\0',8*sizeof(uint8_t));
    for(i=0,z=100;i<3;i++,z/=10){
      inter = val / z;
      tampon[i] = inter+'0';
      val -= inter*z;
    }
    tampon[3] = '\n';
    appState = APP_SLEEP;
    if (call fileADC.write(tampon,4) == FAIL)
      SHOW_ERROR;
  }
  event void fileADC.writeDone(error_t error) {
    if (error == SUCCESS)
      error = call fileADCDescriptor.suspend();
    if (error == FAIL){
      call fileADCDescriptor.close();
      SHOW_ERROR;
    }
  }
```

Le dernier correspond au traitement à proprement parler. Bien que ce soit la partie la plus courte de l'application, c'est aussi la partie qui contient tout le fonctionnement de celle-ci.

readADC.readDone(...) est exécutée lorsque le microcontrôleur a finit la lecture de la température. Après une conversion en chaîne de caractères, une demande d'écriture est faite. Quand cette dernière est finie, le système de fichier est mis en sommeil, jusqu'à la prochaine expiration du timer périodique.

8.6 Test de vitesse et de portabilité

Pour connaître la vitesse d'écriture, une application a été réalisée. Celle-ci est équivalente à celle pour la mesure du débit de la SD en rawWrite. Le test a duré pendant 5 min 41 s présentant une vitesse approximative de 3,2 kB/s. Comparé à la vitesse de rawWrite, trois fois plus élevée, le résultat, compte tenu du nombre d'opérations réalisées, est cohérent.

Cette vitesse est largement suffisante pour une application pratique de stockage de trames GPS – incluant l'utilisation d'un récepteur fournissant l'information de phase tel que le Thalès AC12 – dont le débit mesuré peut être au maximum de 3200 b/s (ou 0,4 kB/s).

L'ensemble du travail de développement des pilotes a été réalisé sur notre plateforme expérimentale. Toutefois, afin de pouvoir valider la bonne intégration de l'ensemble dans TinyOS, nous les avons également testés sur des plateformes commerciales, nativement supportées par TinyOS.

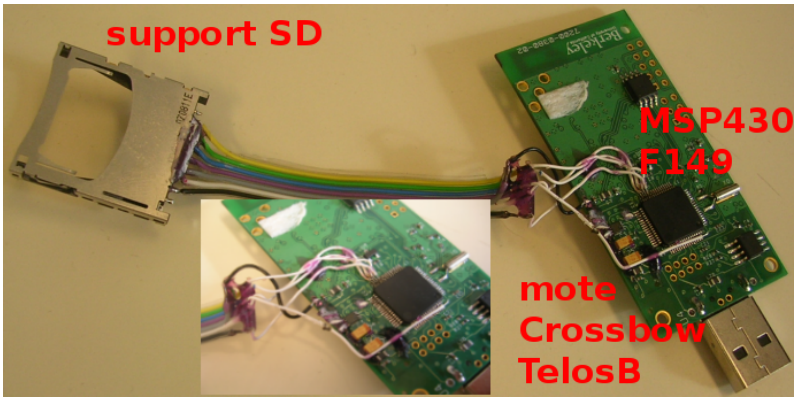

FIG. 21 – Carte TelosB – commercialisée par Crossbow – équipée d'une carte SD

La première, la telosB (Fig. 21), est également basée sur un MSP430. L'utilisation de la SD ainsi que de la FAT16 s'est bornée à copier les fichiers de configuration de la plateforme et à reconfigurer le *SPI* pour faire usage du quartz 32 kHz (la *telosB* ne contient pas de quartz haute-fréquence). Comme l'architecture du microcontrôleur MSP430F1611 est la même que celle utilisée sur notre carte de prototype, la liaison de la carte SD s'effectue en connectant les mêmes broches que dans notre exemple : Chip Select sur P3.0 (broche 28), MOSI sur P3.1 (broche 30), MISO sur P3.2 (broche 30) et l'horloge sur P3.3 (broche 31).

La seconde plateforme est une *micaZ*, celle-ci est sur une base d'un *ATMEGA128* [16]. La mise en œuvre des pilotes sur celle-ci n'a pas pris plus d'une trentaine de minutes, après identification du fonctionnement du SPI sur cette carte. En effet, TinyOS utilise une version logicielle et non matérielle de ce protocole. Sur cette architecture, la carte SD est reliée aux broches suivantes du connecteur 51 broches de la carte *MicaZ* :

- Chip Select est connecté à la broche LED1 (broche 10, PA2),
- MISO est connecté à la broche USART1 RXD (broche 19, PD2),
- MOSI est connecté à la broche USART1 TXD (broche 20, PD3)
- finalement, l'horloge (Clock) est fournie par USART CLK (broche 15, PD5).

9 Conclusion

Nous avons proposé quelques exemples de développements autour de TinyOS-2.x sur plateforme MSP430, en partant des cas les plus simples des entrées/sorties numériques, pour ensuite exploiter les ports de communication série asynchrone et synchrone, et conclure sur l'implémentation d'un système de stockage de fichiers sur support formaté compatible avec la majorité des systèmes d'exploitations sur PC.

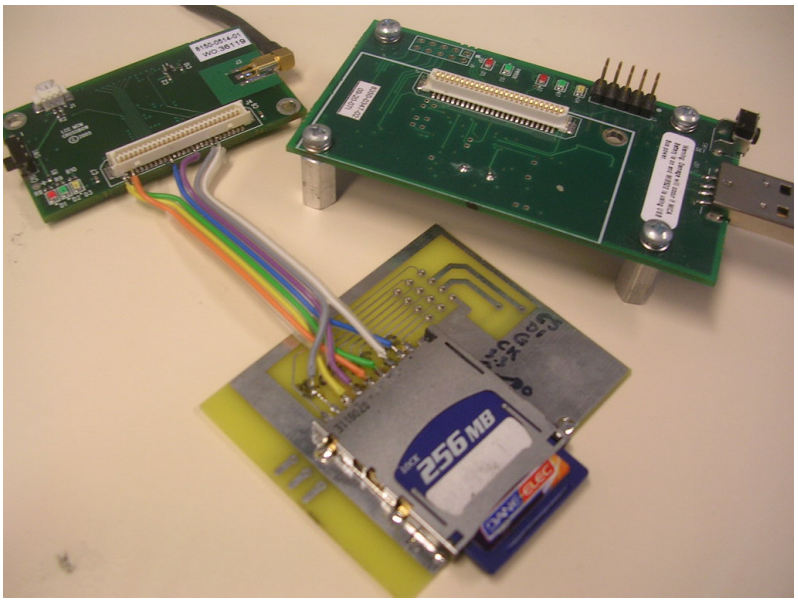

FIG. 22 – Carte MicaZ – commercialisée par Crossbow – équipée d'une carte SD

Le premier résultat concerne les pilotes qui ont étés réalisés : afin d'évaluer la validité à la fois du pilote GPS et du système de fichiers, la plateforme a été exploitée en pratique avec une alimentation sur accumulateurs pour l'acquisition de trames GPS. Une autonomie de plus de 25 h a ainsi pu être validée lors de l'acquisition de traces GPS de plusieurs MB sans corruption de la carte mémoire.

Après récupération du fichier sur un ordinateur, la trace GPS est traitée au moyen de GpsQt [3] pour insertion sur un fond de carte Google Maps (Fig. 23, d'autres itinéraires sont disponibles sur http://www.trabucayre.com/gps).

[3] http ://projetaurore.assos.univ-fcomte.fr/gps/

FIG. 23 – Tracé d'un trajet entre la région de Brest et le nord de Besançon (en rouge), sur un fond de carte Google Maps : 950 km pendant lesquels le circuit a fonctionné sur 4 accumulateurs NiMH pendant 11 h. Le résultat est un fichier de 142411 lignes pour une taille de 9,2 MB.

À temps de développement équivalent qu'en C ou en assembleur, pour une application proposant les mêmes fonctionnalités, le principal intérêt de TinyOS tient en la possibilité de réutiliser le code sans avoir à reprendre l'ensemble de l'implémentation de la communication avec la carte SD et le format FAT.

Par ailleurs, nous avons constaté que l'ajout de pilotes pour des périphériques non-supportés par défaut (par exemple l'écran LCD compatible HD44780) est simplifié par la ressemblance syntaxique du nesC et du C : une implémentation du protocole existante en C est rapidement adaptée aux contraintes de TinyOS. Ceci permet également de réaliser une première implémentation des spécifications sur un ordinateur afin de valider le code avant de le convertir en un module TinyOS comme ce fut le cas avec la FAT.

Les routines offertes par TinyOS donnent la possibilité au programmeur de se concentrer sur une tâche bien ciblée sans avoir à reprendre ou réadapter un autre code de communication avec le matériel. Il est par exemple trivial de réaliser une application utilisant une communication RS232 car le système offre d'emblée toutes les bibliothèques nécessaires à son exploitation.

TinyOS-2.x manque encore d'une certaine maturité. Il est, par exemple, étonnant de se voir contraint à un nombre pré-défini de LEDs, nécessitant dans certains cas de surcharger le module officiel. Le support du stockage n'est, contrairement au reste du système, pas indépendant du composant physique. Cela force donc à réimplémenter toutes les spécifications si le composant (par exemple la carte SD) n'est pas supporté d'origine. Nous avons néanmoins constaté que la hiérarchie des pilotes et la possibilité de réutilisation de méthodes existantes pour l'accès au matériel rend ce travail moins fastidieux qu'en développant du code de bas niveau sans exploiter les fonctionnalités de l'environnement exécutif.

Références

[1] T. Liu, C.M Sadler, P. Zhang & M. Martonosi, *Implementing software on resource-constrained mobile sensors : experiences with impala and zebranet*, Proceedings de MobiSYS'04 (6-9 Juin 2004), disponible à www.cs.princeton.edu/~tliu/p206-liu.pdf

[2] le projet TurtleNet est décrit à prisms.cs.umass.edu/dome/turtlenet

[3] le projet Argo est décrit à www.argo.ucsd.edu, le projet COMMON-Sense Net est à commonsense.epfl.ch, le projet Sensorscope à senseorscope.epfl.ch détaillé dans www.sics.se/realwsn05/slides/schmid-sensorscope.pdf, suivi de volcans par un groupe de Harvard à fiji.eecs.harvard.edu/Volcano, un projet de Berkeley de suivi de croissance d'arbres dans www.eecs.berkeley.edu/~get/papers/tolle05redwoods.pdf, suivi de glaciers à www.sics.se/realwsn05/papers/martinez05glacial.pdf (projet Glacsweb à envisens.org/glacsweb.htm) et projet Seamonsterak décrit dans esto.nasa.gov/conferences/estc2008/presentations/HeavnerA9P3.pdf, observation de la propagation de feux de forets par firebug.sourceforge.net

[4] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman & M. Yarvis, *Design and deployment of industrial sensor networks : experiences from a semiconductor plant and the north sea*, Proc. of the 3rd international conference on embedded networked sensor systems, pp.64-75, (2005)

[5] Référence du language nesC : http://nescc.sourceforge.net/papers/nesc-ref.pdf.

[6] la programmation sur TinyOS 2.x : http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf

[7] Tutorial TinyOS 1.0 concernant la création d'une plateforme : http://docs.tinyos.net/index.php/Platforms

[8] MSP430x1xx Family User's Guide (2006), focus.ti.com/lit/ug/slau049f/slau049f.pdf

[9] J.-M. Friedt & S. Guinot, *Stockage de masse non volatile : un block device pour multimedia-card*, GNU/Linux Magazine France, Hors-série **25** (avril/mai 2006)

[10] J.-M. Friedt & É. Carry, *Enregistrement de trames GPS – développement sur microcontrôleur 8051/8052 sous GNU/Linux*, GNU/Linux Magazine France **81** (février 2006)

[11] Présentation du protocole de communication des SD et MMC en SPI à http://www.retroleum.co.uk/mmc_cards.html

[12] A. Bourgeois, *Croisière au cœur d'un OS : système de fichiers FAT*, GNU/Linux Magazine France **98** (Octobre 2007)

[13] Explication de la structure d'un support de stockage http://www.beginningtoseethelight.org/fat16/

[14] Spécificiations des partitions FAT download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc

[15] M. Tischer, *La bible du PC – programmation système, 6ème édition*, MicroApplications (1996)

[16] D. Bodor, *Découverte du microcontrôleur AVR*, GNU/Linux Magazine France Hors Série **23** (2008)

Gwenhaël Goavec-Mérou a obtenu son master d'informatique de l'université de Franche-Comté, et travaille actuellement dans l'équipe temps-fréquence de l'institut FEMTO-ST sur l'implémentation d'algorithmes de calculs dans les matrices de portes logiques reconfigurables (FPGA). Son projet de master était dédié au stockage de masse sur des nœuds de réseaux de capteurs faible consommation, et la programmation sous TinyOS, activité qu'il a poursuivi durant l'année passée pour obtenir un résultat exploitable par la communauté des utilisateurs TinyOS.

Jean-Michel Friedt est ingénieur, hébergé par l'équipe temps-fréquence de FEMTO-ST, et membre de l'association pour la diffusion de la culture scientifique et technique Projet Aurore. Son intérêt pour les capteurs actifs à faible-consommation est principalement dû aux applications de mesures en environnements sévères ou sur de longues durées, fastidieuses voir impossibles sans ces outils.