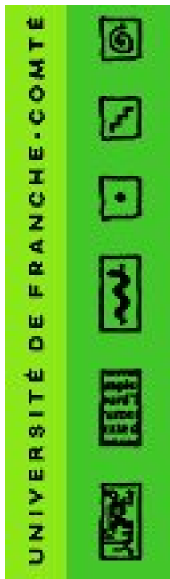

Université de Franche Comté
Master 2

Étude d'un système d'exploitation pour microcontrôleur faible consommation (TI MSP430)

Gwenhaël GOAVEC-MEROU
Benoit PIPART
Julien MASSON
<>

Besançon, Janvier 2009

Tuteurs :
Monsieur Eugène Pamba Capo-Chichi
Monsieur Jean-Michel Friedt

Première partie

Présentation

Chapitre 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, les 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 dû 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 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. Aussi, une programmation bas-niveau limite la portabilité d'algorithmes complexes sur des systèmes informatiques à caractéristiques différentes.

L'entité élémentaire à capacité réduite à laquelle nous nous intéressons est le nœud d'un réseau de capteur qui doit avoir une autonomie de plusieurs jours voir plusieurs années. En effet, ce type de systèmes informatiques, dit "embarqués", utilisés entre autre pour des études climatiques ou de dérives de glaciers, doivent pouvoir fonctionner dans des conditions pouvant être extrême en prenant en compte la contrainte énergétique.

Les données acquises par ces noeuds doivent pouvoir être envoyées par une communication radio ou bien conservées en vue d'un transfert ultérieur :

– La première approche n'est pas toujours adaptée, d'une part parce que la communication n'est pas fiable car soumise aux conditions de l'environnement et d'autre part elle constitue l'activité la plus gourmande en énergie. L'implémentation d'algorithmes de compressions permet de réduire le volume de communication, mais augmente proportionnellement la consommation de ressource de calcul.
– La seconde approche offre une plus grande sécurité sur le point de vue de 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.

Au vu de ce qui précède, le but de ce projet consiste en l'évaluation d'un système

d'exploitation spécialement dédié aux réseaux de capteurs, afin de pouvoir en définir les avantages et les inconvénients.

L'étude se fera à travers un cas concret consistant en la réalisation d'une application faisant l'acquisition de données GPS qui seront stockées sur une carte mémoire utilisant un système de fichier compatible avec n'importe quel ordinateur.
La plateforme (Fig. 1.1) servant de base dispose des caractéristiques suivantes :

- Elle est à base d'un msp430f149 disposant de 60KB de mémoire non volatile flash et de 2KB de RAM.
- Elle est équipée d'une antenne GPS.
- Elle offre une communication série en écriture seule
- Elle fournie un slot pour une carte mémoire de type *secure digital*
- Elle est alimentée à partir de 4 piles classiques pour une tension de 4.8V.

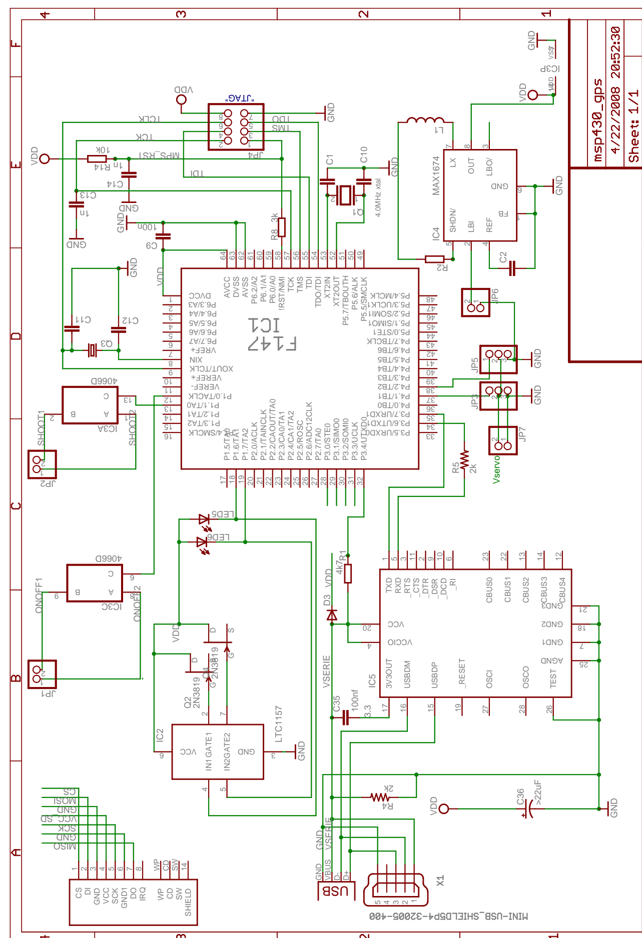

FIGURE 1.1 – Schéma de la plateforme

Chapitre 2

Exploitation d'un capteur

2.1 Généralité

Comme tout matériel informatique un capteur nécessite, utilise un programme pour effectuer sa tâche.

Deux grands axes existent afin de pouvoir rendre un système informatique fonctionnel :

- La première voie est celle empruntée pour l'exploitation des ordinateurs et des plateformes disposant d'un volume de ressources importantes. Le système d'exploitation qui équipe ce type de matériel offre un support natif pour un grand nombre de périphériques. Cette aptitude le rend gourmand en énergie et en ressources. Toutefois son défaut en fait aussi son avantage car l'architecture qui est mise en oeuvre permet à une application d'être en mesure de pouvoir fonctionner sur n'importe quelle machine sans avoir la moindre nécessité de savoir les caractéristiques bas niveau de celle-ci.
- La seconde voie est celle utilisée dans l'informatique embarquée. Elle est basée sur un language bas niveau tel que le C ou l'assembleur. Le programme dialogue directement avec le matériel ce qui le rend plus léger en terme de ressources et de capacités. En effet, il n'est plus sur un modèle de plus grand dénominateur commun comme l'OS mais sur un modèle de spécificité qui lui apporte cette caractéristique. Mais son avantage est aussi son inconvénient. Le logiciel et le matériel étant fortement liés, toutes modifications hardware entraînent nécessairement une modification logicielle.

Un système embarqué impose de part sa composition une solution pointue et économe en ressources. Un système d'exploitation tel qu'utilisé sur un ordinateur n'est clairement pas la solution.
Malgré tout, l'apport d'une solution permettant la portabilité du code est d'un réel intérêt pour des algorithmes complexes, tel que le support d'une solution de stockage commune à un ordinateur et à un capteur.

2.2 TinyOS

TinyOS se présente comme une solution intermédiaire entre une application bas-niveau apportant une faible empreinte mémoire mais trop spécifique au matériel et un système d'exploitation apportant une réelle notion d'abstraction entre l'application et le matériel mais qui n'est pas prévu pour du matériel à faible ressources.

TinyOS n'est pas un système d'exploitation au sens actuel du terme. Il n'offre pas par exemple de notion de multitâches, d'utilisateurs ou de système de fichier. Il n'y a pas de notions de mode utilisateurs et de mode noyau.
En fait, TinyOS est un ensemble de routines mises à la disposition du programmeur en vue de simplifier le développement. Le code nesC(AnnexeC) une fois interprété donne un code C qui est compilé. L'exécutable produit étant en tout point semblable à ce que pourrait générer une application bas niveau.

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 dés lors possible une portabilité quasi transparente sur une multiplicité de plateformes différentes, et d'autre part sans que le programmeur n'ai 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 sont déjà implémentées. Il ne reste donc qu'à ce concentrer sur l'application ou le driver 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 capteurs d'origines et de caractéristiques différentes.

Deuxième partie

Technique

Chapitre 3

Plateforme

La création d'une plateforme n'est nécessaire que lorsque la définition du capteur cible n'est pas fournie d'origine par TinyOS.
Comme les applications se compilent en spécifiant la plateforme cible, c'est l'étape obligée avant de pouvoir développer une application.
Celle-ci est une couche permettant l'abstraction entre les applications et la partie matérielle.
L'explication de la création d'une plateforme se fera à travers la réalisation de celle de la carte fournie et se nommera **projet**.
Cette explication se base sur le tutoriel TinyOS concernant la création d'une plateforme ([TosPlatform])
Le suite de ce chapitre n'a pas pour but de rentrer dans les détails du langage nesC, elle ne couvre qu'une explication globale du rôle de chaque fichier. Pour une explication plus approfondie se référer à l'annexe C.
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 définie 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.

3.1 Définition

Une plateforme permet une couche d'abstraction entre les applications et la carte/Mote utilisée.
En d'autres termes dans l'exemple des leds la hiérarchie du système se compose de la manière suivante :

1. L'application utilise le type *Leds* mais ne sait pas comment elles sont implémentées par le système, ni où elles sont connectées sur la carte. Il n'a besoin que de savoir les actions possibles.
2. Le type Leds ne connaît pas non plus comment sont connectées les leds. Il passe juste l'information (allumage, extinction, bascule) à travers le fichier que le créateur de la plateforme a réalisé.
3. Le fichier de la plateforme définit la liaison entre les leds et les ports du driver du microcontrôleur sans avoir de connaissance sur les adresses et sur la façon d'alimenter

ou non le port en question

4. Le driver du microcontrôleur fait au final l'opération adéquate pour alimenter ou pas celles-ci.

3.2 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. TinyOS utilise une sorte de FrameWork afin d'accéder aux fichiers nécessaires sans que le développeur n'ai la nécessité d'en préciser l'existence.

3.3 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.

3.3.1 projet.target

Dans le répertoire *support/make* nous créeons un fichier *projet.target* qui va contenir les lignes présentes dans le listing 3.1 :

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

projet: $(BUILD_DEPS)
  @:
```

Listing 3.1 – 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.

3.3.2 .platform

Pour la suite tout se déroulera dans *tos/platforms/projet*.
.platform concerne les spécifications sur les répertoires contenant les librairies et les options de compilation.
Contrairement à un logiciel pour un système d'exploitation d'ordinateur 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 3.2 – .platform

Le listing 3.2 contient les configurations dans le cas de la plateforme *projet*. Il est écrit en PERL et contient les deux parties suivantes :

1. les répertoires qui contiennent les librairies.
2. les options qui seront passées au compilateur.

3.3.3 hardware.h

Ce fichier est inclus lors de la compilation d'une application. Il sert à définir des constantes et des headers.

3.3.4 PlatformC.nc

```
#include "hardware.h"

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

Listing 3.3 – PlatformC

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

3.3.5 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 3.4 – PlatformP

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

3.4 Mise en pratique

3.4.1 Exploitation des Leds

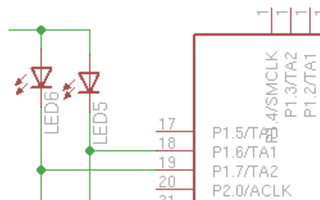

FIGURE 3.1 – câblage des leds

La figure 3.1 présente le câblage physique des 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 3.5 – PlatformLedsC

Cette configuration permet de préciser pour chaque Leds définies par le driver TinyOS leur branchement au niveau hardware.
Les 5 dernières lignes définissent que led0 est raccordé au Port16 (noté P1.6 dans la datasheet du msp430) et led1 est raccordé au Port17 (noté P1.7). Le cas de led2 est particulier car le driver Leds définie une troisième Leds absente du montage physique. celle-ci est connectée sur NoPinC signifiant sa non existence.
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 non présentes physiquement.
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 driver officiel.

3.4.2 Exploitation d'un LCD

Le protocole d'un LCD à base de HD44780 est facilement trouvable sur internet.
C'est un protocole qui nécessite 4 bits pour les données à transmettre ainsi que 2 bits pour les informations de contrôle.

```
#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 3.6 – PlatformLcdC

La figure 3.6 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. Son implémentation est très proche de celle des Leds. 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 est complexifié comme ceci :

```
// 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();
}
```

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

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

Chapitre 4

Communication

4.1 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 transition 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 dans la majorité des microcontrôleurs, et ce sera notre premier exemple de développement sous TinyOS.

4.1.1 Couche communication dans 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, car il permet d'homogénéiser la communication et d'en assurer la fiabilité.
Mais cette encapsulation de la communication est handicapante pour les outils classiques tel que *minicom* ou avec un périphérique exploitant une connection série. Par exemple, dans le cas du GPS, qui fournie ses informations au moyen d'une chaîne ASCII, le modèle TinyOS n'est pas exploitable. En effet ce dernier considérera la chaîne comme non valide et la rejettera. Cette solution n'apporte donc rien sur le noeud utilisé dans le cadre du projet.

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 driver et la couche hardware.
- De spécifier la configuration du port.

Pour une exploitation brut de l'UART il suffit donc d'utiliser ou de redéfinir ces deux fichiers sans passer par les couches de plus haut niveau.
Le listing suivant donne l'interface publique raccordant le driver 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;
}
```

Listing 4.1 – PlatformSerialC

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

- Une commande permettant l'envoie 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 avertie de la fin de l'envoie 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 sera généré à chaque fois qu'un octet arrivera 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 surtout 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} };
```

Une fois ces fichiers à disposition et raccordés à l'application ou à un driver, il est possible de pouvoir obtenir des données telles que des trames GPS.

4.2 GPS

4.2.1 Structure des trames NMEA

Les trames GPS sont des chaînes de caractères en ASCII directement compréhensibles. Elles commencent par un $ et se finissent par un *CRLN*.
Le protocole de communication du GPS définie 6 types de trames distinctes (notées à la suite de **GP**.
Chaque type de trames a sa propre utilité tel que fournir la position ainsi que l'heure, fournir le nombre de satellites, ... Pour de plus amples détails se référer à [1]

4.2.2 Fonctionnement

Le GPS équipant la carte utilise le protocole RS232. Celui-ci utilise le même port que la communication série(Tx pour la communication avec un ordinateur, Rx pour la réception des données GPS). Le second port disponible sur le msp430 est réservé à l'utilisation du SPI.
Le driver exploite directement la couche UART du microcontrôleur, car comme vu précédemment il n'y a pas de compatibilité avec le système de communication de TinyOS.

4.2.3 Module GPS

Comme TinyOS gère les interruptions, il est donc possible d'être avertie en permanence de la réception d'un octet sur l'UART.
Cela permet de réceptionner des trames et de les stocker en même temps.
Le module fonctionne de la manière suivante. A la réception du $ il se met à stocker les caractères arrivants. Quand il détecte le caractère de fin, il envoie à l'application cette chaîne qui sera ou non traitée.
Le démarrage et l'arrêt de l'acquisition se fait grâce à l'interface *SplitControl*. *Init* servira à la gestion de l'UART. Le fonctionnement du module est donc ainsi :

- Lorsque *start()* n'a pas était appelé ou bien que *stop()* a était appelé, l'état est GPS_IDLE. Dans cet état, le module ne s'occupe pas des octets qu'il reçoit.
- Après un appel à *start()* ou quand une trame a était envoyée à l'application, le module se met dans l'état GPS_WAIT. Dans cet état, il ne stocke rien tant qu'il n'a pas reçut le caractère $.

1. http://www.gpspassion.com/forumsen/topic.asp?TOPIC_ID=17661

- Lorsqu'il reçoit le caractère de début de trame, il se met dans l'état GPS_READ et commence à stocker les caractères qu'il reçoit de l'UART.
- à l'arrivée d'une fin de trame, il passe en GPS_FINISH et envoie la chaîne.
- Lorsqu'il a envoyé la chaîne, il se remet en GPS_WAIT et attend la réception d'un nouveau caractère $.

4.2.4 Mise en œuvre

Afin de tester le driver GPS, une application simple a était mise en oeuvre :
Elle lance le GPS, et envoie à l'ordinateur chaque trame reçue. Comme les deux communications ont la même vitesse, un tampon correspondant à la longueur d'une trame permet de rendre les deux actions autonomes.

Après stockage des trames reçues sur le port série de l'ordinateur, il est possible de pouvoir calculer la vitesse d'acquisition :

```
$GPRMC,101236.000,A,4821.6999,N,00446.5093,W,0.94,167.02,281208,,*11
$GPGGA,101237.000,4821.6991,N,00446.5093,W,1,05,8.2,5.1,M,52.5,M,,0000*40
$GPRMC,101237.000,A,4821.6991,N,00446.5093,W,0.81,167.67,281208,,*1F
$GPGGA,101238.000,4821.6987,N,00446.5092,W,1,05,8.2,5.4,M,52.5,M,,0000*4C
```

En comptant le nombre d'octets pour deux trames avec la même estampille temporelle, il est possible de déterminer un débit de 144octets par seconde.
Le fait que cette valeur soit inférieure à la vitesse normal peut s'expliquer par une latence au moment où le module fournit la trame à l'application. Comme le RS232 est asynchrone, c'est à dire que le périphérique possède sa propre horloge et se trouve donc autonome vis à vis du microcontrôleur, tous caractères qui ne seront pas lus seront perdus. Si le module rate le $ il rejettera l'ensemble des caractères de la trame, perdant donc une trame sur deux.
Il serait nécessaire d'améliorer de ce fait l'algorithme de traitement afin de réduire la latence en fin de réception.

4.3 Communication synchrone SPI et carte mémoire

Les cartes SD et MMC donnent un avantage pour la gestion des relevés fait par un capteur. Ce type de média donne la possibilité de stocker un grand nombre d'informations sur un support amovible qui peut ensuite être exploité depuis n'importe quel ordinateur ou PDA.

4.3.1 SPI

Le bus SPI supporte un protocole synchrone (partage de l'horloge entre les divers périphériques du bus) nécessitant 3 fils : horloge, données en sortie (du maître vers l'esclave) et données en entrée (de l'esclave vers le maître). De plus, un signal détermine quel périphérique du bus est actif : seul un périphérique peut être activé à un instant donné.

Ce protocole, 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 [sla049, chap.14] SPI ne fournit que 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 permettant d'éviter les problèmes rencontrés pour le GPS malgré le débit plus élevé.

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 étant de définir, à l'instar de l'UART, les configurations (en terme de vitesse, d'horloge, etc...) ainsi que la broche correspond au signal d'activation du périphérique.

4.3.2 Support de TinyOS

Il n'existe pas à l'heure actuelle de driver TinyOS destiné à l'exploitation d'une carte SD.
La seule implémentation existante était destinée à la version 1.x. Cette solution ne répondant pas au modèle hiérarchique de TinyOS-2.x, il était plus efficace de produire un nouveau driver parfaitement intégré et homogène dans la version actuelle du système, permettant sa totale indépendance vis à vis du matériel.
Cette absence s'explique de par la structure des cartes nativement supportées par TinyOS : celles-ci n'offrent du stockage non volatile que grâce à des chipstets de mémoire.

4.3.3 Fonctionnement de la SD

La carte SD permet de dialoguer soit à travers un protocole natif soit à travers le protocole SPI.

Les articles [LMHS25],[LM81] et le site [Retroleum] 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 driver dans et pour TinyOS.

la 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 au moment de l'initialisation ou lors de l'écriture le nombre.
- Ecriture d'une taille fixée de 512octets (1 secteur), l'adresse de début devant être un multiple de 512.
- Effacement du contenu d'un nombre arbitraire de secteurs.

4.3.4 Implémentation

Connection SPI

La première étape dans la création du driver va consister, à l'instar du GPS, à créer les fichiers qui feront la liaison entre le driver 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 4.2 – PlatformSdC

Le listing 4.2 présente le raccordement avec le module gérant le SPI sur le MSP430 (Msp430Spi0C). Contrairement au GPS, le dialogue avec le matériel se fera 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 :

```
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
    }};
```

Interface d'exploitation

La seconde étape consiste à définir de quelle manière sera faite l'interaction entre le driver SD et les couches supérieures.
L'utilisation d'un mécanisme tel que celui mis en oeuvre pour le RS232 impose que le driver contienne un tampon de 512 octets, soit le nombre d'octets à écrire.
Mais la mémoire d'un microcontrôleur étant relativement faible, cet encombrement risque-

rait de réduire le volume de mémoire disponible pour une application utilisant ce driver. Le module a donc était développé sur un modèle classique de fonctions ne rendant la main qu'une fois l'action finie. Ainsi le choix du fonctionnement pour les couches exploitant la SD sera laissé au soin au 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 4.3 – interface d'utilisation du module SD

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

Chapitre 5

Stockage en mémoire non volatile

L'objectif de l'implémentation d'un système de fichier dans TinyOS est d'être en mesure, sur le capteur, de stocker les données acquises pour ensuite pouvoir les restituer et les exploiter à l'aide d'un ordinateur ou d'un PDA. Et ce sans avoir besoin de demander au capteur d'en faire la restitution.

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

1. Réduire le temps d'arrêt du capteur. Ainsi la carte peut être simplement échangée par une autre. Sans cela il serait nécessaire de passer par une communication RS232.
2. La carte sans système de fichier peut être utilisée en RawWrite. Cette solution impose un post traitement avant son exploitation. Cela impose également de la part de l'utilisateur une certaine compétence. Et pour finir tous les systèmes d'exploitation n'autorisent l'accès à un support de stockage qu'à travers une partition.
3. Le système de fichier permet un transfert in situ, autorisant la libération de la carte et sa réutilisation immédiate.
4. Dans le cas d'un redémarrage sur une carte mémoire non vierge, le capteur ajoutera les informations à la fin du fichier.
5. L'utilisation de fichiers sur la partition permet une cohérence dans les données acquises. Par exemple pour une application faisant l'acquisition de trames GPS et de température, les deux informations ne se trouveront pas mélangées au même endroit.

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

1. A ressources réduites, système économe. Ce qui exclus d'emblée tous les systèmes journalisés, qui bien que réduisant le risque de perte de données, entraînent l'augmentation du nombres 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 concernait un système de fichier originellement développé pour des ordinateurs d'il y a une quinzaine d'années et donc proches des caractéristiques du noeud, tels que minix, CP/M [1] ou FAT [2] par exemple.

1. http ://en.wikipedia.org/wiki/CP/M
2. http ://fr.wikipedia.org/wiki/FAT

Le seul système de fichier qui subsiste et qui est encore extrêmement utilisé étant FAT.

> **Remarque :**
> Tous supports de stockage, pour être compatible et exploitable sur un ordinateur, doit comporter au tout début un MBR qui fournie 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.

5.1 Structures de FAT16

Les détails de la FAT sont documentés sur [SpecFat] et [fatStruct]. Il ne sera donc fait mention que des points importants de la mise en oeuvre.

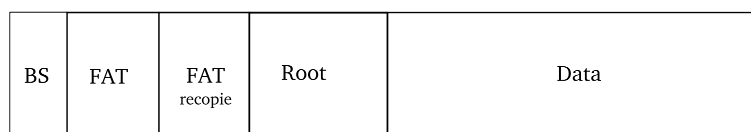

Figure 5.1 – Structure d'une partition FAT

La Figure 5.1 présente la structure de la partition. Elle se découpe de la façon suivante :
- BS (boot Sector) contient l'ensemble des informations sur la partition.
- 2 FAT (File allocation table) qui contiennent les listes chaînées du contenu de chaque fichier
- le répertoire racine (Root) : Contient une entrée par fichier
- La zone des données (Data) contient les fragments des données du fichier à laquelle on accède grâce aux informations de la liste chaînée du fichier contenu dans la zone FAT.

A l'exception du Boot Sector, qui est l'équivalent du MBR pour une partition FAT, les autres zones ont une position et une taille définie dans le Boot Sector.

L'unité de la plus petite zone mémoire de FAT est le **cluster**. C'est un agglomérat d'un certain nombre de secteurs(définie dans le Boot Sector de la partition).Il est toutefois possible d'écrire des données plus petites que cette zone.
Comme la FAT16 ne peut contenir qu'un maximum de 65525 clusters, plus la taille de la partition est grande plus il est possible que la taille d'un cluster soit importante.
La **Table d'allocation des fichiers** est une zone contenant des listes chaînées de clusters. Ces listes chaînées correspondent à l'ensemble des fragments du fichier. Le cluster de début de la liste est définie dans l'entrée de fichier contenu dans le répertoire racine.
La dernière zone avant les données est le répertoire racine. Celui ci contient les entrées pour chaque fichier ou répertoire contenu dans la partition. Pour un fichier, il existe à la

fois un ensemble de blocs définissant son nom long, ainsi qu'un bloc contenant à la fois son nom court et les informations tels que la taille, la date de création et le numéro du cluster de début de son contenu.

5.2 Problématique

L'implémentation d'un système de fichier n'est pas quelque chose de trivial, surtout pour ce type d'environnement.
D'un côté les caractéristiques du microcontrôleur imposent une économie en terme d'utilisation mémoire et d'un autre côté la carte mémoire impose un traitement de l'information par paquet de 512 octets. En plus de cela, la vitesse du msp430 réduit le débit des données.
L'utilisation d'un système de fichier augmente le traitement par rapport à du RawWrite : il devient par exemple nécessaire de remettre à jour les structures de données relatives au fichier ou à la table d'allocation.
L'approche la plus tentante, utilisée sur un ordinateur, correspond à ne pas mettre à jour systématiquement les informations. Le changement de la taille du fichier (2 octets) imposant la lecture/écriture de 1024octets entraîne un ralentissement important de la vitesse d'écriture. Il en va de même pour la gestion des deux tables d'allocation.
Toutefois cette solution, dans le cas d'un capteur, n'est pas du tout adaptée car elle entraîne une nécessité d'augmentation du volume de mémoire utilisée, qui ne serra donc pas disponible pour l'exécution en elle-même. De plus, en cas de coupure de l'alimentation, l'état du système de fichier ne peut être garantie et un certain volume d'informations peut être perdu.
C'est pourquoi le choix a été, au dépend de la vitesse et de la consommation liée à l'accès à la carte, de privilégier au maximum l'état de cohérence du système de fichier.

5.3 implémentation

Piège :
La FAT utilise le cluster comme unité de base. Lors d'une demande de lecture ou d'écriture il est donc nécessaire de faire la conversion vers des tailles en octets.
Les variables dans le BS sont pour la plupart codées en entier non signé 16 bits. Il faut donc faire un cast sur 32 bits sans quoi le résultat sera tronqué, risquant d'écraser des informations stockées ailleurs sur la carte. La formule pour la FAT est la suivante :

```
(((uint32_t) cluster -2)*((uint32_t)BytsPerSec*(uint32_t)SecPerClus))+ FirstDataSector;
```

Pour l'exploitation de la FAT trois drivers ont dû être mis en oeuvre.

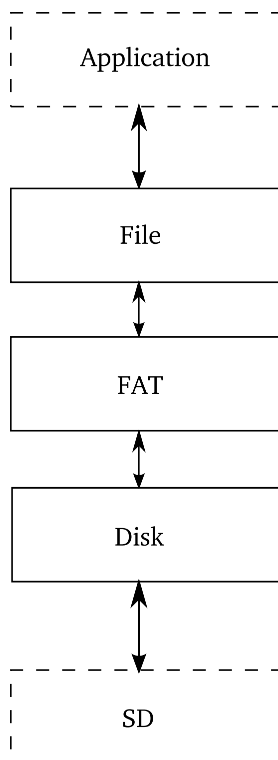

FIGURE 5.2 – Schéma du stockage

En partant du bas :

- Disk concerne uniquement la partie spécifique au support, afin d'en extraire la position de début d'une partition. Cette information sera ensuite ajoutée à chaque demande de lecture ou écriture pour donner la position absolue relative au début du support.
- FAT concerne la partition en elle-même.
- File concerne le descripteur de fichier.

Les détails les plus importants concernent le module FAT en lui même :
Pour être en mesure d'obtenir l'ensemble des informations, il est nécessaire d'exploiter toutes les données pertinentes contenues dans le BS :

```
// Nombre d'octets par secteur
BytsPerSec = (buf[12]<<8)+buf[11];
// taille d'un cluster en secteur
SecPerClus=buf[13];
// Secteur reserves après le BS
RsvdSecCnt = (buf[15]<<8)+buf[14];
// Nombre de FAT dans la partition
NumFATs=buf[16];
// Taille du répertoire racine
RootEntCnt = (buf[18]<<8)+buf[17];
// Taille d'une FAT
FATSz = (buf[23]<<8)+buf[22];
// Debut de la premiere FAT
FirstFatByts = RsvdSecCnt*BytsPerSec;
// Repertoire racine
RootDirByts = (RsvdSecCnt + NumFATs*FATSz)*BytsPerSec;
```

```
// position premier secteur de la zone de données
FirstDataSector = RootDirByts +
  fatAlignSup(RootEntCnt,BytsPerSec)*BytsPerSec;
CountOfClusters = ((buf[20]<<8)+buf[19]) / SecPerClus;
lastFreeCluster = 3;// Commence au début
```

Cette opération se fait à l'initialisation du module.
Une fonction fréquemment utilisée est :

```
void getOffsetAndSector(uint32_t nrClust, uint32_t *fatSec, uint16_t *fatOffset);
```

Celle-ci calcule sur la base du numéro de cluster sa position dans la table d'allocation en terme de numéro de secteur et de décalage par rapport au début du secteur.

La seconde opération importante concerne la réservation d'un cluster. Ceci arrive soit lors de l'utilisation d'un fichier vide, soit quand le programme a complètement remplit **SecPerClus*BytsPerSec** octets.
L'algorithme est en deux parties :
Dans un premier temps, recherche d'un cluster vide (disponible)

```
error_t fatReserveCluster(uint8_t *buf, uint16_t *cluster) {
  uint32_t fatSec, secOr, sector;
  uint16_t off, offOr;
  uint8_t sect;
  error_t error = FAIL;

  // Calcul de la position de départ
  // Selon où la derniere recherche à fini
  getOffsetAndSector(lastFreeCluster,&fatSec,&off);
  if (call disk.read((512*(fatSec-1))+FirstFatByts, buf) == FAIL)
    return FAIL;

  do {
    if (off >= 512) { // Si on dépasse le secteur
      fatSec++; // passage secteur suivant
      off = 0;  // raz de l'offset
      // Lecture d'un nouveau secteur de donnée
      if (call disk.read((512*(fatSec-1))+FirstFatByts, buf) == FAIL)
        break;
    }
    // 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);
```

En tout premier lieu, est demandé une copie du secteur dans lequel se trouve le suivant du dernier cluster vide trouvé (l.8 à 10).
Ensuite la fonction va boucler tant qu'elle n'aura pas trouvée une case décrivant un cluster vide (l.20), avec au besoin un changement de secteur(l.13 à 19).
La seconde partie de l'algorithme concerne la mise à jour des deux tables d'allocation :

```
    // Reservation du nouveau cluster
    (*(uint16_t *)&buf[off])=0xffff;
```

```
    getOffsetAndSector(*cluster,&secOr,&offOr);
    // On travail des clusters différents
    if (fatSec != secOr) {
      call disk.write((512*(fatSec-1))+FirstFatByts, buf);
      call disk.write((512*(fatSec-1))+FirstFatByts+(FATSz*512),buf);
      if (*cluster != 0) {// Pas encore de début donc faut pas écrire
        call disk.read((512*(secOr-1))+FirstFatByts, buf);
        (*(uint16_t *)&(buf[offOr]))=lastFreeCluster;
        call disk.write((512*(secOr-1))+FirstFatByts, buf);
        call disk.write((512*(secOr-1))+FirstFatByts+(FATSz*512),buf);
      }
    } else { // Le même Dans ce cas faut revoir
      if (*cluster != 0)
        (*(uint16_t *)&(buf[offOr]))=lastFreeCluster;
      call disk.write((512*(fatSec-1))+FirstFatByts, buf);
      call disk.write((512*(fatSec-1))+FirstFatByts+(FATSz*512),buf);
    }
```

La première étape (l.2) consiste à noter dans le tableau le cluster comme utilisé (*0xffff*), puis de rechercher la position de précédent cluster pour le fichier (l.4). Celui-ci contiendra le numéro du cluster suivant, c'est à dire de celui qui vient d'être réservé.

A ce niveau, deux possibilités existent :

1. Les deux clusters sont dans le même secteur, dans ce cas l'écriture du secteur concerné modifiera l'ensemble.
2. Les deux clusters sont dans des secteurs différents, il faudra donc les mettre à jour.

Un dernier point important, contenu dans **disk**, concerne la recherche de l'entrée d'un fichier dans le répertoire racine de la partition :

```
do {
  c = buf+offset;
  /* Arrive dans un long file name
   * Ou une entrée supprimée
   */
  if (c[0] == 0x00) { // Fin de fichier
    break;
  } else if ((uint8_t)c[0] == 229 || c[11] == 15) {
  } else if (c[11] == 0x20) { // Trouvé un fichier
    strncpy(fileName,c,7);
    fileName[8]='\0';
    if (!strncmp(fileName,name,strlen(name))) {
      ret = SUCCESS;
      break;
    }
  }
  offset += 32;
  if (offset >= 512) { // Faut passer au secteur suivant
    sector++;
    offset -= 512;
    call fat.fatReadRoot(sector,buf);
  }
}while(c[0] != 0x00);
```

– Le premier cas (l.6) correspond à la fin de la liste des fichiers.

– Le second cas (l.8) correspond aux noms de fichiers longs.
– Le dernier cas (l.9) correspond à une entrée de fichier valide. Dans cette situation, le nom passé lors de l'instanciation de l'objet descripteur de fichier est comparé avec le nom trouvé. Si l'entrée est trouvée, alors la boucle s'arrête.

Une fois l'entrée trouvée, il faut en exploiter les renseignements :

```
/* 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;
// Cluster de fin du fichier
// Si le fichier est vide
if (dataLocation < 2) {
  // Réservation d'un cluster d'emblé
  if (call fat.reserveCluster(&dataLocation) == FAIL){
    //call Lcd.write("reser",5);
    return FAIL;
  }
  lastCluster = dataLocation;
  lastSecteur = 0;
  call fat.fatReadRoot(rootSector,buf);
  (*(uint16_t *)&(buf+rootOffset)[26]) = dataLocation;
  call fat.fatWriteRoot(rootSector,buf);
} else {
  lastCluster = call fat.fatLastCluster(dataLocation);
  lastSecteur = call fat.fatLastSecteur(lastCluster);
}
```

Le point qui est important concerne les lignes 12 à 26. Un fichier vide n'a pas encore de cluster attribué. Si c'est le cas, un cluster lui est automatiquement attribué, et l'entrée de fichier et remise à jour.
Si ce n'est pas le cas, les informations sur la fin du fichier sont recherchés afin d'être en mesure de pouvoir directement exploiter le fichier lors de la première lecture.

5.3.1 Exemple d'utilisation

Un exemple simple est le suivant :
L'application commence par écrire sur la carte dans un fichier la phrase *"Hello World"*. Une fois l'écriture achevée, elle lit le contenu du fichier et l'affiche sur le LCD.
Pour ce faire, il suffit de repartir de l'application présentée en 3.4.2 et modifier ainsi :

```
event void Boot.booted() {
    call fatControl.start();
}
event void fatControl.startDone(error_t error){
    if (error == SUCCESS){
        memcpy(tampon,''Hello World!'',12);
        call file.write(tampon,12);
        error = call file.read(buff, 12);
        if (error == SUCCESS) {
```

```
          call Lcd.write(buff,12);
        }
      } else {
      call Leds.led0On();
    }
  }
```

5.3.2 Test de vitesse

Pour connaître la vitesse d'écriture, une petite application se contentant d'écrire dans un fichier 200 fois un buffer de 512 octets a était réalisée.
Ce test donne une vitesse approximative de 1.6kB/s. Cette vitesse est bien plus faible que sur un ordinateur mais par rapport à la vitesse d'acquisition des données, elle est largement suffisante.

Chapitre 6

Bilan

Le premier bilan concerne les drivers qui ont étés réalisés :
Afin d'évaluer la validité à la fois du driver GPS et du système de fichier, le capteur a été exploité en condition, à savoir sur piles rechargeables sur un trajet d'environ 950 km pour une durée de 11h. Le résultat est un fichier de 68004 lignes pour un poids de 4.7 Mo.
Après récupération du fichier sur un ordinateur et traitement grâce à GpsQt [1], la figure 1.1 Annexe A a put être obtenu (d'autres itinéraires sont disponibles sur http://www.trabucayre.com/gps).
Il est donc possible de dire que les drivers sont fonctionnels.
L'évolution future, concerne l'augmentation de l'aspect autonome de l'écriture par rapport au fonctionnement du reste de l'application.

Le second bilan concerne TinyOS en lui même :

L'étude faite à travers une plateforme non reconnue nativement, a permis de pouvoir analyser en profondeur les avantages et les inconvénients de ce système.
D'une part pour une application équivalente dont le code C ou assembleur étant connu, il est possible d'estimer que le temps nécessaire pour l'adapter à TinyOS est relativement faible.

D'autre part, la plateforme utilisée ayant des périphériques non supportés par TinyOS, nous avons pu ainsi constater que l'implémentation est simplifiée.
En effet, le code nesC étant sur une base de C, il est possible en partant d'une implémentation du protocole existante de l'adapter simplement aux contraintes de TinyOS (comme dans le cas du LCD), ou bien de réaliser une première implémentation des spécifications sur un ordinateur afin de simplifier la validation du code, puis 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 librairies nécessaires pour l'exploitation.

1. http ://projetaurore.assos.univ-fcomte.fr/gps/

Pour finir c'est un système qui est d'une prise en main relativement rapide, le nesC étant descendant du langage C, la compréhension du code des fichiers se fait assez rapidement. D'autre part une importante document est disponible.

Mais TinyOS n'est pas non plus exempt de défaut.
Bien que possédant un volume important de tutoriaux et autres documents, cette documentation ne couvre pas tous les sujets, il est par exemple facile de trouver toutes les informations nécessaire pour l'exploitation de la communication à base de paquets, mais dès lors que l'on souhaite par exemple exploiter une communication RS232 directement aucunes informations n'existent, il n'a pas été, par exemple, possible de trouver d'informations sur comment faire cohabiter deux modules utilisant le même UART, à des vitesses différentes et comment faire le bascule.

TinyOS manque encore d'une certaine maturité. Il est par exemple étonnant de se voir contraint à une somme définie 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 n'est pas supporté d'origine. Par ailleurs ce modèle de stockage bien que sans doute adapté pour le matériel disponible, n'a pas un aspect extensible vers d'autres média de stockage, c'est pourquoi la FAT à dû être intégrée à TinyOS.

Chapitre 7

Conclusion

Ce projet nous a permis de nous confronter à des aspects du développement différents de ceux habituels. En effet, si la notion d'optimisation du code, afin de le rendre plus efficace, est courante, la notion de consommation de mémoire l'est bien moins. La plupart du temps, la vélocité d'une application est une priorité au dépend de sa consommation en terme de ressources.

Le fait d'avoir été confrontés à la compréhension et à l'utilisation de protocoles de communications tels que le RS232 et SPI, ainsi que la nécessité de compréhension de la structure d'un support de stockage et de la structuration d'une partition nous a permis de mieux comprendre certains aspects cachés des systèmes, ainsi que les contraintes liées à leur exploitation et les restrictions. Celle-ci bien que connu sont floues quand à leur raison d'être.

Pour finir ce projet est d'autant plus intéressant que c'est un travail dont le résultat est pérenne dans le temps. En effet, les modules réalisés une fois leur fonctionnement validés et améliorés seront proposés à la communauté TinyOS afin que celle-ci, si ils sont considérés comme pertinents, puisse les intégrer dans le système d'une manière officielle.

Bibliographie

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

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

[TosPlatform] Tutorial TinyOS 10 : Création d'une plateforme
http://docs.tinyos.net/index.php/Platforms

[sla049] MSP430x1xx Family User's Guide (2006), focus.ti.com/lit/ug/slau049f/slau049f.pdf

[LMHS25] FRIEDT (J.-M.), GUINOT (S.), "Stockage de masse non volatile : un block device pour multimediacard"
GNU/LINUX Magazine, Hors-série n° 25, avril/mai 2006

[LM81] FRIEDT (J.-M.), CARRY (E.), "Enregistrement de trames GPS - développement sur microcontrôleur 8051/8052 sous GNU/LINUX"
GNU/LINUX Magazine, n° 81, février 2006

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

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

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

Table des matières

I Présentation 1

1 Introduction 2

2 Exploitation d'un capteur 4

2.1 Généralité 4
2.2 TinyOS 5

II Technique 6

3 Plateforme 7

3.1 Définition 7
3.2 Structure minimale d'une plateforme 8
3.3 Mise en place de la nouvelle plateforme 8
3.3.1 projet.target 8
3.3.2 .platform 8
3.3.3 hardware.h 9
3.3.4 PlatformC.nc 9
3.3.5 PlatformP.nc 10
3.4 Mise en pratique 10
3.4.1 Exploitation des Leds 10
3.4.2 Exploitation d'un LCD 11

4 Communication 14

4.1 RS232 14
4.1.1 Couche communication dans TinyOS 14
4.2 GPS 16
4.2.1 Structure des trames NMEA 16
4.2.2 Fonctionnement 16
4.2.3 Module GPS 16
4.2.4 Mise en œuvre 17
4.3 Communication synchrone SPI et carte mémoire 17
4.3.1 SPI 17
4.3.2 Support de TinyOS 18
4.3.3 Fonctionnement de la SD 18
4.3.4 Implémentation 19

5 Stockage en mémoire non volatile **21**
5.1 Structures de FAT16 22
5.2 Problématique 23
5.3 implémentation 23
5.3.1 Exemple d'utilisation 27
5.3.2 Test de vitesse 28

6 Bilan **29**

7 Conclusion **31**

III Annexes 35

A Trace GPS **36**

B Installation de msp430-jtag **38**
B.1 Récupération des sources sur le cvs de mspgcc 38
B.2 installation des librairies nécessaires 38
B.2.1 Correction de fichiers sources 39
B.2.2 Installation 39
B.3 Installation de msp430-jtag 39

C le Langage nesC **40**
C.1 Structure globale d'un fichier 40
C.2 Structure du langage 41
C.2.1 moteur 41
C.3 Nommage des fichiers 42
C.4 Type Configuration 42
C.5 Type Module 42
C.6 Type Interface 42

Troisième partie

Annexes

Annexe A

Trace GPS

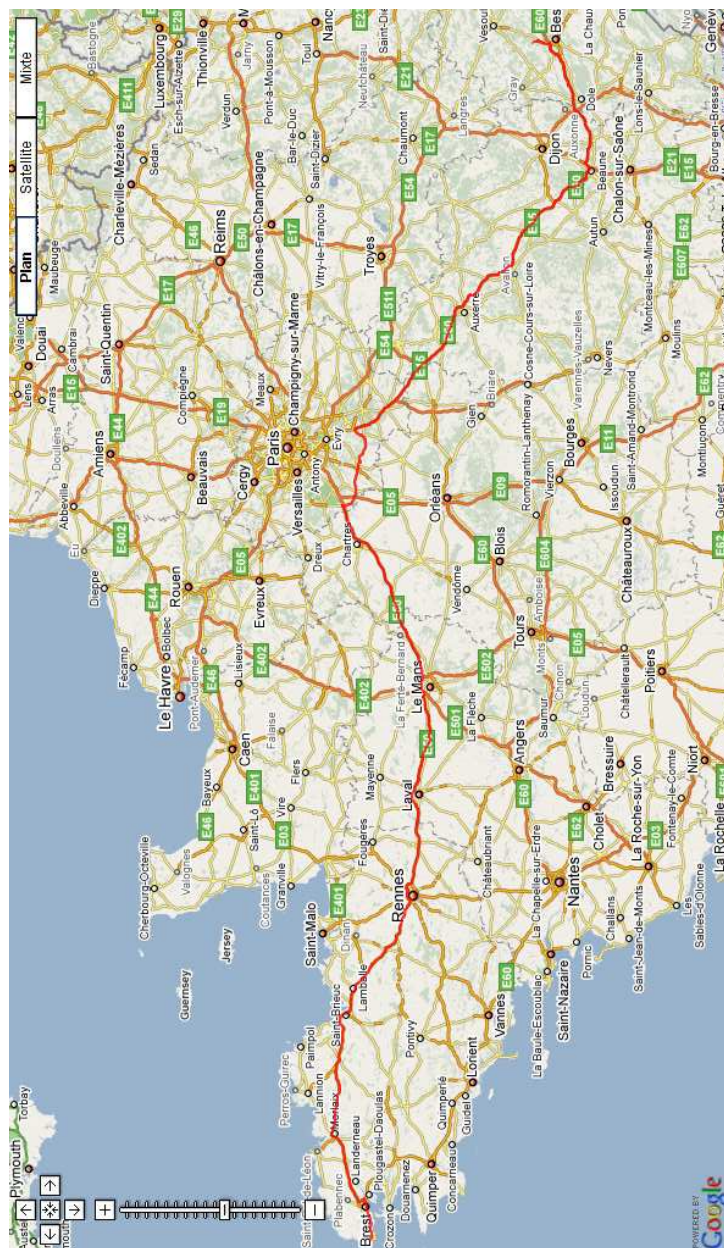

FIGURE A.1 – Tracé d'un trajet entre la région de Brest et le nord de Besanoņ (en rouge), sur un fond de carte Google Maps.

Annexe B

Installation de msp430-jtag

Le MSP430 se programme au travers d'une interface de communication sychrone nommée JTAG. Les sondes JTAG et le matériel dédié étant relativement coûteux, une solution moins performante mais à moindre coût est une émulation du protocole sur port parallèle. Sous unix, une telle fonctionnalité est implémentée par l'outils `msp430-jtag`.

Lors des premières tentatives de flashage de la carte nous avons constaté que msp430-jtag était absent de nos machines.
Pour la Gentoo l'overlay ne fournissait pas ce programme et après avoir contacté celui qui avait réalisé l'overlay il s'est avéré qu'il n'avait pas fait d'ebuild pour msp430-jtag.
Il a donc fallut comprendre comment l'installer par soit même.

B.1 Récupération des sources sur le cvs de mspgcc

Dans un répertoire (par exemple /tinyOS-jtag) Pour ce faire il faut utiliser les commandes qui suivent

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

B.2 installation des librairies nécessaires

Pour cette première partie, l'ensemble des actions se passera dans le répertoire jtag nouvellement créé.
La première étape est optionnelle, elle ne concerne que les personnes ne souhaitant pas installer les diverses librairies dans /usr/local/lib.
Si tel est le cas modifiez la variable LIB_PREFIX des fichiers makefile dans les répertoires **hardware_access** et **msp430**.
Il faut également ajouter dans /etc/ld.so.conf le répertoire de librairies qui sera utilisé. Attention certaines distributions génèrent à partir de fichiers, si tel est le cas il sera

nécessaire de se reporter à la documentation de la distribution pour rendre la modification pérenne.

B.2.1 Correction de fichiers sources

La seconde modification est absolument nécessaire car sans elle il ne sera pas possible de compiler certaines librairies.
dans **jtag/funclets/makefile** :
remplacer msp430x2121 par msp430x149
remplacer ASFLAGS = −mmcu=${CPU} par ASFLAGS = −mmcu=${CPU} −D_GNU_ASSEMBLER_→
↪

dans **jtag/funclets/eraseFlashSegment.S** et dans **lockSegmentA.S**
remplacer toutes les occurrences de LOCKA par 0x0040

B.2.2 Installation

L'installation se fait classiquement par un make intall dans le répertoire *jtag*.
Puis il faudra taper ldconfig.

B.3 Installation de msp430-jtag

Se placer dans le répertoire */tinyOS-jtag*.

tapez python setup.py install
Faite un chmod 755 msp430−jtag.py
Puis copiez msp430-jtag.py dans le répertoire bin de votre convenance (/usr/local/bin, /usr/msp430/bin,etc..) en le renommant msp430-jtag
cp msp430−jtag.py /usr/local/bin

Pour que msp430-jtag fonctionne il est nécessaire de mettre dans votre /.bashrc la ligne :
export LIBMSPGCC_PATH=/usr/local/lib
en admettant que vous ayez installé libMSP430mspgcc.so dans /usr/local/lib

Vous pouvez flasher votre msp430 en jtag sur le port paralléle

Annexe C

le Langage nesC

Le langage nesC est une évolution de langage C avec l'apport de notions plus orientées objets tels que celles retrouvées dans le C++ ou le Java. le document [nesCRef] présente les divers aspects.
Pour sa part le document [TosProg] donne une vision plus précise de la programmation pour TinyOS.
Ce chapitre a pour but principal de donner une vue d'ensemble et d'expliquer globalement la structure des applications, drivers et autres fichiers contenus dans TinyOS.

C.1 Structure globale d'un fichier

Tous les fichiers nesC contiennent au minimum un premier bloc : Celui-ci est définie par un mot clé suivi du bloc entre .
Pour un langage tel que C ou C++ ce bloc correspond à un .h.
Dans le cas de *module* ou *configuration* il contiendra les interfaces qu'il utilise *use* ou implémente 'provide'.
Selon que l'interface est *use* ou *provide* il devra :

- Fournir l'implémentation des callbacks (notées event dans l'interface utilisée) s'il est spécifié 'use XXX'.
- Fournir l'implémentation des fonctions (notées 'command' dans l'interface) si 'provide XXX.

Pour le type interface ce bloc contiendra la signature des fonctions (command) et callbacks (event) que le module devra implémenter.

Sauf pour une interface le fichier contient ensuite l'implémentation revenant au fichier de code .c ou .cpp .
le bloc est commencé par Implémentation et va contenir les fonctions et callback nécessaire ainsi que des variables privées.

C.2 Structure du langage

Pour expliciter la structuration entre configuration, module et interface ainsi que l'abstraction nous allons prendre un exemple simple sans rentrer dans les notions plus techniques ou spécifiques d'une plateforme :
Nous avons deux voitures de type berline ayant chacune un moteur d'une puissance différente mais toutes deux ont la même boite de vitesse.
A ces deux voitures nous ajoutons un utilitaire qui a le même moteur que la voiture la plus puissante mais une boite de vitesse différente.
D'un point de vue utilisateur quel que soit le véhicule, il n'est pas nécessaire de réapprendre à conduire. Il ne sait rien du fonctionnement du moteur. Il n'a à sa disposition que le sélecteur de vitesse, l'accélérateur et l'embrayage.
Nous allons maintenant définir les divers fichiers nécessaire pour spécifier le moteur, la boite de vitesse.

C.2.1 moteur

Au niveau de la plateforme (la voiture) il faut définir une configuration moteur qui offrira à la couche supérieure (en l'occurrence l'utilisateur) quelque chose qui soit générique. En somme il faut fournir les commande et un moteur sans qu'il ne soit nécessaire de dire si c'est un essence, un diesel ou un GPL.
la configuration moteurC est la suivante :
- il utilise de l'essence (uses interface essence)
- il fournie une vitesse de rotation.(provides interface rotation)
- il fournie une commande pour accélérer.(provides interface accel)

ensuite dans le bloc implémentation de notre moteurC il devra se trouver un components→ ↪ moteurXXXC as moteur auquel la configuration raccordera les diverses interfaces fournies précédemment.
Les moteurs réels se trouvent en dehors de la plateforme dans un répertoire qui pourrait être tos/lib/moteur.
le moteurXXXC pourra se nommer :
- moteurDieselC
- moteurEssenceC
- moteurGPLC

Qui raccordera à l'instar de moteurC les interfaces aux fichiers du même noms avec un P à la place du C.
Chacun de ces modules devront implémenter et fournir le code pour les command ou event des interfaces.
L'avantage de ne pas connecter directement les éléments extérieurs au module moteurXXXP sont les suivants :
- L'utilisateur (l'application) n'a pas besoin de savoir explicitement quel est le type du moteur. Elle utilise un moteur et c'est tout.
- Par la double indirection il devient possible d'ajouter une interface à moteurXXXP pour un quelconque usage sans pour autant devoir modifier le fichier de la plateforme ou bien l'application.(ajout d'une sonde de CO dont l'utilisateur n'a pas connaissance).

C.3 Nommage des fichiers

Le nom des fichiers peuvent contenir C ou P en dernière lettre du nom afin d'en préciser la nature.

- C correspond à un fichier qui par abus peut être dit publique. C'est une configuration qui connecte les provide qu'il a définie avec ceux de du fichier finissant par un P.
- P est un fichier dit Private. En d'autres termes il sert à fournir le code spécifique.

Le fait de donner deux types de fichiers apporte une encapsulation du code. Ainsi une modification/ajout/suppression dans le P n'impacte pas sur l'ensemble des modules avec qui il est lié.

C.4 Type Configuration

Ce type de fichier sert exclusivement à raccorder des interfaces ou des modules entre eux. Il n'a pas de code. Les seules instructions contenues dans la partie implémentation concernent les mises en relations entre les interfaces qu'il provide, use et les modules qui le composent.

C.5 Type Module

C'est à proprement le code. Il pourrait être vue de la même manière que le fichier c, ou cpp dans d'autres language. Dans ce fichier sont définies les event et les commands.

C.6 Type Interface

En Java, une interface définit la signature d'un ensemble de méthodes que toutes classes l'implémentant devront obligatoirement définir.
Ainsi, si tu classe c1 “implement” i1, c1 possédera au minimum l'ensemble des méthodes définies dans i1.
en nesC, une interface n'est pas aussi simple qu'en Java. En effet, une interface peut définir deux types de fonctions, celles qui sont de type *event* et celles qui sont de type *command*.
Selon qu'un composant va se déclarer utilisateur (*uses interface XX*) ou fournisseur (*provides interface XX*), le composant devra :

- implémenter les fonctions de type event pour l'utilisateur. Par exemple l'envoi d'un message par le réseau n'étant pas bloquant, le composant utilisateur ne pourra être avertie de la fin de l'envoi, que par la méthode de type event adéquate définie dans l'interface.
- implémenter les fonction de type command pour le fournisseur. Ces fonctions servant à permettre au composant utilisateur de pouvoir envoyer des ordres au fournisseur.

La relation entre les deux composants est multi directionnel. En effet, l'utilisateur qui doit implémenter les events peut utiliser les commands (*call*) et le fournisseur qui doit implémenter les commands peut signaler les événements(*signal*).
Un composant en lui même ne sait pas quel est l'autre composant relié par cette interface,

il ne fait qu'appeler les méthodes ou signaler les événements.
Par exemple, l'interface readStream est définie de la façon suivante :

```
interface ReadStream<val_t> {
  command error_t postBuffer(val_t* buf, uint16_t count);
  command error_t read(uint32_t usPeriod);

  event void bufferDone(error_t result,
                        val_t* buf, uint16_t count);
  event void readDone(error_t result, uint32_t usActualPeriod);
}
```

Soit deux composants c1 utilisateur de ReadStream et c2 fournisseur de c2.
Le composant c1 spécifiera `uses interface ReadStream`.
Lorsqu'il souhaitera demander une lecture fera :
`call ReadStream.read(10)`
Définira une méthode :
`event void ReadStream.readDone(...){...}`
Et le composant C2 spécifiera `provides interface ReadStream`.
Définira une méthode :
`command error_t ReadStream.read(...){...}`
Lorsque la lecture sera fini il utilisera : `signal ReadStream.readDone(....)`
pour avertir l'utilisateur de la fin de la lecture.
Il est visible que ni c1 ni c2 ne connaissent l'autre. Ils ne font qu'utiliser ReadStream.
C'est à la configuration de mettre en relation les deux, en définisant les lignes suivantes.

```
configuration c2 {}
implementation{
components c1, c2;
c1.ReadStream -> c2.ReadStream;
```

La flèche a comme signification "c1 utilise l'interface ReadStream et c2 fournie l'interface ReadStream".