Tomtom Go : une plateforme pour l'acquisition sans fil de mesures géoréférencées

J.-M. Friedt

Association Projet Aurore, Besançon

11 novembre 2009

Nous nous proposons de découvrir les fonctionnalités d'un système embarqué sous GNU/Linux et pour lequel le constructeur respecte l'éthique de la GPL : le récepteur GPS Tomtom Go. Notre objectif est d'enrichir les fonctionnalités de cet appareil en s'en servant comme afficheur d'informations acquises par un dispositif dédié dont les mesures sont transmises par liaison sans fil. Les données ainsi accumulées sont géoréférencées et conservées sur support de stockage de masse non volatile pour exploitation ultérieure.

1 Introduction

Le Tomtom Go est un récepteur GPS, contenant un logiciel propriétaire pour la navigation, exécuté sous GNU/Linux sur un processeur ARM9 [1]. Une liaison bluetooth sur le modèle 720 est par ailleurs fournie pour se connecter à des périphériques ne nécessitant qu'un faible débit de communication tel que les téléphones portables.

Tomtom a documenté les logiciels sous GPL utilisés sur leur récepteur, et fournit une toolchain de crosscompilation. Ceci a permis le projet Opentom qui démontre la capacité à développer des applications fonctionnelles. Les exemples disponibles [1] fournissent les bases de développement utiles pour le reste de cette discussion : une console en mode texte avec un clavier sur l'écran tactile, un enregistreur de traces (pour exploitation sur une application telle qu'Openstreetmaps) [2], les scripts nécessaires à ouvrir un terminal sur la liaison bluetooth ... Ces outils vont nous permettre de nous connecter au Tomtom Go et d'y développer des applications additionnelles. Néanmoins, nous allons ici utiliser le système d'exploitation non pour fournir la capacité à charger dynamiquement un binaire et les bibliothèques associées, mais pour l'interpréteur shell qui est associé à GNU/Linux ainsi que la pile bluetooth du noyau.

Bluetooth est un protocole de communication radiofréquence fonctionnant dans la bande libre (Instrumentation Scientifique et Médicale – ISM) autour de 2,45 GHz. Trop lent pour permettre une communication efficace, trop long à s'associer à ses interlocuteurs pour permettre une gestion efficace de l'énergie, c'est un protocole qui ne doit probablement sa disponibilité dans à peu près tout équipement portable grand public qu'à la puissance commerciale des grands groupes industriels qui ont dirigé son développement. Nous allons néanmoins essayer de trouver une application qui justifie l'utilisation d'une liaison bas débit sans fil : la mesure de grandeurs scalaires (température) sur un objet en mouvement qui ne permet pas de liaison filaire (ni en alimentation, ni en communication) entre l'ordinateur de traitement et stockage des informations et le capteur.

L'objectif que nous visons est donc le stockage d'informations géoréférencées acquises au moyen d'une liaison sans fil. La justification de la liaison sans fil est de placer le capteur sur un support mobile ou rotatif qui ne permet pas une liaison filaire : une application qui en découle naturellement est la mesure de température du pneu d'un véhicule, avec affichage de l'information en temps réel à l'utilisateur.

On s'intéressera à deux aspects du développement : d'une part le système de mesure de température qui doit fonctionner sur la batterie (puisque fixé sur la roue) et fournir une autonomie au moins compatible avec un trajet d'une journée pendant lequel le récepteur GPS Tomtom est alimenté sur batterie de la voiture (on espère donc dépasser l'autonomie du Tomtom sur batterie

[1] www.opentom.org

[2] http://www.opentom.org/TTTracklog

qui n'est que de l'ordre de 1,5 à 2 h pour le modèle 720). De façon plus générale, les outils mis en œuvre seront exploitables pour observer une grandeur physique agissant sur la résistance électrique d'un transducteur (applicable à une vaste gamme de mesures, des jauges de contraintes aux capteurs de gaz à oxydes métalliques en passant par l'intensité lumineuse pour une photorésistance). La communication numérique des informations implique l'utilisation d'un microcontrôleur, ici le MSP430, et la gestion d'énergie associée. D'autre part, la communication des informations par une liaison sans fil et leur restitution à l'utilisateur au travers de l'interface propriétaire mais documentée du Tomtom Go. L'objectif est de conserver la fonctionnalité de guidage, augmentée des informations acquises par le(s) capteur(s), et ce en faisant appel exclusivement par l'utilisation des outils mis à disposition par GNU/Linux au travers du shell `bash`, sans nécessiter de programmation de nouvelles applications.

2 Le Tomtom : un ordinateur sous GNU/Linux

Nous exploiterons au cours de cette présentation le Tomtom Go 720 : cet ordinateur est équipé d'un processeur ARM9 cadencé à 400 MHz, entouré de 64 MB RAM et 2 GB de stockage non volatile (*a priori* de la mémoire flash d'après le nom du *device* `/dev/sdcard` monté dans `/mnt/sdcard` malgré le symbole de disque dur apparaissant à l'allumage)[3]. Ces caractéristiques matérielles ne sont importantes que pour obtenir sur le web les binaires des programmes qui nous manquent, puisque nous allons exploiter au maximum le système d'exploitation pré-installé et ainsi économiser l'effort de programmer des outils déjà accessibles *via* le shell.

Une distribution dédiée de Linux avec un noyau modifié a été mise en place par Tomtom, tandis que les outils accessibles aux utilisateurs sont en grande partie fournis par busybox. Les outils de la bibliothèque de gestion bluetooth `bluez` [2] sont également disponibles et permettent de rapidement communiquer avec le Tomtom sans recompiler les logiciels embarqués (et donc notamment de développer sur Tomtom sans avoir à en modifier le contenu). Nous allons donc nous familiariser avec l'interaction avec le système installé sur Tomtom afin de déterminer les possibilités et contraintes du système de mesure avec lequel nous voudrons communiquer.

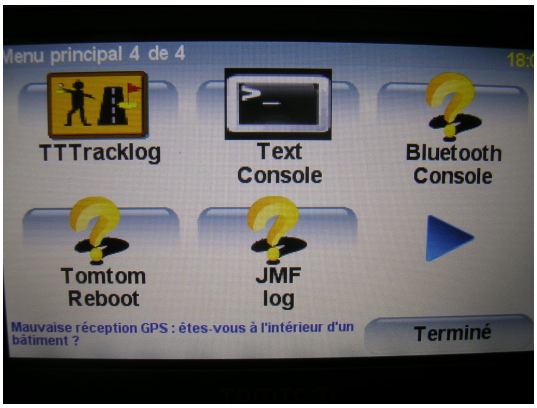

FIG. 1 – La liste d'applications ajoutées à un Tomtom Go 720 : la liste des entrées du répertoire `SDKRegistry` détermine les noms des applications, quel programme est exécuté et éventuellement l'icône associée à chaque application.

L'accès depuis l'interface graphique (Fig. 1) aux scripts et programmes que nous développerons se fait au travers du répertoire `SDKRegistry` (attention aux majuscules et minuscules selon que l'on accède au Tomtom par liaison USB de classe `mass storage` sous MS-Windows ou GNU/Linux). Ce répertoire contient une liste de paramètres de configuration dans un fichier d'extension `.cap` [3], avec notamment le PATH (`AppPath`), l'image bitmap de l'icône, le nom de l'application exécutée

[3] http://www.opentom.org/Hardware_Variants

lorsqu'on appuie sur l'icône (`AppName`) ... Ainsi, ayant accédé au support de stockage non-volatile du Tomtom tel que décrit dans le paragraphe suivant, nous mettrons à jour le répertoire `SDKRegistry` pour y placer notre application (binaire exécutable ou script shell) qui deviendra accessible depuis l'interface graphique, en prenant soin d'ajouter tous les outils nécessaires au PATH.

3 Les outils pour développer sur Tomtom

Le premier outil qui vient à l'idée pour découvrir les fonctionnalités d'une nouvelle plateforme est une console : un tel outil est fourni à http://www.opentom.org/TomTom_Console. Un clavier virtuel permet de taper sur l'écran tactile – il est judicieux à cet effet de se munir d'un stylet dédié à cette opération – les commandes classiques fournies par busybox (Fig. 2). Noter néanmoins que la console ne fonctionne qu'avec un firmware suffisamment ancien puisque les dernières versions du logiciel de navigation reprennent rapidement la main en rafraîchissant l'écran : tous les tests présentés dans ce document ont été effectués sur un Tomtom Go 720 muni de l'application 7.010 (8290/07711), OS version 2618, boot version 5.4062 et GPS version 1.21. Après avoir découvert les fonctionnalités du shell (bash) et les outils mis à disposition par la distribution GNU/Linux embarquée, nous voudrons taper nos commandes *via* une interface plus efficace que le clavier virtuel.

Le support bluetooth fournit ce qui deviendra notre interface de communication privilégiée : activée à l'origine pour s'interfacer avec un téléphone mobile, nous commencerons par reconfigurer cette interface pour créer un port série virtuel au moyen des scripts mis à disposition à [4]. En lançant la *Bluetooth Console*, le Tomtom Go crée un serveur de connexion sur un port série virtuel. Du côté du PC, un convertisseur USB-Bluetooth nous fournit une interface sur ce protocole de communication tel que le valide la commande `hciconfig`. Ayant une interface bluetooth fonctionnelle, l'adresse matérielle (équivalent de l'adresse MAC pour ethernet) `MAC_TOMTOM` s'obtient par `sdptool browse`. Finalement, le port série virtuel `/dev/rfcomm0` est associé au Tomtom par `rfcomm bind 0 MAC_TOMTOM 1`. La liaison est réellement mise en place lors de l'exécution d'un client se connectant au terminal série (par exemple `minicom`) sur `/dev/rfcomm0` au débit de 9600 bauds.

FIG. 2 – Configuration d'un Tomtom Go 720 tel qu'obtenue depuis la console.

À ce point, nous avons pu exploiter la liaison bluetooth pour créer un lien avec une console exécutée sur le Tomtom. Nous avons donc ici une méthode exploitable pour qu'une application de traitement et d'affichage de données reçoive les informations par ce port série virtuel. Cette méthode est d'autant plus attractive qu'elle permet de sérier les problèmes : tester l'application de liaison asynchrone (RS232) sur PC, porter au Tomtom sur la liaison bluetooth, puis ajouter les interfaces avec l'utilisateur. Bien qu'une toolchain [5] pour Tomtom soit disponible, nous allons

[4] http://www.opentom.org/BTconsole
[5] http://www.tomtom.com/page.php?Page=gpl

changer notre habitude de développer un programme compilé depuis un langage de haut niveau (C) pour générer une application, mais nous allons exploiter la disponibilité d'un shell et des outils GNU associés pour ne programmer qu'en `bash`.

4 Électronique de mesure de température et conversion au format numérique

4.1 Mesure de résistance

Le Tomtom va nous servir à enregistrer, localiser et afficher une grandeur mesurée. Nous allons ici développer un aspect original – ou tout au moins inhabituel mais bien documenté [4] : la mesure de température au moyen d'une sonde résistive selon des contraintes compatibles avec une application embarquée faible consommation.

Classiquement, une sonde de température se mesure au moyen d'une source de courant et mesure par amplificateur différentiel de la chute de tension associée à la variation de résistance en fonction de la température, ou par pont de Wheatstone dont un des bras contient la résistance à mesurer [5]. Cependant, ces solutions nécessitent d'une part des amplificateurs opérationnels qui, sauf cas exceptionnel (disponibilité d'un signal de mise hors tension), doivent être continuellement alimentés, et d'autre part la mise en place d'astuces pour fonctionner en alimentation unipolaire (et non alimentation symétrique telle qu'on a l'habitude d'employer sur amplificateur opérationnel).

La mesure de la résistance par circuit RC [4] que nous nous proposons d'exploiter présente l'avantage de ne nécessiter que peu de composants additionnels autour du microcontrôleur en charge de la mesure (un condensateur et une résistance de référence), de ne pas nécessiter de convertisseur analogique-numérique (souvent gourmand en énergie) et d'être désactivé lors de la mise en veille du microcontrôleur.

Néanmoins, certaines précautions doivent être mises en place pour la mesure de résistance sous forme d'une constante de temps. Tout d'abord, la détection précise de seuil (comparateur) et de temps (*timer*) doivent être disponibles sur le microcontrôleur utilisé. Ces pré-requis étant satisfait par le choix du Texas Instruments MSP430F149 [6] (Code 1 et Fig. 3), il nous faut appréhender la technique de mesure pour en identifier les faiblesses :

- la mesure d'une constante de temps s'obtient dans un premier temps en chargeant un condensateur C pour lui permettre d'atteindre un potentiel connu, puis de le décharger au travers de la résistance R à identifier : ce temps de décharge est associé à la constante de temps $\tau = RC$. Nous identifions ici une première contrainte : *le potentiel de charge du condensateur doit toujours être le même*, d'où la nécessité d'une régulation de la tension d'alimentation du microcontrôleur.
- un condensateur est un composant qui évolue considérablement dans le temps (vieillissement) et avec son environnement (température), polluant ainsi la mesure de résistance. Nous ne travaillerons par conséquent jamais sur la valeur absolue de τ mais *sur un rapport* de la mesure de la résistance inconnue R à une résistance connue de référence R_{ref}. Ce rapport de temps $\frac{\tau}{\tau_{ref}}$ rend la mesure indépendante de C : $\frac{\tau}{\tau_{ref}} = \frac{R}{R_{ref}}$
- cette mesure se multiplie facilement à un grand nombre de résistances de mesure, pour une unique résistance de référence. En effet, nous utilisons un port numérique d'entrée sortie (*General Purpose Input Output*, GPIO) comme source de courant lors de la charge du condensateur (potentiel en sortie égal à la tension d'alimentation) ou comme puits de courant lors de la décharge (GPIO en sortie au potentiel de la masse). Les autres broches du port sont pendant la mesure en entrée, c'est à dire en haute impédance, ce qui signifie qu'aucun courant ne circule dans ces broches et par conséquent n'affectent pas la mesure de constante de temps de charge et de décharge du condensateur.

Dans notre implémentation de cette technique de mesure, nous avons exploité 5 sondes de température Pt100 [6] et une résistance de référence. La constante de temps a été sélectionnée en

[6] Référence 1289666, sonde Pt100 classe A, 5,37 euros/pièce chez Farnell

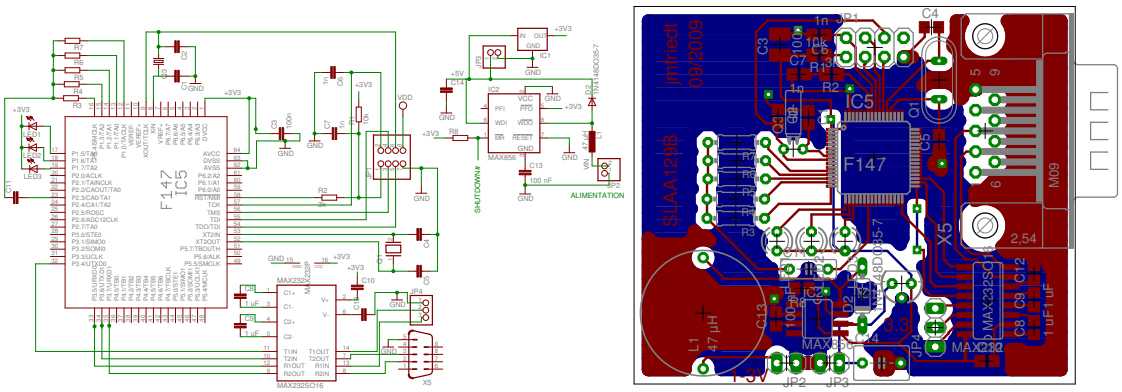

FIG. 3 – Gauche : schéma du circuit exploité pour implémenter la mesure de capteurs résistifs selon les principes décrits par la note d'application SLAA129B de Texas Instruments. Les 5 résistances en haut à gauche du schéma sont les points de mesure. Droite : circuit résultant de l'implantation des composants. La taille du circuit est 60×50 mm^2. Noter la taille de l'inductance (en bas à gauche), nécessaire pour la mise en œuvre d'un convertisseur DC-DC, et de la connectique.

choisissant un condensateur qui permette de travailler sur 10 bits d'un compteur compte tenu de la fréquence de l'horloge de référence fournie par un quartz à 4 MHz. Noter qu'un quartz de fréquence élevée consomme plus, mais permet d'obtenir une mesure précise plus rapidement afin de retourner le microcontrôleur en mode veille au plus vite.

Le calcul de la consommation électrique du circuit de mesure s'obtient comme suit :

- nous savons qu'un condensateur de capacité C chargé à une tension U accumule une énergie $E = 1/2 \times C \times U^2$ issue de la pile
- le temps caractéristique τ de charge et de décharge d'un condensateur C dans une résistance R est de l'ordre de $\tau = RC$.
- ce temps est mesuré par un compteur du microcontrôleur de b bits, cadencé par une horloge de fréquence f : $\tau \sim 2^b/f$.
- nous en déduisons une relation entre la valeur du condensateur de charge C, la résistance des sondes de températures imposées par les standards ($R \sim 100\ \Omega$ pour une sonde Pt100), la fréquence f du quartz haute-fréquence du MSP430 (nous avons sélectionné $f = 4$ MHz) et $b = 12$ le nombre de bits exploités sur le compteur :

$$C \sim \frac{2^b}{R \times f}$$

est de l'ordre de 10 μF et l'énergie nécessaire pour charger le condensateur est

$$E = \frac{2^{b-1}}{Rf} \times U^2$$

soit, pour une tension $U = 3,3$ V, une consommation de 55 μJ par mesure, consommés pendant une durée de $\tau = RC \sim 1$ ms.

Nous pouvons donc nous interroger sur la pertinence d'une mesure de température nécessitant 55 μJ, considérant qu'un périphérique de conversion analogique-numérique consommant 1 mA sous 3,3 V pendant 1 ms ne consomme que 3,3 μJ pour une conversion. La stratégie proposée par la note d'application SLAA129B n'a donc un intérêt que par sa capacité à ne nécessiter aucun composant externe au microcontrôleur pour une mesure de résistance, et en particulier pas d'amplificateur opérationnel généralement associé à une gestion d'énergie complexe (par exemple pour une alimentation bipolaire) et gourmande en énergie. Ainsi, si les composants externes consomment au total 1 mA qui ne peut être désactivé en l'absence de mesure, alors la méthode de SLAA129B devient pertinente pour un rapport cyclique de mesure à veille de l'ordre de 10%. Une application

classique consiste en une mesure toutes les quelques secondes, soit un rapport cyclique proche de 0,1%, pour lequel le gain énergétique est significatif par rapport à une mesure de résistance classique (pont de Wheatstone ou source de courant stable).

```
// gcc 3.3 : msp430-gcc  -Wall -Os -mmcu=msp430x149 slaa129b.c -o slaa129b
// flasher : msp430-jtag -e slaa129b
// point commun sur broche 23=CA0/TA1, resistances sur P1.x

#include <msp430x14x.h>
#include <string.h>
#include <signal.h>
#include <io.h>
#include <stdio.h>

#define ref    0x80

volatile char timeout=0;

void Sommeil(void) {LPM1;} //LPM4 serait mieux

void Pause_ms(unsigned short ms){ // pause de ms millisecondes
 TBCCR0=(unsigned short)(ms*32768/1000);
 TBCTL=TBSSEL_1+TBCLR + MC_1;
 timeout=0;
}

void init_rs1_9600(){
 UCTL1=SWRST;
 UCTL1|=CHAR;
 UTCTL1=SSEL0;
 UBR01=0x03;
 UBR11=0x00;
 UMCTL1=0x4A;
 UCTL1&=~SWRST;
 ME2|=UTXE1+URXE1;
}

unsigned char get_rs1()
 {while ((IFG2 & URXIFG1)==0) {}; return(RXBUF1);}

void snd_rs1(unsigned char cara)
{while ((IFG2 & UTXIFG1)==0);
 TXBUF1=cara;
}

void writeASC(unsigned short ptr)
{unsigned char b;
 b=((ptr&0xf000)>>12);
 if (b<10) snd_rs1(b+48); else snd_rs1(b+55);
 b=((ptr&0x0f00)>>8);
 if (b<10) snd_rs1(b+48); else snd_rs1(b+55);
 b=((ptr&0x00f0)>>4);
 if (b<10) snd_rs1(b+48); else snd_rs1(b+55);
 b=((ptr&0x000f));
 if (b<10) snd_rs1(b+48); else snd_rs1(b+55);
}

void str_rs1(unsigned char* cara)
{int k=0;
 do {snd_rs1(cara[k]);k++;} while (cara[k]!=0);
}

void init_PORT(void){
  P1SEL = 0x00;
  P1DIR = 0xE0;
  P1OUT = 0xF0;

  P3SEL=0xFE;   // ; P3.6,7 = USART select
  P3DIR=0x5B;   // ; P3.6 = output direction
  P3OUT=0x01;   // ; P3.6 = output val

  P2SEL=0x08;   // comparator sur broche 23=P2.3 jmfriedt
  P2DIR=0x00;
  P2IES=0x00;
  P2IE=0x00;
}

void init_TIMER(void){
 TBCCTL0=CCIE; // CCR0 interrupt enabled
 TBCCR0=200;
 TBCTL=TBSSEL_1 + TBCLR;
 TBCTL=TBSSEL_1+TBCLR + MC_0;
}

interrupt(TIMERA1_VECTOR ) wakeup Vector_TIMERA1(void)  // CCR1 ISR
{P1OUT^=0x40;
 TACTL|=TACLR;
 CCTL1&=~0x1;
}

interrupt(TIMERB0_VECTOR ) wakeup Vector_TIMERB0(void)
{TBCTL=TBSSEL_1+TBCLR + MC_0;
 timeout=1;
}

void init_ADC(void){
  ADC12CTL0=SHT0_6+REF0N+ADC12ON; // V_REF=1.5 V
  ADC12CTL1=SHP;
  ADC12MCTL0=INCH_10+SREF_1;      // p.17-11: single conversion
  ADC12CTL0|=ENC;
}

unsigned short temperature()
{ADC12CTL0|=ADC12SC; //  start conversion
 do {} while ((ADC12CTL1 & 1)!=0);
 return(ADC12MEM0);
}

int charge(unsigned char mask)
{unsigned int periode1,periode2;
//Charge
 P1DIR|=mask;              // ;ref output
 P1OUT|=mask;              // ;ref Set
 Pause_ms(10);Sommeil();
 P1DIR&=~mask;             // ;ref = HiZ, charge complete
 P1OUT&=~mask;             // ;ref = reset
 TACTL=TASSEL1+MC1+TACLR ; // Xtal ext + Mode continu + Remise Ã  zÃ©ro

 Pause_ms(30); // PAS de sommeil : soit on se reveille par TB, soit par TA
 //discharge
 CACTL1=CARSEL+CAREF0+CAON;
 CCTL1= CM1+CAP+CCIE + CCIS0;
 CACTL2=P2CA0+CAF;
 periode1=TAR;
 P1DIR|=mask;
 P1OUT&=~mask;
 LPM1;
 if (timeout==1) periode1=0xffff;
 else {periode2=TACCR1;
       periode1=periode2-periode1;
      }
 P1DIR&=~(0xff);
 CACTL1=0;
 CCTL1=0;
 TACTL=0;
 return(periode1);
}

void init_horloge(void){
 volatile unsigned char dco_cnt = 255;
 BCSCTL1 &= ~XT2OFF;          // XT2on
 P5DIR |= 0x10;               // P5.4= output direction
 P5SEL |= 0x10;               // P5.4= MCLK option select
 do                           // wait for MCLK from quartz
  {IFG1 &= ~OFIFG;            // Clear OSCFault flag
   for (dco_cnt = 0xff; dco_cnt > 0; dco_cnt--); // Time for flag to set
 } while ((IFG1 & OFIFG) != 0);  // OSCFault flag still set?
 BCSCTL1 = 0x07;               // LFXT1: XT2on: LF quartz for MCLK
 BCSCTL2 = SELM1 + SELS;       // LFXT2: use HF quartz (XT2) if available
 WDTCTL = WDTPW + WDTHOLD;     // Stop WDT
}

int main (void)
{int k;
 unsigned char sensor=0x01;
 int t;

 init_horloge();
 init_PORT();
 init_ADC();
 init_TIMER();
 init_rs1_9600();
 eint();

 while (1) {
    t=charge(ref);
    writeASC((unsigned short)t);snd_rs1(' ');
    for (sensor=1;sensor<=0x10;sensor*=2)
       {t=charge(sensor);
        writeASC((unsigned short)t);snd_rs1(' ');
       }

    t=0;for (k=0;k<16;k++) t+=temperature(); t/=16;
    writeASC((unsigned short)t);snd_rs1('\n');
    for (k=0;k<9;k++) {Pause_ms(1000);Sommeil();} // 9 s + 1 s pour la mesure
 }
 return(0);
}
```

Tab. 1 – Programme implémentant en C sur MSP430F149 la stratégie de mesure de résistance par temps de décroissance d'un circuit RC et génération d'une interruption sur comparateur de tension. Noter l'utilisation du Timer B comme sécurité en cas de dysfonctionnement d'une des sondes résistives qui se traduit par l'absence d'interruption du comparateur analogique.

Classiquement, un microcontrôleur communique avec son environnement par port série asynchrone (RS232). Ici, notre objectif est de placer le capteur sur un objet mobile, donc les liaisons filaires sont proscrites. Par ailleurs, le Tomtom ne possède qu'une interface Bluetooth et pas de port asynchrone facilement accessible : nous allons donc exploiter un convertisseur RS232-Bluetooth – Free2Move [7], pour communiquer le résultat de mesures au Tomtom Go.

Le dongle Free2Move se configure de diverses façons : soit au moyen d'un logiciel graphique disponible pour MS-Windows uniquement, mais fonctionnel sous GNU/Linux au moyen de `wine` (Fig. 4), soit en suivant les consignes de configuration fournies dans la notice d'utilisation [8] et permettant de reconfigurer le dongle depuis le microntrôleur à chaque mise sous tension (par exemple pour créer une liaison "sécurisée" avec un unique interlocuteur au lieu d'une configuration acceptant une communication de n'importe quel interlocuteur tel que nous allons le faire ici). Dans tous les cas, nous prendrons soin de configurer le module Free2Move en *esclave* (Endpoint) et acceptant la connexion de n'importe que maître, permettant ainsi le debuggage rapide de l'application embarquée soit depuis le Tomtom, soit depuis un PC équipé d'une interface bluetooth. Cette phase de configuration détermine aussi le débit de la liaison asynchrone entre le microcontrôleur et le module Free2Move : nous avons choisi de travailler en 9600 bauds, fréquence accessible depuis le quartz 32 kHz cadençant le MSP430.

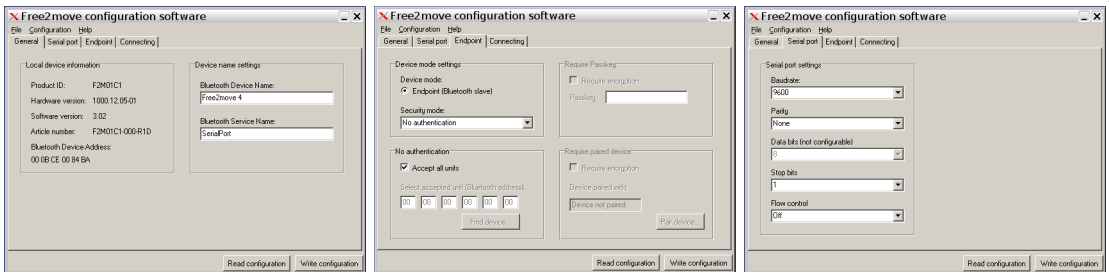

FIG. 4 – Exécution du logiciel de configuration graphique du module Free2Move sous Wine. Penser à créer le lien symbolique dans `.wine/dosdevices` entre `com1` et `/dev/ttyS0` afin que `wine` sache où trouver le module Bluetooth-RS232.

4.2 Gestion d'énergie

Deux alimentations sont nécessaires dans le circuit embarqué : l'alimentation du convertisseur RS232-Bluetooth (5 V) et le microcontrôleur (3,3 V). Afin de limiter le nombre et l'encombrement des batteries, nous avons opté pour un n'utiliser que deux piles, et élevér la tension à 5 V par un convertisseur DC-DC MAX856. Cette stratégie a pour vocation de gérer efficacement l'énergie, et notamment d'illustrer l'économie en désactivant l'alimentation des périphériques inutilisés puisque la majorité des convertisseurs DC-DC possèdent une broche de désactivation de leur oscillateur interne, et donc de l'alimentation des composants qui y sont connectés. Il est ainsi possible de désactiver certaines portions du circuit inutilisées – ici le dongle Free2Move, seul composant alimenté en 5 V – tout en conservant les fonctionnalités d'acquisition de données ou de contrôle, alimenté directement par les piles.

Cependant, le Bluetooth est en ce sens un mauvais exemple car il nécessite près de 10 s pour négocier sa connexion lors de la mise sous tension. Pour l'application qui nous intéresse, il n'est donc pas rentable de couper l'alimentation 3,3-5 V pour économiser l'énergie entre deux communications (une liaison par protocole zigbee, avec son temps de connexion qui se compte en fraction de seconde, est en ce sens plus efficace). Par ailleurs, il est apparu à l'usage qu'alimenter directement le microcontrôleur avec la pile qui est utilisée pour générer le 5 V est source de trop de bruit sur l'alimentation du microcontrôleur pour permettre une mesure exploitable. Nous

[7] http://www.free2move.se et en particulier le F2M01C1, alimenté en 5 V (par exemple depuis USB). Le F2M01SXA est disponible auprès de Lextronic pour 94 euros

[8] www.free2move.net/uploads/downloads/Wireless_UART_protocol_v3.pdf

avons donc élevé la tension des piles (qui doit être au dessus de 1 V pour que le convertisseur fonctionne) à 5 V, pour en dériver l'alimentation du convertisseur RS232-bluetooth d'une part et, d'autre part, abaisser par régulateur linéaire 5V-3,3V (LE33CZ ou LM1117-3.3) la tension qui alimente le microcontrôleur, garantissant ainsi un potentiel de 3,3 V stable quelle que soit la tension aux bornes des piles. Cette tension d'alimentation stable est requise pour garantir un seuil de charge du circuit RC à un niveau connu lors de la mesure de R.

Par conséquent, nous proposons une gestion efficace de l'énergie avec élévation de la tension de deux piles en série vers 5 V pour en dériver une source stable d'alimentation du microcontrôleur sous 3,3 V, garantissant la stabilité nécessaire à une mesure précise de température (Fig. 5).

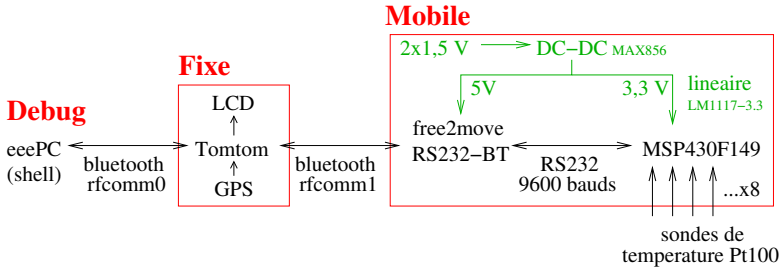

FIG. 5 – Schéma récapitulatif des liaisons entre les divers éléments de mesure

4.3 Robustesse aux défaillances mécaniques

L'utilisation intensive des interruptions matérielles permet une gestion efficace de l'énergie puisque le microcontrôleur passe la majorité de son temps en mode veille (LPM1, *Low Power Mode 1*). Le danger cependant est de ne jamais être réveillé de ce sommeil en cas de dysfonctionnement d'un périphérique. Cette situation est rencontrée lorsqu'une liaison vers une des thermistances est rompue : les charges ne s'accumulent jamais dans le condensateur ou ne peuvent pas s'en évacuer, ce qui implique que l'interruption de passage de seuil de tension n'est jamais activée et le programme bloque pour ne jamais rendre la main au cours d'une mesure.

En effet, le principe de la mesure de résistance est basée sur une mesure de temps de décroissance d'une tension aux bornes d'un condensateur, temps proportionnel à la résistance qui limite le courant de décharge. Cette résistance a pour vocation à être placée au plus près du point de mesure, donc être exposée à un environnement mécanique rude qui peut éventuellement entraîner sa destruction. En l'état, l'ensemble du circuit de mesure est bloqué si une des résistances est détruite, puisque le microcontrôleur passe en mode veille pour attendre le déclenchement d'une interruption de comparaison de tension, interruption qui n'arrive jamais puisque le condensateur ne peut plus se décharger (ni se charger) au travers de la thermistance détruite. La solution classique du chien de garde (*watchdog*) est ici implémentée manuellement au moyen d'un second timer un délai beaucoup plus long que le temps de décroissance de la tension du condensateur : la sortie du mode veille du microcontrôleur s'effectue alors soit lorsque la mesure est correcte, soit lorsque le second timer a atteint sa limite. Dans ce cas, sans réinitialiser le microcontrôleur, nous passons à la mesure de résistance suivante et marquons la mesure comme erronée (le gestionnaire d'interruption étant différent, nous sommes capables de distinguer ces deux cas : code 1).

En conclusion de ce développement électronique, nous obtenons un instrument de mesure de température robuste, de consommation réduite. L'ajout du convertisseur RS232-bluetooth limite à lui seul l'autonomie du circuit puisque

- le convertisseur DC-DC et le microcontrôleur consomment moins de 1 mA
- le régulateur linéaire de tension présente un courant de fuite qui peut être réduit sous le mA dans le cas du LE33CZ (le LM1117-3.3 présente un courant de fuite de près de 10 mA !)
- le convertisseur RS232-bluetooth consomme moins de 1 mA en l'absence d'activité, pour passer à 7 mA lorsqu'un câble RS232 est branché mais pas de communication asynchrone, pour

grimper à 23 mA lors d'une communication RS232 mais absence d'interlocuteur bluetooth, et finalement atteindre 40 à 60 mA lorsque l'association bluetooth est réalisée.

Le précision de la mesure est établie lors d'un calibrage en enceinte climatique (Fig. 6) : nous constatons que une fois stabilisées, les diverses mesures sont obtenues avec un écart type d'environ 0,2^{o}C et une reproductibilité d'une sonde à l'autre du même ordre de grandeur.

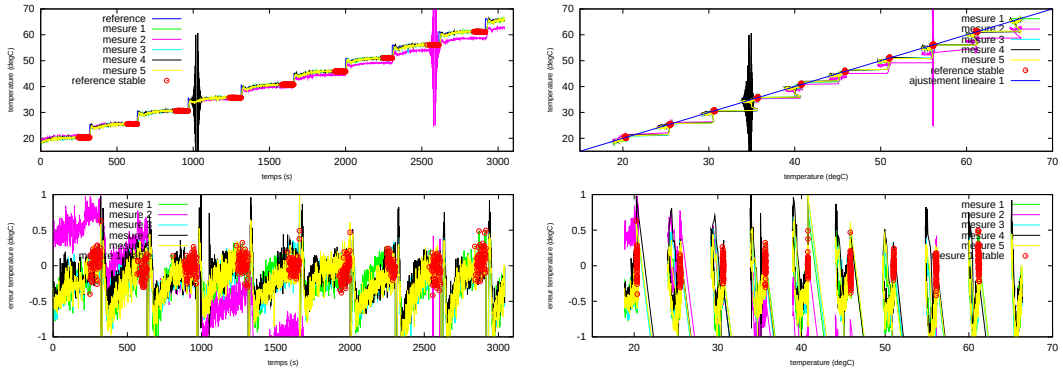

FIG. 6 – Gauche : calibrage des 5 sondes Pt100 en enceinte climatique, par pas de 5^{o}C, afin d'obtenir la correspondance expérimentale entre la mesure de constante de temps et la température du capteur. Droite : analyse de l'erreur en fonction de la température. Sur toute la gamme 15-65^{o}C, l'écart type sur la mesure est inférieur à 0,2^{o}C sauf pour la seconde sonde (magenta) dont le calibrage présente une phase grossièrement erronée de mesures.

5 Affichage et stockage des mesures sur Tomtom

Afin d'acquérir les données émises par le MSP430 sur la liaison bluetooth, nous avons besoin de plusieurs outils, disponibles ou non par défaut au shell du Tomtom Go :

1. configuration de la liaison bluetooth depuis le Tomtom Go vers le convertisseur Free2Move : `rfcomm`
2. lire les informations *ligne par ligne* depuis le port série virtuel au moyen de `read` (au lieu d'attendre une fin de fichier comme le ferait une redirection par > ou | de la sortie du port série vers notre application de traitement et d'affichage)
3. fournir les informations à l'utilisateur (afficher un message sur interface graphique) et stocker les informations dans un fichier : le flux de données est divisé en deux au moyen de `tee` du paquet `sh-utils-2.0-1.armv4l.rpm` dans les archives de www.netwinder.org [9].

Le traitement des informations ligne par ligne est un point important pour rafraîchir régulièrement l'information : l'utilisation en ce sens de la fonction `read` est illustrée par :

```
jmfriedt@eee:~/tomtom$ cat affiche.sh
#!/bin/sh
while true; do
  read DATA
  a="`date`  $DATA"
  echo $a
done
# ./affiche.sh < /dev/rfcomm1 | tee fichier_sauvegarde
Wed Aug 26 21:19:39 CEST 2009 1178 243d 2468 2469 2478 246b fbff
```

[9] de façon, générale, nous piochons dans les exécutables mis à disposition sur ce site web lorsqu'un programme n'est pas disponible sur le Tomtom, tel que décrit à http://wiki.opentom.org/Talk:TomTom_Console

```
Wed Aug 26 21:19:40 CEST 2009 1174 244b 246b 246e 246f 2463 fbfd
Wed Aug 26 21:19:41 CEST 2009 1179 243d 246e 2478 2474 245c fbff
Wed Aug 26 21:19:42 CEST 2009 1175 243b 246a 247b 2479 245d fbf8
...
```

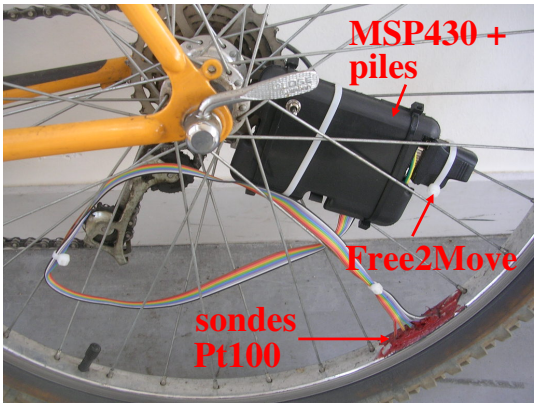

FIG. 7 – Installation du circuit de mesure des 5 sondes Pt100 et transfert de données par liaison bluetooth : bien que les dimensions du circuit n'aient pas été une contrainte forte lors de l'assemblage de ce circuit, il est suffisamment compact pour tenir, incluant les 2 piles et la gestion des alimentations, dans une boîte tenant entre les rayons d'une roue de vélo.

Les informations sont communiquées sur le port série dans la base naturelle d'un système numérique, *i.e.* en hexadécimal, afin de limiter la charge de calcul sur le système embarqué. Les données à transmettre sont ainsi traitées quartet par quartet, en ajoutant les constantes appropriées pour afficher le code ASCII du caratère correspondant à la valeur transmise (0 à F, soit l'ajout de 0x30 à la valeur transmise si celle-ci est inférieure à 10 et 0x37 sinon, les concepteurs de la norme ASCII ayant eu l'étrange idée de placer des symboles mathématiques et de ponctuation entre les caractères '9' et 'A' pour les raisons décrites dans `man ascii`).

Cependant, la majorité des utilisateurs de systèmes embarqués ne lisent malheureusement pas l'hexadécimal couramment – *a fortiori* pour le calcul flottant – et il faut les aider dans l'exploitation des mesures en convertissant les valeurs acquises en informations lisibles par un humain.

Les implications sont, lors du traitement des informations reçues du microcontrôleur :

- lire les données au format hexadécimal – naturel pour un microcontrôleur – sur l'entrée standard
- interpréter les informations fournies dans la bonne base et les exporter dans la base préhistorique des humains (base 10)
- effectuer le calcul avec une précision compatible avec celle des capteurs, donc calcul flottant (Fig. 10).

L'outil classique `bc` répond à toutes ces conditions après avoir renseigné la variable de la base des données en entrée correctement. Par défaut, `bc` prend en entrée des données en décimal et fournit la réponse en base 10, mais ces deux paramètres se modifient en définissant les variables `ibase` et `obabase`. Nous ajoutons donc à la liste des outils nécessaires le paquet `bc-1.06-12.armv4l.rpm` et les dépendances associées, toujours depuis le site www.netwinder.org. Le script `script.bc` suivant va nous permettre de convertir chaque ligne contenant une séquence de nombres en représentation hexadécimale vers une représentation décimale en virgule flottante en tenant compte des coefficients de calibrage qui ont été identifiés auparavant (Fig. 6) :

```
a=$(echo "ibase=16; obase=A; scale=1; ($2-2213)/15.DE7" | bc -l)
b=$(echo "ibase=16; obase=A; scale=1; ($3-21EA)/18.0D0" | bc -l)
c=$(echo "ibase=16; obase=A; scale=1; ($4-2228)/16.1AE" | bc -l)
```

```
d=$(echo "ibase=16; obase=A; scale=1; ($5-222F)/16.1A9" | bc -l)
e=$(echo "ibase=16; obase=A; scale=1; ($6-2221)/16.189" | bc -l)
echo $a $b $c $d $e
# ./script.bc 249D 24AE 24B0 24C0 24AF
29.7 29.4 29.3 29.7 29.5
```

Noter que une fois la base d'entrée (`ibase`) définie à 16, la configuration de la base de sortie (`obase`) à 10 en décimal s'obtient par `A`.

Le résultat de tous ces développements est la capacité de mesurer la température sur une roue (Fig. 7), transférer les informations par une liaison sans fil au TomtomGo et les stocker après les avoir datées (Fig. 8)

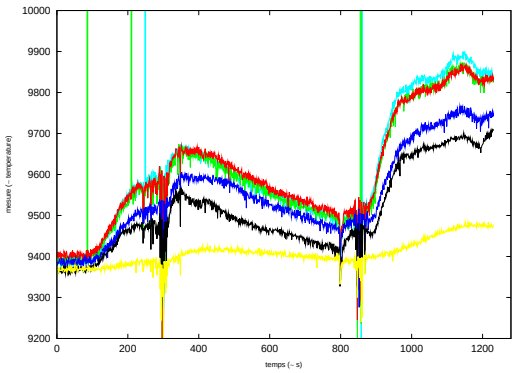

FIG. 8 – Mesure de température sur une jante de vélo. Ces courbes rprésentent le temps de décharge τ du circuit RC formé par chaque thermistance R en série avec un condensateur C. La courbe jaune est obtenue lors de la mesure d'une résistance de référence, illustrant l'évolution de C avec la température (la résistance de référence étant supposée constante).

6 Affichage des résultats dans l'interface graphique du TomtomGo

Tomtom exploite un mode de communication assez surprenant compte tenu des nombreuses possibilités de communication entre processus fournies par unix : un fichier dont le nom suit un standard contient la commande à exécuter par le gestionnaire de l'interface graphique. Ce gestionnaire acquitte l'exécution de la commande en effaçant le fichier qui a servi à l'interaction et en fournissant, dans un autre fichier, le résultat de la commande. Bien que ces échanges de fichiers se fasse sur un RAMdisk, l'efficacité de ces transactions est douteuse.

En pratique, un ordre est donné à l'interface Tomtom en écrivant un fichier nommé
`SDK.TomTomNavigationServer.PID.N.message`
dans le répertoire `/var/run`. `PID` est le numéro du processus qui effectue la requête (obtenu par $$ dans bash) et `N` est un numéro de requête, que nous maintiendrons toujours à 1. Ce fichier contient la commande à effectuer, tel que décrit plus bas dans le paragraphe suivant. L'ordre est effectivement transmis lorsque `/var/run/SDK.TomTomNavigationServer.PID.N.finished` contenant le mot `finish` est écrit. Une fois la commande réalisée, l'interface Tomtom efface ces deux fichiers et crée `/var/run/TomTomNavigationServer.SDK*`, que nous n'exploiterons pas mais nous nous contenterons d'effacer à chaque acquittement.

L'ensemble des interactions possibles depuis le shell avec l'interface graphique du Tomtom Go est répertorié à [3]. La partie qui nous concerne est

– obtenir la coordonnée géographique au moment de la mesure : une solution est d'interagir avec l'interface Tomtom au moyen de la commande `GetCurrentPositionV01`, mais nous

avons préféré exploiter la commande `whereami` fournie par www.opentom.org/Whereami
- fournir les informations à l'utilisateur : la commande `FlashMessageV01` affiche une fenêtre dans un coin de l'interface graphique (Fig. 9) pendant une durée fournie comme second argument, le premier argument étant le message lui même. Les arguments sont séparés par des `|`.

Ainsi, nous proposons dans une boucle infinie

```
read DATA
b=‘script.bc $DATA‘
a="‘date‘ $b "
a="$a ‘whereami | cut -c 1-17 | sed ’s/|/ /g’‘"
rm /var/run/TomTomNavigationServer.SDK*
echo -e "FlashMessageV01|$a|3000|\0\c" > "/var/run/SDK.TomTomNavigationServer.$$.1.message"
echo finish > "/var/run/SDK.TomTomNavigationServer.$$.1.finished"
```

pour continûment afficher les informations acquises sur stdin et traitées par le script faisant appel à `bc` pour une mise en forme exploitable (présentée auparavant) à l'utilisateur. Ce script est appelé par

```
rfcomm bind 2 00:0B:CE:00:84:BA 1
affiche.sh < /dev/rfcomm2 > /mnt/sdcard/jmf/$nom &
```

Noter que cette dernière partie est avantageusement précédée de

```
if test "‘rfcomm | grep rfcomm2‘" ; then
rfcomm release 2
fi
```

afin de garantir que le port série virtuel `rfcomm2` est libre pour une liaison avec le module Free2Move (conflit possible avec une utilisation antérieure de ces scripts lorsque le Tomtom Go est entré en mode veille et bloque la réinitialisation de la liaison bluetooth à son réveil). La fréquence de raffraîchissement de l'information est déterminée par le microcontrôleur : la fonction `read` est bloquante et l'information n'est mise à jour que lors de l'envoi d'une trâme par le MSP430. Nous somme cependant limités à un intervalle de temps de l'ordre de 10 s entre deux mesures à cause de la durée d'obtention de la position par `whereami`.

FIG. 9 – Affichage des données acquises (la résistance de référence, 5 résistances des sondes Pt100 et l'estimation de la température par la diode interne au MSP430) précédés de la date et géoréférencées par les informations fournies par le récepteur GPS du Tomtom. L'affichage, en hexadécimal, retranscrit directement les informations obtenues du microcontrôleur.

7 Exploitation et résultats

Une fois une série de mesures (ici température de jante, Fig. 10) géoréférencées acquise, il nous reste à les exploiter afin d'en tirer un maximum d'informations. La mise en contexte géographique est une étape essentielle : nous utilisons GNU/Octave pour générer un fichier au format KML permettant d'insérer nos mesures dans Google Earth qui fournit alors le contexte de mesure sous forme d'un modèle d'élévation sur lequel sont plaquées des photographies aériennes des régions traversées.

FIG. 10 – Exploitation du Tomtom Go720 fixé sur le guidon d'un vélo, mettant à jour toutes les 10 secondes les 4 estimations de température de la jante (ici, 18,1, 16,9, 17,1 et 17,1 degrés) et la position du récepteur (ici 5,98 degrés est, 47,25 degrés nord) sur l'interface de navigation fournie avec l'instrument. Ces informations sont par ailleurs stockées en mémoire non-volatile pour post-traitement.

Le choix de GNU/Octave pour générer le fichier KML est initialement dicté par la volonté de coder la couleur de l'information de position de la mesure par la température. Il est probable que d'autres outils sont plus appropriés ou plus largement répandus, mais celui-ci répond à nos exigences de souplesse du tracé au travers de gnuplot et de sauvegarde du code de couleur dans le fichier KML au format hexadécimal (Code 2). GNU/Octave supporte une gestion des chaînes de caractères très proche du C, tout en proposant la souplesse d'un langage scripté dont les performances en terme de vitesse ou d'occupation mémoire n'ont ici aucune importance.

Le résultat de l'exploitation des données est visible à Fig. 11 qui illuste le codage de l'information de température par la couleur, et en Fig. 12 la mise en contexte géographique et topographique avec le modèle de terrain inclus dans Google Earth qui facilite l'interprétation des données. Nous constatons en effet que tous les points d'échauffement de la jante (en haut à gauche de Fig. 11 et à la longitude 6,01, latitude 47,235) correspondent à des pentes descendantes dans lesquelles nous avons volontairement serré les freins. Au contraire, toute la partie inférieure de la Fig. 11 correspond à une piste cyclable sans relief le long du Doubs. Noter une élévation significative de la température lors des mesures *postérieures* dans le "carré" à droite de la Fig. 11 par rapport au parcours le long du Doubs : le centre "ville" de Besançon conserve significativement plus de chaleur une fois la nuit tombée (ces parcours ont été effectués entre 20 h et 22 h début septembre).

8 Conclusion

Nous avons présenté les divers outils pour développer un BAN (Bike Area Network) en vue de la mesure de la température d'un objet mobile et la restitution en temps réel de l'information

```
f=fopen('sortie.kml','w');
fprintf(f,'<?xml version="1.0" encoding="UTF-8"?>\n');
fprintf(f,'<kml xmlns="http://www.opengis.net/kml/2.2">\n');
fprintf(f,'<Document>\n');

load gps1251914042.dat
x=gps1251914042;
a=jet

hold on
for ii=1:length(x)
    if (x(ii,8)>14)
      couleur=a(floor((x(ii,8)-13)*5),:);
      b=plot(x(ii,11)/1e5,x(ii,12)/1e5,'o');
      set(b,'color',couleur);

      fprintf(f,'<Placemark><name>temperature</name><Style> <geomColor>ff%02x%02x%02x</geomColor> </Style>\n',\
                floor(couleur(3)*255),floor(couleur(2)*255),floor(couleur(1)*255));
      fprintf(f,'<Polygon> <outerBoundaryIs> <LinearRing> <coordinates>\n');
      fprintf(f,'%f,%f,1000\n',x(ii,11)/1e5,x(ii,12)/1e5);
      fprintf(f,'%f,%f,1000\n',x(ii,11)/1e5+5e-4,x(ii,12)/1e5);  %1 deg=111 km => 1e-5=1 m
      fprintf(f,'%f,%f,1000\n',x(ii,11)/1e5+5e-4,x(ii,12)/1e5+5e-4);
      fprintf(f,'%f,%f,1000\n',x(ii,11)/1e5,x(ii,12)/1e5+5e-4);
      fprintf(f,'</coordinates> </LinearRing> </outerBoundaryIs> </Polygon></Placemark>\n');
  end
end
fprintf(f,'</Document></kml>\n');
fclose(f)
```

TAB. 2 – Script GNU/Octave pour générer un fichier KML à partir des informations géoréférencées : chaque polygone localisé aux coordonnées de la mesure se voit affecté une couleur issue de la palette `jet`.

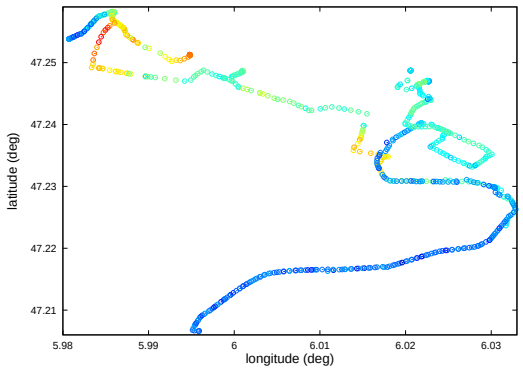

FIG. 11 – Codage par la couleur de la température en chaque point de mesure (mesure toutes les 10 secondes – l'espacement entre les points est donc proportionnel à la vitesse du véhicule mobiles).

à un utilisateur. Les informations sont géoréférencées et converties en un format exploitable par un humain qui ne lit pas communémement l'hexadécimal au moyen des outils classiquement mis à disposition sur un système unix.

Afin d'atteindre cet objectif, nous avons exploité le système GNU/Linux installé par défaut sur un assistant de navigation Tomtom Go 720, accessible grâce aux outils mis à disposition par la communauté de opentom.org. Une console nous a permis de nous familiariser avec les outils disponibles, notamment le shell de busybox et les outils de configuration de l'interface Bluetooth.

L'ajout de quelques outils dédiés – notamment le calculateur `bc` – a complété une interface

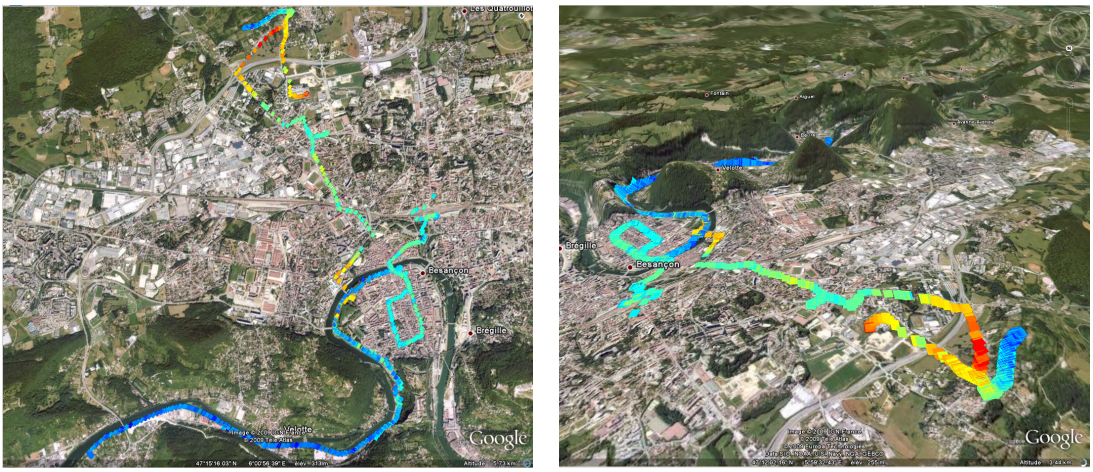

FIG. 12 – Mise en contexte géographique des informations acquises : à gauche sur une vue en 2D, à droite en 3D en exagérant les reliefs. La figure de droite montre clairement l'échauffement de la jante dans les pentes (freins). La figure de gauche illustre un effet plus subtil d'élévation de la température moyenne en milieu urbain (pour autant que le "centre" de Besançon puisse être qualifié de tel) par rapport aux berges du Doubs.

conviviale exploitant le logiciel de navigation propriétaire pour afficher la température de la jante de la roue.

Remerciements

Nous remercions L. Fagot-Revurat (MFPM, Clermont-Ferrand) pour le prêt gracieux du convertisseur Bluetooth-RS232. T. Petazzoni a eu l'excellente initiative de proposer les *lightning talks* pendant la session "systèmes embarqués" des Rencontres du Logiciel Libre 2009 au cours desquels le Tomtom Go a été présenté. T. Rétornaz (doctorant à FEMTO-ST, Besançon) a identifié SLAA129B comme méthode de mesure efficace en consommation électrique de la consommation, algorithme mis en œuvre sur MSP430F149 par B. François.

Références

[1] C. Daniel & T. Kleffel, *Hacking into Tomtom Go – Reverse engineering des Embedded-Linux Navigationsystems*, 22nd Chaos Communication Congress (2005) [en Allemand]

[2] X. Garreau, *Bluetooth : développons*, GNU/Linux Magazine France **80**, Février 2006, 40-49

[3] TomTom Navigator SDK, TomTom B.V. 2004 et www.opentom.org/FileInterface

[4] K. Quiring, *Implementing An Ultralow-Power Thermostat With Slope A/D Conversion*, MSP430 Application Report SLAA129B, Texas Instruments, Janvier 2006

[5] J. Williams, *Bridge circuits – marrying gain and balance*, Linear Technology Application Note 43, Juin 1990

[6] J.-M Friedt, A. Mass, F. Bassignot, *Les microcontrôleurs MSP430 pour les applications faibles consommation – asservissement d'un oscillateur sur le GPS*, GNU/Linux Magazine France **98**, Octobre 2007