Communication asynchrone et interface graphique portables sous Qt

J. Garcia, J.-M Friedt

14 mai 2008

1 Introduction

Malgré la popularité de GNU/Linux dans la communauté des développeurs, de nombreux utilisateurs s'obstinent à utiliser d'autres systèmes d'exploitations. Notre objectif dans cette présentation est de proposer une solution pour générer des applications utilisables pour un maximum d'utilisateurs, donc pour un maximum de plateformes. De plus, la mode aujourd'hui est aux applications graphiques. Alors que des applications en mode texte [1] réaliseraient parfaitement la même fonction, la majorité des utilisateurs est rebutée à l'idée de lancer une fenêtre de commande DOS sous Windows ou un terminal sous Unix. Bien que nous regrettions cette attitude et la perte de performances associée, nous allons ici nous efforcer de répondre aux exigences de l'utilisateur en présentant les méthodes de développement d'applications graphiques portables. Nous allons de plus nous imposer la contrainte de communiquer avec des systèmes embarqués tels que ceux présentés dans les numéros précédents de ce magazine [2, 3, 4]. Ceci implique la capacité à utiliser des ports de communication tels que le port série (RS232) afin de communiquer avec des microcontrôleurs.

Nous avons exploré diverses voies avant de converger sur le choix de la programmation sous environnement Qt : Perl et son environnement graphique Perl/Tk, GTK, Java. Qt4 est la seule solution que nous ayons testé qui réponde à nos critères de portabilité (pas de modification des codes source pour compiler une application GNU/Linux ou Windows) et de performances. Mentionnons la disponibilité d'une classe pour l'accès au port série en Java – `rxtx` disponible à `http://rxtx.org/` – tout à fait satisfaisante mais qui ne sera pas développée ici.

2 Communication par interfaces asynchrones

Malgré la mort annoncée du port série (RS232), la communication asynchrone reste un périphérique communément utilisé sur les applications embarquées. Sa simplicité et sa fiabilité lui garantissent encore de beaux jours malgré la volonté des industriels de l'informatique grand public d'imposer l'USB. L'interface entre la communication asynchrone (côté microcontrôleur) et l'USB (ordinateur personnel) est notamment fournie par un composant très simple d'emploi, le FT232R de FTDI [5]. Ce composant s'interface d'un côté aux broches de communication issues de l'UART (*Universal Asynchronous Receiver Transmitter*, implémentation matérielle du protocole RS232) de tout microcontrôleur, et de l'autre côté fournit une interface USB qui se programme, grâce aux drivers `http://www.ftdichip.com/FTDrivers.htm` fournis par le fabriquant FTDI, comme un port série. Sous GNU/Linux, le module noyau `ftdi_sio` associe `/dev/ttyUSB`x (x=0, 1...) à une interface de programmation classique de port série telle que nous en avions l'habitude avec `/dev/ttyS`x.

Si nous désirons que nos application embarquées communiquent avec un maximum d'ordinateurs personnels, il est fondamental de savoir communiquer avec un port série de façon portable, indépendante du système d'exploitation.

3 L'accès au port série en C

Commençons par poser le problème en présentant les méthodes "classiques" d'accès au port série en C sous Linux, et en Visual C(++) pour Microsoft Windows. Bien qu'il soit techniquement possible de jouer sur les `#define` pour définir deux environnements portables, cette méthode est fastidieuse (bien que fonctionnelle). Elle se couple efficacement avec tout environnement portable d'interface graphique et mérite en ce sens d'être mentionnée.

Notre objectif dans ce document est d'*éviter* le genre de gestion des cas particuliers que nous présentons dans un code couramment utilisé dans nos applications C gérant une communication

```
#ifdef linux       // All examples have been derived from miniterm.c
#include <arpa/inet.h>
struct termios oldtio,newtio;
#define BAUDRATE B38400
#define RSDEVICE "/dev/ttyS0"

extern struct termios oldtio,newtio;

int init_RS232()
{int fd;
 fd=open(RSDEVICE, O_RDWR | O_NOCTTY );
 if (fd <0) {perror(RSDEVICE); exit(-1); }
 tcgetattr(fd,&oldtio); /* save current serial port settings */
 bzero(&newtio, sizeof(newtio)); /* clear struct for new port settings */
 newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;  /* _no_ CRTSCTS */
 newtio.c_iflag = IGNPAR;       // | ICRNL |IXON;
 newtio.c_oflag = IGNPAR;       //ONOCR|ONLRET|OLCUC;
 newtio.c_cc[VTIME]    = 0;     /* inter-character timer unused */
 newtio.c_cc[VMIN]     = 1;     /* blocking read until 1 character arrives */
 tcflush(fd, TCIFLUSH);tcsetattr(fd,TCSANOW,&newtio);
 return(fd);
}

void free_rs232(int fd)         /* restore the old port settings */
{tcsetattr(fd,TCSANOW,&oldtio);close(fd);}

#else   // cas NON-Linux = MS-Windows

// hton[l/s] : use lib ws2_32.lib in
// C:\Program Files\Microsoft Visual Studio\VC98\Lib
// #include <winsock2.h.>

#define RSdelay 10
```

```
HANDLE  portcom ;
DWORD dwBytesRecd,dwBytesWritten;

int init_RS232()
{DCB dcb;COMMTIMEOUTS cmm;
 portcom=CreateFile("COM1", GENERIC_READ | GENERIC_WRITE,1,NULL,
           OPEN_EXISTING,0,NULL ); // 0=NO overlapped IO
 dcb.DCBlength = sizeof( DCB ) ;
 GetCommState(portcom, &dcb ) ;
 dcb.BaudRate=38400;
 dcb.ByteSize=8;
 dcb.Parity=NOPARITY;
 dcb.StopBits=ONESTOPBIT;
 SetCommState( portcom, &dcb ) ;

 GetCommTimeouts(portcom, &cmm ) ;
 cmm.ReadIntervalTimeout=RSdelay;
 cmm.ReadTotalTimeoutConstant=RSdelay;
 cmm.ReadTotalTimeoutMultiplier=RSdelay;
 cmm.WriteTotalTimeoutConstant=RSdelay;
 cmm.WriteTotalTimeoutMultiplier=RSdelay;
 SetCommTimeouts(portcom, &cmm) ;
 return(1);
}

void write(int dummy,char *cmd,int len)          // POSIX write for windows
{WriteFile(portcom,cmd,len,&dwBytesWritten,0);}

int read(int dummy,unsigned char *cmd,int len)   // POSIX read for windows
{do {ReadFile(portcom,cmd,len,&dwBytesRecd,0);}
 while (dwBytesRecd==0);
 return(dwBytesRecd);
}
#endif
```

Tab. 1 – Exemple de code fonctionnel sous GNU/Linux ou MS Windows selon la disponibilité de la variable d'environnement `linux` : en cas d'utilisation de Windows, nous tentons d'émuler grossièrement les fonctions `read()` et `write()` POSIX.

RS232 (code 1). En effet, ce genre de cas particulier se multiplie en fonction des plateformes que nous désirons supporter, jusqu'à rendre le code illisible et difficile à maintenir. Par ailleurs, le C de base manque d'un support portable d'interface graphique : parfait pour les logiciels en console (`stdio`), l'ajout d'une interface utilisateur devient rapidement fastidieuse.

4 L'accès au port série sous Qt

Afin de suppléer à ces déficienses du C, nous proposons de développer une classe Qt – environnement de développement proposé par Trolltech (trolltech.com), libre si l'utilisateur propose ses applications sous license GPL – pour gérer le port série de façon portable, et donc d'exploiter les possibilités d'interface graphique portable fournies par cet environnement de développement. Nous allons dans un premier temps présenter l'installation de l'environnement de développement Qt4, et en particulier la capacité à cross-compiler et tester une application à destination de Windows sur plateforme GNU/Linux. Cette documentation n'a *pas* pour vocation de présenter le développement d'interfaces graphiques sous Qt pour interagir avec l'utilisateur, mais uniquement d'aborder les points de l'affichage des informations récupérées lors de la communication asynchrone avec un système embarqué. Le lecteur pourra se référer à la série d'articles sur Qt3 parue entre 2002 et 2004 dans ces pages pour une présentation plus exaustive de cet environnement de développement [6].

4.1 Qt4 sous Debian GNU/Linux

L'installation se déroule trivialement par `apt-get install libqt4-dev` qui se chargera d'aller chercher les dépendances requises.

La compilation s'obtient ensuite :

- en développant son code Qt en C++ dans son éditeur de texte favori
- en effectuant une première fois `qmake -project` pour générer un fichier `.pro` contenant les sources et les dépendances nécessaires à la compilation
- une première fois `qmake` pour générer le `Makefile`

– finalement, le classique `make` pour chaque remise à jour de l'exécutable lorsque les sources ont été modifiées.

```
/*************************************************
 * http://doc.trolltech.com/4.3/tutorial-t1.html
*************************************************/

#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QPushButton hello("Hello world!");
    hello.resize(200, 30);
    hello.show();
    return app.exec();
}
```

Nous validons la chaîne de compilation en exécutant le programme résultant (Fig. 1).

FIG. 1 – Résultat de la compilation de l'exemple Hello World fourni par TrollTech pour valider la chaîne de compilation, ici sous GNU/Linux.

Nous sommes donc capables de générer une interface graphique sous GNU/Linux. Àprès cette introduction triviale, nous allons dans un premier temps considérer la cross-compilation d'applications pour Windows. Le lecteur qui ne s'intéresse pas à ce système mais désire tout de suite consulter la communication asynchrone sous Qt4 peut se reporter directement à la section 4.3.

4.2 Cross-compilation d'une application Qt4 pour Windows sous Linux

L'installation d'un environement de compilation pour Windows sous Linux nécessite un compilateur croisé capable de traduire un code source en fichier exécutable pour une autre architecture, en l'occurence un programme pour Windows [7, 8] :

– dans cet exemple `mingw32` sera notre compilateur,
– il nous faudra avoir à disposition les librairies Qt pour Windows, le plus simple étant de les télécharger sur le site de Trolltech,
– finalement, le dernier outil nécessaire sera `wine` pour pouvoir tester notre application.

Sous debian, nous installons les paquets appropriés pour la cross-compilation et le test des binaires que nous générerons :

```
su -c 'apt-get install mingw32 wine'
```

Une fois cet environnement fonctionnel, nous ajoutons Qt pour Windows que nous obtenons à partir du site de Trolltech [9] :

```
wget http://ftp.ntua.gr/pub/X11/Qt/qt/source/qt-win-opensource-4.3.4-mingw.exe
```

L'installation de Qt avec wine (figure 2) s'obtient par la commande

```
wine qt-win-opensource-4.3.4-mingw.exe
```

Une fois l'installation de Qt achevée correctement, il faut modifier la base de registres de votre environnement Windows avec `wine` pour y ajouter Qt dans le `PATH` : par défaut l'installation s'effectue dans `C:/Qt/4.3.4/`. Si la clef `HKEY_CURRENT_USE/Environment/PATH` n'existe pas, il faut la créer pour y ajouter : `C:/Qt/4.3.4/bin` comme sur la Fig. 3

Pour achever la configuration de notre environement Windows, il nous reste à placer la librairie `mingwm10.dll` dans le répertoire système approprié :

```
gunzip /usr/share/doc/mingw32-runtime/mingwm10.dll.gz
mv mingwm10.dll  ~/.wine/drive_c/windows/system32/
```

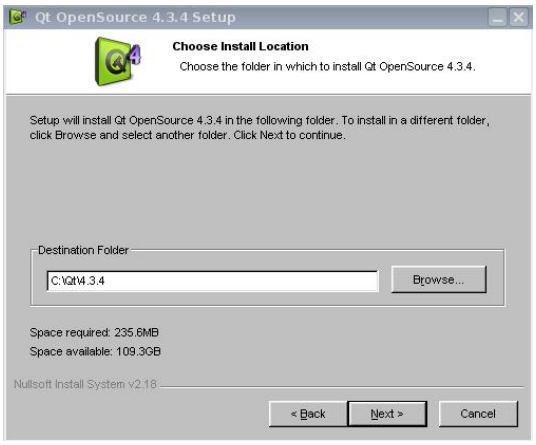


FIG. 2 – Installation de Qt pour Windows avec wine : cette fenêtre s'obtient par `wine qt-win-opensource-4.3.4-mingw.exe`.

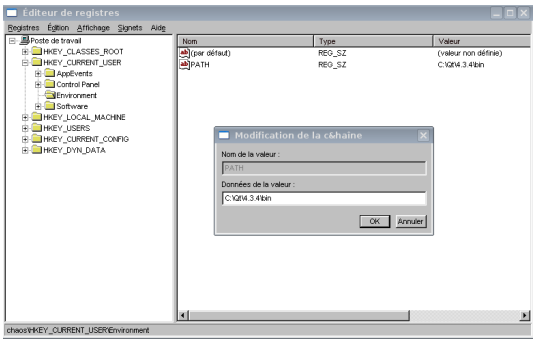


FIG. 3 – Ajout de Qt dans le `PATH` de `wine`, obtenu par l'exécution de la commande `wine regedit`

Nous devons modifier les règles de compilation définies dans le fichier `mkspec`, afin de passer les bonnes options dans nos futurs `Makefile`. Afin de respecter les règles pour ce nouvel environnement, les principales différences avec le fichier original sont décrites ci-dessous :

```
cp -ar /usr/share/qt4/mkspecs/win32-g++ /usr/share/qt4/mkspecs/win32-cross
su -c 'vi /usr/share/qt4/mkspecs/win32-cross/qmake.conf'
```

```
#
# qmake configuration for win32-cross
# Written for MinGW
#

MAKEFILE_GENERATOR = MINGW
QMAKE_CC = i586-mingw32msvc-gcc
QMAKE_CXX = i586-mingw32msvc-g++
QMAKE_INCDIR = /usr/i586-mingw32msvc/include/
QMAKE_INCDIR_QT = /home/julien/.wine/drive_c/Qt/4.3.4/include
QMAKE_LIBDIR_QT = /home/julien/.wine/drive_c/Qt/4.3.4/lib
QMAKE_LINK = i586-mingw32msvc-g++
MINGW_IN_SHELL = $$(MINGW_IN_SHELL)
#isEqual(MINGW_IN_SHELL, 1) {
QMAKE_DIR_SEP = /
QMAKE_COPY = cp
QMAKE_COPY_DIR = cp -r
```

```
QMAKE_MOVE = mv
QMAKE_DEL_FILE = rm -f
QMAKE_MKDIR = mkdir -p
QMAKE_DEL_DIR = rm -rf
#} else {
...
#}

QMAKE_MOC = $$[QT_INSTALL_BINS]$${DIR_SEPARATOR}moc-qt4
QMAKE_UIC = $$[QT_INSTALL_BINS]$${DIR_SEPARATOR}uic-qt4
QMAKE_IDC = $$[QT_INSTALL_BINS]$${DIR_SEPARATOR}idc

QMAKE_LIB = i586-mingw32-ar -ru
QMAKE_RC = i586-mingw32msvc-windres
QMAKE_ZIP = zip -r -9

QMAKE_STRIP = i586-mingw32msvc-strip
```

une fois le `mkspec` créé, nous pouvons commencer les premiers tests en reprenant le programme de test vu précédemment (section 4.1) afin de vérifier, avec un exemple simple, que notre environnement de compilation fonctionne bien.

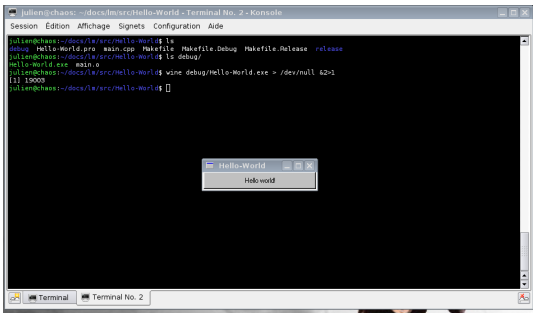


FIG. 4 – Exécution du `Hello-World` avec wine.

Comme précédemment dans le cas de la compilation sous GNU/Linux, pour la première fois, il faut lancer `qmake -project` pour générer un fichier `.pro` qui servira à `qmake` pour faire un Makefile.
L'étape suivante génère un Makefile avec les options adéquates pour compiler un programme exécutable pour Windows.
`qmake -spec win32-cross` suivi de `make` pour créer l'exécutable.

Si la compilation s'est bien terminée, nous pouvons tester notre programme avec la commande suivante comme sur la figure 4.

```
julien@chaos:~/docs/lm/src/Hello-World$ wine debug/Hello-World.exe
```

4.3 Gestion du port RS232 sous Qt4

Nous allons, afin d'exploiter la capacité de Qt4 à fournir une interface unifiée de communication avec les ports série, utiliser la librairie `qextserialport` disponible à http://qextserialport.sourceforge.net/. Pour cela, il nous faut ajouter l'accès à cet objet additionnel dans nos Makefiles. Nous avons vu que le fichier `.pro` servait à `qmake` pour générer les Makefiles automatiquement. Partant d'un `.pro` existant (dans cet exemple un programme nommé `PhotoAutoCapture`, nous ajoutons les références à `qextserialport` dans les champs appropriés (Code 2)

```
TEMPLATE = app
TARGET = PAC
CONFIG += qt warn_off thread release

HEADERS += \
        Sources/PhotoAutoCapture.h \
        Sources/qextserialbase.h \
        Sources/qserialportwidget.h \
        Sources/qextserialport.h

SOURCES += \
        Sources/main.cpp \
        Sources/PhotoAutoCapture.cpp \
        Sources/qextserialport.cpp \
        Sources/qserialportwidget.cpp \
        Sources/qextserialbase.cpp

linux-g++ {
        HEADERS += Sources/Posix_QextSerialPort.h
        SOURCES += Sources/Posix_QextSerialPort.cpp
        DEFINES += _TTY_POSIX_
        LIBS += -lm
}

win32-cross {
        HEADERS += Sources/win_qextserialport.h
        SOURCES += Sources/Win_QextSerialPort.cpp
        DEFINES += _TTY_WIN_
}
```

TAB. 2 – Fichier de configuration `.pro` modifié pour appeler la classe `qextserialport` d'accès au port série. Nous y voyons apparaître les deux configurations qui nous intéressent pour la génération d'exécutables pour GNU/Linux et Windows.

Dans le code source, l'accès au port série se fait de la façon suivante (Code 3) :

1. initialisation du port série (`QextSerialPort *port = new QextSerialPort();`)
2. affectation du port `port->setPortName("COM1")` , Nous constatons que cette phase est encore dépendante du système d'exploitation puisque le nom du port varie selon les OS. ;

3. une ouverture du port en lecture seule `port->open(QIODevice : :ReadOnly))` *suivie* de sa configuration.
4. la lecture de données provenant du port par `port->read()`
5. si en plus de recevoir des informations du système embarqué, nous désirons lui envoyer des commandes, la méthode `port->write(buf,n)` envoie `n` octets du tableau de caractères `buf`.

Le code Qt4 implémentant ces concepts est présenté ci-dessous :

```
#include "qextserialport.h"

QextSerialPort *port = new QextSerialPort();

/* Ouverture du port */
QString portName;
#ifdef Q_OS_WIN
portName="COM1";
#else
portName="/dev/ttyUSB0";
#endif

// L'ordre est important
port->setPortName(portName);
port->open( QIODevice::ReadOnly );
port->setFlowControl(FLOW_OFF);
port->setParity(PAR_NONE);
port->setDataBits(DATA_8);
port->setStopBits(STOP_1);
port->setBaudRate(BAUD57600);
```

```
if(  port->isOpen( ) ) /* reception de donnees */
{
int numBytes = port->bytesAvailable();
if(numBytes > 0) {
if(numBytes > 512) numBytes = 512;
       char buff[512];
int i = port->read(buff, numBytes);
buff[numBytes] = '\0';
}
}

if(  port->isOpen( ) ) /* emission de donnees */
{
  char tmp = 'P';
  int i = port->write(&tmp,1);
}

port->close(); /* fermeture du port */
```

TAB. 3 – Exemple d'initialisation du port série. Ce code met en pratique la récecption de 512 octets puis l'envoi du caractère `P` avant de quitter l'application. Le `#ifdef Q_OS_WIN` permet de s'affranchir de l'OS et ainsi d'être portable.

Un exemple concret d'utilisation de ces codes (Fig. 5) est une application graphique de restitution des trames GPS enregistrées par un microcontrôleur, disponible dans notre dépôt subversion par
`svn co https ://rcs.sequanux.org/gpsqt/tags/stable` (login et mot de passe : anonyme). Ce programme a pour vocation de reproduire – pour une application très précise de récupération et de traitement de trames GPS – les fonctionnalités de `minicom` sous GNU/Linux ou Hyptertermi-nal sous Windows, en éliminant pour l'utilisateur les étapes de configuration du port série puisque nous fournissons des valeurs par défaut "les plus probables". L'exploitation de ce programme pour une application précise nous a conduit à ajouter des fonctions dédiées à l'exploitation des trames GPS (filtrage, conversion de formats) que nous chercherons à réutiliser dans un maximum de circonstances (section 5).

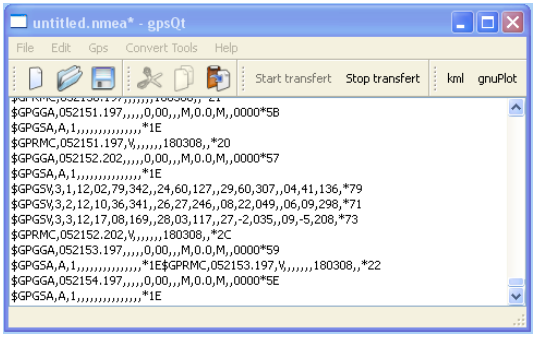


FIG. 5 – Exemple d'application chargée uniquement de recevoir des informations du système présenté dans [2] : après une identification du port série le plus probable (section 4.4), une boucle infinie attend les trames GPS qui avaient été stockées en vue d'être restituées à l'utilisateur à l'issue de son trajet.

4.4 Détection du port série

Un problème récurrent sous Windows est le changement continuel du nom du port série virtuel associé à un système embarqué muni d'un composant FTDI. En effet, alors que sous GNU/Linux le premier périphérique muni d'une telle interface est *systématiquement* associé à `/dev/ttyUSB0`, le second à `/dev/ttyUSB1` ..., sous Windows, nous nous retrouvons rapidement à manipuler des COM15 ou COM20 lorsque 20 circuits différents munis de 20 composants FT232RL ont été utilisés. Afin d'éviter de rechercher pour chaque nouveau système connecté au PC sous Windows, nous proposons de balayer tous les ports disponibles entre COM3 et COM100, à la recherche de ceux qui sont occupés par un périphérique, et de sélectionner par défaut celui dont le nombre est le plus élevé. Cette stratégie a bien fonctionné, *sauf* sur un portable équipé d'une interface Bluetooth qui apparaissait comme un port série de nombre très élevé.

```
/* Parcours COM101 a COM2 pour trouver le port de + haut niveau ouvrable
QStringList gpsQt::testPort()
{
QString portSerieTmp;
QStringList portSerieTmpList;
for (int i = 101 ; i > 2  ; i-- )
{
QString str;
str.setNum(i);
if ( i < 10 )
portSerieTmp = "COM"+str;
else
portSerieTmp = "\\\\.\\COM"+str;

port->setPortName(portSerieTmp);

if ( port->open(QIODevice::ReadOnly))
{
port->close();
portSerieTmpList<<portSerieTmp;
}
}
return portSerieTmpList;
}
...
#ifdef Q_OS_WIN
        listport = testPort();
#else
        portSerie = "/dev/ttyUSB0"
#endif
```

Notez la syntaxe différente pour les ports COM inférieurs ou supérieurs à 10. Cette syntaxe différente est la cause du dysfonctionnement d'un certain nombre de logiciels sous Windows incapables de gérer les ports supérieurs à 10 car ne tenant pas compte de cette nouvelle syntaxe (par exemple des programmeurs de microcontrôleurs, initialement écrits pour fonctionner sur port série "classique" – généralement COM1 à COM4 – qui ne fonctionnement plus lors du remplacement du MAX232 par un FT232R).

L'application doit ici encore tenir compte explicitement des cas Windows et GNU/Linux, d'où la condition associée à `#ifdef Q_OS_WIN`.

4.5 Acquisition d'un flux continu de données

Afin de récupérer des données à partir d'un port série de façon continue il existe plusieurs techniques, dont les *Threads*, qui lisent en permanence ce qui arrive sur le port série mais en parallèle à l'application, et les *Timer* qui déclenchent à intervalle régulier la lecture du buffer. Ces deux méthodes permettent de recevoir les informations en permanence et de les traiter en même temps.

La classe suivante est un exemple de *thread* utilsant la classe `QThread` de `Qt` :

```
//fichier receiverthread.h
#ifndef ReceiverThread_H
#define ReceiverThread_H

#include <QThread>
#include "qextserialport.h"

class ReceiverThread : public QThread
{
public:
    ReceiverThread(QextSerialPort &inPort);
    void run();
private:
    QextSerialPort port;
};

#endif


//fichier receiverthread.cpp
#include <QThread>
#include <iostream>
#include "receiverthread.h"

ReceiverThread::ReceiverThread(QextSerialPort &inPort)
:port(0)
{
  port = inPort;
}
```

```
void ReceiverThread::run()
{
  int numBytes = 0;
  int l = 512;
  while (1){
    numBytes = port.bytesAvailable();
    if(numBytes > l) {
      if(numBytes > l) numBytes = l;
      char buff[l];
      qint64 lineLength = port.readLine(buff, sizeof(buff));
      std::cout<<buff<<std::endl; //affichage des donnees
    }
  }
  exec();
}
```

Dans le code de l'application, une fois le port correctement ouvert, il suffit de définir le *thread* et de le démarrer :

```
#include "receiverthread.h"
[...]
ReceiverThread *receiver = new ReceiverThread(port);
receiver->start();
```

4.6 Application à la communication avec un microcontrôleur

Les systèmes embarqués qui vont nous intéresser désormais doivent être des dispositifs de faible consommation, fonctionnant plusieurs années sur piles, notamment à de fins d'observation de l'environnement. Outre le choix d'un microcontrôleur dédié à ce genre de tâches (MSP430 [4]), la stratégie de base pour augmenter l'autonomie reste toujours de réduire au maximum les ressources afin de ne mettre sous tension que les périphériques utiles.

Par ailleurs, l'utilisateur de tels dispositifs requiert généralement une information datée, soit avec un échantillonage à intervalle de temps régulier, ou une mise en marche à une heure donnée chaque jour. Notre stratégie est donc d'implémenter une horloge temps-réel logicielle dans le MSP430, qui n'est donc réveillé de son sommeil profond (mode LPM3) qu'une fois par seconde.

Le bus USB amène une alimentation de 5 V : en observant l'état de la broche associée sur le connecteur USB du circuit embarqué, nous serons en mesure de savoir si un câble de communication est connecté ou non, et agir en conséquent. Afin de protéger la logique 3,3 V, nous connectons la broche d'alimentation 5 V (broche 1 de l'embase, ou fil rouge dans un câble USB) à une broche de port général (GPIO – dans les exemples qui vont suivre, P3.4) du microcontrôleur *via* une résistance de quelques dizaines de kiloohms. Afin de définir l'état de la broche en l'absence de câble, une résistance de tirage un peu plus élevée est placée entre cette même broche et la masse. Si le MSP430 se rend compte qu'un câble USB ou série est connecté, il tentera de communiquer (recevoir des ordres du PC et retransmettre les informations acquises), sinon il se contente de vaquer à ses tâches (vérifier si une condition sur l'horloge est vérifiée afin de lancer une acquisition de données ou de réveiller un autre périphérique plus gourmand en énergie dont le microcontrôleur autorise l'alimentation) avant de se rendormir (Fig. 6).

4.7 Lecture de températures sur MSP430

```
...
#define         tmp     R9
#define         tmp2    R8
#define         tmp3    R7
...
  ; ADC
  mov.w   #SHT0_6+REFON+ADC12ON,&ADC12CTL0        ; V_REF=1.5 V
  mov.w   #SHP,&ADC12CTL1
  mov.b   #INCH_10+SREF_1,&ADC12MCTL0 ; p.17-11: single conversion
  bis.w   #ENC,&ADC12CTL0         ; MUST be .w jmfriedt
...

temperature:
        push    tmp
        push    tmp2
        push    tmp3

        mov.w   #0,tmp2
        mov.b   #12,tmp3    ; moyenne sur 12 mesures : 12->16 bits
somme:  call    #SetupADC12
        add.w   tmp,tmp2
        dec.b   tmp3
        jnz     somme

        mov     tmp2,tmp  ; envoi de la somme sur 16 bits
        swpb    tmp
        call    #byte_asc
        swpb    tmp
        call    #byte_asc
        mov.b   #' ',tmp
        call    #rs_tx1

        rlc.w   tmp2          ; on fait l'hypothese que bit 15 est 1
        swpb    tmp2          ; => val = (val+256)*1.5/512 sur 9 bits

        mov.b   tmp2,tmp
        call    #byte_asc
        mov.b   #'\n',tmp
        call    #rs_tx1
        mov.b   #'\r',tmp
        call    #rs_tx1
fintemp:
        pop     tmp3
        pop     tmp2
        pop     tmp
        ret

...
```

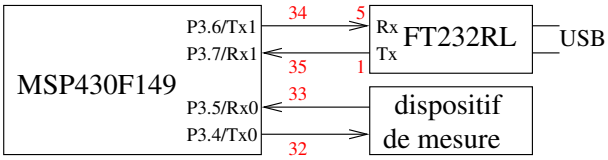
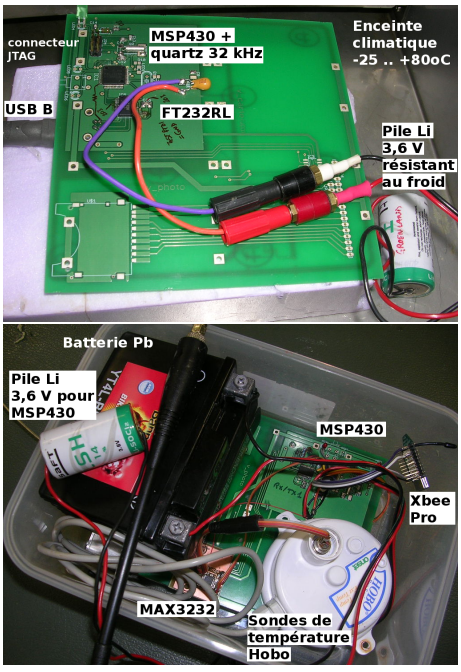


FIG. 6 – Circuit de base utilisé lors des mesures présentées par la suite : un microcontrôleur MSP430, nécessitant comme seul composant additionel un quartz à 32 kHz, communique avec un PC par liaison USB via un convertisseur RS232-USB FT232RL. Le second port série du MSP430 est éventuellement connecté à un autre instrument de mesure pour acquérir des informations additionelles à la température du microcontrôleur obtenue par mesure de la dérive en tension de la sonde interne. Le FT232RL peut être remplacé par un module XbeePro (http://www.lextronic.fr/produit.php?id=1319) si une liaison sans fil est nécessaire. Les chiffres en rouge indiquent les numéros des broches utilisées.

```
SetupADC12: ; book C. Nagy p.90   Vtemp=0.00355(T_C)+0.986
            bis.w   #ADC12SC,&ADC12CTL0             ; start conversion
adc:        bit.b   #1,&ADC12CTL1   ; wait while ADC busy==1
            jnz     adc
            mov.w   &ADC12MEM0,tmp  ; conversion result
            ret
```

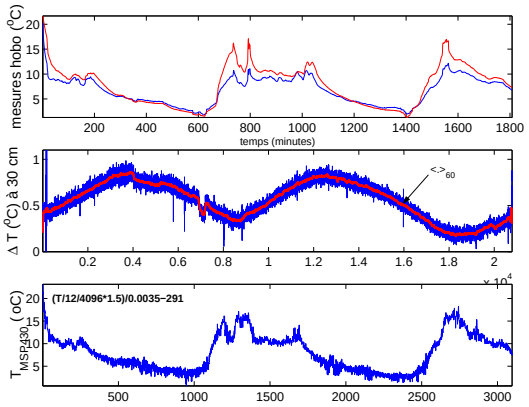
À titre d'exemple d'instrument que nous avons interrogé périodiquement par le port série afin d'en communiquer par liaison sans fil le résultat de mesures, mentionnons la station météorologique Hobo (http://www.onsetcomp.com/solutions/data_logger_kits) communément utilisée lors des mesures de température de l'environnement. Ce dispositif est muni d'un port série (RS232) pour la communication, qu'il s'agisse de la configuration des intervalles de temps entre deux acquisitions ou de la récupération des données. À l'allumage, ce dispositif communique au débit de 1200 bauds. Bien que le logiciel propriétaire fourni avec ces instruments passe rapidement à une communication haut-débit à 38400 bauds, nous avons choisi de maintenir toutes nos transactions au débit initial, compatible avec les performances en mode faible consommation (seul un résonateur basse tension à 32768 Hz alimente le microcontrôleur) du MSP430. La transaction visant à récupérer les deux mesures de température des sondes internes et externes du Hobo se font comme suit :

1. initialiser la transaction par les commandes 'D' suivi de 'E'
2. pour chaque nouvelle mesure, émettre l'ordre 'C' et attendre la réception de 6 caractères. Ces 6 caractères contiennent, dans le modèle de Hobo à notre disposition, deux mesures de températures codées sur 16 bits chacune
3. cette quantité lue est transmise au PC qui se charge d'appliquer les coefficients de calibration obtenus au préalable de la mise en place de l'expérience.



En haut, le montage en enceinte climatique lors de la phase de calibration de la sonde interne de température au MSP430. La même mesure a été faite avec un Hobo connecté au port série du MSP430. En bas, le montage placé en extérieur pour la mesure pendant 1 semaine, en totale autonomie, de la température : chaque minute (déterminé par une horloge temps réelle programmée dans le MSP430), le système se réveille, interroge le Hobo et transmet par liaison Zigbee les températures lues à un PC chargé de la réception, l'interprétation et l'affichage des données tel que nous le décrivons dans ce document. Ce montage a par ailleurs la particularité de récupérer par RS232 les informations d'un montage capable d'interroger un capteur enterré sous quelques dizaines de cm sous terre, alimenté par une batterie au plomb.

Noter le remplacement de la liaison USB utilisée en enceinte climatique par un module Xbee Pro acheté chez Lextronic http://www.lextronic.fr/produit.php?id=1319 : en plus de connecter directement les broches d'émission et de réception du MSP430 sur les broches 3 et 2 respectivement du Xbee Pro, nous connectons une broche de contrôle de la mise sous tension. La consommation de ce module est en effet très élevée (plus de 50 mA en mode écoute et plus de 100 mA en mode émission – 1000 fois plus important que la consommation du MSP430 seul) : afin de conserver une autonomie acceptable du montage, nous exploitons la broche de mise en veille SLEEP# (13) du Xbee Pro afin de ne l'alimenter que lors de la communication (une fraction de seconde chaque minute). Il s'agit là d'une particularité du protocole Zigbee qui le distingue de son concurrent le plus proche, Bluetooth : là où le second protocole met plusieurs secondes à initialiser la communication, avec une consommation électrique associée rédibitoire pour un fonctionnement sur pile avec une bonne autonomie, Zigbee ne met que quelques millisecondes à établir la communication lors de son réveil. Nous n'exploitons ici le Zigbee que pour une liaison point à point, sans utiliser sa capacité à créer un réseau en étoile.



En haut, mesure toutes les minutes de la température lue par les deux sondes du Hobo : ces deux valeurs sur 16 bits sont converties en degrés celsius suite à une étape de calibration. Milieu : mesure de la température à 30 cm sous terre aux mêmes instants. En bas : transmission de la mesure issue de la sonde de température interne au MSP430F149, translatée afin de correspondre au mieux aux valeurs du Hobo. Ces mesures ont été acquises et transmises par zigbee pendant 2 jours (vieillissement accéléré).

4.8 Réception des ordres sur MSP430

Afin de détecter la présence d'un câble USB et ne gérer la communication que dans ce cas, nous connectons la broche alimentée par la tension +5 V sur le câble USB (broche 1) à la broche P3.4 du MSP430 *via* une résistance quelques kiloohms. La valeur de cette broche est testée à chaque réveil de l'horloge temps réel logicielle programmée dans le MSP430 : la latence entre la connexion d'un câble USB et sa détection est au maximum de 1 s. Lorsque le câble est détecté, les broches P3.6 et P3.7 (puisque nous utilisons le second port série) passent de l'état GPIO en entrée vers le mode communication UART. Une fois le câble retiré, ces broches repassent en mode GPIO en entrée. En effet, le FT232RL n'étant pas alimenté en l'absence de câble USB (Fig. 11 du paragraphe 7.1 de [5] : *Bus powered configuration*), nous avons constaté un courant de fuite de plusieurs mA entre le MSP430 et le FT232RL en l'absence de ces précautions. Il est probable que la broche d'émission (Tx) du MSP430 essaie d'alimenter le FT232RL en l'absence de câble série, alimentation rendue impossible si la broche est en mode "entrée".

La fonction `comm`, présentée ci-dessous, est appelée chaque seconde lors de l'activation de l'horloge temps réel par l'interruption timer et implémente les fonctions que nous venons de décrire : scruter le port auquel est connectée la lignee d'alimentation du bus USB (dans notre cas, la broche P3.4), et si la liaison USB est détectée (niveau haut sur le port), alors le microcontrôleur passe les broches associées à la communication en mode UART. En présence du câble USB, nous passons dans la fonction `recoit` qui teste les caractères reçus : chaque caractère est associé à diverses fonctions dans le code (caractères '?', 'P' ...). Notez qu'en l'absence de communication, le microcontrôleur ne bloque pas en attente d'un ordre si le câble est débranché car la fonction `rs_rx` incorpore elle-même le test de présence du câble en scrutant la valeur de P3.4. La sortie de cette fonction est donc conditionnée soit par la réception d'un caractère sur le port série, soit par la déconnexion du câble.

```
...
        mov.b #0x40,&P3DIR ; P3.6 = output direction
        mov.b #0x00,&P3OUT ; P3.6 = output val
...

comm:           ; communication par port serie ?
        bit.b #0x10,&P3IN  ; teste alim du port serie = P3.4
        mov.b #0x00,&P3SEL ; P3.6,7 = GPIO select
        jnz   recoit
        ret

recoit:
        mov.b #0xC0,&P3SEL ; P3.6,7 = USART select
        call  #rs_rx
        cmp.b #'?',tmp
        jne   s1
        call  #dumpall  ; renvoie les parametres
        jmp   comm
s1:     cmp.b #'P',tmp  ; programmation parametres
        jne   s2
        call  #getall
        jmp   comm
```

```
s2:     cmp.b #'S',tmp        ; on ne fait rien !
        jne   s3
s3:     cmp.b #'T',tmp
        jne   s4
        call  #rendtemp
s4:     cmp.b #'U',tmp
        jne   s5
        call  #testxmit
s5:     jmp   comm            ; on n'a pas vu de char connu

rs_rx:
        bit.b   #URXIFG1,&IFG2  ; lecture port1
        jnz     fin_rsrx        ; char recu
        bit     #0x10,&P3IN; teste alim du port serie = P3.4
        jnz     rs_rx
        mov     #0x20,tmp
        ret                     ; si pas recu de char et pas de cable rs232
fin_rsrx:
        ; mov.b &RXBUF0,tmp     ; lecture port0
        mov.b   &RXBUF1,tmp     ; lecture port1
        ret
```

5 Utilisation des classes Qt en mode console

Nous avons exploité Qt pour développer un logiciel d'acquisition et de traitement des trames NMEA issues d'un récepteur GPS (Fig. 5). Or, il est fastidieux de graphiquement sélectionner chaque fichier à traiter quand de nombreuses traces GPS ont déjà été acquises : la ligne de commande présente là tout son intérêt dans sa capacité d'appliquer un filtre sur un grand nombre de fichiers dont le nom est fourni en argument (`for i in *; do ./mon_filtre.qt $i > $i.out ;done`).

Dans le cadre du développement d'interfaces graphiques avec les librairies Qt, nous développons des classes plus ou moins complexes pour des conversions de données ou de validation de codes correcteurs. Ces classes C++/Qt peuvent également être utilisées pour une application en ligne de commande : de cette manière nous bénéficions des même performances en ligne de commande et dans notre interface graphique. Par exemple dans le gestionnaire de trames GPS, il faut clicker sur 'Open a nmea file' puis sur 'convert2kml' et enfin choisir l'emplacement et le nom de son fichier destination pour convertir ses trames. Cette action utilise deux classes distinctes, l'une pour vérifier l'intégrité des données et l'autre pour filtrer et convertir les données. En ligne de commande, le

programme utilise les mêmes classes, mais il accepte en argument (`argv[1]`) un nom de fichier, et lance la conversion automatiquement en envoyant le résultat sur la sortie standard.

6 Affichage graphique de données acquises

Afin de compléter l'interface utilisateur graphique permettant de contrôler et recevoir des informations d'un système embarqué, nous désirons afficher en temps réel des courbes présentant l'évolution temporelle des quantités mesurées. Pour ce faire, nous exploitons la librairie `qwt` disponible à http://qwt.sourceforge.net/, développée pour simplifier l'inclusion de graphiques dans les applications Qt. Les exemples qui suivent sont basés sur la version 5.0(.2) de Qwt, compilés avec Qt4.3.4.

Un graphique est défini dans une classe nommée `FreqPlot`, dans laquelle sont définies toutes les propriétés et l'apparence du schéma (échelle, titre, axes, fréquence de raffraîchissement de l'affichage ...). Cette classe fera appel à une variable statique contenant les données lues sur le port série par le thread afin de complèter la courbe.

```
/* dans le programme */
#include "freqplot.h"
[...]
/* Initialisation d'une nouvelle courbe */
FreqPlot *plot = new FreqPlot();
plot->setTitle("Temperature");
```

Cet objet fait appelle à la classe `FreqPlot` qui exploite les objets et les structures fournies par la librairie Qwt :

```
/** classe FreqPlot **/
FreqPlot::FreqPlot(int sensor, int type):
[...]
/* ajout d'une l\'egende: */
    QwtLegend *legend = new QwtLegend;
    legend->setItemMode(QwtLegend::CheckableItem);
    insertLegend(legend, QwtPlot::RightLegend);

/* ajout d'un quadrillage */
QwtPlotGrid *grid = new QwtPlotGrid();
grid->enableX(true);
    grid->enableY(true);
grid->setXDiv(*(this->axisScaleDiv(QwtPlot::xBottom)));
    grid->setYDiv(*(this->axisScaleDiv(QwtPlot::yLeft)));
    grid->attach(this);

/* on n'a pas ajoute le zoom */
[...]

/* borne min et max en abscisse et ordonnees */
#define HISTORY 600 //seconds
setAxisScale(QwtPlot::yLeft,-50,200);
setAxisScaleDraw(QwtPlot::xBottom, new TimeScaleDraw(My_Time()));
    setAxisScale(QwtPlot::xBottom, 0, HISTORY);

/* initialisation de la courbe (des donnees)*/
    FreqCurve *curve;
    curve = new FreqCurve("Temperature");
    curve->setColor(Qt::blue);
    curve->setStyle(QwtPlotCurve::Lines);
    curve->attach(this);
showCurve(curve, true);

/* initialisation du timer qui rafraichira le graphique */
    refreshTimer = new QTimer();
    connect(refreshTimer, SIGNAL(timeout()), SLOT(timerEvent()));
    refreshTimer->start(1000); // 1/s
}

/* SLOT connect\e au Timer */
void FreqPlot::timerEvent()
{
//remplissage de curve avec les nouvelles donnees
[...]
replot();
}
```

Afin de nous affranchir des problèmes d'endianness, nous avons choisi de transférer la valeur 16 bits en ASCII. La lecture du côté Qt se fait en découpant la chaîne de caractères acquise par le port série :

```
bool ok = true;
double celcius, d;
char buff[20];
QStringList l;
qint64 lineLength = port.readLine(buff, sizeof(buff));
QString paquet = buff;
l = paquet.split(' ');
d = l[0].toInt(&ok,16);
if ( ok )
  {celcius = (((float) d/12 )/4096 * 1.5 - 0.986)/0.00355;}
```

La valeur convertie en température est alors empilée dans la liste `QList<double> temperature` de points à afficher :

```
temperature<<celcius;
```

Le transfert de l'octet de poids le plus fort en premier, afin de respecter la convention de lecture classiquement utilisée en occident de gauche à droite, est sélectionné du côté du MSP430 :

```
        swpb    R9
        call    #byte_asc
        swpb    R9
        call    #byte_asc
        mov.b   #' ',tmp
        call    #rs_tx

byte_asc:           ; parametre d'entree : R9
        push.w  R9
        push.w  R9
        rrc     R9          ; quartet fort
        rrc     R9
        rrc     R9p
        rrc     R9
        and.w   #0x0f,R9
        add.w   #0x30,R9
        cmp.w   #0x3a,R9
        jlo     byte1
        add.w   #7,R9
byte1:  call    #rs_tx
        pop.w   R9          ; quartet faible
        and.w   #0x0f,R9
        add.w   #0x30,R9
        cmp.w   #0x3a,R9
        jlo     byte2
        add.w   #7,R9
byte2:  call    #rs_tx
        pop.w   R9
        ret

rs_tx:  bit.b   #UTXIFG1,&IFG2
        jz      rs_tx
        mov.b   R9,&TXBUF1
        ret
```

`R9` est un registre sur 16 bits, tandis que la fonction `byte_asc` transfère uniquement l'octet de poids faible du registre qui sert au passage de paramètre. Le fait de placer un `swpb` *avant* le premier appel à `byte_asc` permet de transférer l'octet de poids fort en premier.

Du côté de la réception, les chaînes de caractères contenant les valeurs en hexadécimal, séparées d'espaces, sont découpées par la méthode `split(' ');` des chaînes de caractères Qt.

En résumé, nous assemblons toutes ces fonctions dans le code final fonctionnel suivant, donc le résultat est présenté sur la Fig. 7.

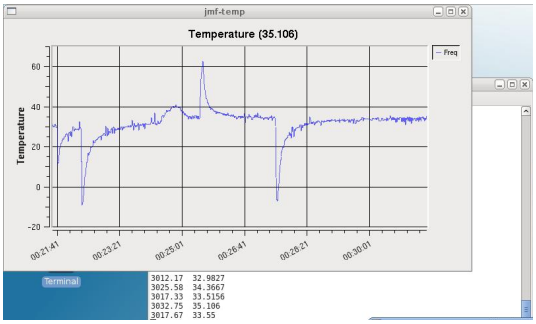


FIG. 7 – Affichage de la température (GNU/Linux et X11) issue du MSP430, transférée sous forme de 2 valeurs ASCII (avant et après moyenne des 12 acquisitions). Le capteur a été artificiellement refroidi deux fois et réchauffé une fois.

7 Mise en œuvre sur un système embarqué : la carte Armadeus

Un atout de Qt apparu en cours de développement est la disponibilité d'une version ne nécessitant pas de serveur graphique (X11) et moins de ressources, et donc compatible avec des systèmes embarqués disposant d'un afficheur. Embedded Qt – connu aussi sous son ancien nom QTopia – fonctionne sur la couche framebuffer ou sur svgalib. La stratégie de cross-compilation d'une application à destination de ARM (le processeur disponible sur la carte Armadeus) est décrit succintement à http://www.armadeus.com/wiki/index.php?title=Qt/Embedded. Chez TrollTech, http://doc.trolltech.com/qtopia2.1/html/environment-setup-build.html propose une autre description de la procédure de cross-compilation. Nous n'avons pas nous mêmes mis en place cet environnement de compilation : afin de démontrer la portabilité sur système embarqué de code développé sur PC (GNU/Linux, Intel), nous nous sommes contentés d'envoyer les codes sources de l'application présentée plus haut à F. Burkhart (Armadeus Systems) qui a pu les compiler en l'état et nous communiquer les résultats d'exécution de l'application (Fig. 8). Pour cet exemple, la broche d'émission des données Tx du MSP430 est directement connectée à ttySMX1 (le second port série) de l'APF9328 pour fournir la température lue sur la sonde interne au microcontrôleur.

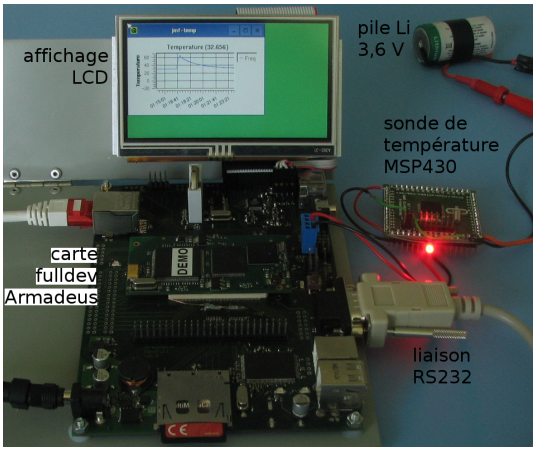


FIG. 8 – Photographie de la carte de développement DevFull de Armadeus, équipée d'un circuit APF9128 sur lequel tourne une application développée sous Qt et compilée ici par Embedded Qt (QTopia) pour fonctionner sur le framebuffer. Il s'agit de la même application de mesure de température que présentée précédemment (section 6) (photographie fournie par F. Burkhart, Armadeus Systems)

.

Nous constatons que l'application se comporte de façon strictement identique sur la carte Armadeus que sur PC fonctionnant sous GNU/Linux ou Windows. La seule modification nécessaire est l'ajout pour cette configuration du nom du port de communication, défini dans la série des `#ifdef _TTY_WIN_` de `qextserialbase.cpp` et `posix_qextserialport.cpp` pour fournir un nom approprié :

```
#elif _TTY_ARMADEUS_
        portSerie="/dev/ttySMX1";
...
```

(ce port est en logique 3,3 V sur la DevLight et en ±12 V sur la DevFull). Par ailleurs, le fichier de configuration `.pro` est complété pour y définir les variables associées à l'environnement de l'Armadeus :

```
contains( CONFIG , armadeus) {
        message("armadeus")
        TARGET = ./bin/$(OUTPUT_DIR)/temp
        HEADERS += src/Posix_QextSerialPort.h
        SOURCES += src/Posix_QextSerialPort.cpp
        DEFINES += _TTY_POSIX_
        DEFINES += _TTY_ARMADEUS_
        LIBS += -lm Lib/$(OUTPUT_DIR)/libqwt.a
        OBJECTS_DIR = bin/$(OUTPUT_DIR)/objs/
        MOC_DIR = bin/$(OUTPUT_DIR)/mocs/
        UI_DIR= bin/$(OUTPUT_DIR)/uis/
}
```

Nous aurons pris soin pour la cross compilation d'installer la toolchain appropriée et d'y faire appel :`QMAKE_CC= arm-linux-gcc`, `QMAKE_CXX= arm-linux-g++` ... et de faire appel aux librairies pour applications embarquées `-lqte -lqte-mt -lqpe -lqtopia`.

8 Conclusion

Nous avons présenté un environnement de développement basé sur Qt4 pour réaliser des interfaces graphiques capables de communiquer avec des systèmes embarqués. Nous avons démontré la capacité de cet environnement à générer des binaires pour un système d'exploitation propriétaire (Windows) depuis GNU/Linux afin de fournir une interface commune pour ces deux plateformes. Nous avons focalisé notre travail sur la communication par protocole asynchrone (RS232) avec des systèmes embarqués faible consommation, éventuellement *via* le convertisseur FTDI FT232 lorsqu'un port série RS232 n'est pas disponible sur le PC. Les interfaces graphiques proposées incluent l'interaction avec l'utilisateur (envoi de commandes et lecture de paramètres) et l'affichage de courbes acquises par le système embarqué. La portabilité de la solution proposée ne se limite pas aux interfaces graphiques puisque les librairies de traitement des données ont aussi été exploitées en ligne de commande afin d'automatiser le traitement d'un grand nombre de fichiers déjà disponibles. Enfin, nous avons étendu ces fonctionnalités à l'affichage sur une carte de développement aux ressources modestes comparées à un PC (carte Armadeus), sans serveur X11, afin d'illustrer l'utilisation de nos codes avec Embedded Qt.

Références

[1] N. Stephenson, *In the beginnning ... was the command line*, Harper Perennial (1999)

[2] J.-M Friedt, É. Carry, *Acquisition et dissémination de trames GPS à des fins de cartographie libre*, GNU/Linux Magazine France, Hors Série **27** (Octobre 2006)

[3] J.-M. Friedt, *Géolocalistion de photographies numériques*, GNU/Linux Magazine France **96**, Juillet/Aout 2007

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

[5] http://www.ftdichip.com/Products/FT232R.htm et la datasheet http://www.ftdichip.com/Documents/DataSheets/DS_FT232R.pdf

[6] Y. Bailly, *À la découverte de Qt*, GNU/Linux Magazine France, (Février 2002) 56-58, suivie d'une série de 20 articles jusqu'à Février 2004.

[7] D. Bodor, *Création d'un périphérique USB avec supports GNU/Linux et Windows*, GNU/Linux Magazine France **100**, (Déc. 2007), 56-63

[8] http://doc.qtfr.org/post/2007/04/10/Cross-Compilation-Native-dapplication-Qt-depuis-Linux

[9] http://doc.qtfr.org/post/2007/04/10/Cross-Compilation-Native-dapplication-Qt-depuis-Linux

J.-M Friedt est ingénieur dans la société Senseor, hébergé par l'institut FEMTO-ST de Besançon, et membre de l'association Projet Aurore pour la diffusion de la culture scientifique et technique (projetaurore.assos.univ-fcomte.fr/). Il n'aime pas les interfaces graphiques.

Julien Garcia est technicien, développeur dans le département Temps-Fréquence de l'institut FEMTO-ST de Besançon. Il est membre de l'association Sequanux (www.sequanux.org) pour la diffusion des logiciels libres en Franche Comté.