Mise en oeuvre d'un port série sous Qt

Thierry Vaira <tvaira@free.fr>

Table des matières

Site : tvaira.free.fr

Mise en oeuvre d'un port série sous Qt

Présentation

Qt est une **bibliothèque logicielle orientée objet** développée en **C++** par Qt Development Frameworks, filiale de Digia.

Qt est une plateforme de développement d'interfaces graphiques (GUI - *Graphical User Interface*) fournie à l'origine par la société norvégienne Troll Tech, rachetée par Nokia en février 2008 puis cédée intégralement en 2012 à Digia (www.qt.io).

Qt permet la portabilité des applications (qui n'utilisent que ses composants) par simple recompilation du code source. Les environnements supportés sont les Unix (dont Linux) qui utilisent le système graphique X Window System, Windows et Mac OS X.

Qt est principalement dédiée au développement d'interfaces graphiques en fournissant des éléments prédéfinis appelés widgets (pour windows gadgets) qui peuvent être utilisés pour créer ses propres fenêtres et des boîtes de dialogue complètement

prédéfinies (ouverture / enregistrement de fichiers, progression d'opération, etc). Qt dispose d'un moteur de rendu graphique 2D performant.

Qt fournit également un ensemble de classes décrivant des éléments non graphiques : accès aux données, connexions réseaux (*socket*), gestion des fils d'exécution (*thread*), analyse XML, etc.

La dernière version est Qt 5.0 qui est sorti le 19 décembre 2012. Bien que marquant des changements majeurs sur bien des points, le passage de Qt4 à Qt5 casse au minimum la compatibilité au niveau des sources.

Version de Qt

Vous pouvez vérifier la version installée sur votre poste :

```
$ dpkg-query -l | grep qt[0-9]
...
ii  libqt4-xmlpatterns:i386 4:4.8.1-0ubuntu4.9 Qt 4 XML patterns module
ii  qt4-demos               4:4.8.1-0ubuntu4.9 Qt 4 examples and demos
ii  qt4-designer            4:4.8.1-0ubuntu4.9 graphical designer for Qt 4 applications
ii  qt4-dev-tools           4:4.8.1-0ubuntu4.9 Qt 4 development tools
ii  qt4-doc                 4:4.8.1-0ubuntu4.9 Qt 4 API documentation
ii  qt4-linguist-tools      4:4.8.1-0ubuntu4.9 Qt 4 Linguist tools
ii  qt4-qmake               4:4.8.1-0ubuntu4.9 Qt 4 qmake Makefile generator tool
...
```

On utilisera donc **Qt4** !

Qt4 sépare sa bibliothèque en modules :

- QtCore : pour les fonctionnalités non graphiques utilisées par les autres modules ;
- QtGui : pour les composants graphiques, maintenant QtWidgets (qt5) ;
- QtNetwork : pour la programmation réseau ;
- etc ...

Mais ici pour une prise en main minimale de Qt, **nous n'utiliserons pas de modules spécifiques de Qt**.

Environnement de Développement Intégré (EDI)

Qt Creator est l'environnement de développement intégré dédié à Qt et facilite la gestion d'un projet Qt. Son éditeur de texte offre les principales fonctions que sont la coloration syntaxique, le complètement, l'indentation, etc... Qt Creator intègre en son sein les outils Qt Designer et Qt Assistant. Il intègre aussi un mode débuggage.

Remarque : même si Qt Creator est présenté comme l'environnement de développement de référence pour Qt, il existe des modules Qt pour les environnements de développement Eclipse et Visual Studio. Il existe d'autres EDI dédiés à Qt et développés indépendamment de Nokia, comme QDevelop et Monkey Studio.

Mais ici pour une prise en main minimale de Qt, **nous n'utiliserons pas Qt Creator**.

Structure générale des classes Qt

L'API Qt est constituée de classes aux noms préfixés par `Q` et dont chaque mot commence par une majuscule (`QString`, `QObject`, ...). Il faut savoir que la classe `QObject` est la classe mère de toutes les classes Qt.

Par exemple, la classe `QString` est une classe Qt pour **manipuler les chaînes de caractères**. Pour pouvoir utiliser une classe Qt, il ne faudra oublier d'inclure son fichier de déclaration :

```
#include <QString>

...

QString myString;
```

Ici pour une prise en main minimale de Qt, **nous utiliserons seulement la classe `QString`**.

qmake

Qt se voulant un environnement de développement portable, il a été nécessaire de concevoir un moteur de production spécifique. C'est ainsi qu'est conçu le programme `qmake`.

Ce dernier prend en entrée un fichier (avec l'extension `.pro`) décrivant le projet (liste des fichiers sources, dépendances, paramètres passés au compilateur, etc...) et génère un fichier de projet spécifique à la plateforme. Ainsi, sous les systèmes UNIX/Linux, `qmake` produira un `Makefile`.

Le fichier de projet est fait pour être très facilement éditable par un développeur. Il consiste en une série d'affectations de variables.

La génération d'une application se fait en plusieurs étapes :

- création d'un répertoire et des sources (cela pourra dépendre de l'EDI utilisé) : le nom initial du répertoire détermine le nom du projet et donc de l'exécutable qui sera produit (par défaut)
- générer un fichier `.pro` qui décrit comment générer un `Makefile` pour compiler ce qui est présent dans le dossier courant : `$ qmake -project`
- générer un `Makefile` à partir des informations du fichier `.pro` : `$ qmake`
- fabriquer l'application en appelant l'outil make : `$ make`

Remarque : on exécute la commande `qmake` seulement si le fichier `.pro` est modifié

Démarrer un projet vierge

À partir de la ligne de commandes (CLI), voici la procédure pour créer un projet Qt "vierge" :

```
$ mkdir mo-qt-0

$ cd mo-qt-0/

$ vim main.cpp
int main()
{
    return 0;
}

$ qmake -project

$ qmake

$ make

$ ./mo-qt-0

$ cat mo-qt-0.pro
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
SOURCES += main.cpp
```

Le fichier de projet est fait pour être très facilement éditable par un développeur. Il consiste en une série d'affectations de variables :

```
$ vim mo-qt-0.pro
TEMPLATE = app
TARGET = monProgramme # le nom de l'exécutable
DEPENDPATH += .
INCLUDEPATH += .

# on peut choisir les chemins :
OBJECTS_DIR = ./tmp # pour les fichiers objets (.o, .obj)
MOC_DIR = ./tmp # pour les fichiers générés par moc
DESTDIR = ./bin # pour l'exécutable
```



```
# Input
SOURCES += main.cpp

$ qmake

$ make

$ ./bin/monProgramme

$ ls -l ./tmp/
-rw-rw-r-- 1 tv tv 1312 sept. 19 17:52 main.o

$ make clean
rm -f tmp/main.o
rm -f *~ core *.core

$ ls -l ./tmp/
total 0
```

Gérer les arguments d'un programme C/C++ avec Qt

Un paramètre (ou argument) est au sens large un élément d'information à prendre en compte pour prendre une décision ou pour effectuer un calcul.

Un paramètre est une donnée manipulée par une section de code (programme, shell script, sous-programme, fonction, méthode) et connue du code appelant cette section.

On distingue deux types de paramètres :

- les paramètres d'entrée : donnée fournie par le code appelant au code appelé
- les paramètres de sortie : donnée fournie par le code appelé au code appelant

En programmation C/C++, on parle de paramètre effectif. Il s'agit de la variable (ou valeur) fournie lors de l'appel du programme (ou d'un sous-programme). **Lorsqu'on saisit des arguments après le nom d'un programme, ceux-ci sont passés en paramètres de la fonction principal `main`.**

Le prototype de la fonction `main` d'un programme C/C++ est `int main(int argc, char *argv[]) { ... }` ce qui correspond à :

- Le paramètre d'entrée `argc` contient le nombre d'arguments (`int`)
- Le paramètre d'entrée `argv` contient l'ensemble des arguments sous la forme de chaînes de caractère : argv[0] contient le nom du programme, argv[1] contient le premier argument, etc ... (`char *argv[]` ou `char **argv`)
- Le paramètre de sortie contient le code de retour à la fin de l'exécution de la fonction `main` (`int`)

Remarque : pour les affichages dans la console, on utilisera exclusivement `qDebug()` (donc pas de `cout` ni de `printf` !).

Voici un exemple qui permet de comprendre la **gestion des arguments d'un programme C/C++** :

```
#include <QDebug>

int main(int argc, char *argv[])
{
   int i;

   qDebug("nb d'arguments = %d", argc);
   qDebug("nom du programme argv[0] : %s\n", argv[0]);

   for(i=0;i<argc;i++)
      qDebug("argv[%d] = %s", i, argv[i]);

   return 0;
}
```

On obtient :

```
$ ./bin/monProgramme hello world
nb d'arguments = 3
```



```
nom du programme argv[0] : ./bin/monProgramme

argv[0] = ./bin/mo-qt-1
argv[1] = hello
argv[2] = world
```

Il est possible de l'adapter à nos besoins : par exemple **récupérer le nom du fichier spécial d'accès à un périphérique USB**

```
#include <QDebug>
#include <QString>

#define PORT    "/dev/ttyACM0"

int main(int argc, char *argv[])
{
   QString nomPort(PORT); // port par défaut

   // un nom a été fourni ?
   if(argc > 1)
   {
       nomPort = QString(argv[1]); // port choisi au lancement du programme
   }

   qDebug() << "nom du port : " << nomPort;

   return 0;
}
```

Maintenant, on obtient :

```
$ ./bin/monProgramme
nom du port :  "/dev/ttyACM0"

$ ./bin/monProgramme /dev/ttyUSB0
nom du port :  "/dev/ttyUSB0"
```

Manipuler des chaînes de caractères avec Qt

La classe `QString` contient de nombreuses méthodes pour manipuler des chaînes de caractère. Il vous faudra consulter très souvent sa documentation : doc.qt.io/qt-4.8/qstring.html.

Remarque : la documentation de Qt est très riche et très bien faite !

Voici quelques exemples d'utilisation de la classe `QString` dans le cadre d'un **traitement d'une trame NMEA 0183** :

```
#include <QDebug>
#include <QString>

int main(int argc, char *argv[])
{
    QString phrase = "$GPGGA,064036.289,4836.5375,N,00740.9373,E,1,04,3.2,200.2,M,,,,0000*0E";
    // Faire des essais :
    //QString phrase = "";
    //QString phrase = "GPGGA,064036.289,4836.5375,N,00740.9373,E,1,04,3.2,200.2,M,,,,0000*0E";
    //QString phrase = "$GPAAM,A,A,0.10,N,WPTNME*32";
    //QString phrase = "$GPGGA,064036.289,4836.5375,N,00740.9373,E,1,04,3.2,200.2,M,,,,0000";

    QString checksum;
    const QString debutTrame = "$";
    const QString typeTrame = "GPGGA";
    const QString debutChecksum = "*";

    // phrase vide ?
    if(phrase.length() != 0)
    {
```



```
        // est-ce une phrase NMEA 0183 ?
        if(phrase.startsWith(debutTrame))
        {
            // est-ce la bonne phrase ?
            if(phrase.startsWith(debutTrame + typeTrame))
            {
                // y-a-t-il un checksum ?
                if(phrase.contains(debutChecksum))
                {
                    checksum = phrase.section(debutChecksum, 1, 1);
                    qDebug() << "checksum : 0x" << checksum;
                }
                else
                    qDebug() << "Attention : il n'y a pas de checksum dans cette phrase !";
            }
            else
                qDebug() << "Erreur : ce n'est pas une trame GGA !";
        }
        else
            qDebug() << "Erreur : ce n'est pas une trame NMEA 0183 !";
    }
    else
        qDebug() << "Erreur : phrase vide !";

    return 0;
}
```

L'autre utilisation fréquente de la classe `QString` est la **conversion des données numériques** (`int`, `double`, ...) :

```
#include <QDebug>
#include <QString>

int main(int argc, char *argv[])
{
    /* Exemple de base */
    int i = 2;
    double d = 3.14;

    // Du numérique -> chaîne de caractères
    QString entier = QString::number(i); // int -> QString
    QString reel = QString::number(d); // double -> QString

    qDebug() << "L'entier i : " << entier;
    qDebug() << "Le réel d : " << reel;

    // De chaîne de caractères -> numérique
    entier = "100";
    reel = "2.71828";
    i = entier.toInt(); // QString -> int
    d = reel.toDouble(); // QString -> double

    qDebug() << "L'entier i : " << i;
    qDebug() << "Le réel d : " << d;

    /* Exemple appliqué */
    QString phrase = "$GPGGA,064036.289,4836.5375,N,00740.9373,E,1,04,3.2,200.2,M,,,,0000*0E";
    QString horodatage;
    unsigned int heure, minute;
    double seconde;

    // découpe la trame avec le délimiteur ',' et récupère le deuxième champ
    horodatage = phrase.section(',', 1, 1);
    // découpe une chaine à partir d'une position et un nombre de caractères
    heure = horodatage.mid(0, 2).toInt();
    minute = horodatage.mid(2, 2).toInt();
    seconde = horodatage.mid(4, 2).toDouble();
```



```
    qDebug() << "Horodatage : " << horodatage;

    horodatage = QString::number(heure) + " h " + QString::number(minute)
                 + " " + QString::number(seconde) + " s";

    qDebug() << "Horodatage : " << horodatage;

    return 0;
}
```

Gestion du port série sous Qt

Malheureusement, la version 4 de Qt ne fournit pas de classes pour gérer un port série nativement. On va donc devoir utiliser une **bibliothèque logicielle** externe à Qt.

Remarque : le problème n'existe plus en Qt5 car on dispose alors de la classe `QSerialPort` !

On va utiliser la classe `QextSerialPort` disponible ici.

On crée un nouveau répertoire et on se déplace à l'intérieur :

```
$ mkdir mo-qt-1

$ cd mo-qt-1/
```

On télécharge l'archive :

```
$ wget http://qextserialport.googlecode.com/files/qextserialport-1.2rc.zip
```

On décompresse l'archive téléchargée :

```
$ unzip qextserialport-1.2rc.zip
```

On déplace les sources et on renomme le répertoire :

```
$ mv qextserialport-1.2rc/src ./qextserialport
```

On peut maintenant supprimer l'archive téléchargée :

```
$ rm -f qextserialport-1.2rc.zip
```

On crée un nouveau fichier de projet :

```
$ vim mo-qt-1.pro
include(qextserialport/qextserialport.pri)

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

OBJECTS_DIR = ./tmp
MOC_DIR = ./tmp
DESTDIR = ./bin

# Input
SOURCES += main.cpp
```

On crée maintenant un programme de **test du port série** :

```
$ vim main.cpp
```



```
#include "qextserialport.h"

#define PORT "/dev/ttyACM0"

// Mode débuggage
#define DEBUG

#ifdef DEBUG
#include <QDebug>
#endif

int main()
{
    QextSerialPort *port;

    // création du port en mode asynchrone -> QextSerialPort::Polling
    port = new QextSerialPort(QLatin1String(PORT), QextSerialPort::Polling);

    // TODO : paramètrer le port (débit, ...)

    // ouverture du port
    port->open(QIODevice::ReadWrite | QIODevice::Unbuffered);
    #ifdef DEBUG
    qDebug("<debug> etat ouverture port : %d", port->isOpen());
    #endif

    // TODO : réceptionner des données

    // fermeture du port
    port->close();
    #ifdef DEBUG
    qDebug("<debug> etat ouverture port : %d", port->isOpen());
    #endif

    delete port;

    return 0;
}
```

On fabrique et on exécute le programme de test du port série :

```
$ qmake

$ make

$ ./bin/mo-qt-1
<debug> etat ouverture port : 1
<debug> etat ouverture port : 0
```

Le paramétrage du port série se fait avec les méthodes suivantes :

- `setBaudRate()`
- `setParity()`
- `setDataBits()`
- `setStopBits()`
- ... voir la documentation

Et pour les opération de lecture et d'écriture, on utilisera les méthodes `read` et `write`.

Retour au sommaire

