Applications Qt Embarquées

Version 1.01 - Michel GRIMALDI

INTRODUCTION

Le framework Qt est très largement employé[1] dans les applications informatiques standard de type *Desktop* mais également dans les applications embarquées sur plate-forme mobiles *Android*, *IOS* ou autre[2].

Par rapport à d'autres frameworks, Qt présente l'avantage d'être *encore* libre et open source, de s'exécuter aussi bien sous Windows, Linux et MasOS, et surtout d'avoir des fonctionnalités de cross compilation directement intégrées à son outil de développement QtCreator.

Dans ce module, nous allons voir comment développer, compiler et déployer des applications Qt sur une plate-forme Raspberry PI.

Nous verrons ensuite comment intégrer un écran tactile pour obtenir un dispositif autonome, sans clavier ni souris, pour héberger une application dédiée.

PRÉPARATION DU RASPBERRY PI

Prérequis :

- avoir une image de linux installée sur la carte DS
- avoir une connexion réseau active (paramétrage du proxy éventuellement).

Dans un premier temps, nous allons installer la chaîne de compilation GNU (pas vraiment nécessaire dans l'absolu, mais on ne sait jamais) :

```
sudo apt-get install build-essential
```

puis les binaires de la bibliothèque Qt 4.

```
sudo apt-get install libqt4-dev
```

La version 5 de Qt n'est pas encore dans le dépôt à l'heure où j'écris ce post, mais si vous y tenez vraiment, vous pouvez la recompiler à partir des codes sources.

Il faut enfin activer le serveur *ssh* avec l'utilitaire **raspi-config**.

Voilà, s'en est fini avec le Raspberry !

1 Pour découvrir Qt et ses applications phares : http://qt.developpez.com/faq/?page=intro

2 Android, iOS, WinRT (incluant Windows Phone), Sailfish OS, Tizen.

PRÉPARATION DE LA MACHINE LOCALE :

Nous allons également devoir installer les composantes suivantes:

- la chaîne de compilation GNU
- une version de Qt
- une version de QtCreator (par exemple la dernière version open source à https://www.qt.io/download-open-source/)
- comme nous avons choisi de *cross compiler* des applications Qt4, il faut également installer les binaires de la bibliothèque Qt 4[3].

```
sudo apt-get install libqt4-dev
```

Pour la suite, tout se passera dans un répertoire de base, que j'ai nommé **raspberry** et que j'ai mis tout simplement sur mon bureau[4].

/home/grimaldi/Bureau/raspberry

De même, dans mon réseau local, le Raspberry PI se trouvait à l'adresse 192.168.0.16.

Montage du *file system*[5] du Raspberry PI dans le *file system* de la machine locale

Le fait de monter la racine du raspberry dans le *file system* local permettra de compiler et de déployer les applications sans avoir à faire des mouvements de carte SD.

Pour cela, il faut installer le paquet ***sshfs***, et autoriser votre utilisateur (moi c'est grimaldi) à l'utiliser en l'ajoutant au groupe ***fuse*** :

```
sudo apt-get install sshfs
sudo usermod -a -G fuse grimaldi
```

On peut maintenant monter le *file system* du Raspberry sur la machine locale (ici 192.168.0.16 avec le login et le mot de passe par défaut).
N'oubliez pas de créer préalablement le répertoire **rpi-mnt** dans votre répertoire de travail.
Chez moi : **/home/grimaldi/Bureau/raspberry/rpi-mnt**

```
mkdir rpi-mnt
```

```
sudo sshfs pi@192.168.0.16:/ ./rpi-mnt -o transform_symlinks -o
allow_other
```

Si tout s'est bien passé, vous pouvez voir le *file system* du raspberry dans le répertoire ***rpi-mnt*** :

```
ls
bin   dev  home  lost+found  mnt  proc  run   srv  tmp  var
boot  etc  lib   media       opt  root  sbin  sys  usr
```

3 Attention, il faut installer Qt4 sur les deux plate-formes : pour compiler et pour exécuter
4 Vous devrez bien entendu choisir un autre nom et un autre emplacement
5 Je voulais dire système de fichiers

création des liens vers le *file system* du raspberry :

```
sudo ln -s /home/grimaldi/Bureau/raspberry/rpi-mnt/usr/lib/arm-linux-gnueabihf/ /usr/lib/arm-linux-gnueabihf
sudo ln -s /home/grimaldi/Bureau/raspberry/rpi-mnt/lib/arm-linux-gnueabihf/ /lib/arm-linux-gnueabihf
```

Ceci, pour permettre à QtCreator et à la chaîne de compilation de trouver leurs bibliothèques[6] dans leur propre *file system*.

Vérification de la présence des liens :

```
ls -ld /usr/lib/arm-linux-gnueabihf
ls -ld /lib/arm-linux-gnueabihf
```

Installation de la chaîne de cross compilation *arm-bcm2708*

Toujours dans le même répertoire :

```
mkdir tools
cd tools
git clone https://github.com/raspberrypi/tools.git
```

à l'Université, derrière le proxy cela risque de ne pas marcher. Vous pouvez alors télécharger une archive que vous décompresserez dans le répertoire ***tools***
(chez moi : **`/home/grimaldi/Bureau/raspberry/tools)`** vous devez alors avoir quelque chose comme :

```
arm-bcm2708  configs  mkimage  pkg  sysidk  test_code  usbboot
```

et dans le sous répertoire *arm-bcm2708*, le cross-compilateur sous la forme de quatre répertoires :

```
arm-bcm2708hardfp-linux-gnueabi  gcc-linaro-arm-linux-gnueabihf-raspbian
arm-bcm2708-linux-gnueabi        gcc-linaro-arm-linux-gnueabihf-raspbian-x64
```

Nota : vous pourrez également récupérer une copie de la chaîne de compilation sur :

https://github.com/raspberrypi/tools/archive/master.zip

Vérification du bon fonctionnement de la chaîne compilation locale:

Avant d'aller plus loin, nous allons compiler le petit programme c++, « hello », suivant pour vérifier l'installation de la chaîne de compilation locale du *raspberry* et la chaîne de cross compilation de l'ordinateur:

```
#include <iostream>
int main(int argc, char **argv)
{
```

6 Bibliothèque = Library en anglais, les informaticiens disent souvent « les librairies »

```
        char c;
        std::cout<<"Hello World !"<<std::endl;
        std::cin>>c;
        return 0;
}
```

Après l'avoir sauvé sur le bureau du Raspberry (par exemple dans hello.cpp), utilisez le répertoire rpi-mnt précédent pour accéder au *file system* du raspberry ou le protocole *sftp* pour cela, dans le premier cas, le bureau se trouve alors à : `rpi-mnt/home/pi/Desktop,`

Ouvrir un terminal ssh sur le *raspberry*

```
ssh pi@192.168.0.16
```

Compiler localement (sur le *raspberry*) le programme hello.cpp

```
cd ~/Desktop
ls
g++ -Wall -o hello hello.cpp
```

si tout s'est bien passé, exécutez le programme :

```
./hello
Hello World !
```

Vérification du bon fonctionnement de la chaîne de cross compilation :

Nous allons maintenant vérifier le bon fonctionnement de la chaîne de cross compilation.

Revenez sur l'ordinateur puis avec *geany*, ouvrez le fichier ci-dessus (*click droit, ouvrir avec geany*). Modifiez les commandes de construction (menu *construire*, *définir les commandes de constructions*) en y mettant le chemin de votre *cross compilateur*, chez moi, dans la rubrique « *construire* » j'ai mis :

```
/home/grimaldi/Bureau/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-g++ -Wall -o "%e" "%f"
```

à la place de **g++ -Wall -o "%e" "%f"**

Dans mon cas, la chaîne de cross compilation a été installée à :
/home/grimaldi/Bureau/raspberry/tools

le chemin du cross compilateur c++ est :
/home/grimaldi/Bureau/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-g++

S'il n'y a pas eu d'erreur, il ne devrait pas y en avoir, après construction vous avez dû créer un fichier exécutable nommé **`hello`** que vous ne pourrez pas exécuter sur l'ordinateur mais que vous pouvez exécuter sur le *raspeberry* après l'y avoir transféré.

Vérifiez sa structure avec la commande :

```
file hello
```

vous devriez obtenir quelque chose comme :

```
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for
GNU/Linux 2.6.32,
BuildID[sha1]=323b4b055be6c8eb6f2267419b2bf9fea67f57d6, not
stripped
```

PARAMÉTRAGE DE QT CREATOR – MACHINE LOCALE

Création d'un nouveau kit de compilation:

par le biais du menu **Outils**, puis **Options...** et la rubrique **Compiler et Exécuter**

Ajouter un compilateur linux-icc :

Nom : **Raspberry**
chemin du compilateur : **/home/grimaldi/Bureau/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-g++**
Chemin de make : **/usr/bin/qmake-qt4**
ABI : **arm linux generic elf 32bit**

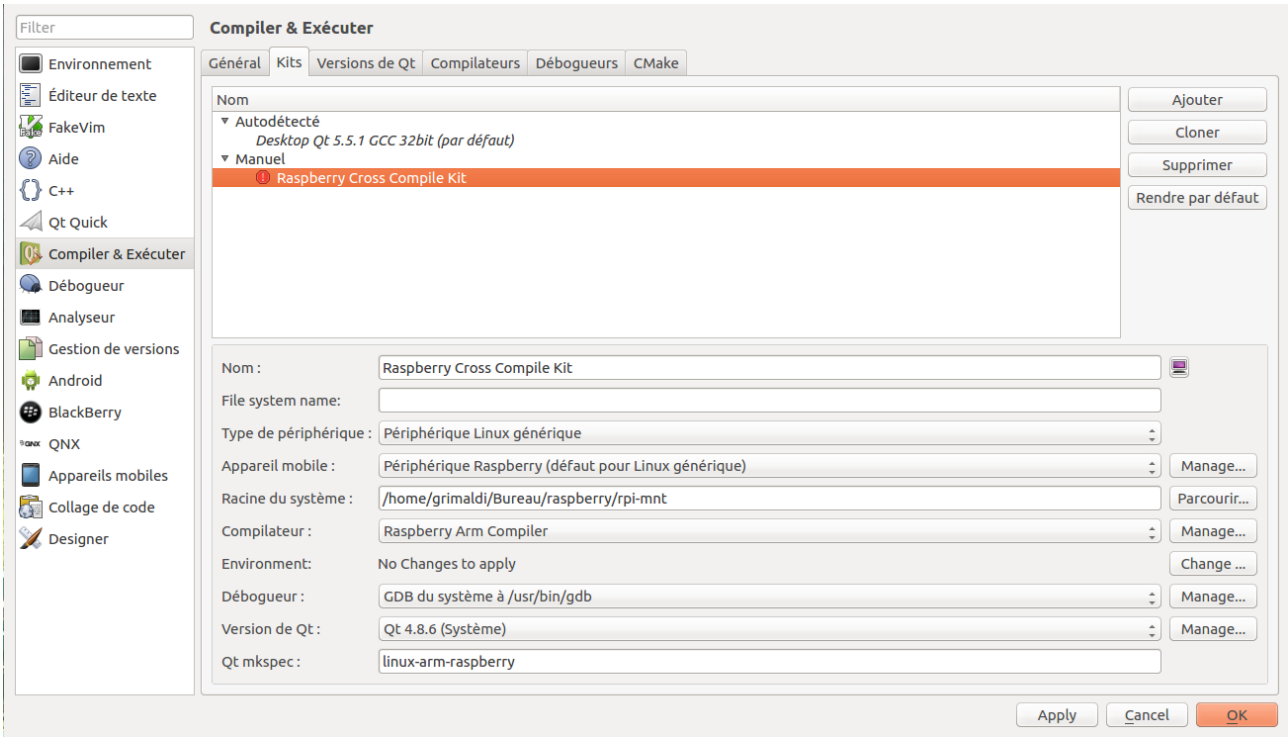
Ajouter une version de Qt :

Nom : **Qt 4 for raspberry**
Emplacement de qmake: **/usr/bin/qmake-qt4**

Ajouter un kit :

Nom : **Raspberry Cross Compile Kit**
File system name :
Type de périphérique : **périphérique Linux Générique**
Appareil mobile :
Racine du système : **/home/grimaldi/Bureau/raspberry/rpi-mnt**
Compilateur : **Raspberry**
Version de Qt : **Qt 4.8.6**

Qt mkspec : **linux-arm-raspberry**[7]



Création de l'environnement QMAKESPEC pour la plateforme linux-arm-raspberry

Pour comprendre la suite, il faut savoir qu'une application Qt est construite en deux phases :

1. avec l'outil **qmake** qui permet de simplifier le processus de construction pour le projet de développement à travers différentes plates-formes. ***qmake* génère un Makefile** sur la base des informations de votre fichier de projet. Les fichiers de projet sont créés par le développeur, et sont généralement simples, mais peuvent être plus sophistiqués pour des projets complexes.
2. Avec l'outil **make**[8] standard, qui construit automatiquement des fichiers, souvent exécutables, ou des bibliothèques à partir d'éléments de base tels que du code source. Il utilise des fichiers appelés *makefile* qui spécifient comment construire les fichiers cibles. À la différence d'un simple script *shell*, *make* exécute les commandes seulement si elles sont nécessaires. Le but est d'arriver à un résultat (logiciel compilé ou installé, documentation créée, etc.) sans nécessairement refaire toutes les étapes. *make* est particulièrement utilisé sur les plateformes UNIX.

qmake nécessite un fichier de description de la plate-forme et du compilateur, QMAKESPEC, qui contient les valeurs par défaut utilisées pour générer les *Makefile* appropriés. La distribution Qt standard contient beaucoup de ces fichiers, situés dans le sous-répertoire *mkspecs* de l'installation de Qt.

```
/usr/share/qt4/mkspecs/
```

Hélas, la description de la plate-forme *raspberry* n'en fait pas partie, nous devrons donc la créer

7 Continuer au paragraphe suivant pour créer cet environnement QMAKESPEC
8 Source https://fr.wikipedia.org/wiki/Make

nous même.

dans le répertoire : **/usr/share/qt4/mkspecs**

créer un sous répertoire **linux-arm-raspberry** en copiant un des répertoires existants, par exemple linux-arm-gnueabi-g++ :

```
sudo cp -r /usr/share/qt4/mkspecs/linux-arm-gnueabi-g++ /usr/share/qt4/mkspecs/linux-arm-raspberry
```

remplacer le contenu du fichier nommé *qmake.conf* par le texte suivant.

Attention, si vous copiez/collez directement le texte de ce document pdf, vous risquez d'obtenir des codes parasites, non visibles, qui empêcheront la compilation.
Vous pouvez trouver le fichier à : grimaldi.univ-tln.fr/files/raspberry-qmake.conf

Vous devrez également remplacer le chemin du répertoire de base de la chaîne de cross compilation par le votre :

```
LOCAL_CROSS_TOOL_CHAIN=/home/grimaldi/Bureau/raspberry
```

pour cela, vous devez avoir les droits administrateur :

```
sudo gedit qmake.conf
```

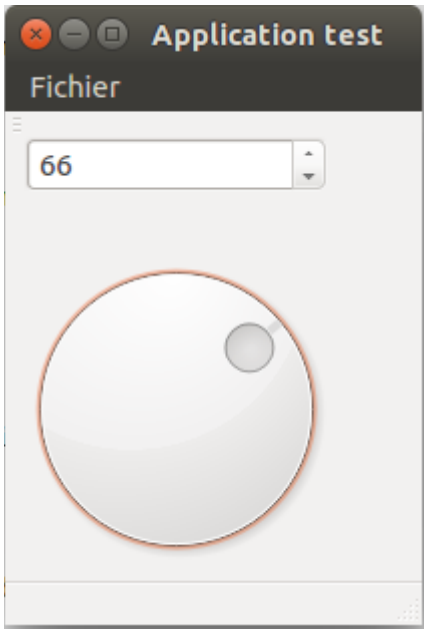
ce qui doit donner quelque chose comme :

```
#
# qmake configuration for raspberry linux-arm-g++ for raspberry pi
#
LOCAL_CROSS_TOOL_CHAIN        = /home/grimaldi/Bureau/raspberry
CROSS_COMPILER_PATH           = $$LOCAL_CROSS_TOOL_CHAIN/tools/arm-bcm2708/gcc-linaro-arm-linux-
gnueabihf-raspbian/bin
CROSS_COMPILE                 = $$CROSS_COMPILER_PATH/arm-linux-gnueabihf
#ROOTFS                       = $$LOCAL_CROSS_TOOL_CHAIN/rpi-localfs
ROOTFS                        = $$LOCAL_CROSS_TOOL_CHAIN/rpi-mnt
MAKEFILE_GENERATOR            = UNIX
TARGET_PLATFORM               = unix
TEMPLATE                      = app
CONFIG                        += qt warn_on release incremental link_prl gdb_dwarf_$
QT                            += core gui
QMAKE_INCREMENTAL_STYLE       = sublib

include(../common/linux.conf)
include(../common/gcc-base-unix.conf)
include(../common/g++-unix.conf)
QMAKE_CXX                     = $$CROSS_COMPILE-g++
QMAKE_LINK                    = $$CROSS_COMPILE-g++
QMAKE_LINK_SHLIB              = $$CROSS_COMPILE-g++
QMAKE_AR                      = $$CROSS_COMPILE-ar cr
QMAKE_OBJCOPY                 = $$CROSS_COMPILE-objcopy
QMAKE_STRIP                   = $$CROSS_COMPILE-strip
QMAKE_LFLAGS_RELEASE          = -Wl,-O1
QMAKE_RPATHDIR                += $$ROOTFS/lib/arm-linux-gnueabihf
QMAKE_RPATHDIR                += $$ROOTFS/usr/lib/arm-linux-gnueabihf
QMAKE_INCDIR                  = $$ROOTFS/usr/include
QMAKE_INCDIR                  = $$ROOTFS/usr/include/arm-linux-gnueabihf
QMAKE_INCDIR_QT               = $$ROOTFS/usr/include/qt4
QMAKE_RPATHDIR                = $$ROOTFS/lib/arm-linux-gnueabihf
QMAKE_RPATHDIR                += $$ROOTFS/usr/lib/arm-linux-gnueabihf
QMAKE_LIBDIR                  = $$ROOTFS/usr/lib
QMAKE_LIBDIR_QT               = $$ROOTFS/usr/lib/arm-linux-gnueabihf
QMAKE_LIBDIR_QT               += $$ROOTFS/lib/arm-linux-gnueabihf
```

```
QMAKE_INCDIR_X11              = $$ROOTFS/usr/include
QMAKE_LIBDIR_X11              = $$ROOTFS/usr/lib/arm-linux-gnueabihf
QMAKE_LIBDIR_X11              += $$ROOTFS/lib/arm-linux-gnueabihf
QMAKE_INCDIR_OPENGL           = $$ROOTFS/usr/include
QMAKE_LIBDIR_OPENGL           = $$ROOTFS/usr/lib

load(qt_config)
```

et un fichier nommé qplatformdefs.h que l'on laissera dans l'état.

Voilà, vous pouvez maintenant écrire une application graphique Qt supportant deux plate-formes : **Desktop Qtxx GCC32 bits** et **Raspberry Cross Compile Kit** dont l'interface sera constituée d'un *QDial* et d'un *QSpinBox*. Vous imaginez bien le fonctionnement de cette application de test : *quand on agit sur le QDial, sa position s'écrit dans le QSpinbox*.



1. construisez cette application et testez la sur machine locale (Desktop Qtxx GCC32 ou 64 bits)
2. changer de kit (Raspberry Cross Compile Kit) et compilez la
3. il ne reste plus qu'a envoyer l'exécutable sur le *raspberry* par *sftp* ou autre et l'exécuter.

> Une autre solution consiste à donner directement le chemin où l'application sera créée sur le *raspberry* dans les options du projet en cochant **Shadow build** et en choisissant la destination dans le *file system* de la carte, exemple :
>
> `/home/grimaldi/Bureau/raspberry/rpi-mnt/home/pi/Desktop/build-essai0`

ECRAN TACTILE ELEDUINO 7"

A partir de l'image du constructeur

Une première manière d'installer l'écran tactile *Eleduino* est donnée à :

http://grimaldi.univ-tln.fr/Ecran%20tactile%20sur%20RPi.html

Comme c'est expliqué à la fin de l'article, cette façon de faire utilisant des fichiers dont nous n'avons

pas les codes source n'est pas totalement satisfaisante.

Une autre solution

Le *touch screen*[9] Eleduino est considéré comme un périphérique HID (Human Interface Device) auquel on peut accéder par le biais d'un driver hidraw**x** (x= 0, 1, 2, … en fonction de l'ordre de montage des périphériques HID).

Dans le cas où le *touch screen* est *pluggé*[10] tout seul au moment du *boot*, il est monté sur : /dev/hidraw0

Nous allons créer une règle de montage udev, afin de donner les droits de lecture à l'utilisateur en créant un fichier **`/etc/udev/rules.d/52-touchscreen.rules,`** sur le *raspberry* et contenant les lignes suivantes :

```
SUBSYSTEM=="usb", ATTR{idVendor}=="0eef", ATTR{idProduct}=="0005", MODE="0666",
KERNEL=="hidraw*", SUBSYSTEM=="hidraw", MODE="0666", GROUP="plugdev"
```

Si l'écran cohabite avec d'autres périphériques HID (souris, clavier,.etc.), on peut également faire en sorte qu'il ait un nom de *device* symbolique fixe, ***touchscreen*** par exemple, en ajoutant à la règle, l'option:

```
SYMLINK+="touchscreen"
```

Un lien symbolique **/dev/touchscreen** sera automatiquement créé quand l'écran sera *pluggé.*

Fonctionnement : Lors de chaque action (touche, déplacement ou relâchement) le *touch screen* envoie sur le driver /dev/hidraw0 une trame constituée de 25 octets constituée comme suit :

- 1 octet de *header* contenant la valeur hexadécimal 0xaa
- 1 octet représentant l'état de l'événement (1 touché, 0 non touché)
- deux entiers sur 16 bits contenant les coordonnées[11] x1 et y1 de l'événement
- 1 octet de séparation contenant la valeur hexadécimal 0xbb
- 1 octets représentant le nombre de touches simultanées sur l'écran – champ de bits(5 maxi)
- 8 entiers sur 16 bits contenant les coordonnées respectives des touches (x2, y2, x3, y3, x4, y4, x5, y5)
- 1 octet de fin contenant la valeur hexadécimal 0xcc

exemple : la trame
aa 01 01 c2 01 1f bb 01 01 e0 03 20 01 e0 03 20 01 e0 03 20 01 e0 03 20 cc
correspond à une touche unique en x=450, y=287

La solution retenue pour supporter l'écran tactile consiste à interpréter ces trames et à envoyer, directement sur le terminal X, les signaux correspondant aux actions sur le *touch screen* :

9 Je voulais dire l'écran tactile
10 Je voulais dire branché
11 Toutes les coordonnées sont codées en *big endian*

- déplacement de la souris
- click gauche de souris (mono-touche court)
- click droit de souris (bi-touche court)
- maintien de la souris (touche longue)
- relâchement de souris

Le code source en C de ce programme est donné en annexe et pourra être récupéré sur : http://grimaldi.univ-tln.fr/files/touchscreen.c

La bibliothèque libX11 devra être installée préalablement :

```
sudo apt-get install libx11-dev
```

TRAVAIL DEMANDÉ

1. Installer l'image Linux sur le Raspberry
2. Installer la chaîne de cross compilation
3. Tester son fonctionnement avec l'application console « hello »
4. Ecrire une l'application graphique Qt de test mettant en œuvre un serveur TCP qui permet d'agir sur l'interface à distance à partir de commandes que vous définirez (exemple : **`radio1 on, radio1 off`**, *.etc*,). Tester avec *telnet.* et vérifiez les échanges avec *Wireshark.*
5. Cross compiler le programme de gestion du *touch screen*[12] et vérifiez son fonctionnement
6. Transformer ce dernier en service linux[13]
7. Tester avec la même application que précédemment mais sans souris
8. Ecrire une application Qt qui démarre en mode plein écran et qui permet
    - de choisir d'afficher une des 4 webcams de la ville de Toulon (http://www.toulon.fr/webcams) par le moyen de votre choix sur l'interface (bouton, imagette, comboBox, .etc.).
    - d'afficher les données météo locales récupérée sur le site de la ville de Toulon : (couverture nuageuse, température, direction et vitesse du vent).
    - Représenter ces données avec les composants de la bibliothèque QWT qu'il faudra cross-compiler également :
        - courbe de température
        - rose des vents
        - icône nuages / soleil
8. Connecter un capteur, qui sera fourni en séance, sur le GPIO du Raspberry et intégrer les

12 Le code source est donné en annexe
13 Attention, ce service devra démarrer après le serveur X pour pouvoir fonctionner

données de ce capteur à une application graphique Qt utilisant QWT.

ANNEXES

Programme de gestion du *touch screen*

```
/*
* touchscreen.c
*
* Copyright 2015 Michel GRIMALDI
*
* allow to control X mouse events from an eleduino 7" touch screen
*
*       release 1.00 - 12-09-2015
*/


#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/input.h>
#include <linux/uinput.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdint.h>
#include <X11/X.h>
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <sys/ioctl.h>
#include <time.h>
#include <math.h>


//      The payload structure sent each time the touchscreen is touched, mouved or relaesed
// header1=aa, x, y, header2=bb, multitouch flag, x2, y2, x3, y3, x4, y4, x5, y5, terminator=cc
// an hex dump bellow
// aa 01 01 c2 01 1f bb 01 01 e0 03 20 01 e0 03 20 01 e0 03 20 01 e0 03 20 01 e0 03 20 cc
typedef struct {
    unsigned char header1;      // aa
    unsigned char touch;
    uint8_t xh;         // first touch position
    uint8_t xl;
    uint8_t yh;
    uint8_t yl;
    unsigned char header2;      // bb
    unsigned char multitouch;
    uint8_t xh2;        // second touch position
    uint8_t xl2;
    uint8_t yh2;
    uint8_t yl2;
    uint8_t xh3;        // third touch position
    uint8_t xl3;
    uint8_t yh3;
    uint8_t yl3;
    uint8_t xh4;        // fourth touch position
    uint8_t xl4;
    uint8_t yh4;
    uint8_t yl4;
    uint8_t xh5;        // fifth touch position
    uint8_t xl5;
    uint8_t yh5;
    uint8_t yl5;
    unsigned char terminator;   // cc
}  HidEleduinoPayload;


#define SIMPLE_CLICK (0.1*CLOCKS_PER_SEC)
//      global variables
Display *display ;                                  // the X terminal display
Window root_window;                                 // the root system Window


// mouse move pointer function
```

```
int mouseMove(int x, int y){
    if(display == NULL)return -1;
    XWarpPointer(display, None, root_window, 0, 0, 0, 0, x, y);
    XFlush(display);
    return 0;
}

// mouse click function
int mouseClick(int button){
    if(display == NULL)return -1;
    // Create and setting up the event
    XEvent event;
    memset (&event, 0, sizeof (event));
    event.xbutton.button = button;
    event.xbutton.same_screen = False;
    event.xbutton.subwindow = DefaultRootWindow (display);
    while (event.xbutton.subwindow)
    {
        event.xbutton.window = event.xbutton.subwindow;
        XQueryPointer (display, event.xbutton.window,
                       &event.xbutton.root, &event.xbutton.subwindow,
                       &event.xbutton.x_root, &event.xbutton.y_root,
                       &event.xbutton.x, &event.xbutton.y,
                       &event.xbutton.state);
    }
    // Press
    event.type = ButtonPress;
    if (XSendEvent (display, PointerWindow, True, ButtonPressMask, &event) == 0)return -2;
    XFlush (display);
    //printf("mouse down\n");
    return 0;
}

// mouse release function
int mouseRelease(int button){
    XEvent event;
    if(display == NULL)return -1;
    memset(&event, 0x00, sizeof(event));
    event.type = ButtonRelease;
    event.xbutton.button = button;
    event.xbutton.same_screen = False;
    event.xbutton.state = 1<<(7+button);
    // get the root window pointer
    if (!XQueryPointer(display, RootWindow(display, DefaultScreen(display)),
                       &event.xbutton.root, &event.xbutton.window, &event.xbutton.x_root,
                       &event.xbutton.y_root,
                       &event.xbutton.x, &event.xbutton.y,&event.xbutton.state))  return -2;
    event.xbutton.subwindow = event.xbutton.window;
    while(event.xbutton.subwindow)
    {
        event.xbutton.window = event.xbutton.subwindow;
        XQueryPointer(display, event.xbutton.window, &event.xbutton.root, &event.xbutton.subwindow,
&event.xbutton.x_root, &event.xbutton.y_root, &event.xbutton.x, &event.xbutton.y,
&event.xbutton.state);
    }

    if(XSendEvent(display, PointerWindow, True, 0xfff, &event) == 0) return -3;
    XFlush(display);
    //printf("mouse up\n");
    return 0;
}

// get time in milliseconds
uint32_t millis(){
    struct timeval  tv;
    gettimeofday(&tv, NULL);
    return  (uint32_t)((tv.tv_sec) * 1000 + (tv.tv_usec) / 1000) ; // convert tv_sec & tv_usec to
millisecond
}

//_____________________________________________________________________
//              MAIN PROG
int main(int argc, char **argv)
{
```

```
    HidEleduinoPayload data;                    // a hid driver payload
    int xloc, yloc;                             // touch position
    int xloc2, yloc2;                           // second touch position (multitouch)
    int multitouch;                             // used for right click
    int thresholdTimeClick=200;                 // the duration between a touch and release sequence
                                                // to consider it as a click in milliseconds

    uint32_t now;                               // the current time stamp in millis
    uint32_t clTouch1, clRelease1;              // first touch/release time stamp
    uint32_t holdTime;                          // the time on which the screen is touched
    //uint32_t clTouch2, clRelease2;            // second touch/release time stamp
    char ch;                                                    // a char

    // verify the name of the driver hidraw0 or hidraw1 or ...
    char hidDriverName[50] = "/dev/hidraw";
    if (argc ==2) strncat(hidDriverName, argv[1], 2);
              else strcat(hidDriverName,"0");   // the touchscreen hid driver

    int state;                                  // init th state machine
    display = XOpenDisplay(NULL);               // connect to X server
    if(display == NULL)
    {
        fprintf(stderr, "cannot find X Display\n");
        exit(EXIT_FAILURE);
    }
    // get the system display
    root_window = XRootWindow(display, 0);
    XSelectInput(display, root_window, NoEventMask);

    // open hidraw eleduino touchscreen device
    FILE *fraw = fopen(hidDriverName,"r");
    if (fraw==NULL) {
        fprintf(stderr, "Unable to open %s (perhaps you don't have the good rights!)\n",
hidDriverName);
        return -1;
    }
    state=0;
    holdTime=0;
    for ( ; ; ){                // super loop
        // a touchscreen event comes
        fread(&data, sizeof(HidEleduinoPayload), 1, fraw);       // read the touch payload
        //printf("%x %x %x %x %x\n", data.header1, data.xl, data.xh, data.yl, data.yh);
        if (data.header1!=0xaa || data.header2!=0xbb ) {
            printf("HID touchscreen driver frame error!\n");
            for (ch=0 ; ch!=0xcc ; )fread(&ch, 1, 1, fraw);       // look for the terminator
            continue;
        }
        // swap the endian
        xloc = data.xl+256*data.xh;     yloc = data.yl+256*data.yh;
        xloc2 = data.xl2+256*data.xh2;  yloc2 = data.yl2+256*data.yh2;
        //printf("s=%d (%d, %d,%c), (%x, %d, %d)\n", state, xloc, yloc, (data.touch ? 'T':'F'),
        //                                                    data.multitouch, xloc2, yloc2);
        now = millis();                                     // now time stamp

        if (data.touch){        // every time screen is thouched, move the pointer
            multitouch = data.multitouch>1;                 // ? multitouch
            mouseMove(xloc, yloc);                          // move it
        }
        switch (state) {
        case 0 :if (data.touch){        // screen is touched
                clTouch1 = now;         // time stamp
                state=1;                // wait for release
            }
            break;
        case 1 : if (!data.touch){      // screen is released
                clRelease1 = now;       // time stamp
                //printf("time stamps : %d %d %d\n",clTouch1, clRelease1, (clRelease1-clTouch1));
                if (clRelease1-clTouch1<thresholdTimeClick) {           // is it a click
                    mouseClick((multitouch ? Button2 : Button1));       // right or left button
                    //printf("click %s\n", (multitouch ? "right" : "left"));
                    usleep(1);                                          // wait before release
                    mouseRelease((multitouch ? Button2 : Button1));
                }
                state=0;
            }
            else {              // screen is maintained
                holdTime = now-clTouch1;
```

```
                if (holdTime>1000){     // maintain click
                    mouseClick((multitouch ? Button2 : Button1));       // right or left button
                    state=2;
                }
            }
            break;
        case 2 : if (!data.touch){      // screen is released
                mouseRelease((multitouch ? Button2 : Button1));
                state=0;
                }
        }
    }
    // close X server
    XFlush(display);
    XCloseDisplay(display);
    fclose(fraw);
    return 0;
}
```