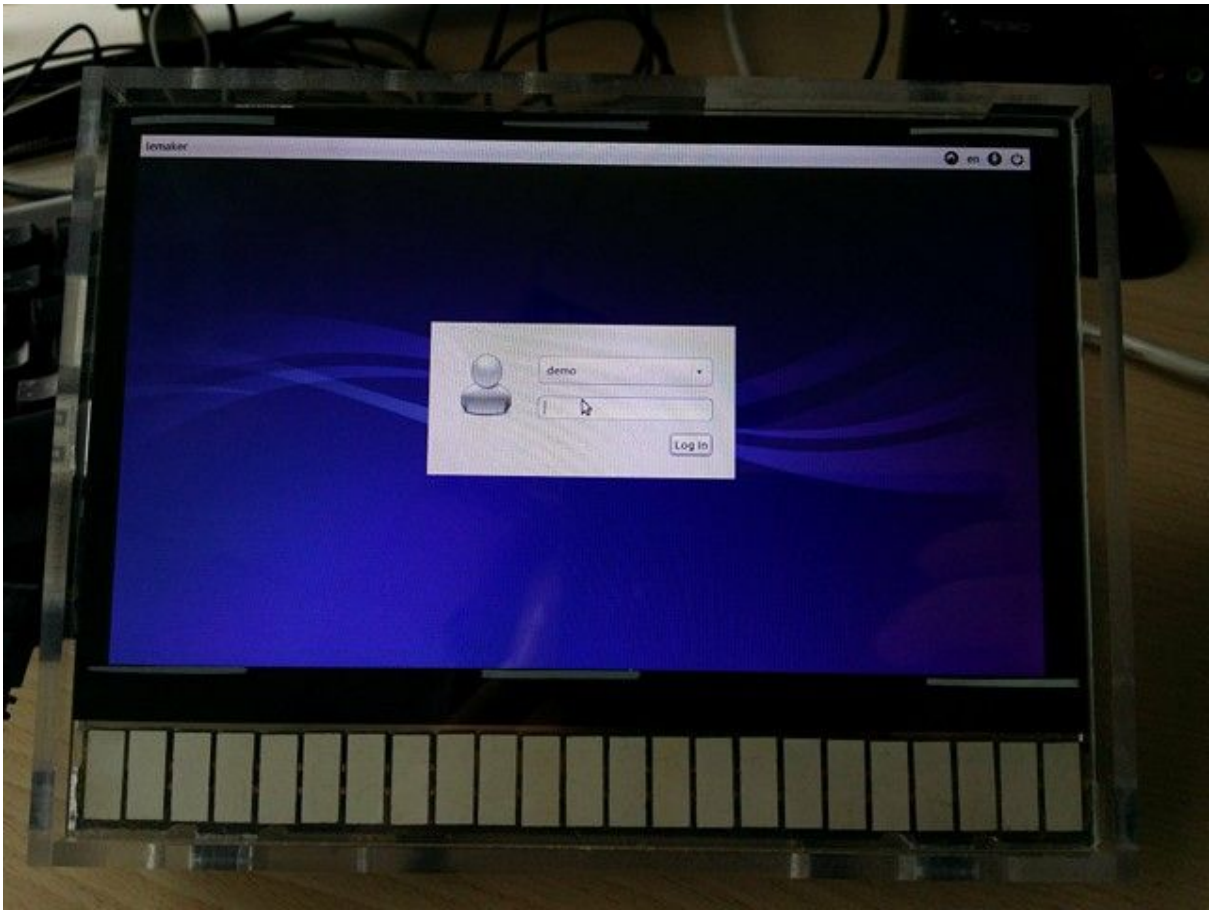
POLYTECH LILLE

Université de Lille

PROJET DE FIN D'ETUDE

Adaptation de la bibliothèque QT à la tablette Evita



Valentin BEAUCHAMP et Vivian SENAFFE
Tuteur : Laurent Grisoni
Années: 2017-2018

REMERCIEMENTS

Nous souhaitons dans un premier temps remercier Laurent GRISONI, notre tuteur, de nous avoir permis la réalisation de ce projet ainsi que de nous avoir donné de précieux conseils. Nous aimerions également remercier Michel AMBERG ainsi que Frédéric GIRAUD qui nous ont aidés à faire avancer notre travail et transmis leurs connaissances techniques sur la tablette.

Table des matières

Contexte

Le but de notre projet est de faire évoluer une bibliothèque de programmation, dans notre cas Qt, afin de la rendre compatible avec une tablette à retour tactile. L'idée principale est de pouvoir recompiler n'importe quelle application avec cette bibliothèque modifiée pour apporter une « tactilisation » de base à celle-ci.

Une application codée spécialement pour la tablette doit pouvoir embarquer des paramètres de retour réglables. Il nous faudra donc trouver un moyen de rentre le paramétrage du retour tactile simple à implémenter pour les futurs programmeurs.
Le concept de la tablette est sémantiquement parlant assez simple. Il s'agit d'une tablette tactile comportant des céramiques piézo-électriques. Lorsqu'on stimule ces céramiques, elles se mettent à vibrer et permettent donc de retourner une information à l'utilisateur de la tablette. Certaines de ces céramiques présentes sur la tablette sont utilisées comme retour. Cela permet une vibration contrôlée, mais on peut aussi l'utiliser en boucle ouverte, c'est-à-dire sans contrôle.

Cette tablette étant encore au stade de prototype, il n'existe pas encore d'application congrès à celle-ci. Mais plusieurs domaines s'y intéressent. On peut imaginer dans un futur assez proche, retrouver cette technologie dans nos smartphones par exemple. Afin de simuler un bouton et de donner à l'utilisateur une sensation de clique sur son téléphone. Cette tablette semble aussi être un bon moyen d'apprentissage de la langue pour les enfants. Elle pourrait aussi être utilisée dans des domaines comme celui du textile et permettrait aux clients de toucher/tester un tissu au travers de la tablette.

Problématique

Durant nos premières semaines de projet, nous avons pu prouver qu'il est possible de modifier Qt afin d'apporter à des applications une tactilisation de base.

Pour le moment la tablette fonctionne en deux parties :

- Un démon calcule les vibrations que le DSP doit effectuer
- Une application envoie des taxtels au démon. Les taxtels étant des zones de vibrations ainsi que leurs textures

Deux possibilités se sont donc offertes à nous :

- Créer des taxtels et utiliser le démon existant
- Intégrer le démon à Qt pour assembler les deux applications

Afin d'apporter un plus au projet, nous souhaitons intégrer le démon à Qt. Si cela n'est pas rendu possible, nous utiliserons l'autre méthode.

La recherche de grandes performances est au coeur de projet, afin de garantir un ressenti agréable et réel pour l'utilisateur. Il nous faut donc parfaitement comprendre le fonctionnement de la récupération de la position des doigts ainsi que de la gestion du DSP.

Matériels

La tablette Evita est constitué de différents blocs :

- la banana Pi supportant le système d'exploitation linux
- l'écran tactile relié à la banana Pi par un bus i2c
- le DSP communicant à la fois avec la banana Pi et les céramiques piézo-électriques
- les céramiques piézo-électriques mettant en mouvement le verre

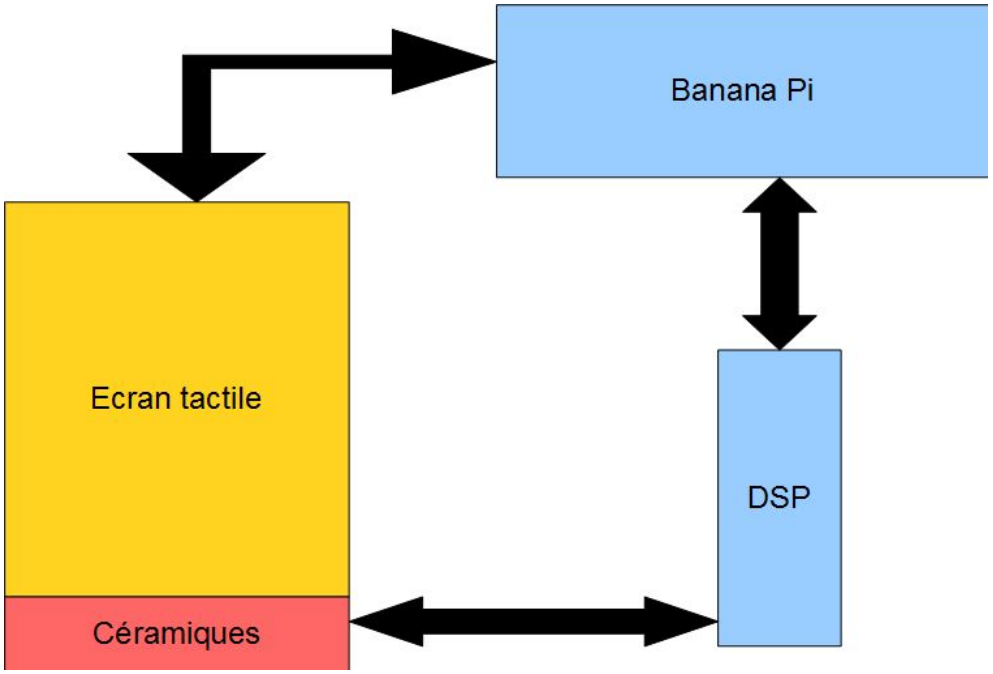


Schéma récapitulatif du dispositif

Par soucis de performances, la banana Pi ne contrôle pas l'écran tactile par son système d'exploitation. En effet, nous ne pouvons pas déterminer la position des doigts de manière sûre à un moment donné car nous ne contrôlons pas les différents éléments externes pris en charge par le système d'exploitation. C'est pourquoi, la lecture de la position des doigts se fait par l'intermédiaire du bus i2c. Celle-ci s'effectue toutes les 18 ms.

Le DSP permet de trouver en permanence la fréquence d'oscillation du verre qui change en fonction des conditions d'utilisation (température, humidité,etc). C'est lui qui génère aussi les signaux envoyés aux céramiques afin de créer des textures sous le doigt de l'utilisateur.

Cross-compilation

Nous nous sommes intéressés à la cross-compilation de Qt dans un premier temps, et avons, en parallèle, travaillé sur l'architecture logicielle que nous pouvions mettre en place.
Qt étant une bibliothèque lourde, elle nécessite un temps de compilation conséquent. Compiler sur la bananaPi est donc impossible, il nous faut donc cross-compiler Qt sur nos propres machines : une compilation complète de Qt prend une trentaine de minutes sur un ordinateur. Qt embarque avec lui des éléments de cross-compilation pour des plateformes définie. La BananaPi ne faisant pas partie de ces plateformes, il nous faut cross-compiler "manuellement" Qt.

La cross-compilation pour plateforme ARM générique sortant des spécificités de Qt, nous nous sommes confrontés à des problèmes inattendus et des comportements hasardeux du compilateur de Qt. Pour cross-compiler Qt correctement, nous avons du créer un sysroot de la tablette sur nos machines.

Un des comportements inattendu observable a été de découvrir que le linker du compilateur n'allait pas chercher ses paramètres dans le sysroot mais dans notre système de base. Ce comportement empêche le compilateur de trouver les librairies nécessaires à la bonne construction de Qt. Le seul message d'erreur obtenu étant que le linker ne trouvait pas les librairies demandées, il a fallu un certain temps avant de nous rendre compte du problème.

Une fois Qt configuré, pour être compiler pour processeur ARM, le compilateur refusait d'utiliser le sysroot pour la compilation. Il semblerait que ceci soit dû à un problème du compilateur que nous utilisions. En changent de compilateur le problème a été résolu, mais il n'a pas été facile de comprendre que le comportement venait du compilateur lui-même.

Pour rendre Qt portable, il est nécessaire de générer une librairie: libqxcb.so. Pour créer cette librairie, il faut installer une autre bibliothèque sur la tablette (libxcb) avant de créer notre sysroot. Dans un premier temps, nous avions utilisé la version 5.2 de Qt pour la cross-compilation. Or un bug connu de Qt empêche l'utilisation de la librairie libqxcb pour cette version, la rendant non portable. Pour corriger ce problème nous sommes simplement passés de la version 5.2 à la version 5.8 de Qt.

Une fois que nous avons réussi a cross-compiler Qt correctement, nous avons écrit un livrable, afin d'aider de futures personnes souhaitant réaliser ceci.

Pour créer une application sur la tablette, nous avons configuré notre IDE comme expliqué dans le livrable fourni en annexe, puis créer un package pour nos applications qui fonctionnent comme voulu.

Architecture logicielle

Architecture pensée

Comme expliqué précédemment, en parallèle de la cross-compilation, nous avons travaillé sur l'architecture logicielle que nous allions mettre en place pour modifier Qt. Une idée, qui a été retenue et approuvée par Michel Amberg, fut de modifier Qt afin que les applications à retour tactile ne passent plus par un service tiers. Celui-ci s'occupait de la captation des doigts et de l'envoie des informations au DSP. En effet, en ne passant plus par ce service, les applications sont plus performantes et les textures ressenties sont plus réalistes.

Afin de mettre en place ce système, nous avions pensé au lancement d'un premier thread au démarrage d'une application. Ce thread gère la communication i2c pour la captation des doigts ainsi que l'envoie des informations au DSP.

Pour savoir quelles actions doivent être effectuées sur la tablette, un thread est démarré à la création d'un widget. Ce thread analyse la position du doigt : en confrontant la position du widget et celle de l'utilisateur, le thread détermine une action tactile à effectuer.

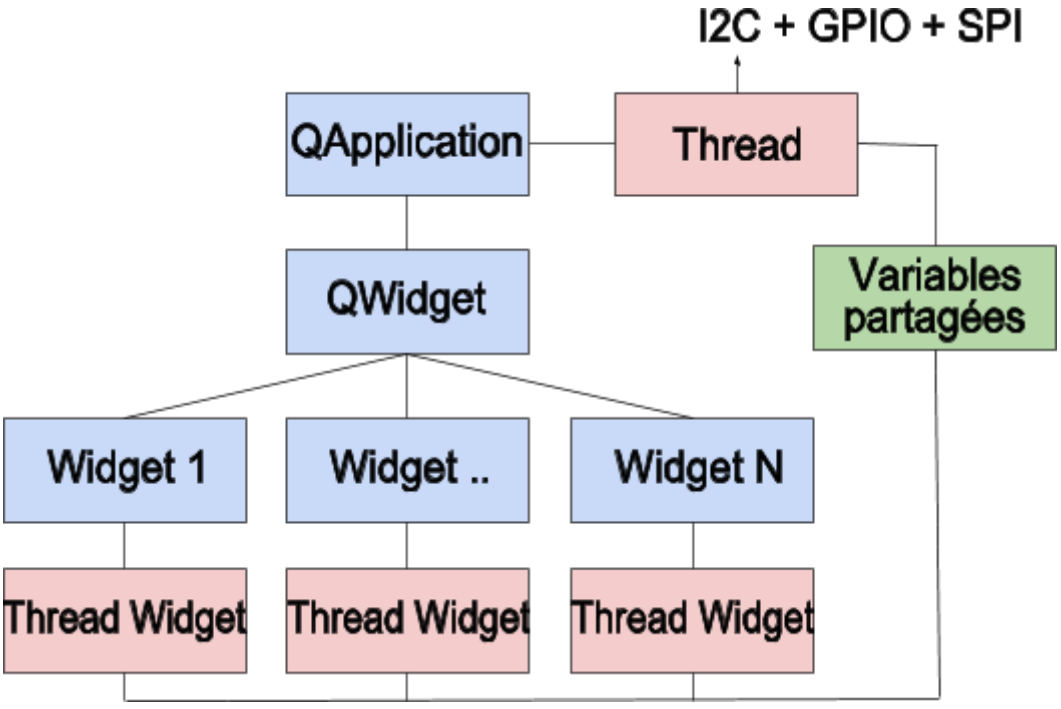


Schéma explicatif de l'architecture pensée

Thread sous Qt

Sous Qt, il existe une bibliothèque QThread qui permet la création de thread de façon multiple. L'utilisation des Qthreads à l'intérieur de la bibliothèque Qt étant peu permissive, il nous a fallu essayer plusieurs méthodes afin de déterminer la solution la plus adéquate à notre utilisation. Une des solutions les plus reconnues est la création de thread par l'intermédiaire de méthode utilisant des événements. Or, la latence des événements n'étant pas contrôlable et souhaitant les meilleures performances possibles, nous avons décidé de limiter cette méthode.
La technique que nous avons retenu, est la ré-implémentation de QThread : en re-définissant sa méthode "run", nous décrivons les actions que le thread effectuera. Devant connaître les informations de l'objet créant ce thread, nous implémentons d'autres méthodes permettant la passation de ces informations.

Timer

La récupération de la position du doigt doit s'effectuer tous les 18 ms comme nous l'a expliqué Michel Amberg. Afin de s'assurer de cette périodicité, nous avons décidé d'utiliser des QTimer. Les QTimer fonctionnent par signaux, pour les mêmes raisons que citées précédemment, nous ne sommes pas pour leur utilisation. Cependant, les signaux générés par QTimer ont un comportement qui se rapprochent des interruptions : ils ont donc une priorité supérieure aux autres et une latence moindre. Nous avons donc décidé de valider leur utilisation, étant la méthode la plus adéquate.
Lors de l'implémentation des QTimer, nous avons rencontré un soucis majeur : pour utiliser des signaux et des slots, nous devons passer par l'utilisation de "méta-objet". Or, après plusieurs tentatives et recherches, nous avons appris que la construction actuelle de Qt empêche l'ajout de méta-objets dans de nouvelles classes internes. Nous avons donc implémenter des nouveaux slots dans les classes créant les threads puisqu'ils utilisent les méta-objets.

Architecture utilisée

Le fonctionnement de la communication entre la BananaPi et le DSP nous a fait réviser l'architecture souhaitée. En effet c'est lorsque le DSP demande de nouvelles instructions que nous devons recalculer la position des doigts et générer un signal adéquat à transmettre au micro-contrôleur. Nous ne pouvons donc pas utiliser de traitement en parallèle si nous voulons assurer la continuité de ressenti.

Le traitement des données s'effectue donc lors d'une interruption sur un port GPIO. Nous n'avons donc plus eu besoin d'utiliser les QTimer ainsi que les méta-objets. Mais il est important de souligner la difficulté de leurs utilisations. L'idée de thread pour chaque widget a été oubliée. Nous avons fait plusieurs implémentations pour mettre en place notre architecture finale:

- Une structure *TabWidget* qui contient:
  - la structure *Textures* qui permet de générer les signaux envoyé au DSP
  - l'Id du Widget
  - Un pointeur du widget
- Un tableau de vecteur contenant la liste des *TabWidgets*
- Des fonctionnalités de set pour les textures dans la classe QWidget
- Une méthode permettant de savoir si l'utilisateur se trouve sur un widget

L'architecture de notre bibliothèque est donc modifier de la sorte:

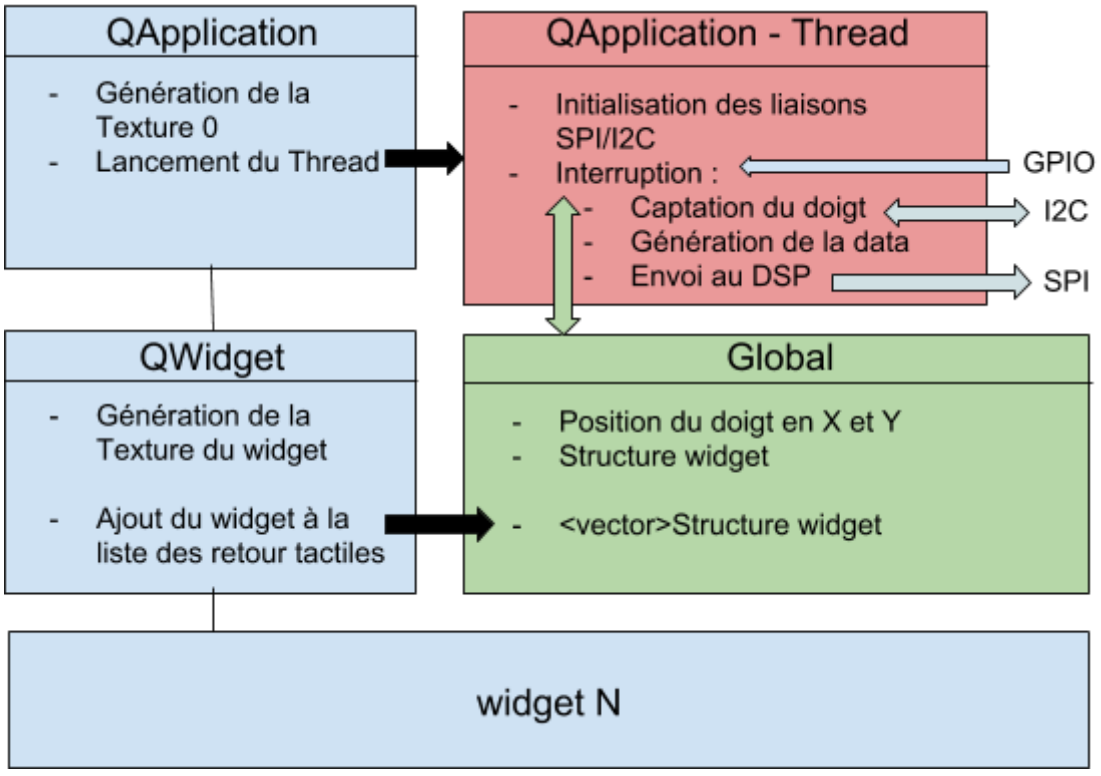


Schéma détaillé de l'architecture retenue

Récupération des positions du doigt

La récupération de la position des doigts sur l'écran tactile se fait par l'intermédiaire d'un bus i2c et non par le système d'exploitation de la banana Pi. Ainsi, la récupération de la position du doigt se fait en deux étapes : on interroge le bus i2c afin de recevoir les informations nécessaires, puis nous traitons les données reçues. Celles-ci se décomposent en sept octets de la manière suivante :

- octet 1 : ignoré
- octet 2 : ignoré
- octet 3 : présence du doigt
- octet 4 : 4 bits pour la force d'appuis + les 4 bits de poids fort pour la position en X
- octet 5 : les 4 bits de poids faible pour la position en X
- octet 6 : 4 bits pour le numero du doigt + les 4 bits de poids fort pour la position en Y
- octet 7 : les 4 bits de poids faible pour la position en Y

Nous récupérons la position du doigt selon X et Y. Pour alléger le traitement des données, si le flag de présence d'un doigt est à 0, alors on stop les vibrations de la tablette en exécutant directement une texture de base qui “ne fait rien”.

Cette position est récupérée en pixel par rapport au coin situé en haut à gauche (0,0). Par défaut les applications tournant sur la tablette sont supposées en plein écran, nous avons donc appliqué cette même idée sur Qt.

Communication avec le DSP

Pour communiquer avec le DSP, la bananaPi utilise une liaison SPI et une broche GPIO. Pour comprendre le fonctionnement, nous pouvons assimiler les données envoyées à un flux audio : le flux d'information doit être ininterrompu pour assurer la cohérence du ressenti de la tablette. Pour assurer la continuité de ce flux, il est nécessaire de surcharger le buffer du micro-contrôleur. Afin d'éviter une coupure, le DSP envoie un signal sur la broche du GPIO pour avertir un manque prochain de données. Dans le cas où le buffer est vidé, le DSP reprend le buffer en mémoire utilisé précédemment.

Le signal reçu par la broche GPIO peut-être interprété comme une interruption sur Qt. Il ne nous est donc plus nécessaire d'utiliser les QTimer pour analyser la position des doigts. Le traitement sera effectué lorsque le DSP exigera un nouvel ordre de vibration.

Ajout de la librairie wiringPi

Pour pouvoir implémenter cette communication, nous avons cherché à introduire la bibliothèque wiringPi dans celle du Qt. La Bibliothèque WiringPi permet la mise en place de communication SPI sur les micro-ordinateur Pi . Pour l'intégrer à Qt nous avons du modifier manuellement le Makefile générant la librairie widget:

$$INCPATH \quad +=- I / < path/to/lib > \quad - lwiringPi .$$

Une fois cette librairie ajoutée à Qt, nous pouvons gérer des interruptions dans le thread créé par le singleton de QApplication.

Fonctionnement du thread de QApplication

Pour faciliter notre développement, nous avons décidé de conserver la méthode de fonctionnement du démon existant et de l'intégrer à Qt. Lors de la création du thread dans QApplication, on initialise les différentes connections, SPI et I2C, puis on génère un signal d'interruption qui prend en compte un front descendant sur le port GPIO.

Lorsqu'un signal d'interruption est capté, le thread entre dans une fonction ISR(). Cette fonction va recalculer la position du doigt puis calculer sa vitesse en fonction de sa position précédente. Grâce à notre liste *TabWidget*, le thread appelle une fonction propre a chaque widget qui permet de savoir si l'utilisateur survole un objet. Si aucun objet est survolé alors on joue une texture de base qui ne génère pas de vibrations pour ne pas couper le flux d'informations. Dans le cas où l'utilisateur passe sur un widget alors le thread récupére les informations de texture de celui-ci. Par la suite, l'application va calculer le signal à envoyer au DSP pour assurer une continuité de vibration sur la tablette. L'idée principale de cette continuité est d'éviter un mauvais ressenti pour l'utilisateur et un passage fluide d'une texture à une autre.

Les Widgets

Lorsqu'un widget est créé, on vérifie dans un premier temps si ce widget est rattaché à la fenêtre de notre application. Sinon, il s'agit de la fenêtre de l'application elle-même. Cette comparaison faite, on va créer un objet *TabWidget* contenant les informations de notre widget. Cet objet est ajouté à la liste de structure *TabWidget* vérifié par le thread de QApplication. En appelant une des méthodes *setTexture* de l'objet, on peut créer une nouvelle texture à jouer lorsqu'un utilisateur passe sur le widget.
Pour le moment, il existe deux méthodes de création de textures:

- void QWidget::setTextureCos(int offset,int amplitude,int period, char* speedFunc)
- void QWidget::setTextureRect(int offset,int amplitude,int period,float ratio,char* speedFunc)

Ces fonctions permettent la génération respective d'un signal sinusoïdale et rectangulaire pour le DSP.

Tests

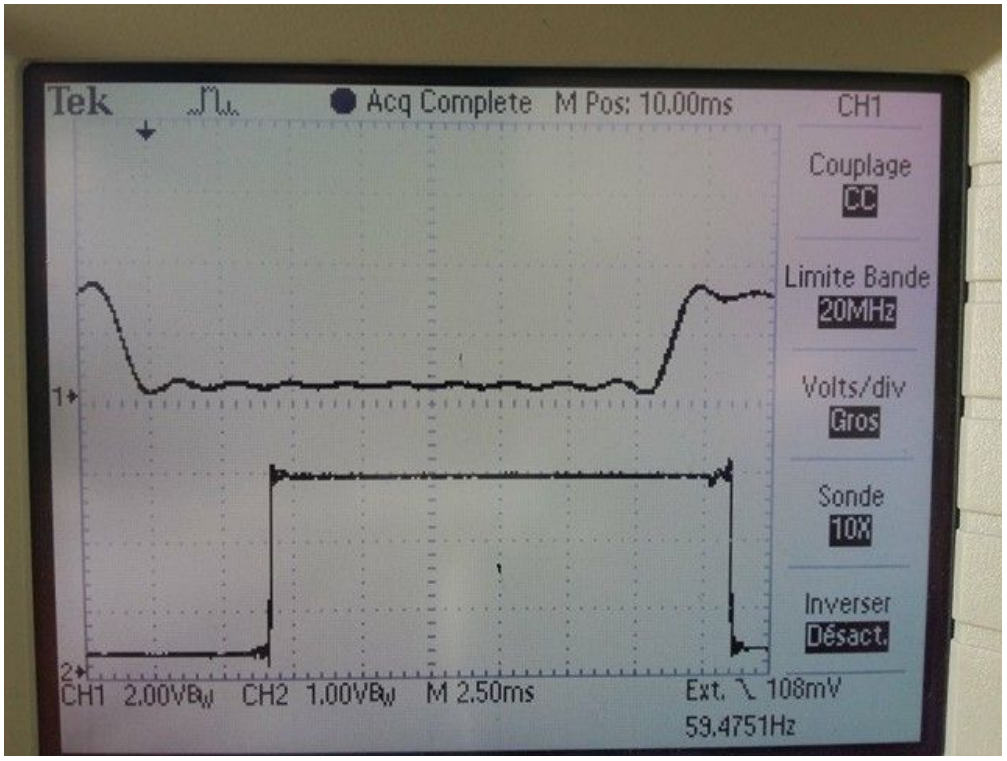
A l'aide d'un oscilloscope et de sondes branchées sur les broches du SPI, nous avons pu déterminer le temps de traitement et donc de réponse de notre système.

Pour une texture correspondant à un signal carré équivalent. Nous observons que le temps de réponse de notre application est d'environs 4.5ms contre 7.5ms pour l'application nous servant de témoin. Cette différence s'explique par le fait que nous ne passons plus par un service tiers pour la communication avec le DSP. Cette amélioration étant attendue, mais la vérification nous a permis de valider nos hypothèses de départ.

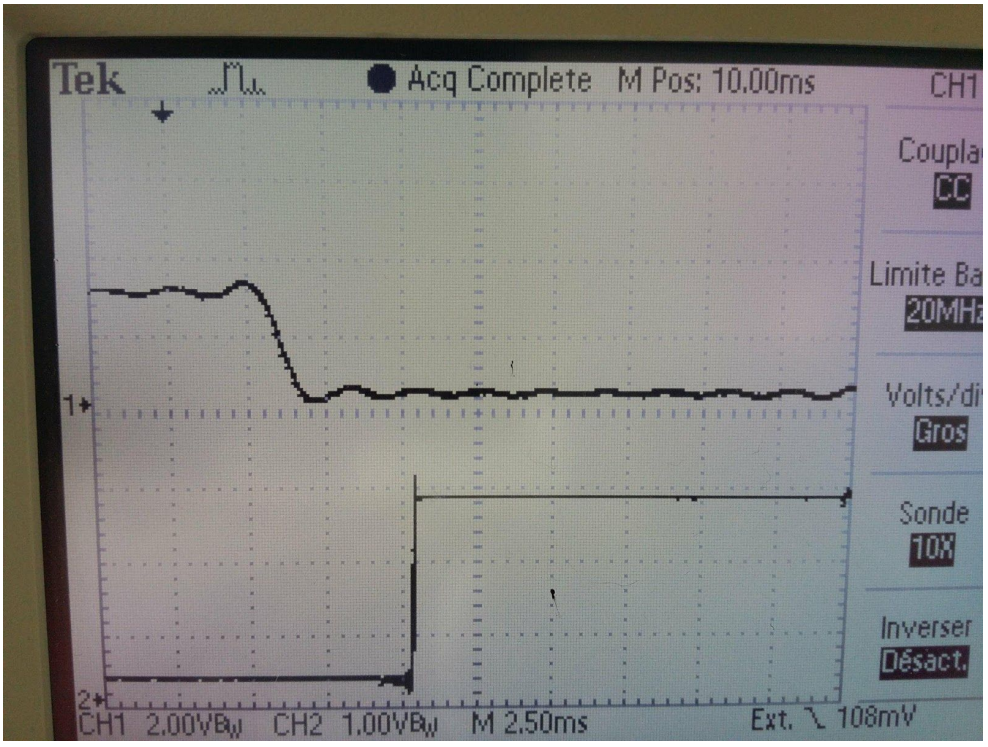
Courbes

Pour les deux graphes:

- Sur la courbe 1, nous observons le signal du DSP envoyé aux céramiques.
- Sur la courbe 2, nous observons la connexion MOSI du SPI.



Courbe observée avec l'utilisation de Qt



Courbe observée avec l'utilisation du service tiers

Pour le premier graphe, on voit que l'application mets 1.75*2.5ms de temps de réaction soit 4.5ms. Par ailleurs, on constate la continuité du signal transmis du DSP aux céramiques.

Pour le second graphe, on à un temps de réponse de 3*2.5ms soit 7.5ms.

Nous gagnons donc 3ms de temps de traitement lors de l'utilisation de notre application, ce qui est loin d'être négligeable.

Détails des modifications apportées à Qt

Pour faciliter le travail des personnes souhaitant reprendre le projet, nous avons formalisé les modifications de Qt :

Dans qtbase/src/widgets/kernel/ :

- Ajout des fichiers tactileGlobal.h et tactileGlobal.cpp

- Ajout des fichiers qapplicationthread.h et qapplicationthread.cpp

- qapplication. cpp :
    - ajout des libs tactileGlobal.h et qapplicationthread.h
    - création de la texture null dans le constructeur de QApplication
    - lancement du thread QApplicationThread dans le constructeur.

- qwidget.cpp/.h :
    - ajout de la lib tactileGlobal.h
    - ajout des méthodes de setTextures
    - initialisation des textures pour les widgets dans le constructeur

- kernel.pri :
    - ajout des fichiers tactileGlobal.h et .cpp pour la compilation
    - ajout des fichiers qapplicationthread.h et .cpp pour la compilation

Dans qtbase/src/widget :

- Makefile :
    - ajout de la ligne : INCPATH += -I /path/to/wiringPiLib -lwiringPi pour l'ajout de la librairie à Qt.

Points d'amélioration du projet

Une des premières étapes à effectuer dans la suite du projet, est l'implémentation d'un thread gérant la souris. C'est-à-dire transformer le toucher de l'écran au pointeur de la souris.

Si un développeur souhaite dorénavant utiliser une application qui n'est pas en plein écran, il est possible de modifier la méthode isInElement() afin de connaître la position du widget à l'intérieur de la fenêtre mais par rapport au coin supérieur gauche de l'écran. Pour cela, il faudrait se pencher sur l'utilisation des méthodes *mapTo() présents dans la classe QPoint.*

Si nous souhaitons générer une texture par défaut différente pour chaque type de widget, il faut implémenter dans les constructeurs des widgets un appel vers une des méthodes de setTexture présentes dans QWidget.

Une possibilité pour améliorer les performances des applications, et donc leurs temps de réponse, est de trouver une méthode pour ne plus passer par des structures et des variables globales. En effet l'utilisation de variables globales semble être mal géré par Qt et ralentit donc le système.

Pour le moment les widgets utilisables ne peuvent être que rectangulaires. Pour utiliser des widgets de toute formes, il semble que l'utilisation de la classe QRegion soit la meilleure.

Conclusion

Nous avons pu démontrer qu'il était possible d'adapter la bibliothèque Qt afin de lui ajouter des fonctionnalités tactiles. Il est désormais possible d'utiliser une application avec des retours tactiles par simple recompilation. Bien que des points d'améliorations soient envisageables, nous sommes satisfaits du travail que nous avons pu effectuer au sein de l'IRCICA.

Les recherches et le travail que nous avons menés durant notre projet, nous ont permis d'acquérir de nouvelles compétences en C++, Qt, système ainsi qu'en analyse de code et rétro-engineering.

Travailler à l'IRCICA, nous a donné un aperçu du monde de la recherche, et nous donne l'envie d'explorer de nouveaux domaines d'activités. Nous espérons que la contribution que nous avons apporté au projet aidera l'IRCICA à faire évoluer la tablette Evita.

Bibliographie

- Documentation C++ : http://devdocs.io/cpp/
- Documentation Qt : http://doc.qt.io
- Forum Qt : https://qt.developpez.com/
- Wiki Qt : https://wiki.qt.io/Main

Annexes

- Archives du code utilisé (donné au tuteur)
- Tutoriel de cross-compilation

Cross Compilation d'application Qt pour Processeur ARM

Après plusieurs essais infructueux, j'ai réussi à cross compiler Qt pour processeur ARM et à faire tourner une application compilée depuis mon ordinateur sur une BananaPi. Etant débutant en la matière, je n'ai pas trouvé d'explications claires à tous les problèmes que j'ai pu rencontrer.
Je vais donc vous expliquer la procédure que j'ai effectué pour cross compiler Qt comme il se doit.

1/ Configurer et Compiler Qt

Pour commencer, j'ai utilisé la dernière version du compilateur arm-linux-gnueabihf:

apt-get install gcc-arm-linux-gnueabihf g++-arm-linux-gnueabihf

J'utilise la version de Qt 5.8 everywhere que vous trouverez facilement sur internet.

Une première étape afin de cross compiler Qt est de le configurer correctement. Pour cela, vous devez créer un sysroot de votre target.

1.1/ Créer le sysroot:

Pour rendre Qt portable, il faudra que Qt génére un fichier libqxcb.so lors de sa compilation. Pour générer ce fichier, il faut installer les libxcb sur votre target:

apt-get install "^libxcb." libx11-xcb-dev libglu1-mesa-dev libxrender-dev*

Une fois les librairies installées, vous pouvez copier le /usr et /lib et /etc/ld.so.conf.d de votre target sur votre ordinateur en les plaçant dans un dossier sysroot

1.2/ Configurer Qt:

Pour configurer Qt j'ai dans un premier temps modifier le fichier [path/to/qtbase/dir/]mkspecs/linux-arm-gnueabi-g++/qmake.conf:

```
#
# qmake configuration for building with arm-linux-gnueabi-g++
#

MAKEFILE_GENERATOR      = UNIX
CONFIG                 += incremental
QMAKE_INCREMENTAL_STYLE = sublib
#QMAKE_LFLAGS = -static -static-libgcc

include(../common/linux.conf)
include(../common/gcc-base-unix.conf)
include(../common/g++-unix.conf)

# modifications to g++.conf
QMAKE_CC                = arm-linux-gnueabihf-gcc
QMAKE_CXX               = arm-linux-gnueabihf-g++
QMAKE_LINK              = arm-linux-gnueabihf-g++
QMAKE_LINK_SHLIB        = arm-linux-gnueabihf-g++

# modifications to linux.conf
QMAKE_AR                = arm-linux-gnueabihf-ar cqs
QMAKE_OBJCOPY           = arm-linux-gnueabihf-objcopy
QMAKE_NM                = arm-linux-gnueabihf-nm -P
QMAKE_STRIP             = arm-linux-gnueabihf-strip
load(qt_config)
```

puis j'utilise la ligne de commande suivante pour configurer qt:

> *./configure -opensource -confirm-license -prefix /usr/local/Qt-5.8-arm -xplatform linux-arm-gnueabi-g++ -nomake examples -nomake tests -sysroot /home/valentin/sysroot -xcb -no-opengl -no-pch*

-xplatform: indique le compilateur utilisé
-sysroot: chemin vers votre sysroot
-prefix: chemin où s'installe la bibliothèque
-xcb: impose la création de la lib qxcb
-no-opengl: ignorer la lib opengl
-no-pch: ignorer les headers précompilé (précompilé pour processeur x86 donc à ignorer)
-nomake: ne pas compiler les fichiers exemples et tests pour gagner du temps
- opensource: licence opensource
-confirm-license: confirmer la licence

A ce moment, vous pouvez avoir une erreur liée au linker ld. En effet pour une raison inconnue, mon linker ne prenait pas les paramètres du sysroot pour sa config. Il prenait la configuration de mon linker natif. Ne trouvant pas comment modifier ce défaut, j'ai ajouté à la config de mon linker deux lignes:

vim /etc/ld.so.conf.d/arm-linux-gnueabihf.conf

/home/valentin/sysroot/lib/arm-linux-gnueabihf
/home/valentin/sysroot/usr/lib/arm-linux-gnueabihf

Une fois ces lignes ajoutées, plus de problème de linker.

1.3/ Compiler Qt

Qt se configure correctement, pour compiler Qt faite un:

make -j4
make install

Qt s'installera dans le dossier indiqué par la commande -prefix de la configuration.

2/ Déploiement d'application

Une fois la bibliothèque Qt compilée, vous pouvez l'installer sur votre BananaPi, pour programmer des applications directement dessus.

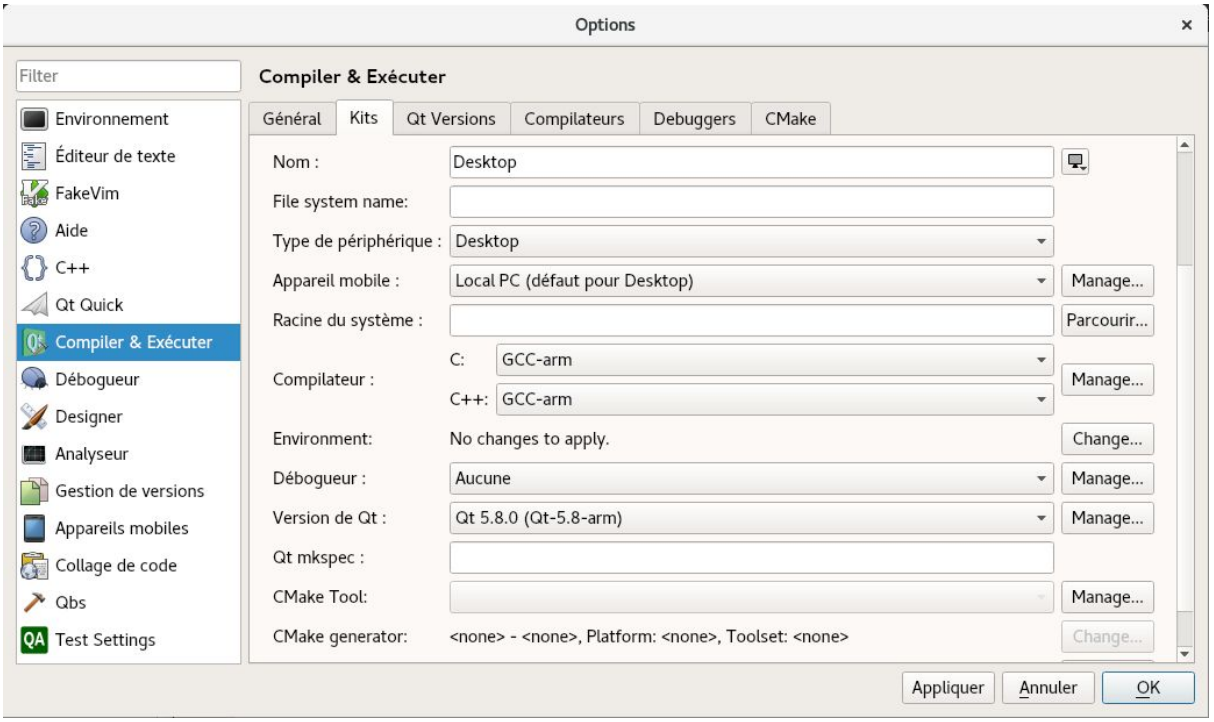
Si vous souhaitez compiler une application sur votre ordinateur et la lancer sur votre Banana, il vous faudra configurer votre IDE pour cela, puis créer un package pour votre application.

Personnellement j'utilise l'IDE de base de Qt, Qt creator.

2.1/ Configuration de Qt Creator

Pour compiler une application ARM sous Qt creator, il faut indiquer l'emplacement de votre compilateur ainsi que la bibliothèque Qt ARM à votre IDE.

Dans l'onglet, outils>options de Qt creator rendez-vous dans l'onglet "Qt version" pour ajouter votre version de Qt qui se trouve dans votre dossier sysroot. Puis ajoutez les compilateurs GCC et G++ pour ARM dans l'onglet "compilateurs".
Une fois la version de Qt et les compilateurs ajoutés, modifier la configuration de Qt ou créez en une nouvelle dans l'onglet "kit" en indiquant à Qt creator d'utiliser votre version ARM de Qt et le compilateur ARM.



Un fois cette étape finie, il ne reste plus qu'à aller dans l'onglet "projet" de Qt Creator pour ajouter

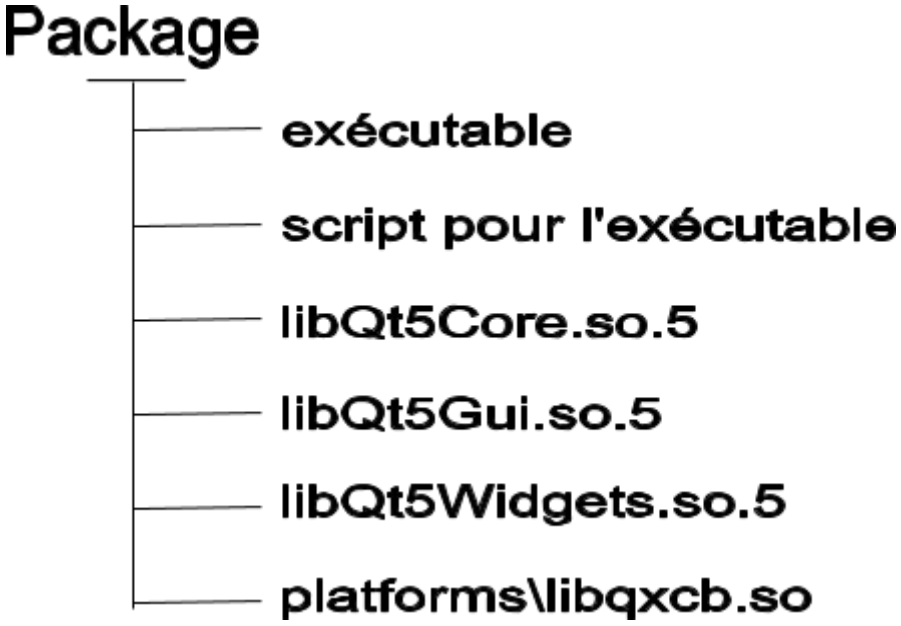
-spec linux-arm-gnueabi-g++ : en argument supplémentaires au Qmake

ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- : en argument du Make

Vous pouvez maintenant compiler votre application, il ne reste plus qu'a créer un package pour que votre BananaPi puis exécuter votre projet.

2.2/ Créer un package

Pour créer votre package, je vous conseille de créer un nouveau dossier où vous placerez vos fichiers selon cette arborescence:



Le script pour lancer une application nous est donné par Qt:

```
#!/bin/sh
appname=`basename $0 | sed s,\.sh$,,`

dirname=`dirname $0`
tmp="${dirname#?}"

if [ "${dirname%$tmp}" != "/" ]; then
dirname=$PWD/$dirname
fi
LD_LIBRARY_PATH=$dirname
export LD_LIBRARY_PATH
$dirname/$appname "$@"
```

Le script doit s'appeler [nom_de_votre_application].sh

Une fois votre package créé, vous pouvez le transférer sur votre bananaPi et exécuter le script pour démarrer votre application.

2.3/ Dépendance pour l’application

Si vous n'installez pas votre version de Qt ARM sur votre BananaPi, il se peut qu’il vous manque des dépendances pour lancer votre application.

Vous pouvez corriger ce problème en utilisant la commande “ldd” sur les .so de votre package pour voir si des liens sont brisés. Vous pouvez ajouter les librairies manquantes à votre package et les lier par un lien symbolique avec la commande “ln” afin de résoudre le problème.