RAPPORT DE PROJET/TP

GUILLEMAILE Timothée - MONTANGE Aimeric

LO52 – A20

Établissement :
Université de Technologie de Belfort-Montbéliard

Responsable UV :
Fabien BRISSET

Sommaire

Séance 1 : Mise en place de l'environnement de développement

Objectif :

Les objectifs de cette première séance étaient de choisir son groupe de TP, créer et configurer son environnement Git avec la création d'une branche. Et enfin de mettre en place un environnement de développement d'applications Android.

Mise en place de l'environnement de travail

Notre groupe de travail est composé de deux personnes : MONTANGE Aimeric et GUILLEMAILLE Timothée. Lors de ce premier TP nous avons créé l'espace de travail que nous allons utiliser tout au long de ce semestre.

Premièrement nous avons récupérer les sources héberger sous GitHub. Afin de par la suite créer sa propre branche correspondante à notre groupe de travail. Ce sera ici que nous déposerons tous nos travaux lors de cette période de travail. Grace au TD effectuer auparavant nous avions les principales commandes et manipulations à effectuer pour créer notre nouvelle branche et nous positionner dessus.

- *git clone https://github.com/gxfab/LO52_A2020* (récupération du contenu du dossier)
- *git branch GuillemailleTimothee_MontangeAimeric* (création de notre branche)
- *git checkout GuillemailleTimothee_MontangeAimeric* (positionnement sur notre branche de travail)
- *git push -u* (envoyer sur le serveur nos modification du dossier de travail)

Notre branche s'appelle donc : GuillemailleTimothee_MontangeAimeric

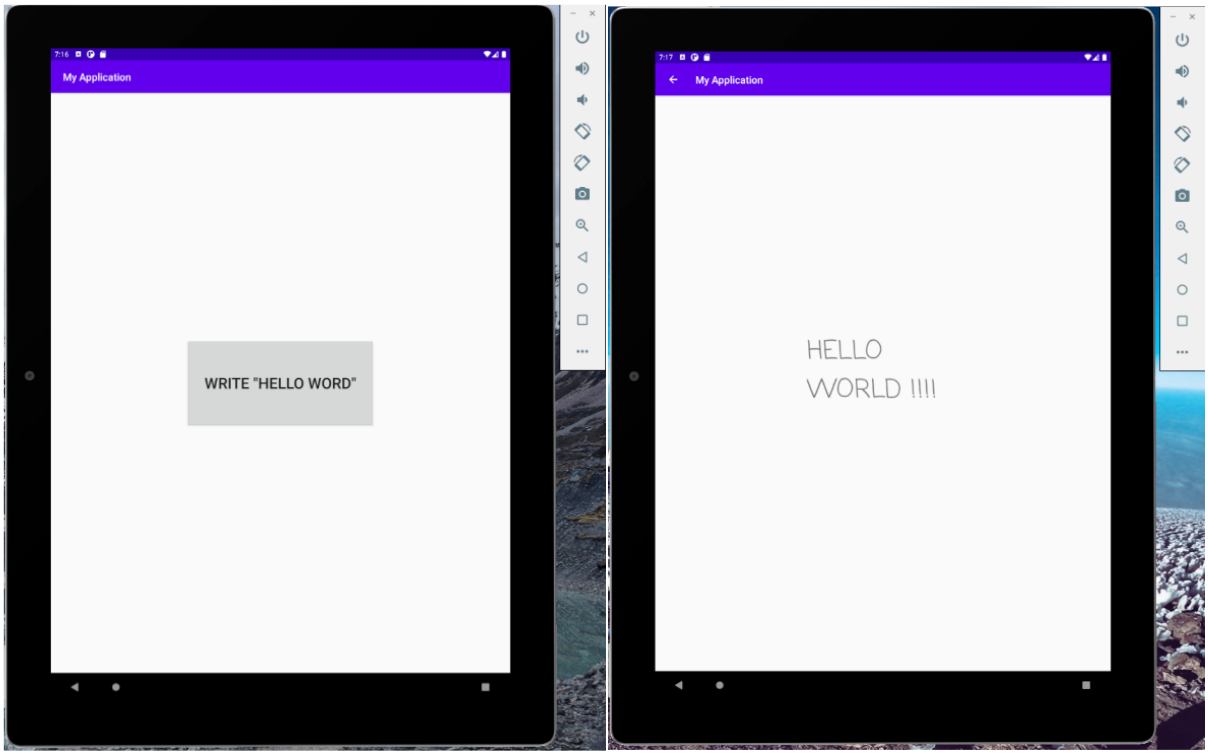
Deuxièmement il nous était demandé de créer une petite application qui permettait l'affichage d'un message lors d'un appui sur un bouton. C'est pour ils nous faillaient avoir un IDE permettant ce travail.

Nous avons ensuite installé et configurer notre outil de travail : Android Studio. Pour cela nous avons renseigner au logiciel le jdk disponible. Un fois fonctionnel nous avons créé notre première application, et nous avons travaillé sur le tutoriel de la documentation Android Studio

(https://developer.android.com/training/basics/firstapp). Nous avons pu prendre connaissance des différents modules disponibles pour pouvoir créer une application. Le fichier .kt, le fichier .xml (graphique)… Nous avons ensuite téléchargé notre émulateur d'Android afin d'exécuter notre application. Nous avons donc vu la création de l'activité principale Bouton, utilisation du layout editor et la liaison entre le bouton et la fonction d'affichage du message.

Nous avons essayé deux manières différentes d'exécuter cette application. Sur notre propre device (téléphone) et sur l'émulateur intégré à Android Studio qui suffit de télécharger. Il faut donc soit brancher son téléphone en USB avec l'option débogage activé, soit télécharger les fichiers correspondants à l'émulateur souhaité, de choisir une version d'API correspondante a notre version d'application et d'avoir assez d'espace de stockage pour l'héberger.

Voici le résultat de notre application dans l'émulateur : Création d'un bouton affichant « Hello World !!!! »



Problèmes rencontres

Nous avons rencontré quelques difficultés lors de cette séance notamment lors de la configuration du SDK qui n'était pas disponible sur nos postes de travail local à l'UTBM. C'est pourquoi

afin d’être à l’abri lors de ce semestre, ils nous étaient préférable de travailler sur notre propre appareil afin d’avoir un SDK configuré et ne changeant pas.

Séance 2 : Manipulation d'un kernel Linux

Environnement de configuration du Kernel

L'objectif de cette deuxième séance était de mettre en place un environnement de développement propice aux applications Android. Il était demandé de faire manipulations sur un kernel Linux. Dans un premier temps il nous a été demander de configurer un noyau hikey-linaro.

Configuration noyau hikey-linaro :

Voici les commandes que nous avons utilisé afin de gérer la configuration de ce noyau :

mkdir android-kernel && cd android-kernel

repo init -u https://android.googlesource.com/kernel/manifest -b hikey-linaro-android-4.19

repo sync -j 8

Nous avons ensuite exporté les variables ARCH et CROSS_COMPILE afin de ne pas les redéfinir à chaque utilisation.

export ARCH=arm64

export CROSS_COMPILE=~/toolchain/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-

Il nous a été demandé de configurer une carte ranchu64 et d'identifier sa configuration. On copie tout d'abord la configuration dans la configuration ARM64 afin de faciliter les commandes comme *make* ensuite.

cp arch/mips/configs/generic/board-ranchu.config arch/arm64/configs/

Nous avons pu observer et identifier la configuration par défaut relative au noyau pour une carte ranchu64 grâce au chemin suivant :

hikey-linaro/arch/{plateforme}/configs/defconfig .

La configuration par défaut est donc celle-ci : *DEFCONFIG=hikey960_defconfig* .

Après avoir chargé cette configuration par défaut, nous avons pu observer les différences entre les config grâce à la commande suivante exécuté dans le dossier *./android-kernel*:

Git diff build.config hikey-linaro/arch/{plateforme}/configs/defconfig .

Il fallait ensuite sauvegarder ses configurations et de leurs fusions :

make savedefconfig

mv defconfig arch/arm64/configs/defconfig

Nous avons écrit un script pour automatiser ses commandes. Afin de pas réeffectuer les mêmes opérations à chaque fois. (Script.sh)

Nouvelle configuration de Kernel

Ensuite nous avons modifié la configuration comme décrite dans le TP. Nous avons donc :

- Gérer la compatibilité pour la carte ARMv8 Versatile mais aussi Qualcomm et Realtek
- Activer du NFC et du protocole NFC HCI
- Activer de l'option Frequency Scaling de la CPU
- Activer du support de l'HDMI CEC et activation du support LED
- Désactiver toutes les autres options superflues

Pour désactiver les options superflues nous avons utiliser l'éditeur graphique grâce à la commande *make xconfig.* Nous avons retiré notamment les support UEFI, le Bluetooth, la carte son, et d'autres options …

Et enfin nous avons générer une configuration par défaut pour cette nouvelle configuration.

Pour pouvoir nettoyer les sorties de compilations et les configurations chargés nous avons utilisé les commandes suivantes :

make clean

make mrproper

Etape bonus

Nous avons compilé une première fois pour Android 64bits, la première version de ce noyau.

Make -j 8 cela permet une première compilation.

La toolchain est récupérer grâce au code qui est décrit dans le script.

cd ~/toolchain

wget https://releases.linaro.org/components/toolchain/binaries/latest-7/aarch64-linux-gnu/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu.tar.xz

tar -xf gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu.tar.xz

Nous avons compilé une deuxième fois la version optimisée pour Android 64. Nous avons utilisé la même commande « make » afin de faire ceci.

Make -j 8

Pour eviter une réecriture de la variable CROSS_COMPILE nous l'exportons :

export CROSS_COMPILE=~/toolchain/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-

Nous avons comparé les deux images et une différence de taille est notable entre ses deux entités :

Image de config par défaut (11MB)>image de config modifiée (8,5MB)

Séance 3 : Création d'un device Android

L'objectif de cette troisième séance de TP était d'écrire des fichiers Makefile Android relatifs à l'intégration d'un composant et d'un produit Android.

Implémentation de la Libusb

Il nous a été demander de déterminer les fichiers sources et les headers pour la compilation d'une libusb sur Android.

Voici le résultat de notre trie :

- Pour les fichiers sources :

core.c // descriptor.c // io.c // os/darwin_usb.c // os/linux_usbfs.c // sync.c

- Pour les fichiers headers :

libusb.h // libusbi.h // os/darwin_usb.h // os/linux_usbfs.h

Nous avons ensuite écrit le fichier Android.mk :

LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)

LOCAL_SRC_FILES:= core.c descriptor.c io.c sync.c os/linux_usbfs.c

LOCAL_C_INCLUDES += external/libusb-1.0.3/ external/libusb-1.0.3/libusb/external/libusb1.0.3/libusb/os

LOCAL_MODULE:= libusb

include $(BUILD_SHARED_LIBRARY)

Notre fichier Android.mk est ainsi écrit dans le repertoire.

Une erreur se produit sur la macro TIMESPEC_TO_TIMEVAL car celle-ci n'est pas définie. Nous avons donc corrigé ce petit défaut : nous avons donc rajouter dans le fichier io.c le code suivant afin de définir cette macro.

 # define TIMESPEC_TO_TIMEVAL(tv,ts)

do {

(tv)-> tv_sec =(ts)->tv_sec;

(tv)-> tv_usec =(ts)->tv_nsec/ 1000

} while (0)

Une autre erreur s'est produite lors de l'essai de compilation : build/tools/apriori/prelinkmap.c(137) : library "libusb.so" not in pre-link map

Pour résoudre ce problème nous avons ajouté l'adresse de la librairie libusb dans le fichier qui se trouve à l'emplacement suivant : /build/core/prelink-linux-arm.map

libusb.so 0x99000000

Implémentation d'un nouveau produit Android

Pour la définition du nouveau produit Android que nous devions faire, il y a eu quelques contraintes : Le nom du produit devait respecter une codification : dans notre cas ce fut lo52_GuillemailleTimothee_MontangeAimeric. Ce produit devait également hériter du produit Hikey de Linaro.

Nous avons vu dans le cours que le produit est défini au même niveau que l'AOSP (device/utbm/lo52_GuillemailleTimothee_MontangeAimeric). Les personnalisations des propriétés ro.hw, net.dnsl et net.dns2 ont été apporté grâce au fichier lo52_GuillemailleTimothee_MontangeAimeric. Nous avons également surchargé le fichier « sym_keyboard_delete.png » Le fichier lo52_GuillemailleTimothee_MontangeAimeric regroupe les modifications et personnalisations des différentes propriétés.

Voici donc le fichier lo52_GuillemailleTimothee_MontangeAimeric.mk

 # Inherit from those products. Most specific first.

$(call inherit-product, device/linaro/hikey.mk)

*PRODUCT_PACKAGES += *

libusb_1_0_3.hikey

*PRODUCT_PROPERTY_OVERRIDES := *

*ro.hw=lo52 *

*net.dns1=8.8.8.8 *

net.dns2=4.4.4.4

DEVICE_PACKAGE_OVERLAYS := device/lo52_GuillemaileTimothee_MontangeAimeric/overlay

PRODUCT_NAME := lo52_GuillemaileTimothee_MontangeAimeric

PRODUCT_DEVICE := lo52_GuillemaileTimothee_MontangeAimeric

PRODUCT_BRAND := UTBM

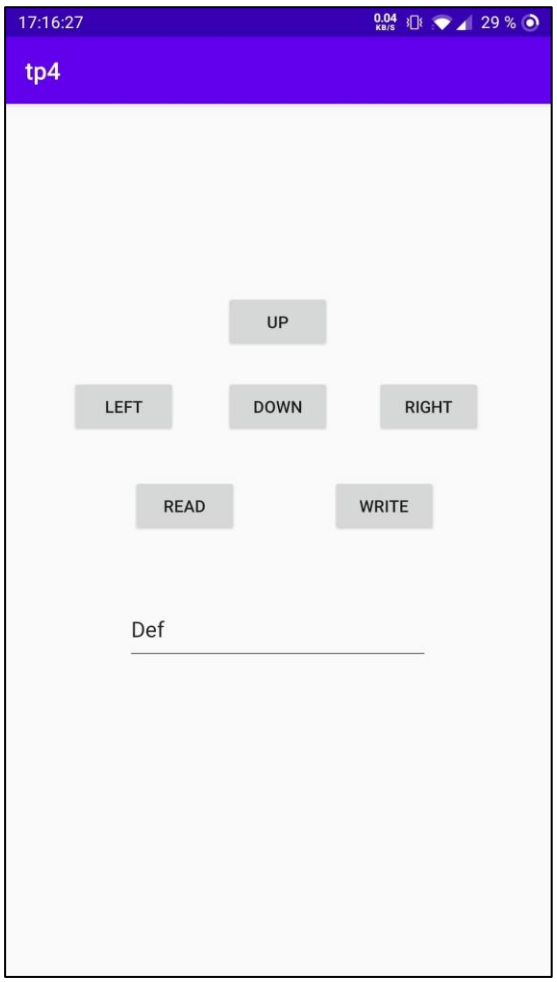
PRODUCT_MODEL := LO52

Pour la surcharge du fichier, nous avons défini la variable DEVICE_PACKAGE_OVERLAY dans le fichier .mk et nous avons ensuite placer ce fichier dans le répertoire suivant : device/utbm/lo52adurycsanchez/overlay/frameworks/base/core/res/res/drawable-hdpi/sym_keyboard_delete.png

Séance 4 : Utilisation de la JNI

Tout d'abord nous avons démarré Android Studio et créé une application C++ native en Java. Une fois cela fait l'IDE nous demandais d'installer le NDK Android nécessaire pour travailler sur ce type d'application. Car en effet celui-ci permet de développer directement dans le langage cible ici le C++ avec quelques fonctions.

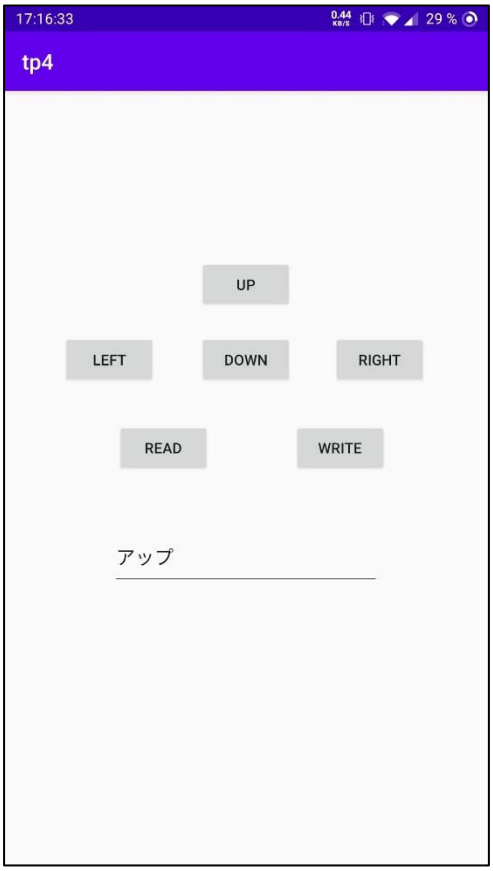
Premièrement nous avons créé les objets (boutons, textView) de l'application. 6 boutons et un PlainText sont alors ajoutés, la valeur du Plaintext est mis par défaut sur Def :



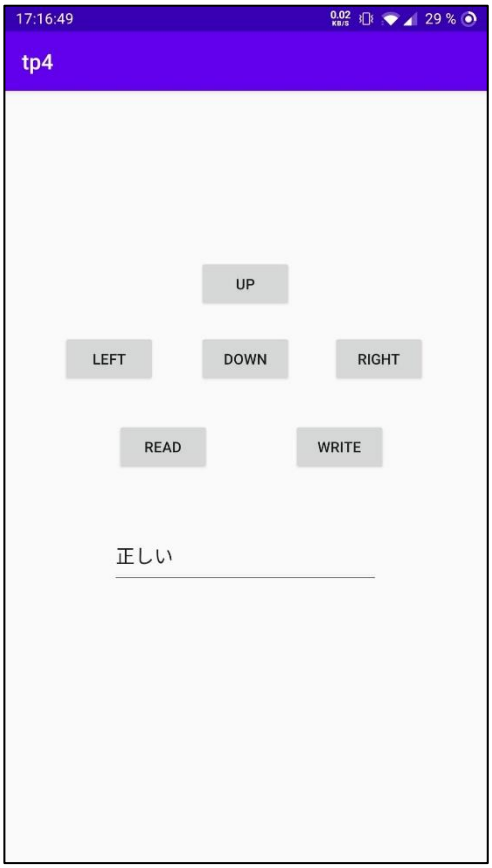
Nous avons ensuite implémenté les trois fonctions évènements qui vont se déclencher lors de l'appui d'un des 6 boutons : OnClick, OnRead et OnWrite.

La première fonction Java permet de récupérer la valeur du texte présent sur le bouton appuyé pour ensuite envoyer cette information à la fonction native C++. Cette fonction permet avec cette information l'affichage du bon résultat dans la PlainText en fonction du bouton. Deux exemples :

Click UP :

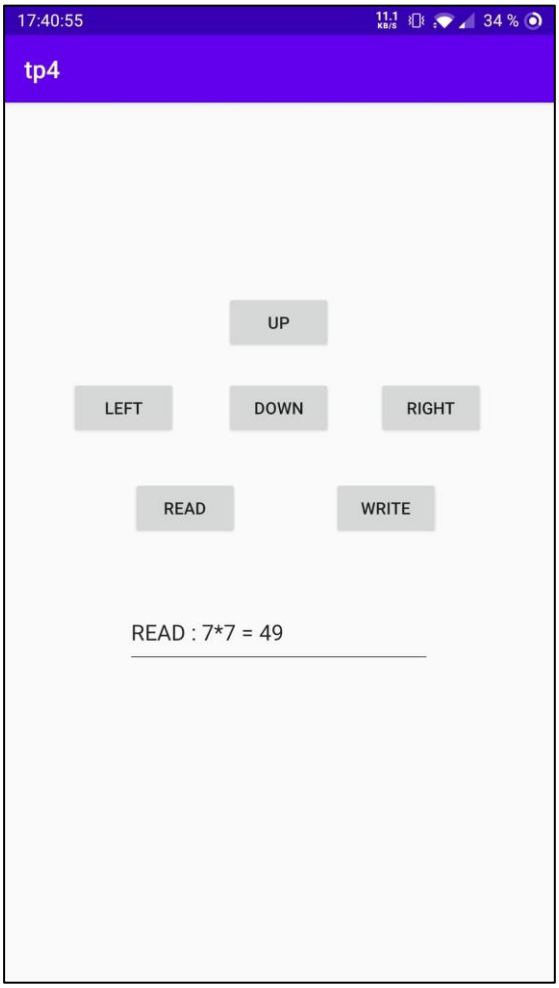


Click RIGHT :

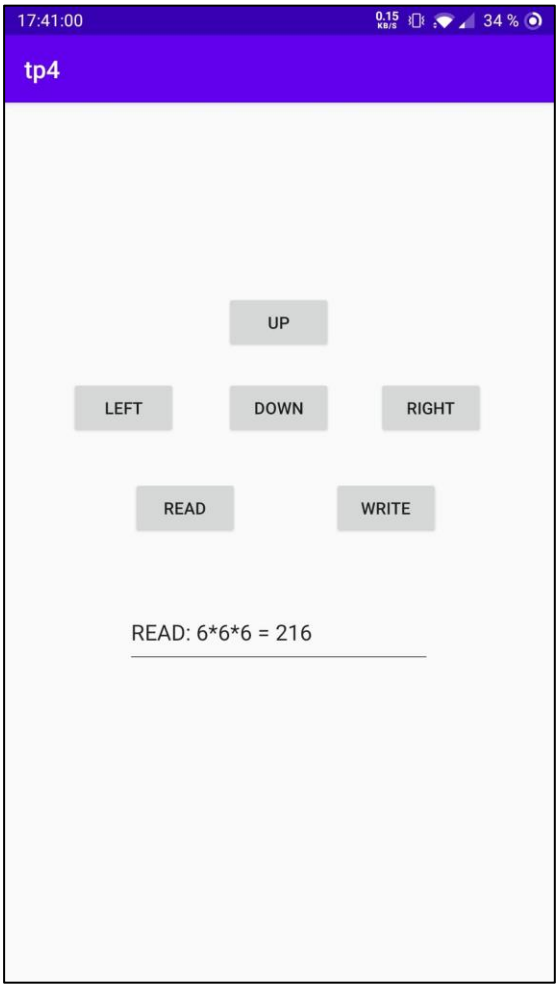


La fonction native DirectionFunction récupère le texte du bouton et en fonction de celui-ci, envoie la traduction japonaise correspondante.

La deuxième fonction Java, OnRead est appelée lors de l'appui sur le bouton Read, elle créera un nombre aléatoire entre 0 et 10 et affichera dans le PlainText le résultat de la fonction native C++ ReadFunction. Cette fonction retournera le carré du nombre avec un formalisme d'affichage.



La troisième fonction Java, OnWrite est appelée lors de l'appui sur le bouton Write, elle créera un nombre aléatoire entre 0 et 10 et affichera dans le PlainText le résultat de la fonction native C++ WriteFunction. Cette fonction retournera le cube du nombre avec un formalisme d'affichage.



Dans notre application, il y a donc 3 types de boutons : le bouton up, left, right, down qui affiche la traduction du mot du bouton en japonais. Le bouton Read qui renvoie le carré d'un nombre aléatoire compris entre 0 et 10. Et le bouton Write qui renvoie le cube d'un nombre aléatoire compris entre 0 et 10.

Nous avons donc dans le mainActivity.java declaré les fonction natives :

public native String DirectionFunction(String buttonName);

public native String ReadFunction(int numberGenerated);

public native String WriteFunction(int numberGenerated);