utbm

université de technologie
Belfort-Montbéliard

LO52

INFORMATIQUE MOBILE ET COMMUNICATIONS COURTES PORTEES

Rapport TPS

COMPTE RENDU DES TPS : TP1, TP2, TP3 et TP4

Rédigé par :

- NOUASSI Zechirine
- MANOU Abbaltu Auguste

Responsable de l'unité de valeur :

- M. BRISSET Fabien

Année académique 2020 / 2021

TABLE DE MATIERES

TABLE D'ILLUSTRATIONS

INTRODUCTION

Android est un système d'exploitation mobile crée par google et qui équipe la majorité des smartphones actuels. D'où son utilisation dans notre unité de valeur LO52 (Informatique mobile et communication courtes portées) que nous suivons dans le cadre de notre cursus d'ingénierie informatique à l'UTBM (Université de Technologie de Belfort Montbéliard).

Pour ce semestre d'automne 2020, les objectifs sont les suivants : la mise en place de l'environnement de développement ensuite, la définition du modèle de données pour une application Android puis, la manipulation du kernel linux, la création d'un device Android enfin, le développement de l'application et l'utilisation du JNI. Ce document retrace les différentes étapes suivies pour la réalisation de ces objectifs.

PARTIE I: MISE EN PLACE DE L'ENVIRONNEMENT DE DEVELOPPEMENT

Le but de cette partie est de nous familiariser avec le gestionnaire de code source Git et l'environnement de développement Android Studio. Pour cela nous avons évolué sous plusieurs étapes. Il sera donc question par la suite de développer toutes ces étapes qui nous ont permis de réaliser cette première partie.

I.1) TELECHARGEMENT, INSTALLATION ET CONFIGURATION DE GIT

Git étant déjà installé sur nos ordinateurs, ils nous restaient qu'à configurer git. Pour ce faire nous avons modifier le fichier de configuration de Git pour y faire figurer notre email UTBM et notre nom d'utilisateur au travers des commandes ci-après :

- git config --global user.name "<user_name>"
- git config --global user.email "<email>"

Clonage du dépôt https://github.com/gxfab/LO52_A2020

Après la configuration de Git, nous avons procédé au clonage du dépôt du TP à travers la commande :

- git clone **https://github.com/gxfab/LO52_A2020**

Cette commande crée un répertoire du même nom que le dépôt distant, initialise un répertoire. git à l'intérieur et récupère toutes les données de ce dépôt.

Création d'une branche

Le dépôt cloné, nous avons créé une branche pour notre groupe afin d'éviter les conflits lors des push des uns et des autres. Nous avons nommé notre branche **ManouAbbaltu_NouassiZechirine** (combinaison de noms des membres de l'équipe). Une fois créé nous nous sommes positionnés dessus à travers la commande :

- git checkout -b ManouAbbaltu_NouassiZechirine

Nous avons exécuté les commandes suivantes afin de mettre à jour le dépôt distant en uploadand nos changements :

- git add . (Pour indexer tout le contenu du répertoire courant)
- git commit -m ''Initial commit''
- git push origin ManouAbbaltu_NouassiZechirine

I.2) CREATION DE NOTRE PREMIERE APPLICATION HELLOWORLD

Cette première application a pour but de constituer deux vues que nous allons appeler activité. La première activité doit afficher un bouton, au clic de ce bouton nous devons naviguer vers une autre activité affichant le label *Hello World*.

Pour le faire nous avons créé un projet sur Android Studio et nous l'avons sauvegardé dans le dossier TP1 du dépôt. A l'aide de l'assistant de création de projet, nous avons pu créer un projet avec déjà une activité vide appelé *Empty Activity*. Le langage de programmation choisi était *Kotlin* car nous avons déjà une bonne base en JAVA mais pas en KOTLIN et donc profiter pour apprendre davantage sur ce langage.

A l'ouverture du projet nous avons déjà le *MainActivity* avec ces composants *MainActivity.kt* (fichier kotlin) et *activity_main.xml* (fichier pour le layout). Dans le fichier *activity_main.xml* nous avons ajouté un bouton et un label.

Afin de mieux gérer nos valeurs, dans le fichier de ressource *String.xml* nous avons ajouté plusieurs valeurs :

```
<resources>
    <string name="app_name">HelloWorld</string>
    <string name="app_message">Hello World</string>
    <string name="welcome_message">Bienvenue dans notre Application</string>
    <string name="button_message">Cliquer ici</string>
</resources>
```

Figure 1: Configuration fichier String.xml

Nous avons également appliqué des contraintes au bouton pour qu'il soit au milieu et le label pour qu'il placé comme un titre.

Ensuite nous avons créé une nouvelle activité avec ces composants bien sûr *HelloWordActivity.kt* et *activity_hello_word.xml* afin qu'il affiche le texte « Hello World »

Une fois les deux activités misent en place, nous avons donc créer la méthode, *afficher* dans le fichier *MainActivity.kt* et lier cette méthode au layout *activity_main.xml* en tant que propriété « OnClick ». Ci-après le code qui nous a permis de réaliser ceci.

```kotlin
package com.lo52.tp1.helloworld

import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.View

class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    fun afficher(view : View){
        val intent = Intent(this, HelloWordActivity::class.java)
        startActivity(intent)
    }
}
```

Figure 2 : Méthode afficher de l'activité MainActivity

```xml
<Button
    android:id="@+id/button"
    android:layout_width="174dp"
    android:layout_height="66dp"
    android:onClick="afficher"
    android:text="@string/button_message"
    android:textSize="14sp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Figure 3: Propriété OnClick affectée au bouton

Cette méthode permet de passer de l'activité *MainActivity* à *HelloWordActivity* grâce à la méthode *startActivity* tout ceci une fois le bouton cliqué.

Nous avons pu tester l'application grâce à notre Android. Nous avons pu constater qu'au click du bouton, il affiche bien la deuxième activité.

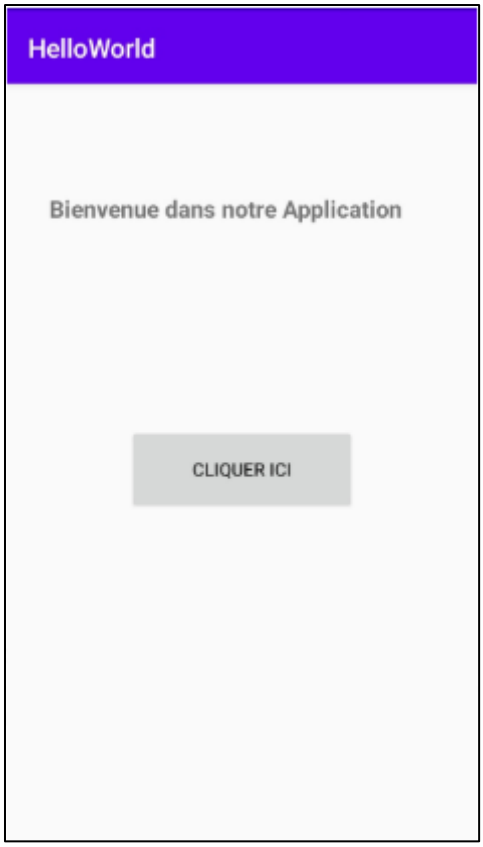

Figure 5: Activité 1

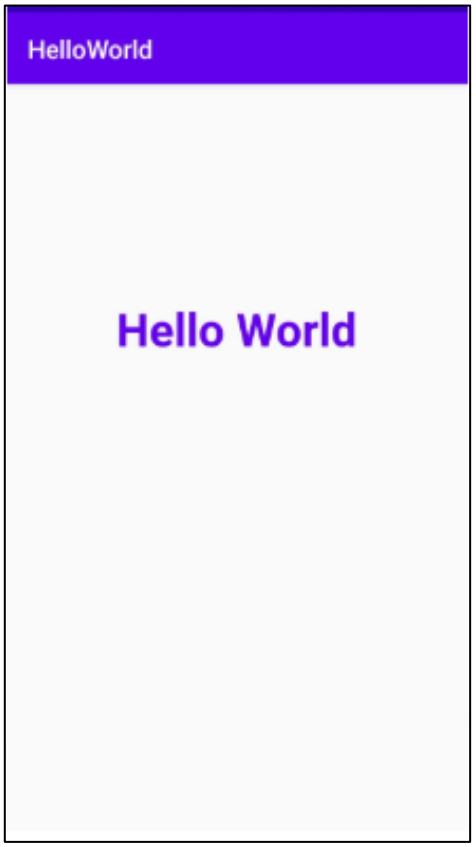

Figure 4 : Activité 2

PARTIE II: MANIPULATION D'UN KERNEL LINUX

II.1) PREPARATION D'UN ENVIRONNEMENT PERMETTANT DE CONFIGURER UN NOYAU HIKEY-LINARO

Pour ce faire, nous avons suivi les recommandations indiquées sur https://source.android.com/setup/build/building-kernels . Nous sommes passés par les étapes suivantes :

Installation de l'outils repo

Repo est un outil aidant à gérer plusieurs dépôts Git. Il automatise certaines parties des flux de développement.

- $ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
- $chmod a+x ~/bin/repo

Création du dossier contenant les fichiers du noyau :

Nous avons créé le dossier à travers la commande suivante.

- $ mkdir android-kernel && cd android-kernel

Initialisation de repo et téléchargement dans le répertoire courant avec la branche *hikeylinaro*

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

Téléchargement des outils de compilation dans un dossier :

Ainsi nous créerons un dossier toolchain dans lequel nous décompresserons le fichier gcc-linaro-7.5.0-2019.12 téléchargé.

- $ mkdir ~/toolchain && 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 aarch64-linux-gnu/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu.tar.xz

Définition de l'architecture CPU et spécification du chemin (jusqu'au préfixe des outils de compilation) :

Cette partie sera essentielle pour la compilation du noyau.

- $ export CROSS_COMPILE=~/toolchain/gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-
- $ export ARCH=arm64

II.2) CONFIGURATION ET COMPILATION (ETAPE BONUS)

Dans cette partie nous compilons notre noyau en la spécifiant les configurations correspondantes.

- **Configuration par défaut relative à un noyau ranchu64 :** cette configuration se retrouve dans le dossier *arch/mips/configs/generic/board-ranchu.config*
- **Chargement de la configuration par défaut :** Pour ce faire nous avons copié le fichier *board-ranchu.config* vers le dossier *arch/arm64/configs* et charger le fichier avec la commande suivante. Bien avant de charger cette configuration, nous avons effectué une sauvegarde de la configuration existante (vers le dossier configuration) à l'aide de la commande ***make savedefconfig***.

  $ make board-ranchu.config
- **Différence entre les deux configurations :** Nous avons utilisé l'outil ***kdiff3*** pour afficher la comparaison des deux configurations dans une interface graphique à travers la commande suivante.

  $ kdiff3 defconfig ../../configurations/defconfig_first_version

  A l'aide de kdiff3, nous avons exporté la comparaison dans le fichier difference.pdf dans le dossier configurations.

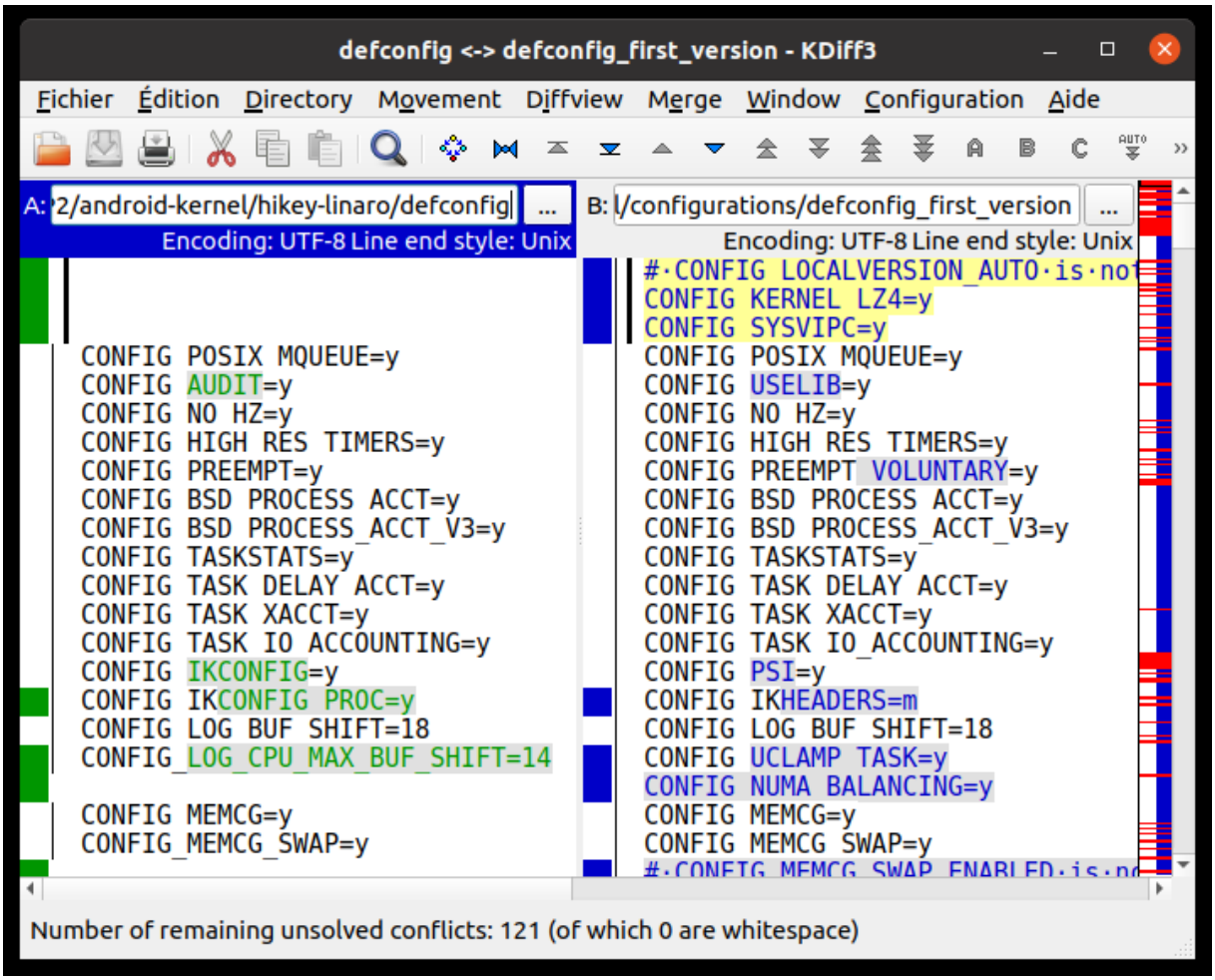

Figure 6: Interface de kdiff3

- **Script d'automatisation des commandes** : Nous avons créé un fichier script.sh dans lequel nous mettons la liste des commandes tapées précédemment afin d'automatiser les commandes précédentes. Ce fichier peut être exécuté à l'aide de la commande suivante :

  $ sh script.sh

- **Modification et Génération de la configuration par défaut :** Nous avons modifier la configuration par défaut à l'aide de la commande **make xconfig.** Cette commande exécute un utilitaire qui nous présente graphiquement. En ce qui concerne la désactivation des options superflues, nous n'avons pas pu détecter quelle option pouvait être considérée comme superflue et nous avons donc désactivé les options suivantes :
  - Bluetooth system support
  - SPI support

- Mailbox Hardware Support

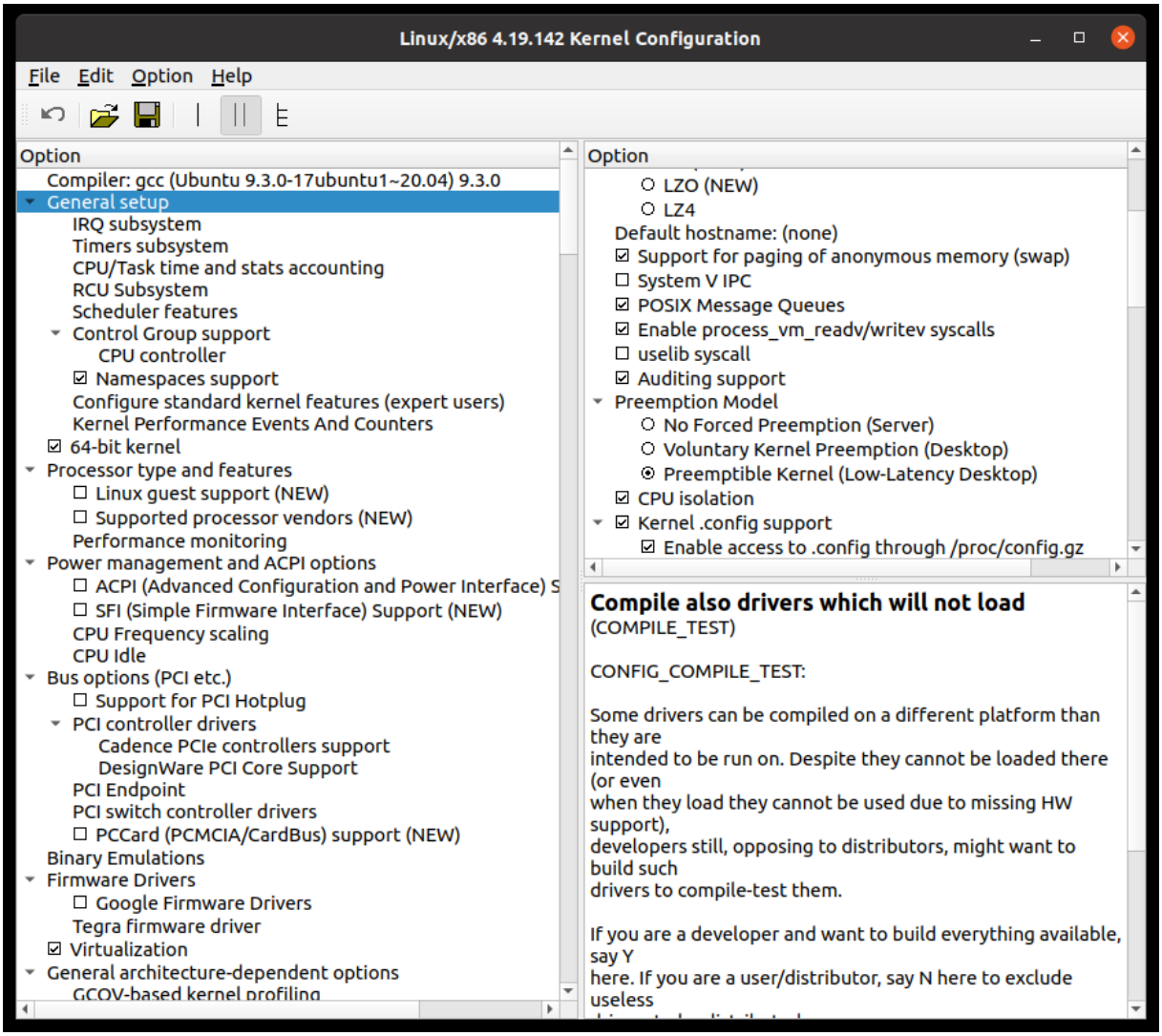

Figure 7: Interface présentée par le make xconfig

- **Compilation (Etape bonus) :** la compilation se fait avec la commande ***make.*** Afin que chaque configuration puisse être compilée, elle doit être chargée par défaut avant l'exécution de la commande ***make***.
- **Envoi par git**

- **Fichier. gitignore :** Afin de pouvoir envoyer uniquement les configurations et les scripts nous avons créé un fichier .gitignore à la racine de notre dossier(TP2) qui contient la liste des fichiers/dossier à ne pas envoyer. Ce fichier contient la ligne « **android-kernel**/ » qui permet d'envoyer tous les fichiers exceptés le dossier android-kernel.
- **Envoi :** Nous envoyons notre projet à travers les commandes suivantes :

- $ git add .
- $ git commit -m “TP2”
- $ git push

PARTIE III: CREATION D'UN DEVICE ANDROID

Dans cette partie il sera question d'écrire des Makefiles relatifs à l'intégration d'un composant et d'un produit Android.

Dans le répertoire TP3 qui est notre espace de travail, se trouvent deux répertoires pour deux tâches distinctes :

- Libusb-1.0.3 : Pour l'implémentation de la libusb ;
- Device : Pour l'implémentation d'un nouveau produit.

III.1) IMPLEMENTATION DE LA LIBUSB

Les fichiers sources et les headers pour la compilation d'une libusb sur Android sont dans le répertoire libusb-1.0.3/libusb. En identifiant, ces deux types de fichiers on obtient :

Fichiers sources :

- Core.c
- Descriptor.c
- Io.c
- Sync.c
- Darwin_usb.c
- Linux_usbfs.c

Fichiers headers :

- Libusb.h
- Libusbi.h
- Darwin_usb.h
- Linux_usbfs.h

L'écriture du fichier Android.mk afin d'implémenter la libusb est présenté dans la figure ci-après :

Pour la réalisation de ce fichier, nous avons utilisé des variables comme *LOCAL_PATH* afin d'identifier du dossier courant, ensuite nous avons procédé au nettoyage des variables précédemment positionnées. Nous avons également utilisé les variables locales afin de définir les sources à compiler grâce à *LOCAL_SRC_FILES* et de la variable créé (*commonSources*) comprenant l'ensemble de ces sources. Nous avons par la suite défini le nom du module au travers de la variable local *LOCAL_MODULE*. Enfin nous avons défini ce que l'on doit produire finalement au travers de *include $(BUILD_SHARED_LIBRARY)*.

```
# Copyright 2020 The Android Open Source Project
# By MANOU Abbaltu Auguste
#  &
#    NOUASSI Zechirine
#
# Android.mk for libusb
#

#set current directory as our LOCAL_PATH variable value
LOCAL_PATH := $(call my-dir)

#clear all env. variables
include $(CLEAR_VARS)

# Define all sources files in commonsSources variable
commonSources := \
        core.c \
        descriptor.c \
        io.c \
        sync.c \
        os/darwin_usb \
        os/linux_usbfs.c

# Setting LOCAL_SRC_FILES value from commonSources
LOCAL_SRC_FILES := \
        $( commonSources )

#Define module name
LOCAL_MODULE := libusb

#Define final build
include $(BUILD_SHARED_LIBRARY)
```

Figure 8: fichier Android.mk

Lors de la compilation une erreur se produit sur la macro TIMESPEC_TO_TIMEVAL. Celle-ci n'est pas définie. Pour la corriger, nous modifions le fichier io.c dans le répertoire libusb en ajoutant les lignes présentées ci-après en début de fichier.

```
35  #ifndef TIMESPEC_TO_TIMEVAL
36  #define TIMESPEC_TO_TIMEVAL(tv, ts) { \
37  (tv)->tv_sec = (ts)->tv_sec; \
38  (tv)->tv_usec = (ts)->tv_nsec / 1000; \
39  }
40  #endif
```

Figure 9:Lignes à ajouter dans io.c

Une deuxième erreur survient : build/tools/apriori/prelinkmap.c(137) : library "libusb.so" not in prelink map.

III.2) IMPLEMENTATION D'UN NOUVEAU PRODUIT ANDROID : LO52_MANOUABBALTU_NOUASSIZECHIRINE

Pour l'ajout du nouveau produit, nous avons créé l'arborescence suivante sous TP3/device/utbm/ LO52_MANOUABBALTU_NOUASSIZECHIRINE :

- overlay/
- AndroidProducts.mk
- BoardConfig.mk
- CleanSpec.mk
- LO52_MANOUABBALTU_NOUASSIZECHIRINE.mk
- vendorsetup.sh

Etant donné que ce produit héritera du produit hikey de Linaro, nous récupérons les sources du device hikey à l'adresse https://android.googlesource.com/device/linaro/hikey avec la commande git clone. Nous copions ensuite les fichiers dont nous aurons besoin vers TP3/device/linaro/hikey. Nous obtenons alors l'arborescence suivante :

- hikey/BoardConfig.mk
- BoardConfigCommon.mk
- hikey.mk

Ci-dessous le contenu des différents fichiers créés :

```
1   PRODUCT_MAKEFILES := $(LOCAL_DIR)/lo52_ManouAbbaltu_NouassiZechirine.mk
```

Figure 10: fichier AndroidProducts.mk

```
1  include device/linaro/hikey/hikey/BoardConfig.mk
```

Figure 11; fichier BoardConfig.mk

```
1  $(call add-clean-step, rm -f $(PRODUCT_OUT)/system/build.prop)
```

Figure 12: fichier CleanSpec.mk

```
1  $(call inherit-product, device/linaro/hikey/hikey.mk)
2  PRODUCT_PROPERTY_OVERRIDES := \
3                                                ro.hw = lo52 \
4                                                net.dns1 = 8.8.8.8 \
5                                                net.dns2 = 4.4.4.4
6  DEVICE_PACKAGE_OVERLAYS := device/utbm/lo52_Manou_Abbaltu_Nouassi_Zechirine/overlay
7  PRODUCT_PACKAGES += libusb
8  PRODUCT_NAME := lo52_Manou_Abbaltu_Nouassi_Zechirine
9  PRODUCT_BRAND := lo52_Manou_Abbaltu_Nouassi_Zechirine
10 PRODUCT_DEVICE := lo52_Manou_Abbaltu_Nouassi_Zechirine
11 PRODUCT_MODEL := lo52_Manou_Abbaltu_Nouassi_Zechirine
```

Figure 13: fichier lo52_ManouAbbaltu_NouassiZechirine.mk

```
1  add_lunch_combo lo52_ManouAbbaltu_NouassiZechirine-eng
2  add_lunch_combo lo52_ManouAbbaltu_NouassiZechirine-user
3  add_lunch_combo lo52_ManouAbbaltu_NouassiZechirine-userdebug
```

Figure 14: fichier vendorsetup.sh

Le fichier sym_keyboard_delete.png, se trouve dans le projet à l'adresse https://android.googlesource.com/platform/frameworks/base. Nous avons donc récupéré les sources via git clone. Le fichier se trouve dans le répertoire. L'arborescence du dossier utbm/lo52_Manou_Abbaltu_NouassiZechirine/overlay est la même que celle des sources Android. Nous l'avons donc fait ainsi : utbm\Manou_Abbaltu_NouassiZechirine\overlay\frameworks\base\core\res\res. Le répertoire res contient d'autres répertoires utilisés en fonction du type d'écran :

- drawable-en-hdpi/sym_keyboard_delete.png

- drawable-en-ldpi/sym_keyboard_delete.png

- drawable-en-mdpi/sym_keyboard_delete.png

- drawable-ldpi/sym_keyboard_delete.png

- drawable-mdpi/sym_keyboard_delete.png

- drawable-xhdpi/sym_keyboard_delete.png

- drawable-xxhdpi/sym_keyboard_delete.png

PARTIE IV: UTILISATION DE JNI

Dans cette partie nous allons écrire une librairie c++ qui sera manipulée à travers une application de type NDK. Dans notre cas nous avons utilisé **Andoid Studio.**

IV.1) INSTALLATION ET CONFIGURATION

Pour ce faire, nous avons installé les composant de NDK dans les paramètres en cliquant sur l'icône d'Android Sdk. Il suffira de cocher sur NDK et CMake (qui permet avec Gradle de créer une librairie native) et valider en cliquant sur **ok**.

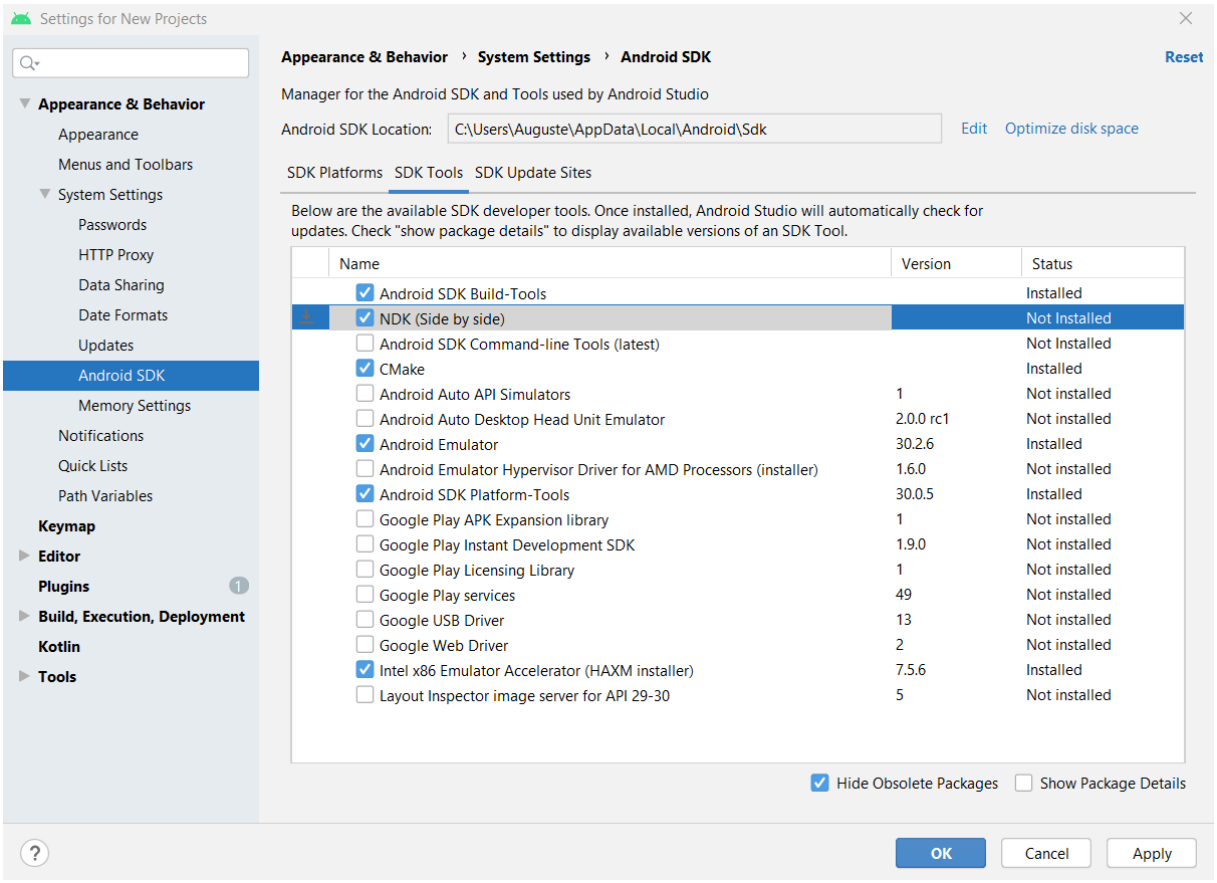

Figure 15: Installation des composants de NDK et CMake

Ensuite nous avons suivi les étapes suivantes :

- Création de notre libraire c++ dans un fichier .cpp en faisant un clic droit sur le dossier du fichier(cpp), **new** →file → C/C++ source file.
- Ajout d'un fichier CMakeList.txt avec les configurations suivantes :

```
# Definition de la version minimale de CMake requise
cmake_minimum_required(VERSION 3.4.1)

# Specification de notre library écrite en c++
add_library( # Nom de notre library.
             native-lib

             # Définition en tant que library partagée.
             SHARED

             # Chemin relatif de notre fichier source
             src/main/cpp/native-lib.cpp )

find_library( # Nom de la variable qui
              #contient le chemin de la librairie
              log-lib

              # Nom de la librairie NDK
              log )

# Liaison de notre libraie native au Log
target_link_libraries( # Notre librairie
                       native-lib

                       # Librairie log
                       ${log-lib} )
```

Figure 16: cmakelist.txt

- Déclaration de notre fichier CMakeList.txt dans le fichier build.gradle

```
    externalNativeBuild {
        cmake {
            path "CMakeLists.txt"
        }
    }
```

Figure 17: ajout du fichier cmakelist

Après cela, nous pouvons passer à l'implémentation de notre application.

IV.2) IMPLEMENTATION

IV.2.1) Interface graphique

Nous avons une activité principale qui est liée à un layout (vue) dans lequel sont définis les boutons qui s'afficheront à l'écran.

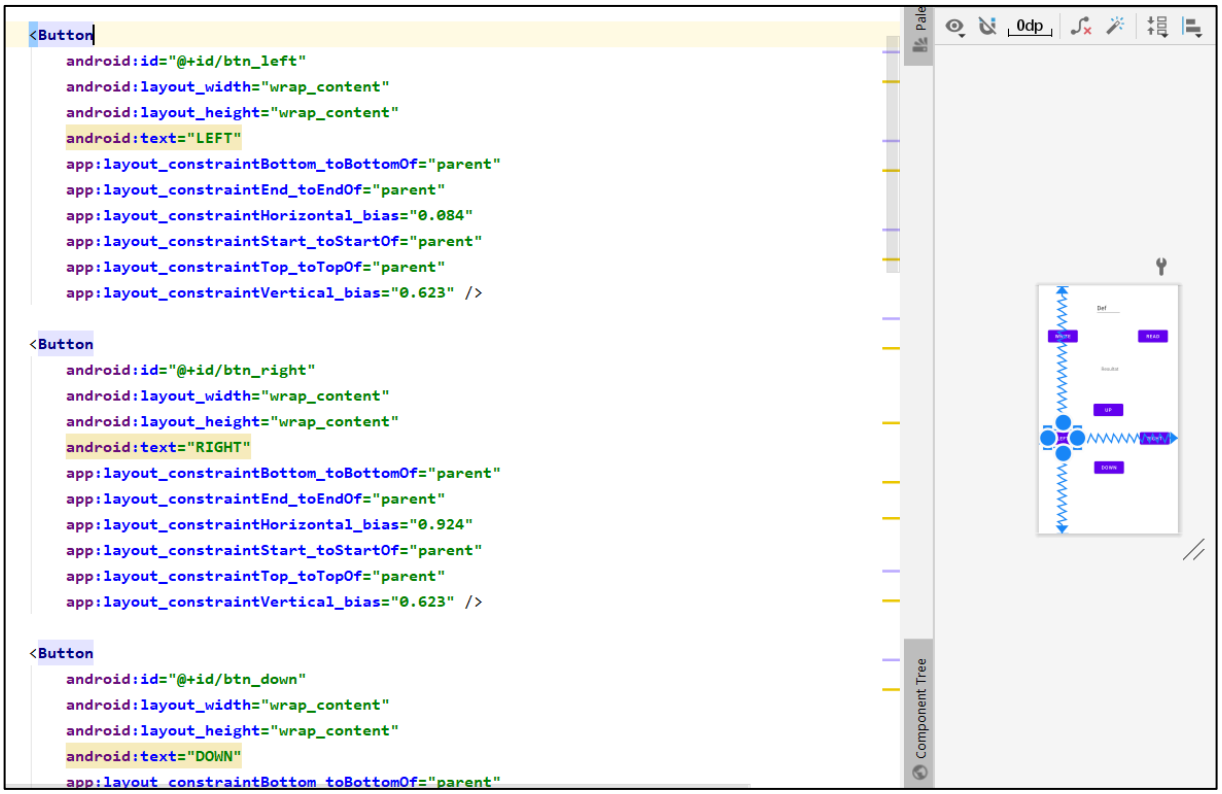

Figure 18: layout de notre activité

Notre activité créée en Kotlin récupère les informations de notre layout et manipule les action à effectuer lors de la manipulation des objets de la vue (boutons, textview et edittext).

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val btnWrite = findViewById<Button>(R.id.btn_write)
    btnWrite.setOnClickListener { fn_write() }

    val btnRead = findViewById<Button>(R.id.btn_read)
    btnRead.setOnClickListener { fn_read() }

    val btnUp = findViewById<Button>(R.id.btn_up)
    btnUp.setOnClickListener { fn_translate_japonais( mot: "UP") }

    val btnDown = findViewById<Button>(R.id.btn_down)
    btnDown.setOnClickListener { fn_translate_japonais( mot: "DOWN") }

    val btnLeft = findViewById<Button>(R.id.btn_left)
    btnLeft.setOnClickListener { fn_translate_japonais( mot: "LEFT") }

    val btnRight = findViewById<Button>(R.id.btn_right)
    btnRight.setOnClickListener { fn_translate_japonais( mot: "RIGHT") }

}
```

Figure 19: lien avec les objets depuis notre activité

IV.2.2) Importation de notre librairie et utilisation

Dans notre activité nous avons importé les fonctions de notre librairie grâce au code suivant :

```
private external fun translateJaponais(direction: String): String
private external fun write(a: Int): String
private external fun read(a: Int): String

companion object {
    // Chargement de la librairie native 'native-lib' au démarrage de l'application.
    init {
        System.loadLibrary( libname: "native-lib")
    }
}
```

Figure 20: import de notre librairie

Une fois notre librairie importée, nous pourrons utiliser les fonctions natives derrière à utiliser au clic de nos boutons.

- Fonction fn_write : cette fonction est exécutée au clic du bouton btnWrite. Dans cette fonction nous utilisons la fonction **write** native pour retourner le message à afficher en fonction du nombre entré.

```
fun fn_write(){
    val editDef = findViewById<EditText>(R.id.saisie)
    val resultat = findViewById<TextView>(R.id.result_textview)
    try {
        val nombre =  editDef.text.toString().toInt()
        val text = write( nombre )
        resultat.setText( text )
    }
    catch (e: NumberFormatException) {
        resultat.setText( "Erreur: Veuillez entrer un nombre valide" )
    }
}
```

Figure 21: appel de la fonction native write

```
extern "C" JNIEXPORT jstring JNICALL
Java_com_utbm_nouassi_manou_ndktp4_MainActivity_write(
        JNIEnv* env,
        jobject, jint a)  {
    std::stringstream ss;
    if(a < 0 || a > 10){
        ss << "ERREUR: le nombre saisi doit être compris entre 0 et 10";
    }
    else{
        ss << "READ: " << a << "*" << a  << "*" << a << "=" << a*a*a;
    }

    return env->NewStringUTF(ss.str().c_str());
}
```

Figure 22: Foncion native write

- Fonction fn_read : cette fonction est exécutée au clic du bouton btnRead. Dans cette fonction nous utilisons la fonction native **read** pour retourner le message à afficher en fonction du nombre entré.

```
fun fn_read(){
    val editDef = findViewById<EditText>(R.id.saisie)
    val resultat = findViewById<TextView>(R.id.result_textview)
    try {
        val nombre =  editDef.text.toString().toInt()
        val text = read( nombre )
        resultat.setText( text )
    }
    catch (e: NumberFormatException) {
        resultat.setText( "Erreur: Veuillez entrer un nombre valide" )
    }
}
```

Figure 23: appel de la fonction read

```
extern "C" JNIEXPORT jstring JNICALL
Java_com_utbm_nouassi_manou_ndktp4_MainActivity_read(
        JNIEnv* env,
        jobject, jint a)  {
    std::stringstream ss;

    if(a < 0 || a > 10){
        ss << "ERREUR: le nombre saisi doit être compris entre 0 et 10";
    }
    else{
        ss << "READ: " << a << "*" << a << "=" << a*a;
    }

    return env->NewStringUTF(ss.str().c_str());
}
```

Figure 24: fonction read

- Fonction fn_translateJaponais : cette fonction est exécutée au clic d'un des boutons LEFT, RIGHT, UP, DOW. Dans cette fonction nous utilisons la fonction native **translateJaponais** pour retourner le message à afficher en Japonais.

```
fun fn_translate_japonais(mot: String){
    val resultat = findViewById<TextView>(R.id.result_textview)
    resultat.setText( translateJaponais(mot) )
}
```

Figure 25: Appel de la fonction translateJaponais

CONCLUSION

Nous voici à la fin de ce projet. Tout au long de celui-ci, nous avons mis en place l'environnement de développement pour Android en installant entre autres git et Android Studio. Par la suite, nous avons manipulé le kernel Linux et ajouté de nouveaux composants Android. Cela étant, nous avons découvert l'univers de la programmation Android avec Kotlin et le NDK. Ce projet a été enrichissant pour nous car nous avons découvert des notions qui nous étaient inconnues et développer en un langage nouveau pour nous comme Kotlin. Les difficultés rencontrées furent au niveau de la manipulation du kernel linux (Partie 2), principalement l'ajout de la toolchain. Mais avec des recherches, nous avons pu trouver la solution et avancer, ce qui est d'ailleurs naturel pour des futurs Ingénieurs (être confrontés à des difficultés et à les surmonter). En plus, nous avons pu adopter des méthodes pour le travail en équipe afin d'optimiser le travail. Git étant une plateforme idéale pour le travail en équipe.