



UTBM – *Université de Technologie de Belfort-Montbéliard*

A2020

L052 - Rapport Travaux Pratiques

TP1 : Mise en place de l'environnement de développement	2
1. Environnement de base	2
2. Découverte d'Android Studio et développement d'une application HelloWorld.....	3
TP2 : Manipulation d'un Kernel Linux.....	5
1. Environnement de configuration du Kernel	5
2. Nouvelle configuration de Kernel.....	6
3. Différences entre les configurations.....	6
4. Bonus : Création d'images	7
TP3 : Création d'un device Android	8
1. Implémentation de la Libsub.....	8
2. Définition d'un nouveau produit Android.....	10
TP4 : Utilisation de JNI.....	12
1. Mise en place de l'ensemble des composants nécessaires pour le NDK Android	12
2. Codage de l'Application NDK.....	12

Auteurs

Anaïs JARNO
Aymeric ROBITAILLE

Encadrants

Fabien BRISSET
Pierre ROMET

TP1 : Mise en place de l'environnement de développement

1. ENVIRONNEMENT DE BASE

L'objectif, ici, est de mettre en place un environnement propice au développement d'applications Android.

La première étape était de se procurer un dépôt Git pour la gestion des sources.

Pour cela, il faut bien évidemment commencer par installer Git si ce n'est pas fait. Dans notre cas, Git était déjà installé sur nos machines. Aussi, il nous restait juste à configurer Git pour nous identifier lors des commits. Ainsi, nous avons exécuter les commandes suivantes pour la configuration de notre nom et adresse mail :

```
git config --global user.name "xxx"  
git config --global user.email xxxx@xxx.com
```

Ensuite, le dépôt Git avait déjà été créé pour tous les élèves de l'UV LO52. Il nous fallait donc le récupérer en local. Nous avons ainsi cloné le dépôt à l'aide de la commande suivante :

```
git clone https://github.com/gxfab/LO52\_A2020.
```

Cela a créé un dossier `LO52_A2020`, lié à nos sources Git, dans lequel nous nous sommes positionnés pour la suite du travail.

Pour différencier notre travail de celui des autres groupes, nous avons créé une branche portant le nom des différents membres de notre groupe de travail, et nous nous sommes directement positionnés dessus pour être prêts à débiter le travail :

```
git checkout -b RobitailleAymeric_JarnoAnais
```

Enfin, après avoir reporté dans le fichier texte TP1.txt, les commandes réalisées jusque-là, nous avons mis à jour le dépôt distant en uploadant nos changements (branche et modifications) :

```
git add . && git commit -m "TP1 Création de branche »  
git push -u origin RobitailleAymeric_JarnoAnais
```

La seconde étape consistait à installer/configurer l'environnement de développement Android Studio.

À ce niveau-là, nous avons eu un problème général concernant le proxy UTBM et l'installation du SDK. Bien que ces soucis aient été résolus par la suite, nous avons eu le temps d'installer Android Studio sur nos machines personnelles et avons donc poursuivi le TP dessus. La configuration fut rapide, elle est simple et très bien guidée par l'assistant d'installation de l'IDE.

2. DÉCOUVERTE D'ANDROID STUDIO ET DÉVELOPPEMENT D'UNE APPLICATION HelloWorld

L'objectif de cette deuxième partie de TP était de prendre en main l'IDE, comprendre le fonctionnement et la liaison entre les fichiers d'une application réalisée avec Android Studio, via le développement d'une application HelloWorld.

Cette application devait se constituer de deux principales vues, ici on nommera ces vues des Activités. Elle doit comprendre une première Activité ne possédant qu'un simple bouton. Et sur clic de ce bouton, on est redirigé vers une seconde Activité comprenant le texte « Hello World ».

Pour cela, nous avons tout d'abord créé un projet que nous avons placé dans le dossier sources de TP1. L'assistant de création de projet, nous a permis de créer directement un projet comprenant une activité vide – *Empty Activity*. Nous avons choisi de travailler avec le langage Kotlin ce qui nous permet de suivre les dernières recommandations de Google et donc les tendances de programmation actuelles.

Nous disposons ainsi déjà d'une activité MainActivity. Celle-ci comprend deux composants : MainActivity.kt (fichier kotlin présent dans le dossier de sources java) et activity_main.xml (dans les ressources, plus précisément dans le dossier de layout).

Le fichier activity_main.xml détaille les composants de la vue. Nous lui avons donc simplement ajouté un Button depuis l'assistant graphique et design. Dans le fichier de ressource string.xml, nous avons ajouté une nouvelle entrée, navigate_hello_activity correspondant au texte « Naviguer vers une nouvelle activité ». Nous avons ensuite assigné ce string à la propriété « text » du bouton de MainActivity. Finalement, nous avons contraint ce bouton de manière à ce qu'il soit au milieu de l'écran.

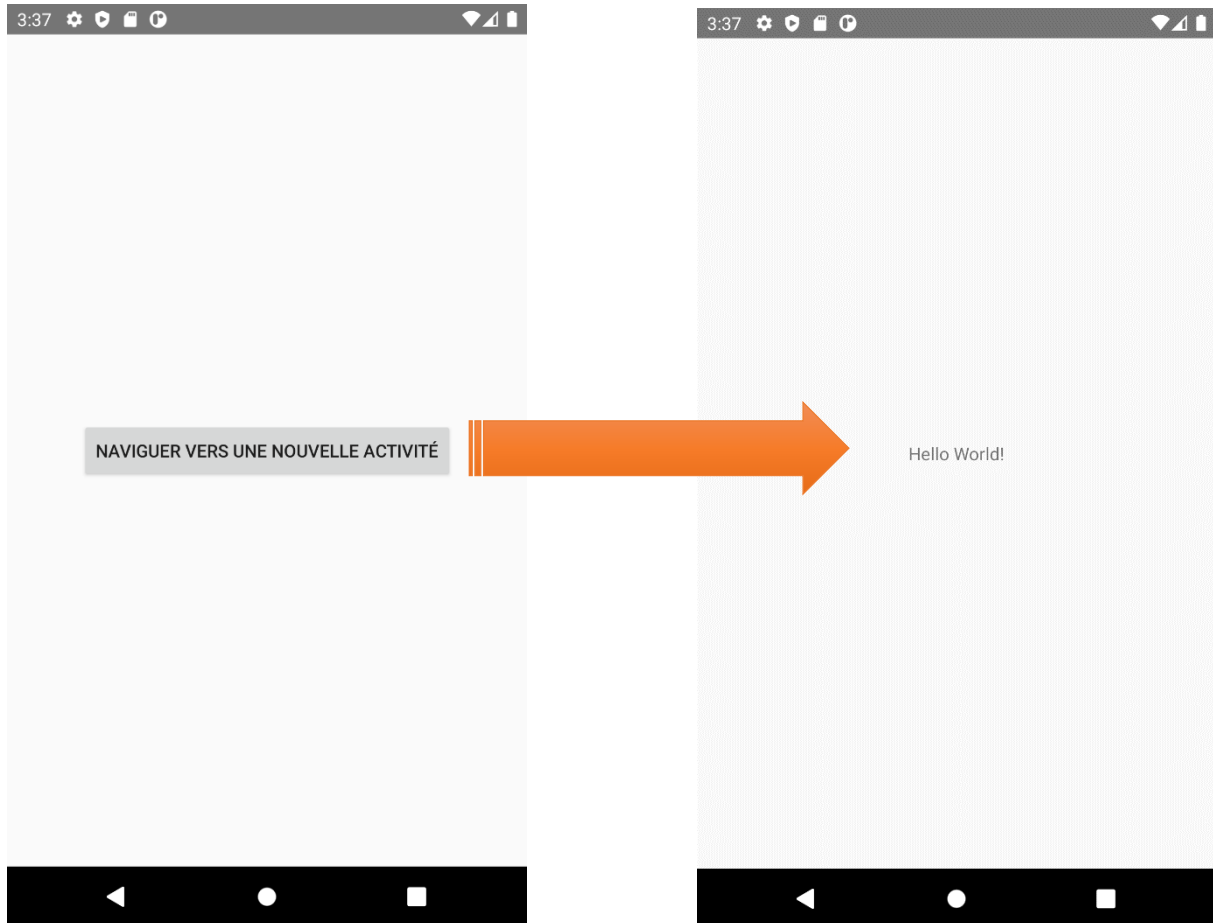
Ensuite, nous avons créé une autre activité dans laquelle nous avons ajouté un TextView avec le texte « Hello World » et les mêmes contraintes, pour qu'il soit au milieu.

Nos deux activités étant à présent créées, nous avons ouvert le code de MainActivity.kt pour y ajouter la méthode « navigateToHello ». Nous avons ensuite lié cette méthode au layout du MainActivity en tant que propriété « onClick » du bouton. De cette manière, le code sera exécuté lorsque le bouton est appuyé. Le code de cette méthode est simple et se résume à un changement d'Activité en appelant la méthode « startActivity » de la manière suivante :

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

Nous avons pu tester l'application aussi bien sur un téléphone Android que sur un émulateur directement lié à l'IDE. Après avoir lancé l'application, nous avons pu constater que l'activité avec le texte était bien lancée lors de l'appui sur le bouton.

Ainsi, on obtient le résultat suivant :



TP2 : Manipulation d'un Kernel Linux

1. ENVIRONNEMENT DE CONFIGURATION DU KERNEL

L'objectif, ici, est de mettre en place un environnement propice au développement d'applications Android.

```
sudo apt-get install repo
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
```

Ensuite, pour informer le système de l'architecture utilisé et éviter de définir celle-ci à chaque commande, nous exportons de manière globale (au terminal en cours) la variable ARCH à arm64 qui correspond au type de processeur demandé.

```
export ARCH=arm64
```

L'étape suivante consiste à fusionner la configuration d'une carte ranchu, qui ne définit que quelques paramètres particuliers, avec une configuration bien plus complète. Dans un premier temps on copie donc la configuration pour processeur MIPS dans les configurations pour processeur ARM64, afin que les commandes make le trouve.

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

Puis, on nettoie les sorties de compilation et les configurations chargés à l'aide des commandes suivantes :

```
make clean
make mrproper
```

Ce qui nous permet alors de passer au chargement de la configuration de base pour hikey :

```
make hikey_defconfig.
```

À présent, il nous fallait identifier la configuration par défaut relative à un noyau pour une carte ranchu64. C'était donc la configuration nommée `board-ranchu.config`, que nous avons copié précédemment et nous l'avons chargée avec la commande : `make oldconfig board-ranchu.config`.

Pour observer la différence entre les différentes configurations, il nous a suffi de lancer la commande `diff <nom-fichier1> <nom-fichier2>`. On observe donc, après chargement de la configuration pour la carte ranchu, l'activation d'options liées à Goldfish.

On choisit ensuite de faire une sauvegarde de la fusion des deux configurations.

```
make savedefconfig
mv defconfig arch/arm64/configs/ranchu_defconfig
```

Ainsi, nous avons regroupé ces diverses commandes dans un script auquel on a ajouté la commande `set -e` pour interrompre le script en cas d'erreur.

2. NOUVELLE CONFIGURATION DE KERNEL

Par la suite, nous avons à modifier la configuration précédemment obtenue afin d'assurer certaines fonctionnalités. Pour cela, nous avons donc dû :

- Assurer la compatibilité pour la carte ARMv8 Versatile, Qualcomm et Realtek
- Activer le NFC et le protocole NFC HCI
- Activer l'option Frequency Scaling de la CPU
- Activer le support de l'HDMI CEC et activer le support LED

Pour ce faire nous utilisons l'éditeur graphique intégré au makefile, grâce à la commande :
`make xconfig`

Par la suite, il nous a été demandé d'optimiser la configuration en désactivant toutes les autres options superflues. Au premier abord, il nous a été très complexe de savoir si une option pouvait, ou non, être considérée comme « superflue ». Finalement, voici les différentes optimisations qui ont été mises en œuvre :

- Suppression des plateformes non demandées
- Désactivation des options qui suivent :
 - PCI, *on suppose ici qu'aucun slot PCI n'est disponible sur la carte*
 - Support de l'UEFI
 - Bluetooth
 - Plan 9 Ressource sharing
 - Carte son
 - Support de carte SD/MMC/SDIO
 - DRI, *étant donné que nous n'avons aucune information particulière sur le module de la carte graphique*



3. DIFFÉRENCES ENTRE LES CONFIGURATIONS

Nous avons enregistré les différentes configurations dans un dossier nommé *configs/* du TP2.

On retrouve ainsi les fichiers de configuration sous les noms suivants :

- *hikey*, la configuration de base
- *ranchu*, la configuration de base merge avec *board-ranchu.config*
- *custom*, la configuration de ranchu avec les activations demandés (armv8, hdmi, nfc, led, ...)
- *custom_opti*, la configuration custom suite à la suppression d'options superflues

Quant aux différences remarquées entre les différentes configurations, on peut les retrouver dans le tableau suivant :

Configuration	Hikey	Ranchu	Custom	Custom_opti
Hikey				
Ranchu	Ajout de la configuration pour une carte ranchu <ul style="list-style-type: none"> activation d'options liées à GoldFish 			
Custom		Activation des options telles que : ARMv8, HDMI, NFC, LED...		
Custom_opti			Suppression des options superflues	

4. BONUS : CRÉATION D'IMAGES

Les images compressées ont été enregistrées dans le dossier *boot/* du TP2.

Pour obtenir ces images il a fallu dans un premier temps installer une toolchain permettant la cross-compilation de notre noyau, le choix se porte ici sur la toolchain bien connu : aarch64-linux-gnu. On utilise les commandes suivantes pour l'ajouter à notre système dans le dossier toolchain situé à la racine du dossier utilisateur :

```
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 -xvf gcc-linaro-7.5.0-2019.12-x86_64_aarch64-linux-gnu.tar.xz
```

Ensuite, tout comme pour la variable ARCH, il est nécessaire d'indiquer qu'elle chaîne de compilation doit être utilisé, on fait donc :

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

On peut enfin procéder à la compilation avec un simple :

```
make -j 8
```

On copie ensuite les fichiers images vers le dépôt git pour les garder en mémoire et procéder à la comparaison, ces derniers sont brut de compilation, c'est à dire compressé. Nous pouvons ainsi observer des changements de taille du kernel en fonction de l'ajout ou la suppression de fonctionnalités, validant bien les changements.

TP3 : Création d'un device Android

Durant ce TP, nous avons dû écrire des Makefile Android relatifs à l'intégration d'un composant et d'un produit Android.

1. IMPLÉMENTATION DE LA LIBSUB

Première chose du TP3, il nous a fallu déterminer les fichiers sources et les headers pour la compilation d'une libusb sur Android. Ceux-ci sont les suivants :

FICHIER SOURCES	HEADERS
➤ core.c	➤ libusb.h
➤ descriptor.c	➤ libusb.h
➤ io.c	➤ os/darwin_usb.h
➤ os/darwin_usb.c	➤ os/linux_usbfs.h
➤ os/linux_usbfs.c	
➤ sync.c	

Ensuite, il nous était demandé d'écrire le fichier Android.mk. On remarque deux fichiers Android.mk dans le dossier du TP3.

a. libusb-1.0.3/Android.mk

Le fichier Android.mk contenu dans le dossier libusb-1.0.3 contient les lignes suivantes :

```
LOCAL_PATH := $(call my-dir)
ubdirs := $(addprefix $(LOCAL_PATH)/,$(addsuffix /Android.mk, \
    libusb \
))
include $(subdirs)
```

Nous avons, tout d'abord, cherché à le comprendre. Ainsi, nous l'avons décomposé ligne par ligne pour en comprendre son fonctionnement :

- La première ligne `LOCAL_PATH := $(call my-dir)` est indispensable au début d'un fichier Android.mk. Elle correspond à la définition de la variable `LOCAL_PATH`, qui sert à localiser les fichiers sources. Ici, `my-dir` correspond au chemin du répertoire actuel, contenant le fichier Android.mk lui-même.
- La deuxième ligne correspond aux fichiers à inclure :
 - La fonction `addprefix` permet d'ajouter un préfixe dans la variable des fichiers à ajouter. Ici, on utilise la variable `LOCAL_PATH`, donc tous les fichiers devront être situés dans le dossier par la variable.

- La fonction `addsuffix` permet d'insérer des suffixes dans les fichiers à ajouter. Ici, on ajoute donc tous les fichiers *Android.mk* présents là où le suffixe fait référence, c'est-à-dire le dossier courant, et également dans le dossier *libusb*.
- Cela permet donc de désigner le chemin vers tous les autres fichiers *Android.mk* à traiter.
- La troisième et dernière ligne permet donc d'inclure les fichiers décrits à la ligne 2, et enregistrés dans la variable *subdirs*. *On notera d'ailleurs une faute à la ligne 2 où la variable est nommée ubdirs et non subdirs, faute que nous avons donc corrigée.*
- Pour simplifier considérablement la syntaxe, ce fichier peut également être réduit à une simple ligne : `include $(all-subdir-makefiles)`

b. **libusb-1.0.3/libsub/Android.mk**

Le second fichier *Android.mk*, contenu dans le dossier *libsub*, est vide et sera compilé comme module grâce au précédent fichier. Nous l'avons ainsi complété de la manière suivante :

```
# Get the path of the current directory, containing the Android.mk file itself
LOCAL_PATH := $(call my-dir)

# Clear variables defined by another module
include $(CLEAR_VARS)

# Define the name of the module to build
LOCAL_MODULES := libusb

# Specify a list of paths to include when compiling all sources
LOCAL_C_INCLUDES += $(LOCAL_PATH) $(LOCAL_PATH)/os

# Enumerate the source files
LOCAL_SRC_FILES := core.c descriptor.c io.c sync.c os/darwin_usb.c os/linux_usbfs.c

# Collect all the information about the module from the LOCAL_XXX variables
# And determine how to build it
include $(BUILD_SHARED_LIBRARY)
```

Nous avons recherché les informations nécessaires à la réalisation de ce fichier dans la documentation https://developer.android.com/ndk/guides/android_mk. Nous l'avons également commenté pour nous rappeler de l'utilité de chaque commande.

Ainsi, notre fichier *Android.mk* configuré comme ci-dessus permet de « build » le module en prenant en compte les fichiers sources et les headers nécessaires à la compilation d'une *libusb* sur Android, comme précédemment déterminé.

c. Erreurs lors de la compilation

Une première erreur se produit sur la macro `TIMESPEC_TO_TIMEVAL`. Celle-ci n'étant pas définie. Pour corriger l'erreur il nous a suffi de définir `TIMESPEC_TO_TIMEVAL` en rajoutant dans `io.c` les lignes suivantes :

```
# define TIMESPEC_TO_TIMEVAL(tv,ts)
do {
    (tv)-> tv_sec =(ts)->tv_sec;
    (tv)-> tv_usec =(ts)->tv_nsec/ 1000
} while ( 0 )
```

Une seconde erreur subvient : `build/tools/apriori/prelinkmap.c(137) : library "libusb.so" not in prelinkmap`. Pour la résoudre, nous avons trouvé qu'il fallait ajouter l'adresse de la librairie `libusb` dans le fichier `build/tools/apriori/prelinkmap`.

Cette solution était valable sur les versions antérieures. Malheureusement, nous avons une version d'AOSP trop récente (branche master) et avons donc eu une erreur lors de la compilation, probablement due à la migration vers un nouveau système de compilation Soong. Notre solution semble tout de même être la bonne.

2. DÉFINITION D'UN NOUVEAU PRODUIT ANDROID

Dans cette partie nous devons nous occuper de la définition d'un nouveau produit Android.

a. Définition de base du produit

Celui-ci devait, tout d'abord, répondre aux contraintes suivantes :

- Le nom de produit doit être `lo52_nom branche`
Ainsi, dans notre cas : `lo52_RobitailleAymeric_JarnoAnais`
- Et il doit hériter du produit `hikey` de Linaro

On commence ainsi par créer l'arborescence suivante : «

`/device/utbm/lo52_RobitailleAymeric_JarnoAnais` », dans lequel on ajoute les fichiers de base pour la définition de notre produit.

On crée donc les fichiers suivants :

- `Android.mk`
- `AndroidProducts.mk`
- `BoardConfig.mk`
- `lo52_RobitailleAymeric_JarnoAnais.mk`
- `vendorsetup.sh`

Associé à cela, on complète le fichier `lo52_RobitailleAymeric_JarnoAnais.mk` avec les lignes suivantes :

```
PRODUCT_NAME := lo52_RobitailleAymeric_JarnoAnais
PRODUCT_DEVICE := lo52_RobitailleAymeric_JarnoAnais
PRODUCT_BRAND := UTBM
PRODUCT_MODEL := LO52
PRODUCT_MANUFACTURER := utbm
```

Cela nous aura permis de respecter la première contrainte, concernant la définition du produit.

A cela, on ajoute auparavant la ligne `$(call inherit-product, device/linaro/hikey/hikey.mk)`. Celle-ci nous permet justement de respecter la seconde contrainte, c'est-à-dire l'héritage vis-à-vis du produit hikey linaro. Enfin, si ce n'est pas déjà fait il faut ajouter un fichier `init.rc` avec la ligne `mount usbfs none /proc/bus/usb -o devmode=0666` pour pouvoir utiliser la librairie au runtime. Cependant, dans la version de l'AOSP téléchargée, cette étape est déjà réalisée dans le device défini par hikey.

b. Personnalisation du produit

Pour ce qui est de la personnalisation de notre produit, on réalise les opérations suivantes :

- Personnalisation de propriétés dans le fichier `lo52_RobitailleAymeric_JarnoAnais.mk` en définissant la variable `PRODUCT_PROPERTY_OVERRIDES` :
 - `ro.hw` avec la valeur `lo52`
 - `net.dns1` avec la valeur `8.8.8.8`
 - `net.dns2` avec la valeur `4.4.4.4`
- Surcharge du fichier « `sym_keyboard_delete.png` » et ajout du nouveau dossier d'overlay
- Ajout de la libusb aux packages du produit

TP4 : Utilisation de JNI

1. MISE EN PLACE DE L'ENSEMBLE DES COMPOSANTS NÉCESSAIRES POUR LE NDK ANDROID

Dans ce TP, il n'est pas nécessaire d'utiliser la ligne de commande. Android Studio gère toutes les commandes pour nous.

Ainsi, la mise en place de l'environnement et l'installation des composants nécessaires pour le NDK Android se résument aux étapes suivantes :

- Ouvrir Android Studio et créer un nouveau projet, sélectionner l'extrait de code Native C++
- Durant la synchronisation de Gradle et du projet, une erreur devrait se produire en cas d'absence du NDK. Il suffit alors de cliquer sur l'erreur et le téléchargement et l'installation de la dernière version de NDK se fait tout naturellement.

On remarquera qu'il est possible de laisser le téléchargement en arrière-plan pour commencer le codage. Toutefois, la synchronisation n'étant pas terminée, l'analyse du code et la complétion sont désactivées.

Une fois ces étapes terminées, il est alors possible de coder et d'exécuter l'application de la même manière que celle expliquée dans le TP1.

2. CODAGE DE L'APPLICATION NDK

a. Définition de l'interface

Pour réaliser cette application, il nous a donc fallu créer son interface graphique.

Pour cela, nous nous sommes rendus dans le dossier `res/layout/` où nous avons trouvé le fichier `activity_main.xml`. On a alors utilisé l'éditeur de layout pour définir l'interface utilisateur.

Nous avons commencé par supprimer le `TextView` utilisé pour le `HelloWorld` par défaut. Nous l'avons remplacé par un `Input`, pour réceptionner la saisie utilisateur. Celui-ci a comme valeur par défaut « Def ».

Nous avons aussi rajouté six boutons comme exigé par le sujet. Ainsi, on retrouve les boutons `LEFT`, `UP`, `DOWN`, `RIGHT`, `READ` et `WRITE`.

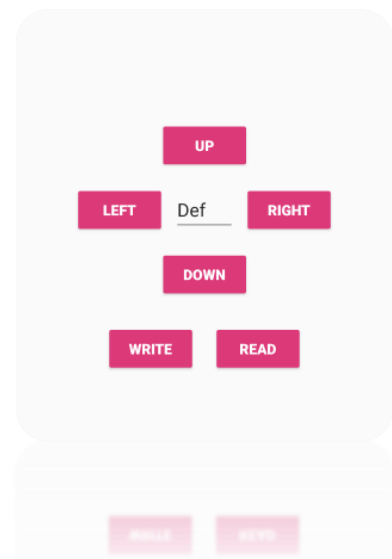
Pour chacun d'eux, nous avons rajouté :

- un ID, pour pouvoir le récupérer depuis le code
- des contraintes, pour leur positionnement.

Aussi, étant donné que les boutons READ et WRITE font appel à la valeur de l'input et qu'ils nécessitent un nombre compris entre 0 et 10, nous avons défini que l'objet Input était de type `numberSigned` pour orienter l'utilisateur vers un bon usage de l'application.

Comme nous venons de le préciser, les boutons READ et WRITE font appel à la valeur de l'input au travers d'une fonction. En effet, lors du clic sur l'un de ces boutons, il fallait afficher un certain texte. De même pour les quatre autres boutons de direction. Ainsi, nous avons placé un label pour afficher ces données.

On obtient ainsi l'interface minimaliste que vous pouvez retrouver à droite.



b. Définition du comportement

Pour définir le comportement réponse des boutons, nous nous sommes rendus dans le code de `MainActivity.kt` où nous avons lié l'événement « `onClick` » à une fonction précise.

Nous avons, dans un premier temps, dû récupérer les différents éléments graphiques présents sur notre interface. Pour cela, plusieurs options s'offraient à nous, telles que :

- l'utilisation de `findViewById`
- ou alors, le **View Binding**

Nous avons opté pour la seconde option pour découvrir d'autres façons de faire qui nous semblait fortement utilisées et intéressantes. En effet, le View Binding est une fonctionnalité qui permet d'écrire plus facilement du code interagissant avec les vues.

Le View Binding n'est pas activé par défaut. Ainsi, la première étape pour son utilisation est son activation dans le projet. Pour cela, il suffit de définir l'option `viewBinding` à la valeur `true` dans le fichier `build.gradle` au niveau du module en rajoutant les lignes suivantes :

```
android {  
    ...  
    buildFeatures {  
        viewBinding true  
    }  
}
```

Une fois la liaison de vue activée dans le module, elle génère une classe de liaison « binding » pour chaque fichier de mise en page XML présent dans le dossier layout. Une instance d'une classe de liaison contient des références directes à toutes les vues qui ont un ID dans la disposition correspondante.

Ainsi, ayant un fichier *activity_main.xml*, une classe *ActivityMainBinding* aura été générée.

Pour obtenir une instance de cette classe de liaison dans notre activité, nous avons suivi les étapes suivantes dans la méthode *onCreate()* de l'activité :

- 1- Tout d'abord, nous avons appelé la méthode *inflate()*, c'est une méthode statique incluse dans les classes de liaison. Cela nous a alors créé une instance de la classe de liaison pour notre activité.
- 2- Ensuite, nous avons pu obtenir une référence à la vue racine en appelant la propriété *root*.
- 3- Enfin, nous avons passé cette vue à *setContentView()* pour en faire la vue active à l'écran.

Voici donc le code qui en résulte :

```
private var binding: ActivityMainBinding? = null

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    val view = binding.root
    setContentView(view)
    ...
}
```

À partir de là, nous pouvons faire référence aux différents objets de la vue, et à leurs propriétés, avec la syntaxe simple : `binding.<objectViewId>.<property>`.

De cette manière, nous avons lié l'événement *onClick* de chacun des boutons en suivant cette démarche :

```
binding.btnId.setOnClickListener { method(arg) }
```

Ainsi, les boutons de direction ont reçu comme méthode à appeler la fonction *translateDirection* avec pour argument le nom de la direction. Celle-ci appelle une fonction native qui nous transmet le nom de la direction correspondante en Japonais, puis on change alors la valeur du label (via le binding) pour l'affecter à la traduction qui nous a été transmise.

Quant à la déclaration des fonctions natives, nous avons explicité ci-dessous la démarche qui a été appliquée.

Tout d'abord, il nous a fallu déclarer les diverses fonctions natives dans le fichier *cpp/native-lib.cpp*, où nos fonctions ont été formatées sous le principe suivant :

```
extern "C" JNIEXPORT jstring JNICALL
Java_fr_utbm_hellondk_MainActivity_nomMéthode(JNIEnv* env, jobject, [arg]) {
    ...
    return env->NewStringUTF(...);
}
```

La première ligne est nécessaire pour créer la fonction native. Ensuite, on retrouve le nom du module, de l'activité et enfin le nom de la méthode créée. Cette méthode nous renvoie ensuite l'objet désiré après traitement (ici nous ne renvoyons que des String).

Ensuite, il faut déclarer ces méthodes dans notre fichier MainActivity.java pour pouvoir les utiliser.

```
private external fun nomMéthode(arg): String
```

On rajoute également les lignes suivantes pour préciser l'origine de ces méthodes dites "externes" :

```
companion object {
    init {
        System.loadLibrary("native-lib")
    }
}
```

Celles-ci étant alors incorporées à notre fichier kotlin (puis java), il nous a donc suffi de les appeler directement, comme tout autre méthode kotlin, dans nos méthodes de réaction à l'événement *onClick*.

Le même principe, c'est-à-dire, l'appel d'une fonction kotlin suite au clic sur un bouton, suivi de l'appel d'une fonction native et le changement de texte dans le label, a été appliqué pour tous les boutons (avec les méthodes appropriées bien évidemment).

On notera que la plus grande difficulté de ce TP aura été de se souvenir comment faire du C++ pour la définition des fonctions natives.

Vous retrouverez une démonstration de l'application en double-cliquant sur l'image ci-contre. Si elle ne se lance pas automatiquement, la cause la plus probable étant la lecture de ce fichier sous le format pdf, [cliquez ici pour lancer la démonstration](#).



Ce compte-rendu touche maintenant à sa fin.

**Nous vous remercions pour le temps et
l'attention que vous nous avez prêté.**

Bonne journée !

–le groupe d'Aymeric ROBITAILLE et d'Anaïs JARNO