

TP1-LO52

Dans ce TP nous étions amenés à installer d'abord l'environnement de travail Android-Studio et le Git.

Concernant l'installation d'Android-Studio nous avons suivi les étapes décrites dans le lien suivant: https://doc.ubuntu-fr.org/android_sdk

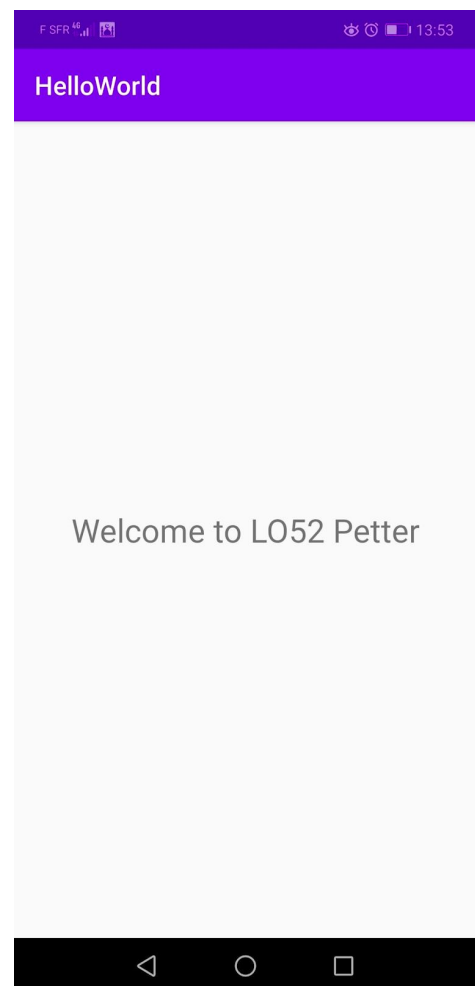
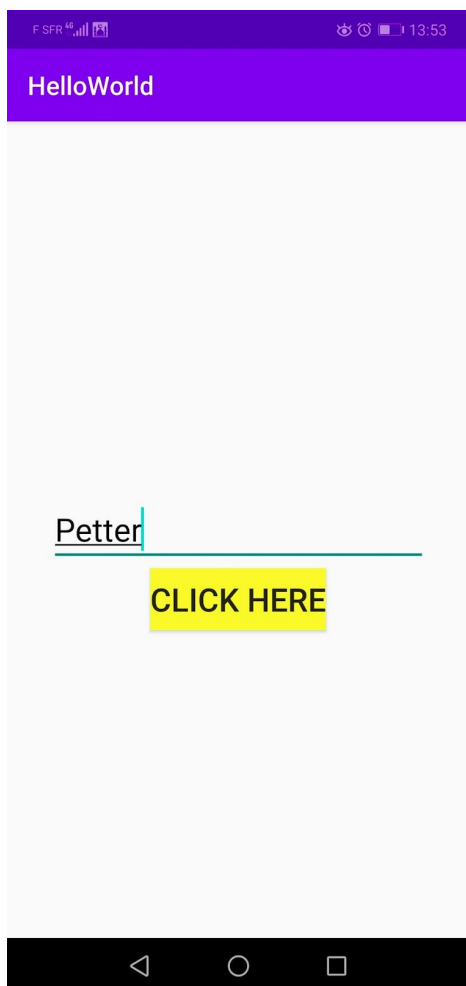
Pour mettre en place le Git sur la machine nous avons exécuté les commandes décrites par le lien suivant: <https://git-scm.com/download/linux>

Ensuite, nous avons cloné la branche "master" et créé notre propre branche.

Puis à chaque modification nous tapons les commandes suivantes:

```
git add .  
git commit -m "msg"  
git push
```

Notre première application "HelloWorld" se compose de deux activités, la première demande à l'utilisateur d'entrer son nom dans le champs de type "EditText" et de cliquer sur le bouton de type "Button" afin de le renvoyer sur un deuxième "view" sur lequel est écrit "welcome to LO52 <Someone Name>" par le biais d'un "TextView".



Lors de l'appui du bouton, la méthode 'ChangeActivity()' est lancée, ce qui crée un objet Intent qui contient le message et le passe à la nouvelle activité. Une fois la nouvelle activité lancée, il récupère l'objet Intent, contenant le message à afficher et le place dans le TextView.

TP2

Pour le TP2, l'objectif était la prise en main du kernel Android. Tout d'abord, nous devons récupérer le kernel depuis les repo Google en utilisant les commandes données par le guide :

```
mkdir android-kernel && cd android-kernel
repo init -u https://android.googlesource.com/kernel/manifest -b
hikey-linaro-android-4.14
repo sync
```

Pour pouvoir réaliser notre première compilation d'android, nous nous sommes rencontré face à des bibliothèques manquantes, que nous devons installer, notamment : Bison, autoconf, automake, ncurses, qt, m4, flex, help2man.

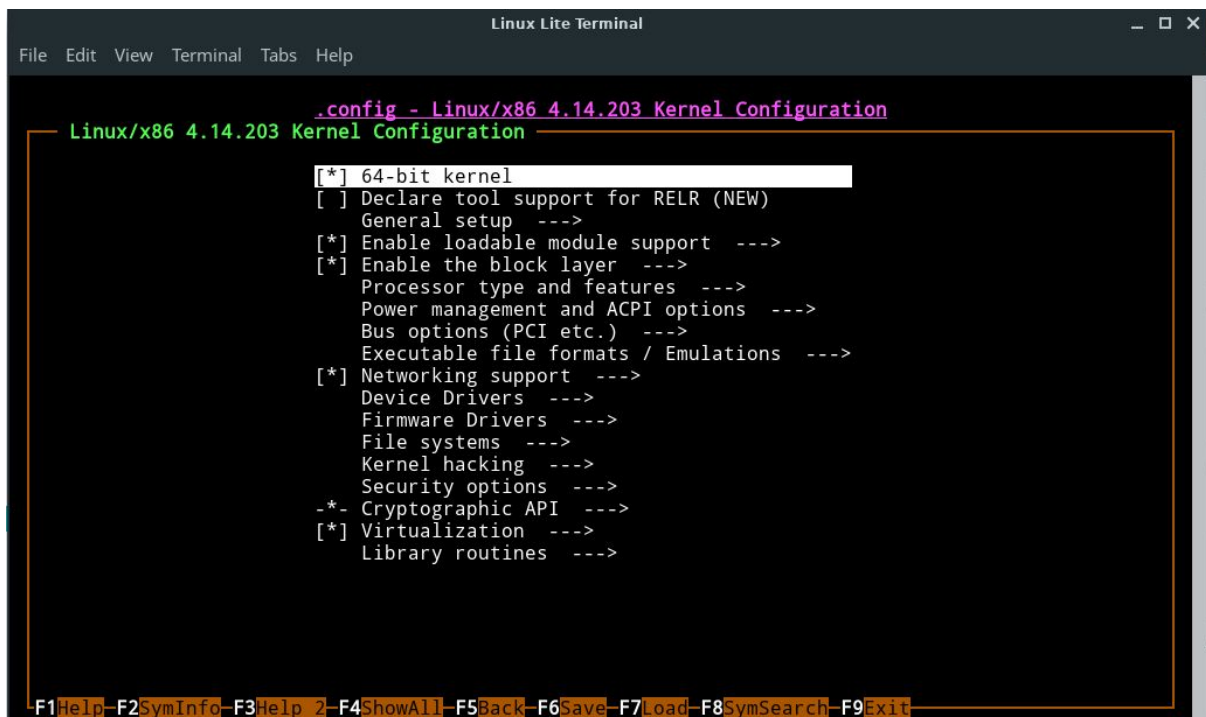
Ensuite nous devons configurer le kernel de façon à compiler certaines fonctionnalités dans la version finale du kernel, notamment:

- Compatibilité pour la carte ARMv8 Versatile mais aussi Qualcomm et Realtek
- Activation du NFC et du protocole NFC HCI
- Activation de l'option Frequency Scaling de la CPU
- Activation du support de l'HDMI CEC et activation du support LED

Pour y parvenir nous avons utilisé un des outils fournis qui permettent de configurer le kernel:

make nconfig

Cette commande nous fournit une interface dans le terminal où on peut sélectionner les options qu'on veut:



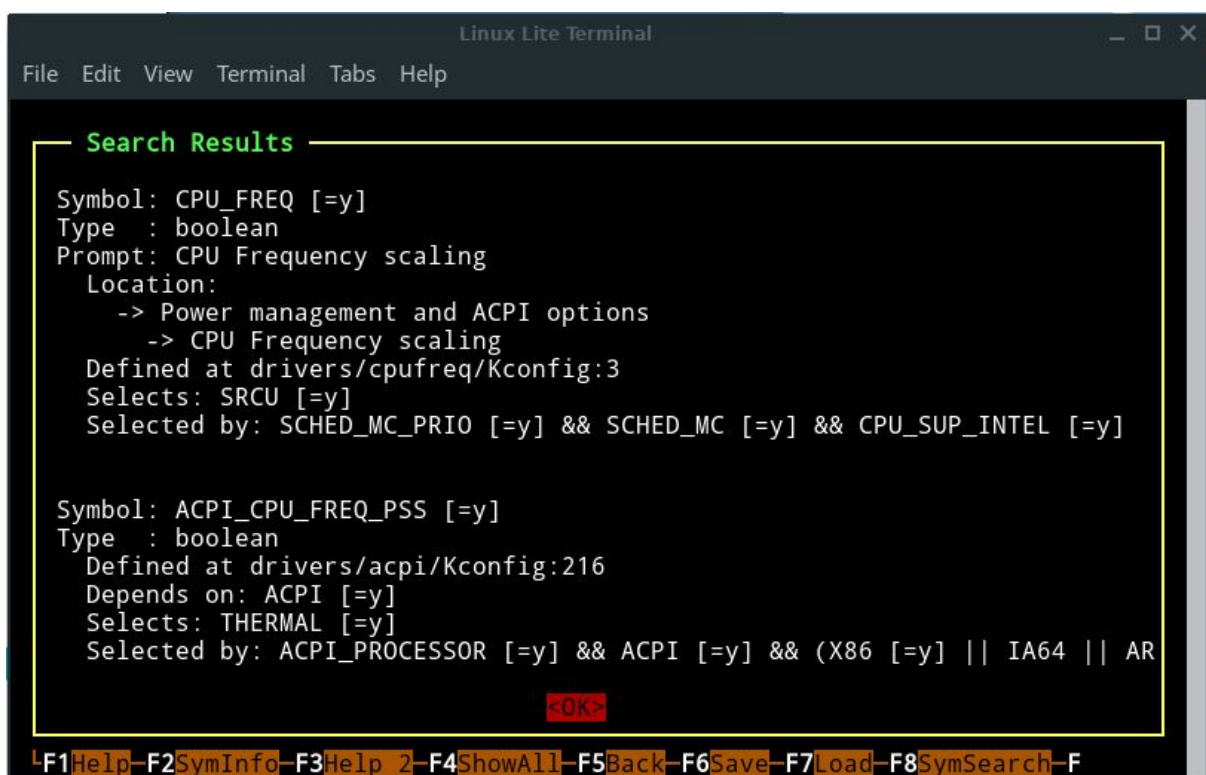
```
Linux Lite Terminal
File Edit View Terminal Tabs Help

.config - Linux/x86 4.14.203 Kernel Configuration
Linux/x86 4.14.203 Kernel Configuration

[*] 64-bit kernel
[ ] Declare tool support for RELR (NEW)
    General setup --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
    Processor type and features --->
    Power management and ACPI options --->
    Bus options (PCI etc.) --->
    Executable file formats / Emulations --->
[*] Networking support --->
    Device Drivers --->
    Firmware Drivers --->
    File systems --->
    Kernel hacking --->
    Security options --->
-* Cryptographic API --->
[*] Virtualization --->
    Library routines --->

F1 Help F2 SymInfo F3 Help 2 F4 ShowAll F5 Back F6 Save F7 Load F8 SymSearch F9 Exit
```

Après avoir chargé la configuration 'board-ranchu.config' (voir steps.sh) , nous avons simplement effectuer une recherche (F8) sur les options désirés, ce qui nous donnait un résultat comme ceci:



```
Linux Lite Terminal
File Edit View Terminal Tabs Help

Search Results

Symbol: CPU_FREQ [=y]
Type : boolean
Prompt: CPU Frequency scaling
Location:
    -> Power management and ACPI options
    -> CPU Frequency scaling
Defined at drivers/cpufreq/Kconfig:3
Selects: SRCU [=y]
Selected by: SCHED_MC_PRIO [=y] && SCHED_MC [=y] && CPU_SUP_INTEL [=y]

Symbol: ACPI_CPU_FREQ_PSS [=y]
Type : boolean
Defined at drivers/acpi/Kconfig:216
Depends on: ACPI [=y]
Selects: THERMAL [=y]
Selected by: ACPI_PROCESSOR [=y] && ACPI [=y] && (X86 [=y] || IA64 || AR

<OK>

F1 Help F2 SymInfo F3 Help 2 F4 ShowAll F5 Back F6 Save F7 Load F8 SymSearch F9 Exit
```

Dans cet exemple, nous avons cherché les termes 'cpu_freq', ce qui nous donne une liste de résultats. Dans les résultats nous avons l'information sur le lieu de l'option, dans ce cas Power management and ACPI options -> CPU frequency

scaling, ainsi que les dépendances. Dans certains cas, les options qu'on cherche ne sont pas toujours disponibles pour être paramétrées, les dépendances sont donc des conditions qui doivent être respectées pour que l'option apparaisse dans l'interface de configuration. Dans ce cas, l'option SCHED_MC_PRIO, SCHED_MC et CPU_SUP_INTEL doivent tous être activées pour avoir accès à CPU_FREQ. Ici, elles le sont déjà.

Pour chaque option nous avons procédé ainsi et nous avons activé toutes les dépendances nécessaires pour répondre au besoin.

Nous avons utilisé la commande diff pour comparer les deux versions. La sortie de la commande se trouve dans le fichier diff.txt.

TP3

Dans ce TP, nous étions amenés à écrire des Makefiles afin d'intégrer un composant et un produit Android.

A. Implémentation de la libusb:

Les fichiers sources et header nécessaires à la compilation de libusb sont les suivants :

- libusb.h
- libusb.h
- os/linux_usbfs.h
- core.c
- descriptor.c
- io.c
- sync.c
- os/linux_usbfs.c

Par la suite nous avons écrit le fichier Android.mk se trouvant dans libusb-1.0.3\libusb:

```
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 := $(LOCAL_PATH) $(LOCAL_PATH)/os/

LOCAL_MODULE := libusb-module

include $(BUILD_SHARED_LIBRARY)
```

Enfin nous compilons en suivant les recommandations qui existent dans le fichier install:

```
./configure
make
make check
make install
```

Cette étape est faite sans encombre.

B. Implémentation d'un nouveau produit Android:

Nous avons paramétré le nouveau produit de la manière suivante:

Héritage du produit : hikey de linaro

Nom du produit : lo52_BayoudeIzoukaYosef_kammounyessine

Propriétés customisées : ro.hw=lo52 net.dns1 = 8.8.8.8 net.dns2 = 4.4.4.4

Surcharge du fichier : sym_keyboard_delete.png

Ajout de la libusb aux packages du produit.

Cette configuration existe dans le fichier

lo52_BayoudeIzoukaYosef_kammounyessine.mk

```
# Heritage from the product hikey linaro
$(call inherit-product, device/linaro/hikey.mk)

# Append libusb
PRODUCT_PACKAGES += libusb

# customized property
PRODUCT_PROPERTY_OVERRIDES += ro.hw=lo52 \
net.dns1=8.8.8.8 \
net.dns2=4.4.4.4

# overlay directory
PRODUCT_PACKAGE_OVERLAYS :=
device/lo52_BayoudeIzoukaYosef_kammounyessine/overlay

# define the name of the product
PRODUCT_NAME := lo52_BayoudeIzoukaYosef_kammounyessine
PRODUCT_DEVICE := lo52_BayoudeIzoukaYosef_kammounyessine
PRODUCT_BRAND := lo52_BayoudeIzoukaYosef_kammounyessine
PRODUCT_MODEL := lo52_BayoudeIzoukaYosef_kammounyessine

include $(call all-subdir-makefiles)
```

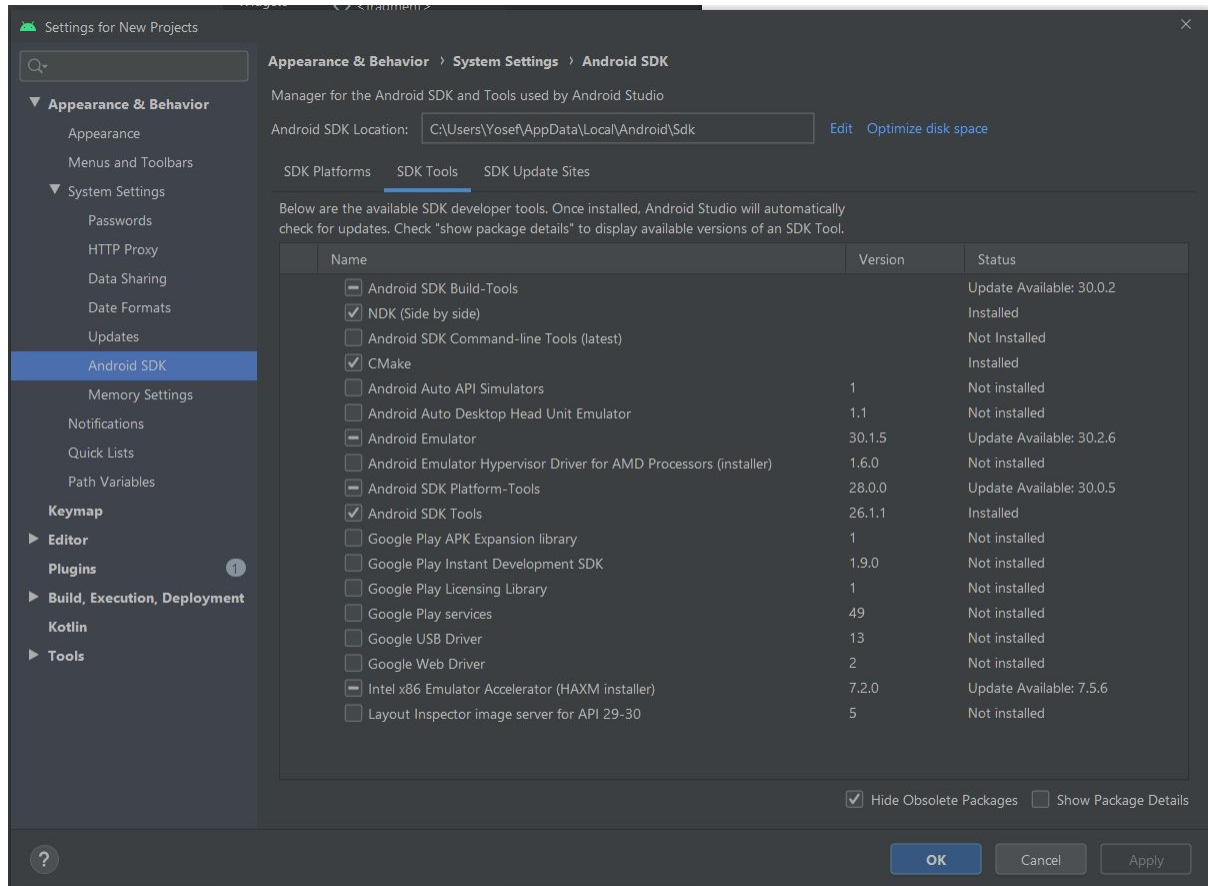
voici le contenu du dossier final du produit:

```
kammoun@galaxy:~/AndroidStudioProjects/TPs/L052_A2020/TP3/device/lo52_BayoudeIzoukaYosef_kammounyessine$ ls -ao
total 32
drwxr-xr-x 4 kammoun 4096 janv. 2 00:18 .
drwxr-xr-x 3 kammoun 4096 janv. 1 23:03 ..
-rw-r--r-- 1 kammoun 131 janv. 2 00:15 AndroidProducts.mk
-rw-r--r-- 1 kammoun 87 janv. 2 00:16 BoardConfig.mk
drwxr-xr-x 5 kammoun 4096 janv. 1 23:11 libusb-1.0.3
-rw-r--r-- 1 kammoun 634 janv. 2 00:18 lo52_BayoudeIzoukaYosef_kammounyessine.mk
drwxr-xr-x 3 kammoun 4096 janv. 1 23:03 overlay
-rwxr-xr-x 1 kammoun 234 janv. 2 00:18 vendorsetup.sh
```

TP 4

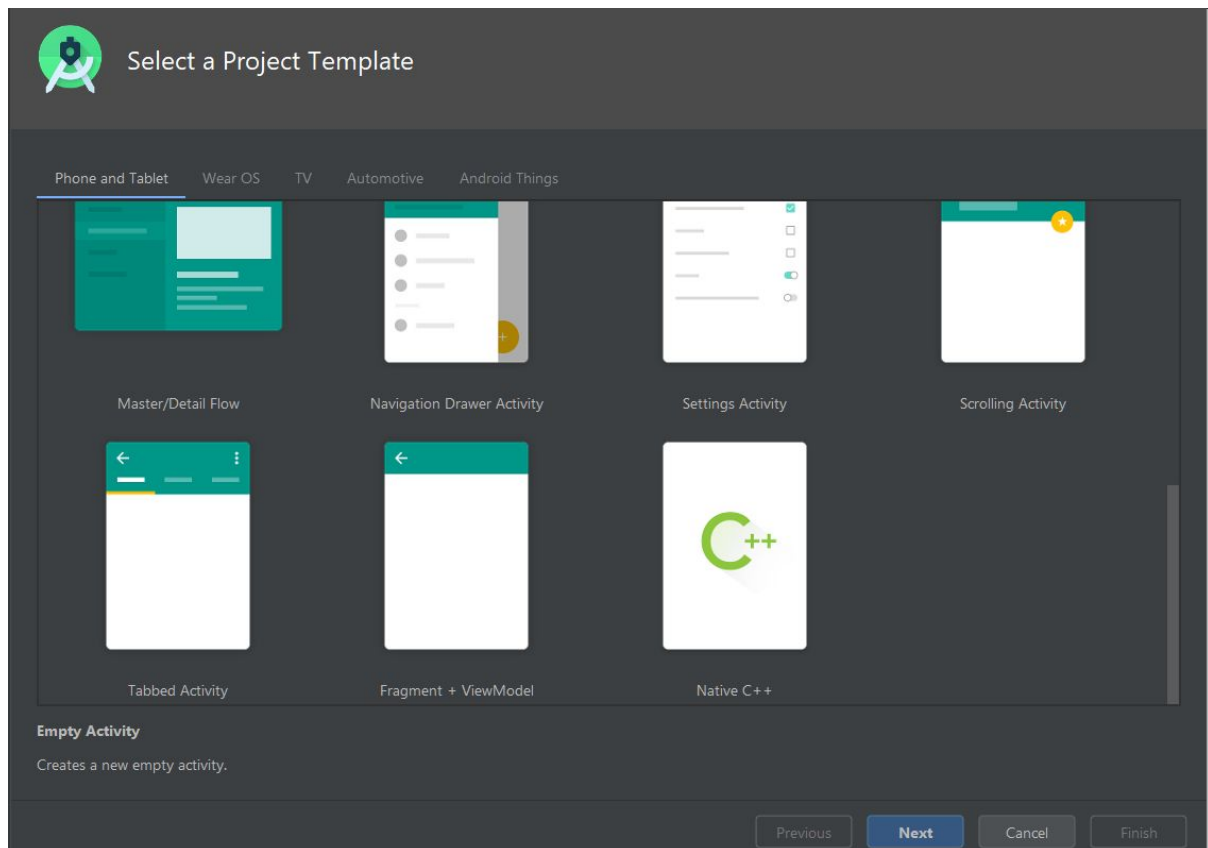
Le but de ce TP est de créer une application Android qui fait appel à des fonctions C ou C++ dites “fonctions natives”. Le TP 4 se déroule donc entièrement dans l’application Android Studio.

Pour pouvoir créer un projet de ce type là, nous devons d’abord installer l’environnement et les outils nécessaires, c’est-à-dire, en téléchargeant le NDK - Native Development Kit. L’application le gère automatiquement grâce à la fenêtre SDK.



Il suffit de cocher NDK(side by side) et CMake ensuite OK pour commencer l’installation.

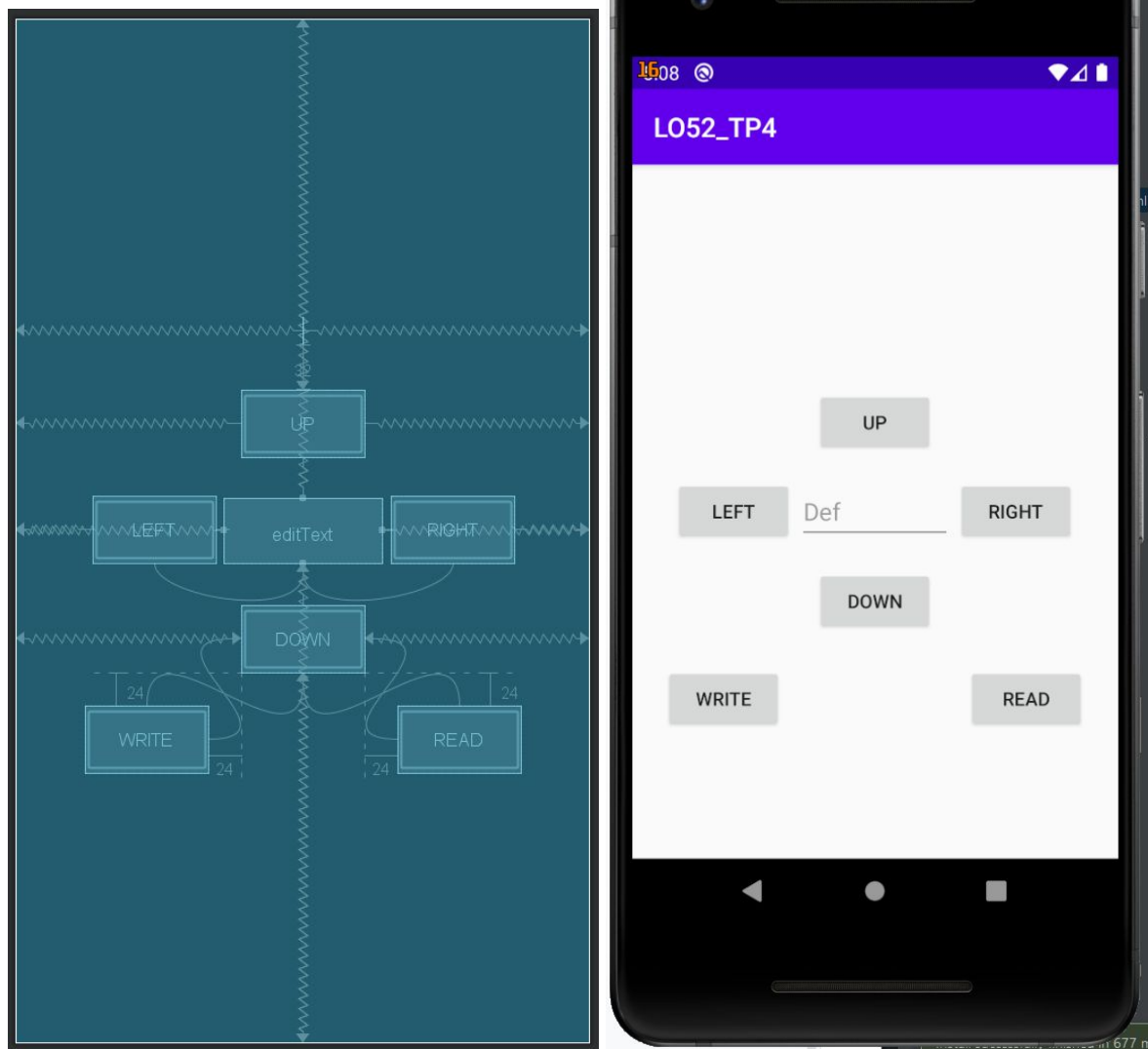
Une fois les outils en place, on peut créer un nouveau projet de type “Native C++” :



Cette option nous permet d'avoir un projet, avec un répertoire “cpp” où se trouvent les includes, les fonctions C++ ainsi que le CMake pour compiler cette partie de l'application.

Développement application

Tout d'abord nous avons défini l'interface comme indiqué dans le sujet, en utilisant l'outil “design” que nous avons découvert lors du premier TP. Notre interface ressemble donc à ceci :



Nous avons défini les attributs de chaque composant et lié chaque bouton à sa méthode propre. Chaque méthode utilise donc une fonction native dans sa description.

Pour pouvoir faire cette liaison, les fonctions C++ doivent être déclarer en suivant la structure suivante : `<nom_du_package>_<nom_activity>_<nom_fonction>()`; Pour pouvoir les utiliser dans notre code Java, il suffit de les déclarer de la façon suivante: `public native String <nom_fonction>()`;

Android fait la liaison tout seul, il suffit d'appeler la méthode déclaré dans la partie Java pour l'utiliser. Dans le cas ou nous voulons passer des paramètre de Java vers C++, nous utilisons les primitives fourni par la JNI :

Java primitive type	Native primitive type
void	void
byte	jbyte
int	jint
float	jfloat
double	jdouble
char	jchar
long	jlong
short	jshort
boolean	jboolean

Pour finir nous avons ajouté des gestions d'erreurs dans la partie Java pour ne permettre que les valeurs entre 0 et 10. On obtient le résultat suivant :

