

Pilotage à distance d'un appareil de mesure sous Arduino avec une application Android



TABLE DES MATIERES

1.	L'application Android	4
1.1.	Installation	4
1.2.	Appairage Bluetooth	4
1.3.	Utilisation de l'application	4
2.	Ecosystème Android	5
2.1.	Script Gradle de l'application Android	5
2.2.	Manifest de l'application Android	5
2.3.	Éléments d'architecture Android	5
2.4.	Cycle de vie Android	6
3.	Représentation des données sous Android	6
3.1.	Package models.....	6
3.1.1.	Enum Unit	6
3.1.2.	Enum Scale	6
3.1.3.	Classe Channel	7
3.1.4.	Classe Generator	7
4.	Echange des données avec Arduino	7
4.1.	Package services Android.....	7
4.1.1.	Classe JsonConverterService.....	7
4.1.2.	Classe BluetoothService	8
4.2.	Fichiers Arduino	8
4.2.1.	220 bluetooth_communication	8
4.2.2.	200 parsing_json	9
5.	Manipulation des données sous Android	9
5.1.	Package interfaces	9
5.1.1.	Interface FragmentSwitcher	9
5.1.2.	Interface BluetoothConstants.....	10
5.1.3.	Interface BluetoothParent	10
5.1.4.	Interface BluetoothChildren	10
5.1.5.	Interface BluetoothConnect	10
5.2.	Package racine	10
5.2.1.	Classe AppCompatActivity	10

5.3.	Package fragments.....	11
5.3.1.	Classe ConnectFragment.....	11
5.3.2.	Classe MainBoardFragment.....	11
5.3.3.	Classe BackupFragment.....	11
5.4.	Package recyclers.....	12
5.4.1.	Classes MainBoardAdapter et MainBoardHolder.....	12
5.4.2.	Classe BackupStoreAdapter.....	13
5.4.3.	Classe BackupConfigAdapter.....	13
6.	Gestion des ressources et des visuels.....	14
6.1.	Dossier Values.....	14
6.1.1.	Integers.....	14
6.1.2.	Strings.....	14
6.1.3.	Colors.....	14
6.1.4.	Styles.....	15
6.1.5.	Dimensions.....	15
6.2.	Dossiers Drawable & Mipmap.....	15
6.2.1.	Drawable.....	15
6.2.2.	Mipmap.....	15
6.3.	Dossier Menu.....	15
6.4.	Dossier Layout.....	15
6.5.	Construction d'un Layout.....	16
6.5.1.	Onglet Code.....	16
6.5.2.	Onglet Design.....	16
Annexe 1 :	Visuels de l'application sur tablette.....	17
	ConnectFragment.....	17
	MainBoardFragment.....	17
	BackupFragment.....	18
	Version smartphone.....	18
Annexe 2 :	Architecture sous Android.....	19
Annexe 3 :	Cycle de vie Android.....	19
Annexe 4 :	Communication Bluetooth.....	20

1. L'APPLICATION ANDROID

1.1. INSTALLATION

L'installation de l'application est très simple : il suffit d'exécuter le fichier APK (Android Package Kit) « MIVS contrôleur » sur l'appareil Android.

Pour générer cet APK, sous Android Studio, il suffit de cliquer sur « Build » puis « Build Bundle(s) / APK(s) » et « Build APK(s) ». Après un temps d'exécution, le fichier se trouvera dans le dossier « app/build/outputs/apk/debug » du projet.

1.2. APPAIRAGE BLUETOOTH

Lors de la première utilisation suivant l'installation, l'application ne montrera que les appareils Bluetooth déjà appairés. Si le module HC-06 n'a jamais été appairé avec l'appareil Android, il faut d'abord aller dans les paramètres systèmes de l'appareil -> Bluetooth -> Activer le Bluetooth si ce n'est pas déjà fait -> Effectuer une recherche/découverte (le module HC-06 doit évidemment être branché et clignoter) -> Cliquer sur "HC-06" ou un autre nom s'il a été renommé -> Entrer "1234" comme mot de passe par défaut, ou le mot de passe défini si celui-ci a été modifié.

Il est possible de modifier le nom et le mot de passe du module Bluetooth HC-06 sous Arduino, grâce aux AT command (AT+NAMExyz et AT+PIN1234).

Maintenant que le module Bluetooth HC-06 et l'appareil Android sont appairés, au retour dans l'application, l'appareil apparaîtra dans la liste. S'il n'apparaît toujours pas, il faut ouvrir le gestionnaire de tâches de fond intégré à Android et fermer l'application. En la rouvrant, la liste se rafraîchira et la connexion au module deviendra possible.

1.3. UTILISATION DE L'APPLICATION

L'application se veut assez intuitive et est très proche de l'expérience du touchscreen du générateur Modular I-V source. On retrouve sur l'écran principal tous les channels actuellement utilisés (actif et inactif) et les interactions possibles. Une seconde page permet d'accéder au stockage des sauvegardes. Tous les visuels sont disponibles en [annexe 1](#).

2. ECOSYSTEME ANDROID

2.1. SCRIPT GRADLE DE L'APPLICATION ANDROID

Ce script gère un ensemble de règles sur les versions d'Android supportées, ainsi que le type et la version du langage utilisée. Enfin, il s'occupe de gérer les dépendances du projet.

Notre application prend donc en charge les appareils Android basés sur le SDK 16 (Android 4.1) et supérieur, couvrant ainsi 99.8% des appareils du marché à ce jour.

Le langage utilisé pour compiler le projet est le Java, dans sa version 1.8, pour l'accès aux lambda fonctions. Cependant, la version ancienne du SDK amène d'autres restrictions (streams).

Les dépendances du projet sont les suivantes :

- Material Design, pour bénéficier de composants plus variés et évolués ;
- AppCompat, pour prendre en charge les versions d'Android les plus anciennes ;
- ConstraintLayout, pour pouvoir appliquer des contraintes verticales et horizontales ;
- RecyclerView, pour pouvoir utiliser ce conteneur doublement dynamique ;
- Gson, pour construire et manipuler des JSON plus facilement.

2.2. MANIFEST DE L'APPLICATION ANDROID

Ce fichier, décrit comme la carte d'identité de l'application, définit plusieurs éléments clés :

- Les permissions de l'application :
 - Permission d'utilisation du Bluetooth,
 - Permission d'utilisation du vibreur.
- L'identité de l'application, via des références aux ressources : Son nom, son icône (version carrée comme ronde, usage selon le téléphone) ainsi que son thème général.
- L'écran principal, de démarrage, et ses options : Ici, notre seule activité : AppCompatActivity. Comme option supplémentaire, nous avons ajouté le support de la rotation d'écran.

2.3. ELEMENTS D'ARCHITECTURE ANDROID

Une application peut contenir plusieurs activités. Chaque activité peut déléguer tout ou une partie de l'écran à un ou plusieurs fragments et interchanger ses fragments sans redémarrer l'écran. Ce fonctionnement est illustré en [annexe 2](#).

Les activités, comme les fragments, déploient leur propre vue : un fichier xml définit dans le dossier layout des ressources. Ces layouts sont composés de divers composants, allant du simple texte au conteneur permettant d'accueillir tout un fragment.

Chaque composant des layouts ont leur affichage et comportement par défaut, mais il faut parfois les redéfinir en créant respectivement un nouveau layout ou une nouvelle classe. Certains composants nécessitent d'implémenter leur comportement, ne pouvant en avoir par défaut.

Dans notre application, nous avons été contraints de n'utiliser qu'une activité et de multiples fragments en raison du service Bluetooth, devant n'être instancié qu'une seule fois.

Les activités et les fragments peuvent afficher un layout de menu et gérer leurs interactions. Dans notre projet, nous avons un seul layout de menu, dont on a défini qu'une fois les interactions. On vient ensuite, selon le fragment, afficher ou masquer les éléments du menu.

2.4. CYCLE DE VIE ANDROID

Les activités, comme les fragments, suivent un cycle de vie : l'appel d'un écran, comme le retour au menu ou l'extinction de l'écran vont mener à l'appel de méthodes on, suivant le schéma en [annexe 3](#).

Dans notre application, nous avons géré le cycle de vie au sein de l'activité, afin que toute cette logique soit regroupée. Pour cela, nous avons implémenté onCreate / onCreateView (nécessaire pour chaque activité / fragment), mais aussi onResume et onPause. Ces deux méthodes supplémentaires suffisent à gérer tous les cas d'usages, de façon à déconnecter l'application de l'appareil connecté lorsqu'on quitte ou met en pause l'application, ou simplement l'écran.

3. REPRESENTATION DES DONNEES SOUS ANDROID

3.1. PACKAGE MODELS

Les classes de ce package servent à représenter la donnée telle qu'elle sera manipulée dans l'application. Cette représentation doit se rapprocher des données manipulées côté Arduino.

3.1.1. Enum Unit

Symboles représentant les unités que peuvent délivrer les canaux, valeurs numériques implicites.

3.1.2. Enum Scale

Symboles représentant les échelles que peuvent prendre les valeurs des canaux, valeurs numériques explicites : puissances de 10.

Méthodes clés :

- ❖ double changeScale(final double value, final Scale initial, final Scale target) : opère un changement d'échelle sur une valeur.
 - Suppression du signe, conversion de value en string et localisation du séparateur,
 - Déplacement du séparateur selon le changement d'échelle, ajout de 0 si nécessaire,
 - Rajout du signe et conversion du string en double.

3.1.3. Classe Channel

Représentation d'un canal : un attribut par information à stocker, avec les valeurs par défaut attribuées d'office.

Chaque attribut a un booléen associé, qui sert à savoir si la valeur appliquée est celle par défaut.

Enfin, chaque attribut est référencé sous forme de string pour le service de conversion JSON.

Les setters retournent leur instance pour permettre des constructions de canal chaînées.

Il y a un jeu de valeurs limites par unité mais un seul jeu de getter et setter : les valeurs sont lues ou appliquées en interne selon l'unité enregistrée dans le canal. Pendant l'initialisation, il faut donc commencer par préciser l'unité avant d'appliquer les valeurs limites.

3.1.4. Classe Generator

Représentation d'une liste de canaux : un seul attribut, une liste dynamique.

4. ECHANGE DES DONNEES AVEC ARDUINO

4.1. PACKAGE SERVICES ANDROID

Les classes de ce package servent à faire transiter les informations entre Android et Arduino. Elles sont indépendantes de l'application : pas d'import d'autres classes.

4.1.1. Classe JsonConverterService

Le JSON est un format de parsing n'utilisant que 3 règles :

- ❖ Un objet est une valeur, une chaîne de caractère, un tableau ou un objet ;
- ❖ Les tableaux, délimités par des crochets, contiennent des suites de valeurs ;
- ❖ Les objets, délimités par des accolades, contiennent des associations clés valeurs.

Cette classe regroupe toutes les fonctions de formatage JSON, que ça soit instances vers string (sérialisation) ou l'inverse (désérialisation). La (dé)sérialisation automatique via la librairie Gson implique que les clés de la string JSON respectent l'écriture des attributs des modèles de données.

La méthode `int applyJsonData(final Generator generator, final String json)` applique directement les données extraites du JSON à l'instance de Generator.

Cette étape de formatage est nécessaire, le transfert des données (envoi et réception) se faisant sous forme de texte grâce à la classe BluetoothService.

4.1.2. Classe BluetoothService

Cette classe renferme toute la logique de connexion et de communication Bluetooth.

Ses attributs clés sont :

- ❖ Le UUID (Universally unique identifier), qui permet de se connecter au module Bluetooth ;
- ❖ La liste des strings de commandes, transmis via le constructeur ;
- ❖ Le handler, transmis via le constructeur. Ce dernier permet au service de transmettre des informations à l'application sans appeler de méthode externe.

Elle renferme 3 classes internes qui héritent de Thread :

- ❖ AcceptThread, qui sert à faire la découverte Bluetooth et héberger une connexion (serveur) ;
- ❖ ConnectThread, qui sert à se connecter à un appareil Bluetooth (client) ;
- ❖ ConnectedThread, qui sert à communiquer avec l'appareil Bluetooth et à s'en déconnecter.

Ses méthodes clés sont :

- void send(final String buffer) : sert à envoyer un message
- void run() : sert à réceptionner les messages. Tant que l'appareil est connecté :
 - On vérifie que le buffer est non vide.
 - Si non, on récupère le contenu du buffer pour compter le nombre d'accolades qu'il contient (ouvrante : +1, fermante : -1).
 - S'il y a eu des accolades et qu'on est à 0, on a reçu un JSON complet : on transmet le message à l'application via le handler puis on nettoie les variables.
 - Sinon, s'il n'y a pas eu d'accolade, on cherche une correspondance avec une des strings de contrôle. Si c'est le cas, on transmet le message à l'application via le handler puis on nettoie les variables.

L'initialisation du service Bluetooth est très spécifique : un schéma l'illustre en détail (appel des méthodes) en [annexe 4](#). Cette annexe illustre aussi les embranchements du handler.

4.2. FICHIERS ARDUINO

Des modifications ont été apportées au code fourni en partie pour avoir une simulation fonctionnelle sur la carte DUE nue. Toutes les modifications au sein des fichiers originaux ont été notifiées en commentaire de la ligne avec le mot-clé "changement" suivi du code initial. De plus, nous avons ajouté deux nouveaux fichiers : 200 parsing_json et 220 bluetooth_communication.

Pour la plupart des fonctions entraînant un changement de valeurs ou de paramètres, un exemple de la fonction à utiliser dans le code Arduino est mis en commentaire.

4.2.1. 220 bluetooth_communication

Le fichier 220 bluetooth_communication ne fait qu'instancier la réception des données Bluetooth et vérifier si les données reçues correspondent à une commande explicite ou s'il s'agit de données JSON. Dans tous les cas, une fonction du fichier parsing_json est appelée.

4.2.2. 200 parsing_json

Ce fichier contient plusieurs tableaux à plusieurs dimensions simulant chacun une liste de canaux, avec pour chaque canal la liste de ses attributs. 2 fonctions s'occupent de la (dé)sérialisation JSON : `jsonDeserialize` et `execJsonCommand`. Il peut arriver que plusieurs actions rapprochées sur Android soient réceptionnées côté Arduino comme une seule string, ne correspondant plus aux règles JSON. C'est pourquoi la fonction `jsonDeserialize` découpe les JSON concaténés, le cas échéant. Ensuite chaque string JSON vérifiée est envoyée à la fonction `execJsonCommand`, qui déséréalise les données et, en fonction du résultat obtenu, exécutera la fonction (Arduino) appropriée.

Il existe également 5 fonctions, appelées lorsque la string reçue est un mot clé de commande :

- ❖ "init_main" -> `initPlz`, pour envoyer la liste des canaux et de leurs attributs reliés à la carte Arduino,
- ❖ "init_stores" -> `getAllStores`, pour envoyer l'état – vide ou non – de chaque emplacements mémoires, et donc leur nombre,
- ❖ "store_get_" suivi d'un entier, pour envoyer la liste des canaux et de leurs attributs stockés à l'emplacement mémoire désigné par l'entier,
- ❖ "store_save_" suivi d'un entier, pour stocker la liste des canaux courants et leurs attributs à l'emplacement mémoire désigné par l'entier,
- ❖ "store_del_" suivi d'un entier, pour vider l'emplacement mémoire désigné par l'entier.

5. MANIPULATION DES DONNEES SOUS ANDROID

5.1. PACKAGE INTERFACES

Les interfaces servent à faire le lien entre plusieurs classes : une première classe implémente l'interface, qui interagira avec ses attributs, et une seconde classe instanciée par la première pourra appeler la ou les méthodes de cette interface, après un cast vers la classe parent de l'instance ou de l'activité.

5.1.1. Interface FragmentSwitcher

Implémentée par l'activité principale, cette interface regroupe la pile de fragments manipulée, ainsi que la méthode de permutation de fragments, accessible par les fragments enfants.

En plus d'implémenter cette interface, il faut réécrire :

- ❖ La méthode `onBackPressed()`, afin de correctement traiter les retours en arrière ;
- ❖ La méthode `onConfigurationChanged(Configuration newConfig)`, afin de supporter les rotations d'écran.

5.1.2. Interface BluetoothConstants

Implémentée par l'activité principale, BluetoothService et ConnectFragment, cette interface regroupe des déclarations de type Integer pour faire la liaison entre les messages envoyés par le service Bluetooth via le handler et leur réception dans l'activité principale. Cette interface regroupe aussi des déclarations de type string qui permettent d'enregistrer dans la mémoire du téléphone (SharedPreferences) et de récupérer les informations du dernier appareil connecté.

5.1.3. Interface BluetoothParent

Implémentée par l'activité principale, cette interface regroupe le Generator affiché dans MainBoardFragment et la liste des Generators stockés dans les emplacements mémoires. Cette liste est liée à une liste de booléens permettant de mémoriser les emplacements remplis et ainsi, différer les appels Bluetooth. Enfin, cette interface regroupe les méthodes de connexion, déconnexion, reconnexion automatique, d'envoi de données et enfin un getter setter pour savoir si la liste des emplacements remplis est initialisée.

5.1.4. Interface BluetoothChildren

Implémentée par MainBoardFragment et BackupFragment, cette interface regroupe la méthode applyChanges(Generator generator, int index), accessible depuis l'activité principale par un cast du fragment du dessus de la pile de FragmentSwitcher, qui permet d'appliquer spécifiquement les changements reçus par Bluetooth aux composants des fragments enfants. Cette méthode est donc une réception des données déguisée.

5.1.5. Interface BluetoothConnect

Implémentée par ConnectFragment, cette interface regroupe la méthode updateTitle(final String title), accessible depuis l'activité principale par un cast du fragment du dessus de la pile de FragmentSwitcher, qui permet de changer le titre affiché par le fragment.

5.2. PACKAGE RACINE

Dans notre cas, nous n'avons qu'une seule activité, donc qu'un seul fichier. C'est pourquoi le point de départ de l'application n'est pas dans un package nommé « activities ». Ici, l'activité est le conteneur des fragments, elle gère donc le cycle de vie de l'application, met en place les services généraux et enregistre les données communes à plusieurs fragments via les fichiers d'interfaces. Elle implémente aussi les interfaces nécessaires pour accéder à tous ces éléments.

5.2.1. Classe AppCompatActivity

AppCompatActivity hérite de AppCompatActivity plutôt que de AppCompatActivity pour bénéficier de la barre de menu, même sur les appareils plus anciens.

Ici, le layout de l'activité se résume à un conteneur de fragments, mais l'activité gère le cycle de vie de l'application, met en place le service Bluetooth avec notamment la définition du handler et enregistre les données communes aux fragments MainBoard et Backup via les d'interfaces. Elle implémente aussi les interfaces nécessaires pour accéder à tous ces éléments.

5.3. PACKAGE FRAGMENTS

Les classes de ce package héritent de Fragment et servent à préparer l'affichage initial d'une vue, ainsi que les interactions utilisateur avec ses composants. Dans notre cas, ces affichages occupent tout l'écran.

5.3.1. Classe ConnectFragment

Cette classe assure l'aspect connexion à un appareil Bluetooth dans onCreateView() :

- ❖ Déconnexion à la création de la view par sécurité ;
- ❖ Si reconnexion automatique est sélectionnée, connexion directe au Bluetooth ;
- ❖ Récupération des appareils déjà appairés par l'appareil et affichage dans un ListView ;
- ❖ Activation d'un listener à la sélection d'un élément du ListView :
 - Tentative de connexion ;
 - Si connexion réussie et reconnexion automatique sélectionnée, enregistrement des informations de l'appareil connecté ;
 - Affichage du fragment MainBoard.

Son seul attribut est un lien vers le composant titre, pour mettre à jour la string affichée « on runtime » via la méthode de l'interface BluetoothConnect.

5.3.2. Classe MainBoardFragment

Cette classe remplit son RecyclerView avec les informations du Generator de l'interface BluetoothParent de l'application et gère les interactions des boutons d'activation et désactivation généraux.

Ses attributs sont des liens vers les composants boutons globaux, ainsi que l'Adapter du RecyclerView, pour mettre à jour les affichages « on runtime » via la méthode de l'interface BluetoothChildren.

5.3.3. Classe BackupFragment

Cette classe commence par vérifier qu'elle possède l'information des emplacements mémoires remplis ou non et réalise un appel Bluetooth si nécessaire. Ensuite, elle initialise son RecyclerView secondaire, dédié à l'affichage du contenu d'un emplacement mémoire, puis son RecyclerView principal, chargé d'afficher les emplacements mémoires et de piloter le RecyclerView secondaire. Si elle possède déjà l'information des emplacements mémoire, elle remplit le RecyclerView principal.

Son seul attribut est l'Adapter de son RecyclerView principal, pour mettre à jour l'un ou l'autre des RecyclerView « on runtime » via la méthode de l'interface BluetoothChildren.

5.4. PACKAGE RECYCLERS

Les RecyclerView sont des composants de type conteneur : ils servent à afficher de manière dynamique une liste sous forme d'items (ici en fonction du nombre de canaux) où les entrées de la liste sont mises en forme selon une vue prédéfinie. Les forces de ce conteneur sont d'une part qu'il « recycle » les items non affichés à l'écran et d'autre part qu'il peut employer différentes vues. En revanche, ce conteneur nécessite d'implémenter deux classes : une pour le conteneur général (qui hérite de AdapterView) et une autre pour l'item spécifique (qui hérite de ViewHolder). La seconde peut être une classe interne comme externe au premier.

Les classes de ce package servent donc à préparer l'affichage initial des RecyclerView, ainsi que les interactions utilisateur avec ses items. Ici, nous avons utilisé 3 RecyclerView : un pour le fragment MainBoard, où les classes ViewHolder et AdapterView sont distinctes ; et deux pour le fragment Backup où les deux sont regroupés.

5.4.1. **Classes MainBoardAdapter et MainBoardHolder**

La classe MainBoardAdapter gère la liste de ses items, internes au RecyclerView, et leur mise à jour selon les réceptions Bluetooth. C'est aussi elle qui assure la sélection d'un seul item et digit de l'item à la fois. Enfin, c'est elle qui se charge des interactions entre l'item sélectionné et les composants de modification des items, en dehors du RecyclerView.

Ses méthodes clés sont :

- ❖ void setLastHolderSelected(final MainBoardHolder holder, final Channel channel, final int position) : actualise l'instance de l'item « actif » et met à jour les valeurs et les légendes des inputs.
- ❖ Le keyListener associé aux trois inputs dans le onBindViewHolder :
 - Si valeur courante, change d'échelle si nécessaire et fixe longueur max,
 - Sinon si min ou max, fixe longueur max selon l'unité,
 - Convertit string de valeur courante en double, rétablit affichage si échec,
 - Si min ou max, vérifie limites absolues et applique nouvelle valeur Android / Arduino,
 - Sinon, vérifie des limites sur même échelle et applique nouvelle valeur.
- ❖ void variation(final Context context, final int step) associé aux boutons + et - :
 - Passage de la valeur en Integer via le type String,
 - Application de la variation en integer (valeurs exactes),
 - Vérification des limites, bornage si nécessaire et vibration dans ce cas,
 - Upscaling si nécessaire ($\text{abs} > 9.999$) et possible,
 - Application de la nouvelle valeur et transmission Bluetooth.

Les points d'améliorations de cette classe sont :

- ❖ Remplacer l'attribut local mChannelList par le global BluetoothParent.mGenerator. En effet, le second est une duplication du premier et la liste peut ne pas être interne.
- ❖ Dans le keyListener, corriger la ligne suivie du commentaire « rustine » : application de l'échelle de la limite de l'unité avec échelle affichée (flou).

La classe MainBoardHolder, elle, gère les nombreux composants internes à chaque item : affichage initial (setInitialDisplay) et interactions (setInteractions). Ses attributs les plus complexes sont :

- ❖ Les Spinner de sélection de l'échelle et de l'unité. Chacun possède son propre adapter par défaut, contenant la liste des données possibles, et chaque interaction entraîne une série de vérifications – notamment sur les limites, de modifications, et donc d'envoi Bluetooth.
- ❖ La représentation de la valeur courante en digits, en réalité plusieurs Button regroupés dans un MaterialButtonToggleGroup. La difficulté de ce composant réside dans l'explosion du nombre en caractères, ce qui est le rôle de la méthode setDigitDisplay, et la modification d'un digit en particulier, travail effectué par la méthode variation de MainBoardAdapter.

5.4.2. Classe BackupStoreAdapter

La classe BackupStoreAdapter gère aussi bien l'affichage des emplacements mémoires, sous forme de tableau de booléens (utilisé ou non), que le RecyclerView BackupConfigAdapter et ses boutons de contrôle. Comme l'Adapter précédent, il assure la sélection simultanée d'un seul item, avec un setLastHolderSelected analogue. Contrairement à l'Adapter précédent, il se réfère aux données globales plutôt que de dupliquer ces données en local.

Ses méthodes clés sont les listeners des trois boutons, qui transmettent les strings de contrôle et actualisent tous les données internes de l'application.

La classe interne Holder s'occupe d'afficher l'emplacement mémoire et la bonne couleur de fond : vide non consulté, plein non consulté, vide consulté ou plein consulté. A la sélection, elle récupère les données par appel Bluetooth si nécessaire (mémoire non vide et données pas encore réceptionnées), met à jour les données du BackupConfigAdapter et actualise l'affichage de ce RecyclerView pour afficher les données.

5.4.3. Classe BackupConfigAdapter

La classe BackupConfigAdapter se borne à afficher les données que le BackupStoreAdapter lui a transmis – ou rien sinon.

Sa classe interne se contente d'insérer les données dans la vue de l'item, sans interaction supplémentaire.

6. GESTION DES RESSOURCES ET DES VISUELS

Le dossier res regroupe toutes les ressources de l'application sous forme de fichiers xml.

6.1. DOSSIER VALUES

Les values prennent différentes formes et sont toujours désignées par des identifiants :

6.1.1. Integers

Les valeurs entières de l'application pouvant être configurées « de l'extérieur ». On y retrouve :

- ❖ Les limites pour chaque unité sous écriture scientifique, tant pour leur valeur concrète que pour leur affichage (Ampère : $\pm 5 \cdot 10^{-3}$ I, affiché en 10^{-6}),
- ❖ Les échelles que peuvent prendre les valeurs selon leur unité, à l'aide de tableau d'integers, renseignés en puissance de 10,
- ❖ Le retour haptique signifiant qu'une limite est atteinte sous forme de temps de vibration,
- ❖ Une référence à la largeur de changement de layout : cette valeur doit correspondre à celle inscrite entre w (width) et dp (unité de longueur) lorsqu'on va dans le dossier item_chanel_edit de layout et que, sur le second fichier, on réalise : clic droit -> refactor -> move file.

6.1.2. Strings

Les textes de l'application. Il y a deux fichiers : un par défaut, en anglais, contenant des éléments non traduisibles, et un autre de traduction pour la langue française :

- ❖ Les textes sont regroupés par layout ou ensembles. Ils suivent le formalisme suivant : *groupe_sous-groupe_description*.
- ❖ Les balises xliif sont des zones à remplir :
 - Le numéro après le %, optionnel et suivi d'un \$, indique la position de l'argument. Plusieurs zones peuvent partager le même argument.
 - Le caractère de fin désigne le type de remplissage : décimal, flottant, string.
- ❖ Pour éviter les doublonnages, il est possible de faire référence à d'autres strings, mais pas d'étendre leur texte. C'est ce que nous avons fait pour assembler le tableau des strings de commandes Bluetooth.

6.1.3. Colors

Les couleurs utilisées par l'application, au format hexadécimal. Il est possible de préciser la transparence après le #. On retrouve deux groupes :

- ❖ Les couleurs primaires et secondaires, notamment utilisées par le style de l'application, et fixées grâce à l'utilitaire suivant : <https://material.io/resources/color/>
- ❖ Les couleurs spécifiques aux utilisations plus ponctuelles.

6.1.4. Styles

Les ensembles de règles pouvant définir le thème de l'application dans son ensemble ou d'un composant spécifique. Ici, nous avons uniquement le thème de l'application, qui étend du thème de Material Design avec barre de menu et qui redéfinit certaines couleurs.

6.1.5. Dimensions

Les valeurs définissant certaines tailles d'éléments et de marges en dp, ainsi que certaines tailles de textes en sp. Les identifiants des dimensions suivent le même formalisme que les strings : *groupe_sous-groupe_description*.

Comme pour les strings, il y a plusieurs fichiers de dimensions, chacun dans un dossier intermédiaire : Cela permet d'adapter les dimensions à l'orientation de l'écran et à sa largeur.

6.2. DOSSIERS DRAWABLE & MIPMAP

6.2.1. Drawable

Ce dossier regroupe les éléments visuels de l'application de type icônes, arrière-plan... au format SVG. On peut ajouter des icônes dans Android Studio avec un clic droit -> New -> Vector Asset. Dans la nouvelle fenêtre, il suffit de cliquer sur Clip Art pour avoir accès à une série d'icônes.

6.2.2. Mipmap

Ce dossier regroupe les différentes tailles d'icône de l'application, faisant généralement référence aux fichiers du dossier Drawable. Le format SVG est pour les SDK supérieurs à 25, les images pour les SDK inférieurs. Ces icônes peuvent être générés avec un clic droit -> New -> Image Asset. Il est aussi possible de les générer avec l'utilitaire suivant, avec ajout d'ombrage, mais nécessité de positionner les fichiers dans les bons dossiers à la main : <https://romannurik.github.io/AndroidAssetStudio/icons-launcher.html>.

6.3. DOSSIER MENU

Ce dossier peut contenir plusieurs fichiers dédiés aux menus. Il consiste essentiellement en une liste d'éléments, avec la possibilité d'associer à chaque élément une icône, de les ordonner et de les afficher en dehors de la liste déroulante.

6.4. DOSSIER LAYOUT

Ce dossier contient les affichages de chaque Activity et Fragment, dans plusieurs orientations et largeur d'écran pour certains Fragment, ainsi que les affichages des items des RecyclerView.

Comme pour les strings et les dimensions, les noms des fichiers sont formalisés : *type_nomClasse*.

6.5. CONSTRUCTION D'UN LAYOUT

Un layout peut être créé ou modifié avec un éditeur WYSIWYG (what you see is what you get) avec l'onglet Design, ou directement dans le xml avec l'onglet Code.

6.5.1. Onglet Code

Le code xml permet de rapidement modifier, dupliquer ou supprimer un composant de la vue. Il permet aussi de facilement ajouter des propriétés aux composants. Cependant, il est assez difficile de faire davantage dans ce mode sans avoir de solides connaissances des descriptions xml des composants.

Ce mode d'édition sert toutefois à améliorer la vue de l'onglet Design :

- ❖ Dans le layout de plus haut niveau, ajouter la propriété `tools:context` et préciser la classe associée permet d'avoir le contexte, donc la vue englobante.
- ❖ Dans les `ListView` et les `RecyclerView`, ajouter la propriété `tools:listitem` et préciser le layout des items permet de visualiser l'affichage des items dans ces containers.

6.5.2. Onglet Design

Cet onglet permet de construire le layout par glisser-déposer de composants parmi la liste des composants possibles. Cet onglet permet aussi, pour chaque composant, de voir l'essentiel des propriétés possibles et ainsi modifier certains aspects plus facilement.

Les composants doivent être agencés dans un layout, que ce soit un `LinearLayout` ou un `ConstraintLayout`. Ces éléments permettent de contraindre la disposition des composants à l'intérieur : chaque composant aura deux ou quatre contraintes possibles. On peut les associer aux bords de page ou aux bords d'autres composants. Les marges appliquées aux côtés opposés positionneront leur composant à leur centre relatif, quoiqu'il soit possible d'y appliquer un biais pour les placer plus proche d'un ou de l'autre côté avec un pourcentage.

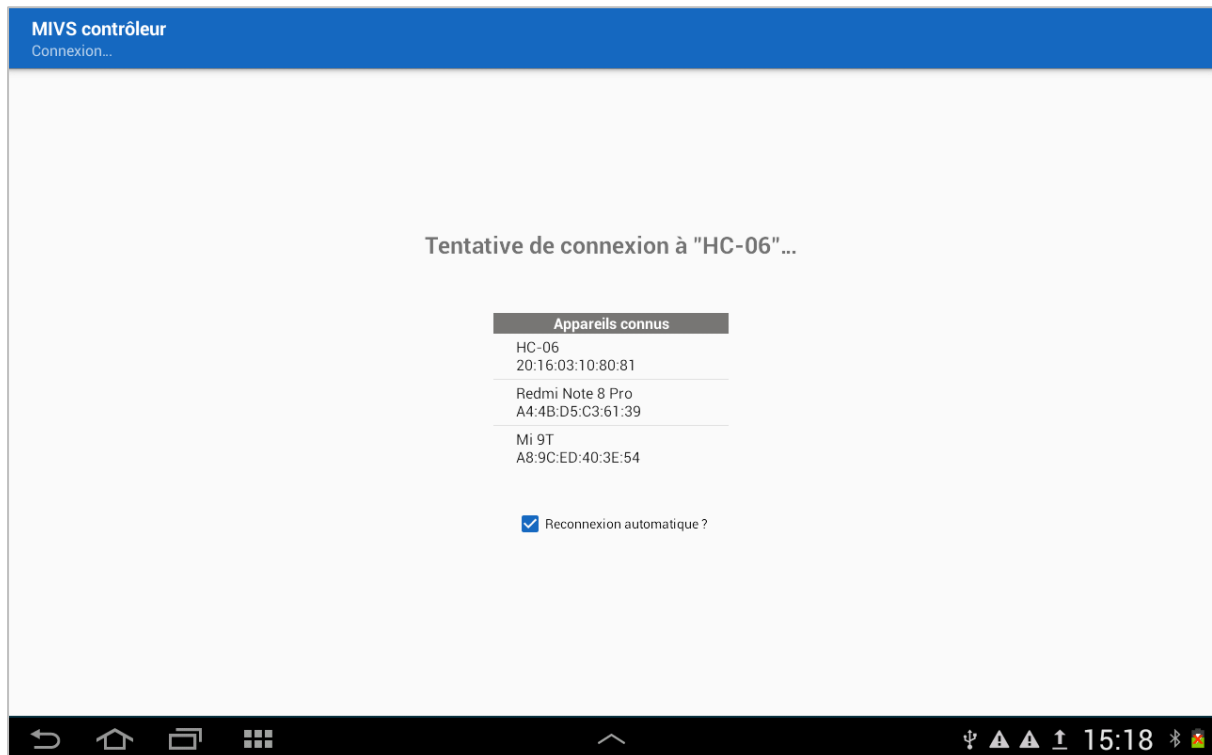
Enfin, chaque composant peut soit prendre :

- ❖ Toute la place disponible (`match_parent`) ;
- ❖ Que la place définie par leurs enfants (`wrap_content`) ;
- ❖ L'espace défini par les contraintes (`match_constraint`) ;
- ❖ Un espace défini (une valeur en dp).

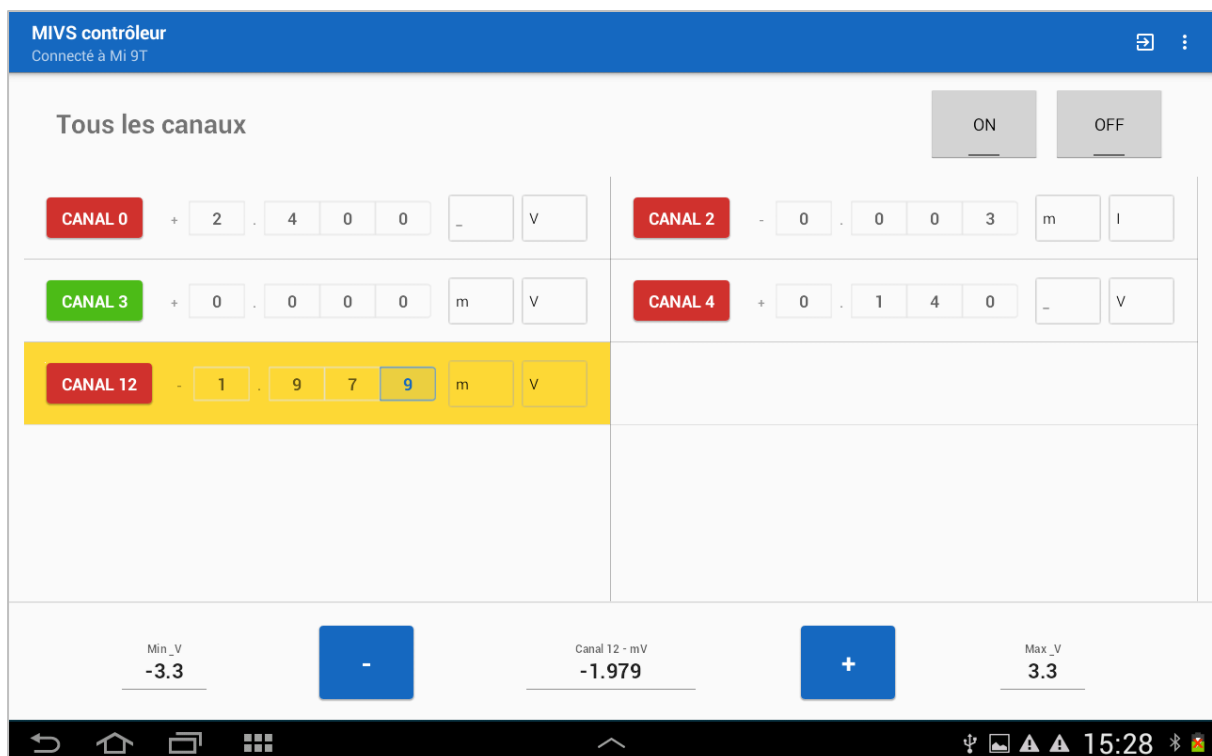
Les éléments du fichier sont listés dans un menu et l'ensemble des modifications effectuées dans l'onglet Design sont aussitôt répercutés dans la version code et vice versa.

ANNEXE 1 : VISUELS DE L'APPLICATION SUR TABLETTE

ConnectFragment



MainBoardFragment



BackupFragment

MIVS contrôleur
Connecté à Mi 9T

Store 0	Canal 0	$-3.3_V \leq 2.4_V \leq 3.3_V$	Canal 2	$-5000.0ml \leq -0.003ml \leq 5000.0ml$
	Canal 3	$-3.3mV \leq 0.0mV \leq 3.3mV$	Canal 4	$-3.3_V \leq 0.14_V \leq 3.3_V$
Store 1	Canal 12	$-3.3_V \leq -1.0_V \leq 3.3_V$		
Store 2				
Store 3				

ENREGISTRE COURANT
CHARGE AFFICHÉ
SUPPRIME AFFICHÉ

Version smartphone

MIVS contrôleur
Non connecté

Sélectionnez un appareil

Appareils connus
GT-N8010
30:D6:C9:17:1C:7F
tel Thib
A4:70:D6:73:A2:32
HC-06
20:16:03:10:80:81
☒ Reconnexion automatique ?

MIVS contrôleur
Connecté à GT-N8010

Tous les canaux

CANAL 0

+ 2 . 4 0 0 _ V

CANAL 2

- 0 . 0 0 3 m l

CANAL 3

+ 0 . 0 0 0 _ V

CANAL 4

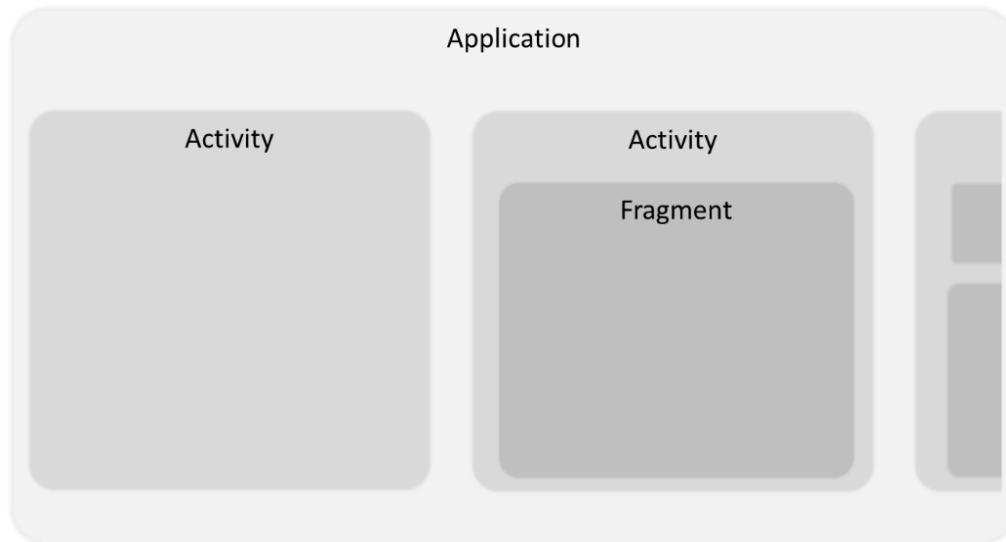
Min ui -5000.0 - + Max ui 5000.0

Canal 2 - ml -0.003

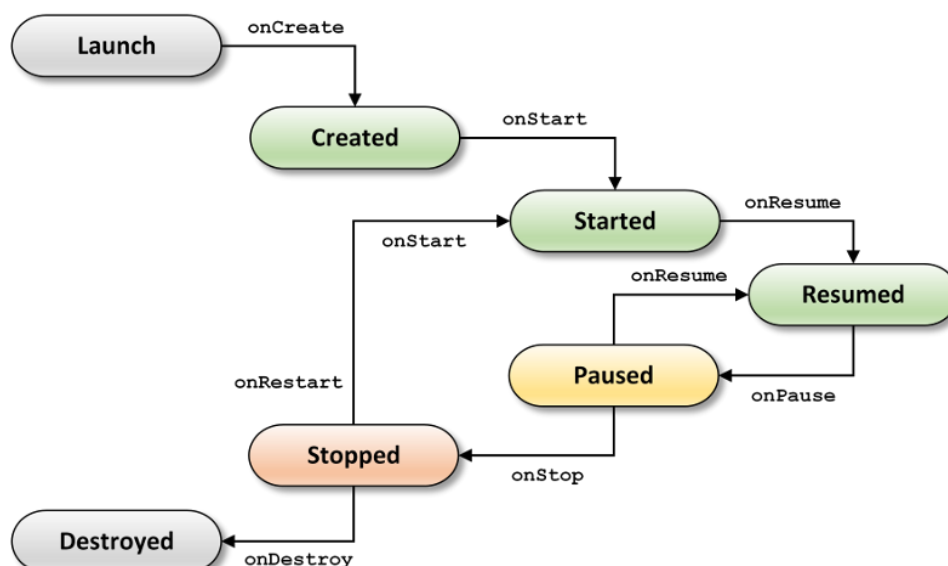
Store 1
Store 2
Store 3

ENREGISTRE COURANT
CHARGE AFFICHÉ
SUPPRIME AFFICHÉ

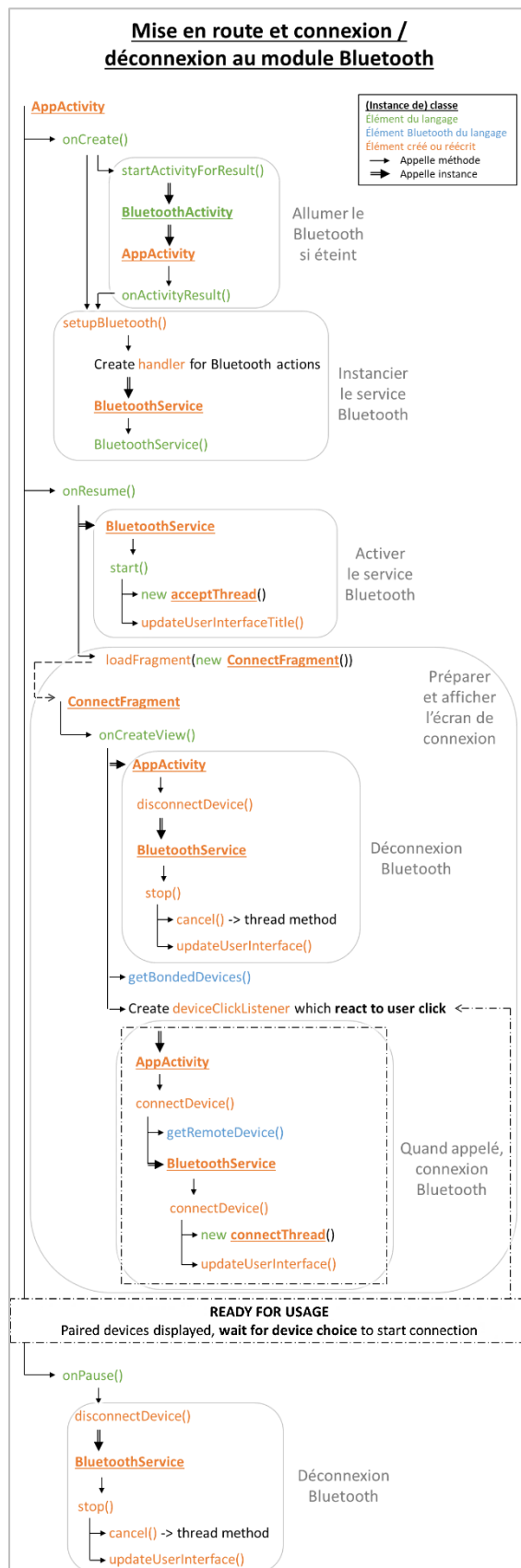
ANNEXE 2 : ARCHITECTURE SOUS ANDROID



ANNEXE 3 : CYCLE DE VIE ANDROID



ANNEXE 4 : COMMUNICATION BLUETOOTH



Handler du service Bluetooth

```

switch (msg.what) {
    case BluetoothConstants.MESSAGE_STATE_CHANGE:
        switch (msg.arg1) {
            case BluetoothService.STATE_NONE:
                update BluetoothConnect display
            case BluetoothService.STATE_CONNECTING:
                update BluetoothConnect display
            case BluetoothService.STATE_FAILED:
                update BluetoothConnect display
            case BluetoothService.STATE_CONNECTED:
                update BluetoothConnect display
                request init_main
            case BluetoothService.STATE_DISCONNECT:
                load ConnectFragment
                clear data
        }
    case BluetoothConstants.MESSAGE_RECEIVE:
        if (data.startsWith("channelList", 2))
            string to data
            apply data
            load MainBoardFragment
        else if (data.startsWith("init_stores", 2))
            string to data
            apply data
            update BackupFragment display
        else if (data.startsWith("store_get_", 2))
            string to integer
            apply data
            update Fragment display
        else if (data.startsWith("store_save_"))
            string to integer
            string to data
            if not initialized
                request init_stores
            apply data
            update BackupFragment display
        else if (data.startsWith("store_load_"))
            string to integer
            if not initialized
                request init_stores
            if not initialized
                request store_get_ + integer
            apply data
            update MainBoardFragment display
        else if (data.startsWith("store_de_"))
            string to integer
            if not initialized
                request init_stores
            apply data
            update BackupFragment display

        else if (data.equals("init_main"))
            send fake data
        else if (data.equals("init_stores"))
            send fake data
        else if (data.startsWith("store_get_"))
            send fake data

        else //some channel attributes only
            string to data
            update Fragment display
}
  
```