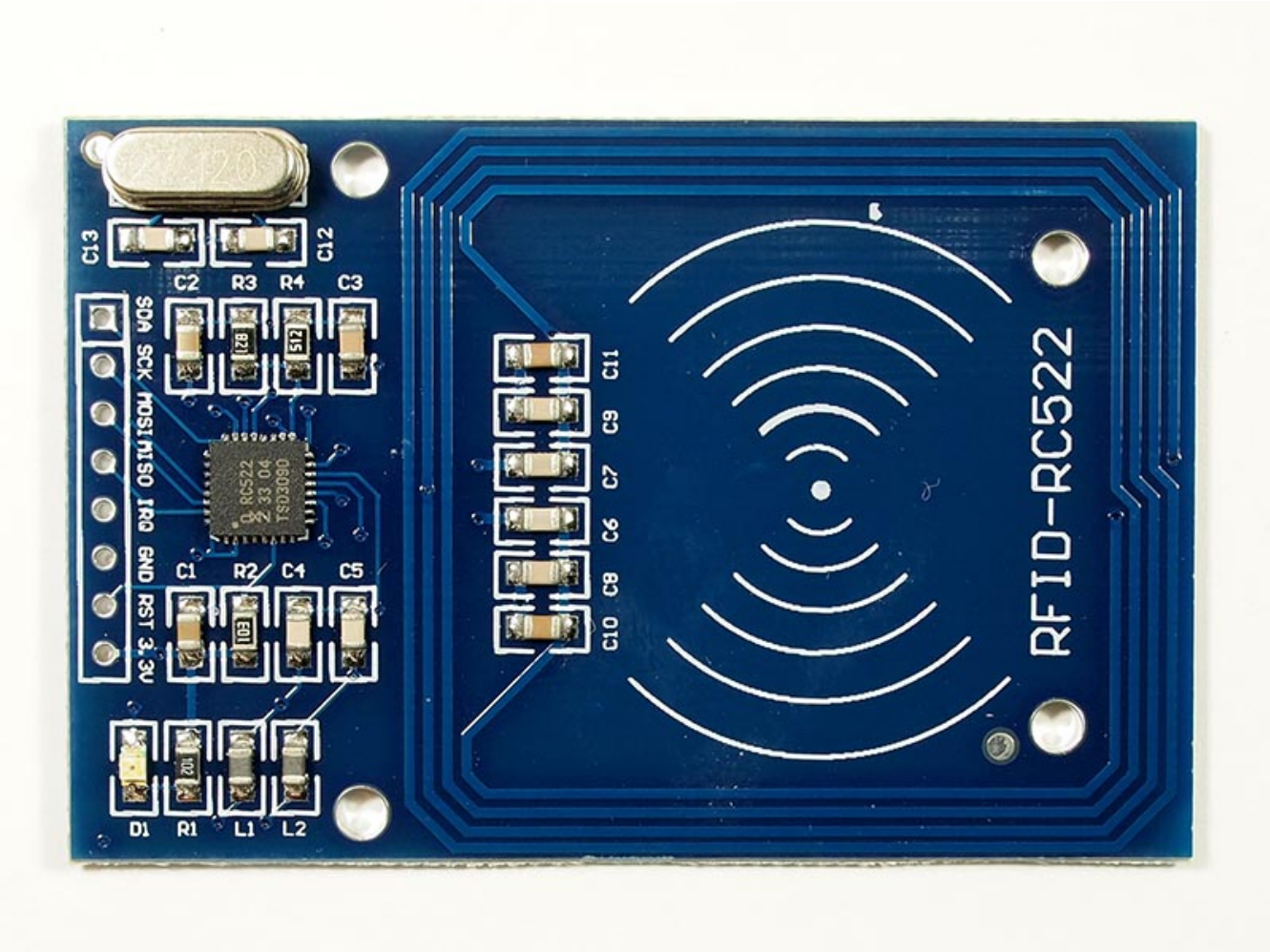
Driver pour MFRC522

Table of Contents

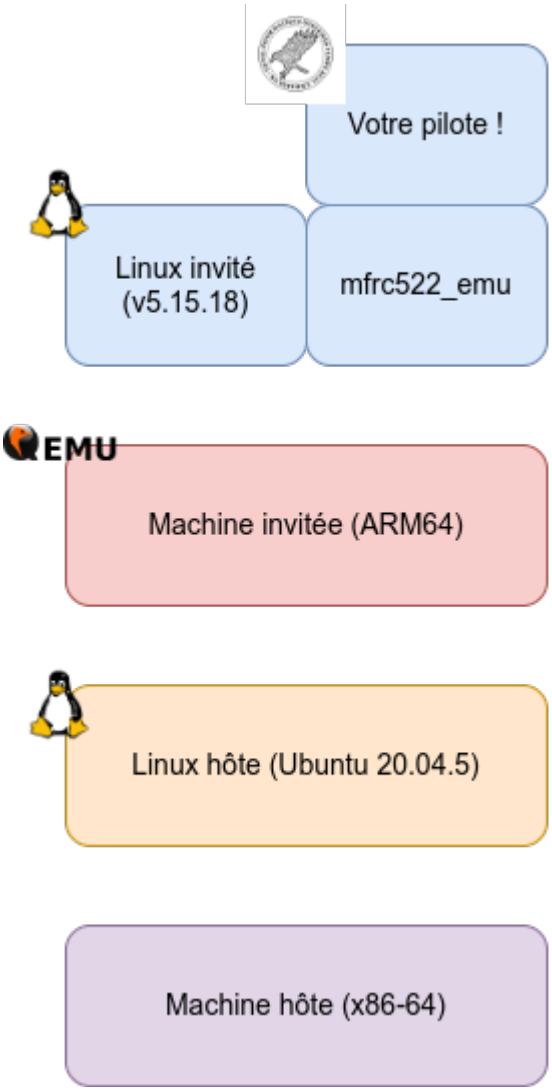


En deux mots

Nous allons écrire un **driver pour le chip MiFare RC522** (MFRC522), qui permet de communiquer avec des puces RFID. C'est donc ce qu'on appelle communément un **"lecteur de cartes";** une "carte" contenant une puce avec une mémoire extrêmement limitée.

Comme vous allez le voir, écrire le code pour communiquer avec le module de base (sans RFID) sera déjà bien assez pour apprendre. Les premières parties de ce sujet introduisent les divers sujets techniques autour du projet, puis en fin de document vous trouverez les paliers à franchir et modalités de rendu.

Voici un joli diagramme en couches du système sur lequel vous allez travailler. Plus d'infos dans la suite !



Présentation du matériel

Ce projet se faisait auparavant avec un vrai lecteur de cartes physique, une carte iHaospace MFRC522. On se propose désormais de le mener à bien avec une version **émulée** de la carte.

On émule le MFRC522 mais aussi la machine sous laquelle Linux va tourner: tous les étudiants utilisent ainsi le même environnement de travail.

Lecteur de carte MFRC522

L'émulateur MFRC522 est un module Linux utilisant l'API *regmap* du kernel Linux pour exposer une interface haut-niveau type SPI/I2C, sauf que les *callbacks* de lecture/écriture y sont personnalisées entièrement, permettant ainsi d'émuler des registres, avec une machine à états imitant le comportement du matériel originel.

A la base, il s'agit vraiment d'une petite carte avec un processeur minimaliste, un ensemble de registres, et des antennes standards ISO/IEC 14443 pour communiquer en RFID. La carte expose quelques fils pour se connecter, originellement en SPI.

L'implémentation d'un driver pour la vraie carte fait partie des bonii.

Board ARM hôte

L'utilisation d'un *device tree* impose de ne pas être sous architecture Intel commune à nos laptops et autres ordinateurs de bureau. Pour éviter de dépendre, là encore, de matériel, on fera tourner un **Linux v5.15.18** sur une **cible ARM "virt" émulée par QEMU.**

La machine "virt" est purement virtuelle et créée de toutes pièces par les développeurs de QEMU; vous n'en trouverez pas sous forme physique. Pourquoi utiliser une machine qui n'existe pas dans la vraie vie ? Il y a plusieurs raisons, la première étant que la machine "virt" a été créée pour servir de *machine de référence* pour les tests de non-régression sur ARM 64-bits. Pour nous, cela signifie la garantie de continuer à avoir accès à des noyaux Linux à jour, car les deux communautés travaillent de concert. Ce n'était plus du tout le cas sur la machine utilisée les années précédentes: la machine Versatile PB utilisait un noyau v4.9 datant de 2016.

A l'origine, ce projet se réalisait sous Raspberry Pi. Vous pourrez toujours faire le projet sur Raspberry Pi avec la vraie carte, une fois les paliers de base complétés: c'est un bonus.

Mais soyez prévenus: je n'ai pas testé…

Un lecteur de cartes, comment ça marche ?

Cette partie est plutôt pour votre culture, à moins que vous ne soyiez très motivés. :)

Le lecteur: MFRC522

Ce chip implémente la technologie *MIFARE*, de *NXP Semiconductors*. Celle-ci propose un système de carte à puces, basé sur le standard ISO/IEC 14443; celle-ci est à l'origine des puces RFID communiquant sur 13,56MHz que l'on retrouve un peu partout (badges, tickets pour évènements…).

MIFARE englobe plus d'aspects que ces simples tags RFID puisque même notre carte "Vitale" utilise une de ses sous-familles, cette fois pour faire une carte à puce autonome. Dans tous les cas, le but est **l'identification** du porteur de carte.

On a donc deux objets à considérer, définis par le standard:

- Un "lecteur" de puces, qui peut les lire voir en modifier le contenu; on l'appelle le PCD (**Proximity Coupling Device**).
- Une puce "cliente", contenant une petite mémoire, qui suit un format particulier; on l'appelle le PICC (**Proximity Integrated Circuit Card**).

Le tag RFID: ISO/14443

Un tag RFID contient une antenne passive, qui ne s'alimente qu'en présence d'un champ RF (celui du lecteur). De plus, il contient une mémoire suivant un certain format.

Les cartes MIFARE sont des tags contenant chacune un **UID sur 32 bits**, des **clés** servant à

l'authentification, et des données utilisateur.

En l'état actuel de l'émulateur, les cartes MIFARE ne sont hélas pas disponibles; je vous laisse des liens dans les ressources, pour les curieux qui voudraient savoir comment leur carte Vitale marche (en gros).

NOTE	Si vous entrepenez le projet avec le vrai matériel, vous êtes libre d'aller jusqu'à programmer la communication avec les PICC. Ca rapportera plus de points !

L'API Regmap

C'est l'API que vous allez utiliser pour communiquer avec le MFRC522.

Regmap, pour quoi faire ?

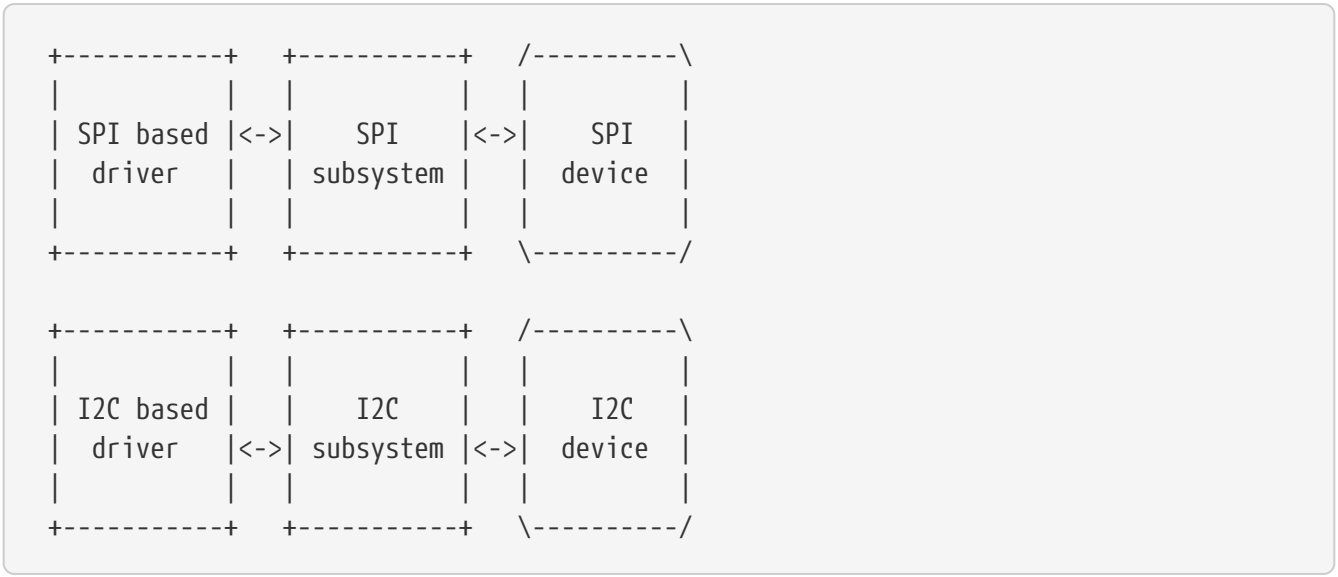
Le chip MFRC522 expose trois interfaces physiques, toutes des bus série:

- UART
- SPI
- I2C

Normalement, on aurait utilisé l'interface SPI.

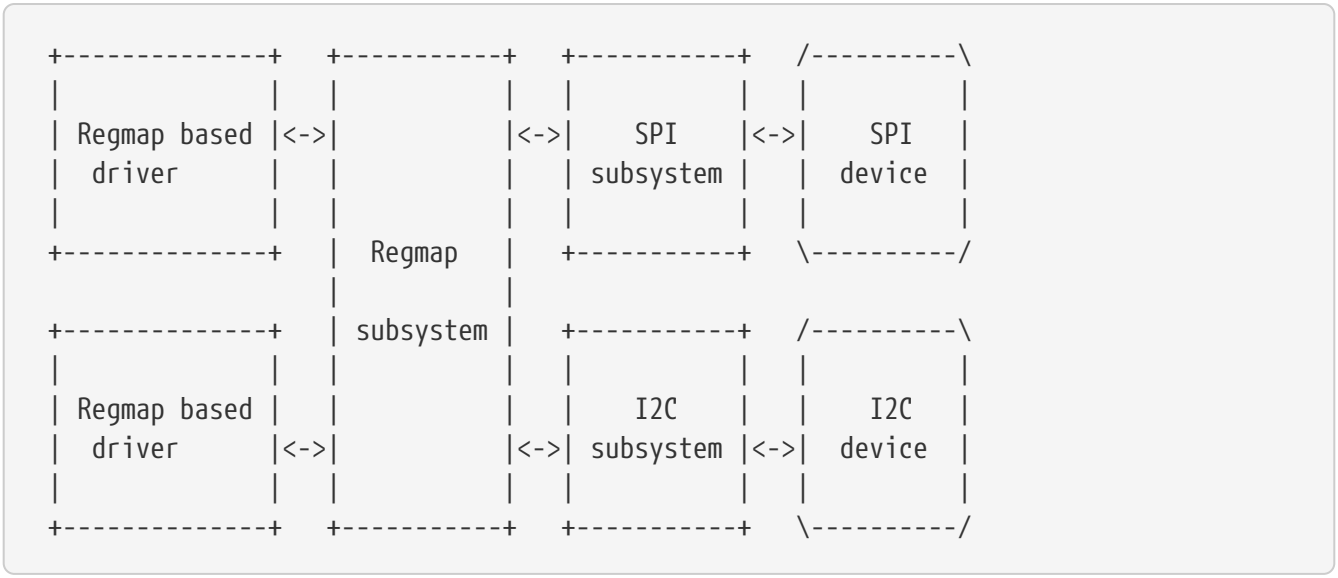
Cependant, nous allons travailler sur une version émulée qui **ne nécessite pas de comprendre ces bus.** En effet, l'API *regmap* avec laquelle l'émulateur est implémentée, va aussi servir à toutes vos communications avec celui-ci. Dans son utilisation "normale", l'API *regmap* est conçue pour faciliter la vie de l'écrivain de pilote Linux, qui aurait besoin d'utiliser le *SPI* ou l'*I2C*. Un peu de contexte…

Ces deux bus simples sont utilisés depuis longtemps dans l'industrie. Historiquement, deux APIs dédiées coexistent: une pour le SPI (`linux/spi/spi.h`), et une pour l'I2C (`linux/i2c.h`). Ces deux bus ont des câblages différents, une logique d'horloge différente (synchrone vs asynchrone)…ce qui justifie de les traiter par des APIs différentes. Sur le fil, on ne s'y prend pas de la même manière pour transmettre la même information. Ce sont en revanche des bus très simples, et on parle de débits avoisinant de base les 100KHz. Mais le peu de fils nécessaires (3 pour le SPI, 4 pour l'I2C) convainquent les ingénieurs électroniciens de s'en servir pour s'interfacer avec des périphériques simples: *watchdogs*, contrôleurs d'écrans LCD, et bien d'autres…



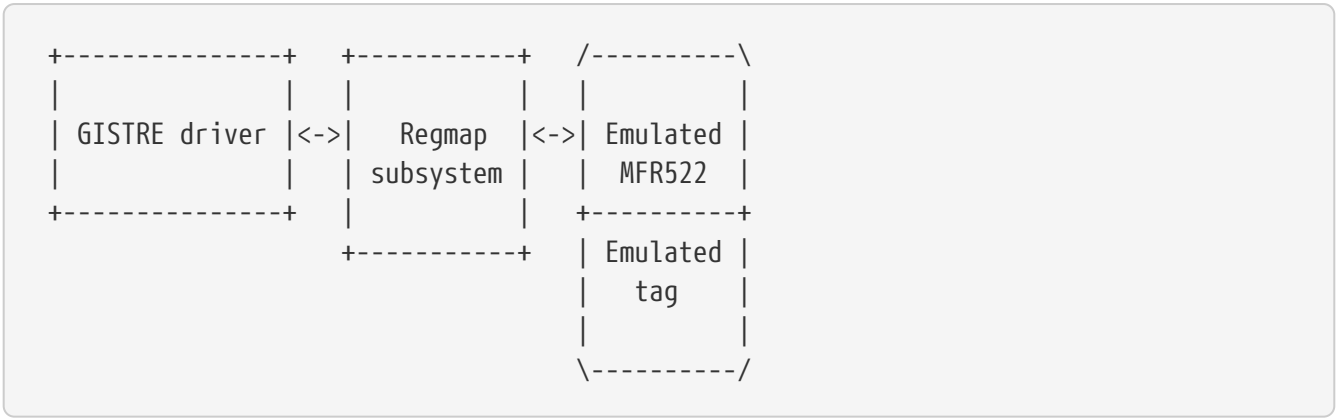
En revanche, en y regardant de plus près, les deux APIs Linux sus-citées exposent quasiment la même chose, à une configuration en amont près. **SPI et I2C se manipulent de façons très similaires.** Pourquoi donc ne pas essayer de **"cacher" les détails sous une API unique**, qui ne demanderait que le type de bus visé et saurait s'occuper du reste ? De là naquit *regmap*. Les fonctions de lecture/écriture qu'elle offre font appel aux "bonnes" fonctions du bus concerné, et s'arrange même pour configurer correctement; une fois qu'on a dit que le périphérique derrière utilise SPI ou I2C, on utilise toujours les mêmes fonctions, dont les plus basiques sont:

```
int regmap_read(struct regmap *map, unsigned int reg, unsigned int *val);
int regmap_write(struct regmap *map, unsigned int reg, unsigned int val);
```



Et l'émulateur, comment fait-il ?

Avec la possibilité de configurer intégralement les fonctions de lecture et d'écriture (normalement pour écrire des procédures qui ont besoin de faire des opérations que les APIs standards ne comportent pas), *regmap* est un outil puissant. Ici, comme dit plus haut, on en profite pour manipuler une `struct mfr522_dev` qui émule tous les registres qui nous intéressent; et au final, on ne sort jamais de *regmap*. **Votre driver va donc s'interfacer comme ceci:**



De votre côté, c'est un plus: **les fonctions d'écriture/lecture sont donc déjà prêtes,** et **vous n'avez à vous soucier ni de comment configurer un bus, ni de comprendre comment se baladent les octets** sur le fil.

Vous voulez écrire dans un registre: vous appelez `regmap_write()` dessus.

Comment coder le projet ?

Pour ce projet, je vous fournis un SDK contenant:

- Un *rootfs*, **rootfs.ext2**
- Un noyau Linux précompilé (non modifié), **Image**
- Un émulateur QEMU précompilé (non modifié), lançable avec **start-qemu.sh**
- Un SDK précompilé, comprenant une chaîne de compilation croisée x86-64 vers ARM64, **aarch64-buildroot-linux-gnu_sdk-buildroot/**
- Un module noyau précompilé, **mfrc522_emu/**
- Un *device tree* dans ses formes binaire et source, **virt.dtb** et **virt.dts**

Une fois décompressé, le SDK devrait ressembler à cela:

```
$> tar xvf gistre24-dril-sdk.tar.bz2
$> cd gistre24-dril-sdk/
$> ls
aarch64-buildroot-linux-gnu_sdk-buildroot  Image  linux-build  mfrc522_emu
rootfs.ext2  rootfs.ext4  start-qemu.sh  virt.dtb  virt.dts
```

Si vous cherchez comment vous servir du SDK pour compiler, lancer Linux, etc. vous pouvez sauter à ce paragraphe.

Sinon, lisez la suite pour comprendre ce qu'il contient.

Contenu du SDK

Rootfs

Le *rootfs*, c'est le système de fichiers que vous allez passer à votre Linux lancé sur une émulation de "Versatile PB" par QEMU, sous forme d'une archive compressée.

Il contient un shell, et la plupart des outils en ligne de commande que vous êtes habitués à utiliser. Il reste limité, mais largement suffisant.

Emulateur MFRC522

L'émulateur est un module *.ko*. Le but est de s'en servir comme d'un *device* "normal", et pour cela vous devez utiliser les symboles exportés par `mfrc522_emu.ko`. Souvenez-vous des premiers cours…

- Utilisez le *header* `mfr522.h` qui décrit une API pour récupérer l'accès à l'émulateur
- Compilez en pointant vers le `Module.symvers`

Du reste, je vous laisse copier le module sur Linux cible et le charger vous-mêmes.

Pièges de l'émulateur MFRC522

Mon émulateur est très modeste. Vous recevrez le code `-EIO` en retour d'une opération de lecture ou écriture, si le registre visé n'est pas supporté.

Pour info, votre module dépendant du mien, vous pouvez forcer Linux à refuser l'insertion de votre module si elle ne vient pas après celle de l'émulateur. Pour cela, rajoutez dans le *scope* global:

```
MODULE_SOFTDEP("pre: mfrc522_emu");
```

Device Tree

Pas de matériel (même virtuel !) sans description dans le *device tree*. Cette version modifiée de QEMU expose le lecteur de cartes émulé sous un contrôleur I2C de type "Versatile PB". Jetez un oeil au *device tree* pour réussir l'exercice associé.

NOTE	Les plus attentifs auront remarqué que je dis que je fais d'habitude utiliser l'interface SPI du MFRC522. En fait, il n'y a pas de SPI qui ne soit supporté par QEMU pour cette *board*. Je "dis" que c'est de l'I2C. Avec *regmap*, je peux mentir si je veux. :)

Utilisation du SDK

Comment compiler mon code ?

Le noyau Linux utilise un système de build permettant aisément la compilation croisée. Commencez par rendre ledit compilateur, fourni par le SDK, visible dans votre environnement shell en mettant à jour `PATH`.

On précise l'architecture pour laquelle on compile avec la variable `ARCH`, le préfixe de la toolchain avec `CROSS_COMPILE`.

Puisque vous voulez compiler pour la version du noyau que je vous fournis, la variable `KDIR` que vous connaissez bien doit pointer vers la racine du répertoire de build, soit `gistre24-dril-sdk/linux-build/`.

Enfin, l'émulateur exporte des symboles. Comme dans l'un des premiers exercices du cours, expliquez où les trouver grâce au fichier `Module.symvers`.

```
$> cd gistre24-dril-sdk/
# Ajoutez le chemin vers le compilateur croisé dans $PATH
$> export PATH=$(pwd)/aarch64-buildroot-linux-gnu_sdk-buildroot/bin:"$PATH"
$> export KDIR=$(pwd)/linux-build
$> export KBUILD_EXTRA_SYMBOLS=$(pwd)/mfrc522_emu/Module.symvers
$> export ARCH=arm64
$> export CROSS_COMPILE=aarch64-buildroot-linux-gnu-
$> cd chemin/vers/votre/module/
$> make modules
```

Comment lancer QEMU ?

```
$> cd gistre24-dril-sdk/
$> ./start-qemu.sh
```

Il y a un utilisateur *root* sans mot de passe. Vous pouvez vous connecter en *root* par SSH/SCP sans mot de passe sur le port 10022:

```
# Attention, "p" minuscule avec SSH, "P" majuscule avec SCP..
$> ssh -p 10022 root@localhost
$> scp -P 10022 mfrc522_emu.ko root@localhost:/root/
```

TIP Il est possible que SSH refuse la connexion sous prétexte que cette configuration n'est pas sécurisée; tentez alors d'ajouter les options `-o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null` à votre ligne de commande.

Contraintes à respecter

Config kernel

Vous n'avez pas à recompiler de noyau Linux pour réaliser ce projet, votre module doit être développé "out-of-tree".

Le projet ne nécessite pas non plus de rédiger un fichier `Kconfig` pour votre module.

Coding style

La *coding style* du kernel Linux doit obligatoirement être suivie. Je passerai *checkpatch* sur votre rendu avant même de le tester, ou de le lire. Vous devez appliquer *checkpatch* sur le code avec la commande suivante, qui est celle que j'utiliserai:

```
$> ./gistre24-dril-sdk/linux-build/scripts/checkpatch.pl --no-tree -f votre_code.c
```

NOTE	Rappelez-vous qu'on utilise Linux v5.15.18; utilisez la version de *checkpatch* qui va avec et que je vous fournis.

TIP	L'option `--no-tree` permet de lancer le script depuis n'importe-où..

Du reste, je vous demanderai la même chose que tout le monde: du bon sens. Si votre variable s'appelle *a*, que votre code ne définit pas de sous-fonctions là où cela aurait aidé à le rendre plus digeste, etc. ne vous étonnez pas si je vous pénalise.

NOTE	La sortie du kernel 5.7 a amené un changement: la limite de 80 colonnes passe à 100.

Questions courantes

Comment sera compilé mon driver ?

Avec votre `Makefile` et le SDK que je vous fournis.

Quel environnement sera utilisé pour tester mon driver ?

J'utiliserai le même SDK que vous, avec la machine virtuelle Ubuntu demandée en début d'année.

TIP	Je n'essaierai pas de tester avec un autre environnement, et ne vous aiderai pas si ledit environnement "ne marche pas". Exemple: sous Arch Linux, il est arrivé qu'avec GCC *bleeding-edge* on ne puisse pas compiler le code du driver.

De quelles APIs ai-je besoin ?

L'API *regmap*, qui est nouvelle pour vous, se trouve dans `include/linux/regmap.h`. Je vous laisse le soin de la découvrir, les deux fonctions citées plus haut sont la base absolue, mais d'autres peuvent s'avérer pratiques.

Côté émulateur, comme dit plus haut, vous n'avez besoin que de `mfrc522.h`. Il vous permet de trouver le MFRC522, et à partir de lui, de sortir une instance de `struct regmap`.

Niveau *format* de données, on ignore complètement le bus: si vous voulez écrire *val* dans le registre *toto*, vous faites `regmap_write(dev, toto, val)`.

Comment communiquer avec le MFRC522 ?

Ce lecteur de cartes expose un **jeu de commandes** accessible depuis le bus I2C émulé. Je vous laisse le soin de lire le manuel du MFRC522 indiqué dans "Ressources" pour comprendre comment parler le langage de la bête.

Le manuel est verbeux. C'est normal: c'est aussi un exercice de tri de l'information pour vous, et l'occasion de lire une *datasheet* raisonnablement complexe.

Implémentation attendue

Cette implémentation de base tire parti de l'émulateur. Les plus motivés souhaitant travailler sur la carte pourront porter leur code sans problème une fois cette partie terminée.

Palier 0 (2 pts)

Posons les bases, en ajoutant un module `gistre_card` simple. Ce n'est même pas un driver en mode caractère, juste un module simple pour l'instant. Votre module doit:

- **S'initialiser et se clore** correctement
- A l'initialisation, **afficher "Hello, GISTRE card !"** dans les traces du kernel

Palier 1 (5 pts)

Préparons les échanges avec le MFRC522. Vous allez faire de votre module un driver en mode caractère simple, qui va trouver le MFRC522 et exploiter une unique commande. Une fois cela fait, ajouter de nouvelles commandes devrait s'avérer assez aisé.

- Commencez par faire de votre module un **driver en mode caractère,** en faisant ce qu'il faut à l'initialisation et au retrait. Vous devez obtenir un majeur alloué *dynamiquement*, et pas statiquement.
- Appelez `mfrc522_find_dev()` pour récupérer un pointeur sur un objet compatible avec l'API Regmap. Ce point me permet de vérifier que vous savez écrire le `Makefile` complet (je vous conseille de lire la doc' de Kbuild)
- **Sur appel de `write()`,** interprétez les données envoyées comme une chaîne de caractère qui donne le nom de la commande à soumettre, et ses éventuels arguments, sous le format `commande:arg_1:arg_2:⋯:arg_N`. J'explicite dans le tableau ci-dessous.
- **Sur appel de `read()`,** renvoyez les données que la commande aura générées.
- En lisant le manuel du MFRC522, vous vous apercevrez que certaines commandes peuvent adopter différents comportements. Pour palier à cela, je vous propose une "API" simple, qui différenciera ces cas.

Votre driver doit supporter les commandes suivantes:

Commande	Arguments	Description
mem_write:<len>:<data>	`len` est le nombre d'octets que fait `data`. Si ce nombre excède la capacité du matériel (indiqué dans le manuel), le code doit "couper"pour ajuster. `data` est une suite d'octets quelconque.	Correspond au mode "écriture" de la commande *Mem*. Ecrit forcément 25 octets, en complétant avec des zéros si nécessaire. Ecrase les données précédentes.
mem_read	Aucun.	Correspond au mode "lecture" de la commande *Mem*. S'il y a des données à lire, renvoie forcément 25 octets, en complétant avec des zéros si nécessaire. Sinon, renvoie zéro. La lecture est *destructrice*: on ne gère pas d'*offsets*, et une fois les données lues, elles sont écrasées par les suivantes.

TIP Ne renvoyez pas zéro dans `write()`, quoiqu'il arrive: ça fera tourner indéfiniment un programme simple comme `echo`.

Comment tester ?

```
# Admettons que votre numéro de majeur soit 255
#> mknod /dev/card c 255 0

#> echo -n mem_write:5:hello > /dev/card
#> echo -n mem_read > /dev/card
#> cat /dev/card
hello
```

Palier 2 (1 pt)

En se basant sur le palier 1, rajoutons le support pour la commande de génération de nombre aléatoire du MFRC522.

Commande	Arguments	Description
gen_rand_id	Aucun.	Correspond à la commande *GenerateRandomId*.

Comment tester ?

```
# Admettons que votre numéro de majeur soit 255
#> mknod /dev/card c 255 0

#> echo -n gen_rand_id > /dev/card
#> echo -n mem_read > /dev/card
#> hexdump /dev/card
0000000 42a9 852e 9200 3990 7476 0000 0000 0000
0000010 0000 0000 0000 0000 0000
0000019
```

Bonus 0 (1 pt)

Ajoutez le support pour `select()` ou `poll()` sur votre *device*, en lecture uniquement.

Bonus 1 (3 pts)

Exposez des statistiques de votre driver au *userspace*, via le *sysfs*.

- Vous pouvez **créer votre propre classe**, ou **réutiliser une existante.**
- Les statistiques attendues sont:
  - Nombre de **bits** (pas octets !) lus dans le buffer interne du MFR522, dans un attribut `bits_read`.
  - Nombre de **bits** (pas octets !) écrits dans le buffer interne du MFR522, dans un attribut `bits_written`.
- On comptera les bits du point de vue de votre *driver* et de son implémentation, pas du point de vue de l'espace utilisateur.

Bonus 2 (2 pts)

Ajoutez une mini-commande de debug.

- **Sur appel de `write()`** avec une nouvelle commande, **activez des traces de debug** supplémentaires dans votre code, qui s'afficheront dans les logs kernel.
- Les traces doivent être **désactivées par défaut.**
- Les traces doivent refléter la taille d'une transaction mémoire, soit, comme vu précédemment, 25 octets. On complètera par des zéros si nécessaire.
- Les traces doivent montrer tous les **octets lus depuis** / **écrits vers** le MFRC522, en se limitant à 5 octets par ligne, sous le format:

  ```
  <OP>\n
  %02x %02x %02x %02x %02x\n
  %02x %02x %02x %02x %02x\n
  <etc.>
  ```

…où <*OP*> est soit "WR" pour l'écriture, soit "RD" pour la lecture.

Commande	Arguments	Description
debug:<mode>	`mode` vaut "on" pour activer les traces, "off" pour les désactiver. Si pas l'un de ces deux arguments, ne rien changer.	(Dés)active les traces de debug avec le format ci-dessus.

Bonus 3 (2 pts)

Adaptez votre code pour pouvoir **gérer plusieurs *devices*** dans votre *driver*. Le nombre maximal de devices à gérer doit être modifiable par un **paramètre de module.**

NOTE	L'émulateur n'émule qu'un seul MFRC522: ce bonus ne demande que d'adapter légèrement votre code.

Bonus 4 (1 pt)

A l'initialisation, récupérer l'instance de `struct mfr522_dev` via l'API que je vous fournis, et **afficher la valeur de la propriété** `version` contenue dans le noeud *device tree* associé. Vous devez bien évidemment utiliser l'API de `linux/of.h` pour cela, et pas me répéter ce que vous êtes allés lire dans le *device tree*…

Bonus 5 (5 pts)

Faites ce projet sur Raspberry Pi, avec la vraie carte comme cible.

Vous trouverez des liens pour vous y aider (comment brancher la carte, etc.) dans la partie "Ressources".

Vous devrez également compiler un kernel pour celle-ci, en arrangeant le *device tree* correctement et en rajoutant mon module d'émulation. Pour cela, vous devrez me demander les sources du `mfrc522_emu`. La procédure de compilation d'un kernel compatible est bien documentée sur le site officiel.

Bonus 6 (2pts)

Dans la continuité du bonus précédent **ne passez plus par l'API Regmap** pour piloter le MFRC522, mais faites-le directement avec l'API SPI ou I2C.

Bonus 7 (5 pts)

Faites ce projet en Rust.

Attention: je ne pourrai pas vous aider. L'utilisation croissante de Rust dans le noyau Linux est un sujet d'actualité, et le défi consistera aussi à marcher sur un terrain peu sûr et à la documentation éparse.

TIP	Ce défi a été relevé la première fois par un groupe de trois GISTRE22.

Bonus N

Ce que vous voulez (enfin, demandez-moi d'abord!) :)

Organisation

Perso, ou en groupe?

Le projet devrait se réaliser en groupe de deux ou trois. Si vous ne pouvez/voulez pas respecter cela, vous devez m'en informer en le justifiant.

Votre groupe doit avoir un numéro unique.

Format de rendu

Le rendu doit se faire sous la forme d'une archive *groupeX.tar.gz*, où "X"est votre numéro de groupe. Son contenu est le suivant:

- **groupeX/**
  - **README**: Contient:
    - Les noms des membres du groupe;
    - Un bref descriptif de la façon dont vous avez organisé votre code;
    - Si vous avez des questions ou remarques particulières, c'est aussi l'endroit où les mettre. Sur le cours, le projet…
  - **src/**
    - **Makefile**: Pour compiler votre/vos module(s) en *out of tree*.
    - ***.c** : Vos fichier d'implémentation.
    - ***.h** : Vos fichiers d'en-tête.

Je vous rendrai, de mon côté:

- Une correction individuelle (par groupe, j'entends): le code source commenté par mes soins, si je trouve une remarque pertinente à vous faire.
- Les réponses à vos questions, si vous en aviez posé dans le *README*.

Notation

Les *paliers* doivent être réalisés dans l'ordre indiqué. Une fois tous les trois paliers réalisés, vous pouvez passer aux *bonii*.

Les *bonii* peuvent être réalisés dans n'importe-quel ordre.

Un palier ou *bonus* rapporte tous ses points s'il est fonctionnel, et moins s'il est incomplet, ou que je trouve quelque chose à redire à l'implémentation (comprendre: vous avez conçu une solution excessivement compliquée ou alambiquée). Je vous encourage à commenter le code que vous écrivez, en particulier si vous souhaitez me soumettre une partie incomplète; je pourrai ainsi mieux comprendre quelle "bille" vous manquait pour compléter la solution.

Je peux également décider de *retirer des points* pour pénaliser du code fonctionnel, mais avec une forme discutable.

NOTE	Il y aura un **dementor** le lundi 12 juin. Le dernier rendu sera organisé à cette date, en fonction des progrès de la classe.

WARNING	Pour le dernier rendu: ça ne compile pas / non-respect de la *coding style* du kernel / non-respect du format de rendu → zéro pointé.

Ressources

MFRC522 & MIFARE

- La carte iHaospace utilisée pour ce projet: https://www.amazon.fr/gp/product/B0716T7R1Y/ref=ppx_yo_dt_b_asin_title_o00_s00?ie=UTF8&psc=1
- Manuel de référence du MFRC522: https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf
- *Layout* d'une carte à puce cliente classique (l'image est lourde): https://upload.wikimedia.org/wikipedia/commons/3/39/MiFare_Byte_Layout.png
- Tuto haut-niveau concis, avec une petite lib' en Python (attention à ne pas copier bêtement le code de la lib'...): http://espace-raspberry-francais.fr/Composants/Module-RFID-RC522-Raspberry-Francais/

ISO 14443

- La partie protocolaire (4ème et dernière partie du standard); attention, il s'agit d'une vieille version (2001), mais l'essentiel reste le même: http://www.emutag.com/iso/14443-4.pdf
- La phase d'entrée en communication avec un tag, succinctement: https://www.redfroggy.fr/le-nfc-et-la-norme-isoiec-14443/
- Vue schématique de la mémoire d'une carte: https://upload.wikimedia.org/wikipedia/commons/3/39/MiFare_Byte_Layout.png

API regmap

- La série de patchs originelle qui a amené l'API, et l'explication de pourquoi elle avait un intérêt: https://lwn.net/Articles/451789/
- Un article introductif (et bien plus exhaustif que ce dont vous avez besoin) à cette API: https://opensourceforu.com/2017/01/regmap-reducing-redundancy-linux-code/ (manifestement copié-collé du livre de John Madieu, "*Linux Device Drivers Development*", où j'ai moi-même découvert l'API)

Raspberry Pi

- Compiler un kernel Linux pour Raspberry Pi: https://www.raspberrypi.org/documentation/linux/kernel/building.md
- *Pinout*: https://fr.pinout.xyz/pinout/spi