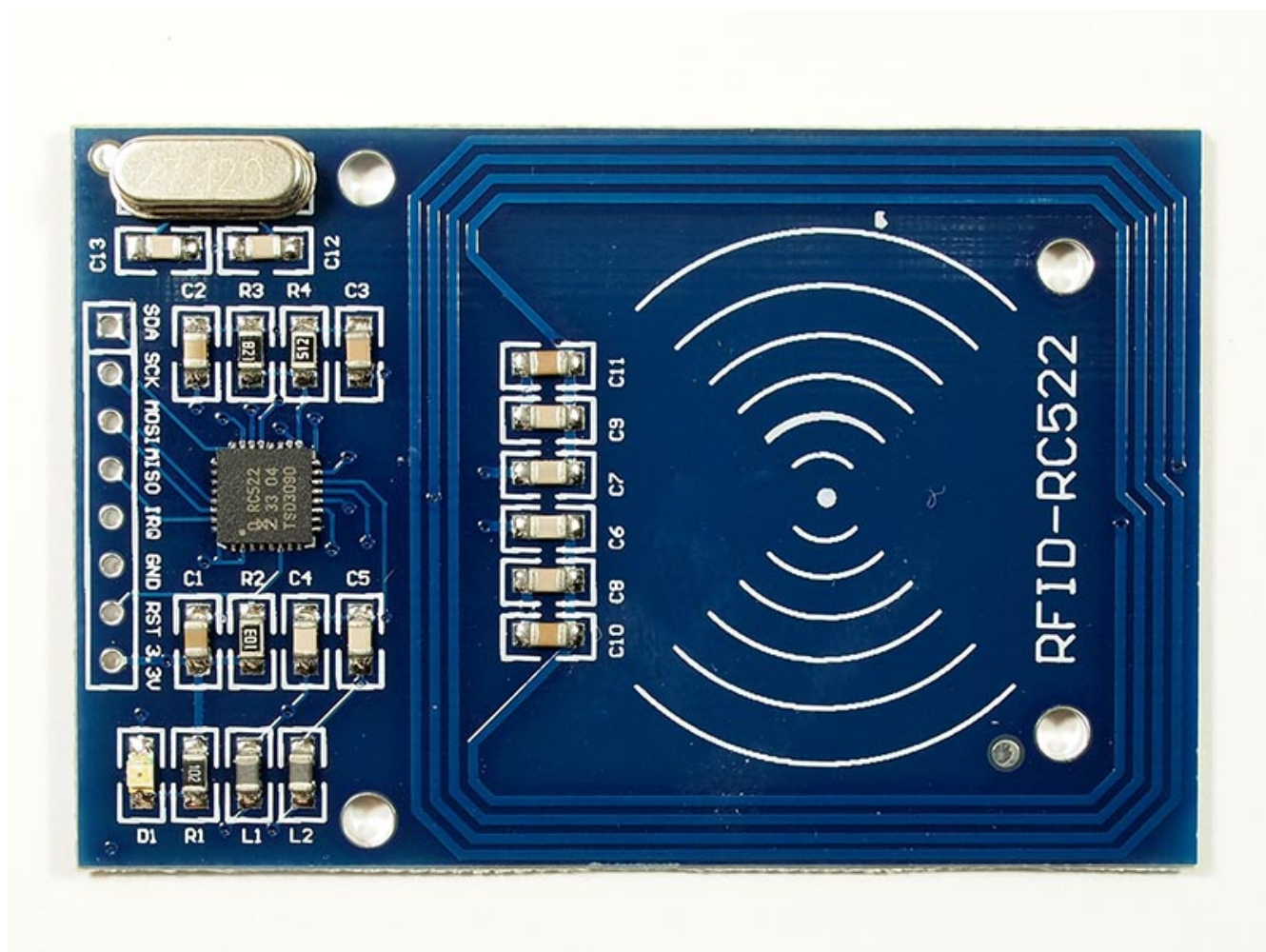


# Driver pour MFRC522

## Table of Contents

En deux mots .....	2
Présentation du matériel .....	2
Lecteur de carte MFR522 .....	3
Board ARM hôte .....	3
Un lecteur de cartes, comment ça marche ? .....	3
Le lecteur: MFRC522 .....	3
Le tag RFID: ISO/14443 .....	4
L'API Regmap .....	4
Regmap, pour quoi faire ? .....	4
Et l'émulateur, comment fait-il ? .....	5
Comment coder le projet ? .....	6
Rootfs ? .....	6
Que contiennent les patchs ? .....	6
Pièges de l'émulateur .....	7
Coding style .....	7
Comment sera appliqué mon patch ? .....	8
Quel environnement sera utilisé pour tester mon patch ? .....	8
Où écrire mon driver ? .....	8
Mais alors, on ne compile pas un module ? .....	8
De quelles APIs ai-je besoin ? .....	9
Comment communiquer avec le MFR522 ? .....	9
Comment lancer QEMU ? .....	9
Implémentation attendue .....	10
Palier 0 (2 pts) .....	10
Palier 1 (2 pts) .....	10
Palier 2 (4 pts) .....	10
Bonus 0 (1 pt) .....	12
Bonus 1 (2 pts) .....	12
Bonus 2 (3 pts) .....	12
Bonus 2 (2 pts) .....	13
Bonus 3 (2 pts) .....	13
Bonus 4 (1 pt) .....	13
Bonus 5 (5 pts) .....	13
Bonus N .....	14
Organisation .....	14
Perso, ou en groupe? .....	14

Format de rendu .....	14
Notation .....	14
Ressources .....	15
MFRC522 & MIFARE .....	15
ISO 14443 .....	15
API regmap .....	15
Raspberry Pi .....	16



## En deux mots

Nous allons écrire un **driver pour le chip 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.

## Présentation du matériel

# Lecteur de carte MFR522

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.

L'émulateur consiste en un module kernel, accompagné de sa représentation dans le *device tree*, tous deux fournis sous forme de patches. Il utilise 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 4.9** sur une cible ARM **Versatile PB émulée par QEMU**. Cette cible a déjà été mentionnée rapidement dans le cours; c'est une carte industrielle relativement ancienne, mais largement suffisante pour nos besoins et bien supportée par QEMU.

A l'origine, ce projet se réalisait sous Raspberry Pi. Si vous en avez l'envie, vous pouvez compiler un kernel pour celle-ci, en arrangeant le *device tree* correctement et en rajoutant mon module d'émulation. La procédure est [bien documentée](#) sur le site officiel.

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 soyez 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 événements...).

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 entreprenez 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 MFR522.

## Regmap, pour quoi faire ?

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

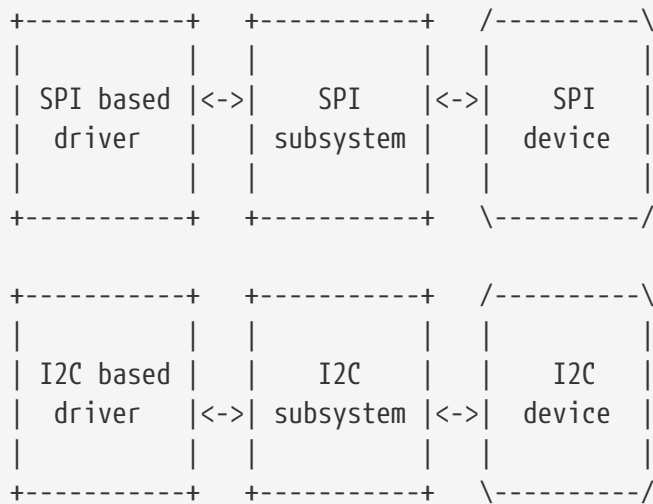
- UART
- SPI
- I2C

**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...

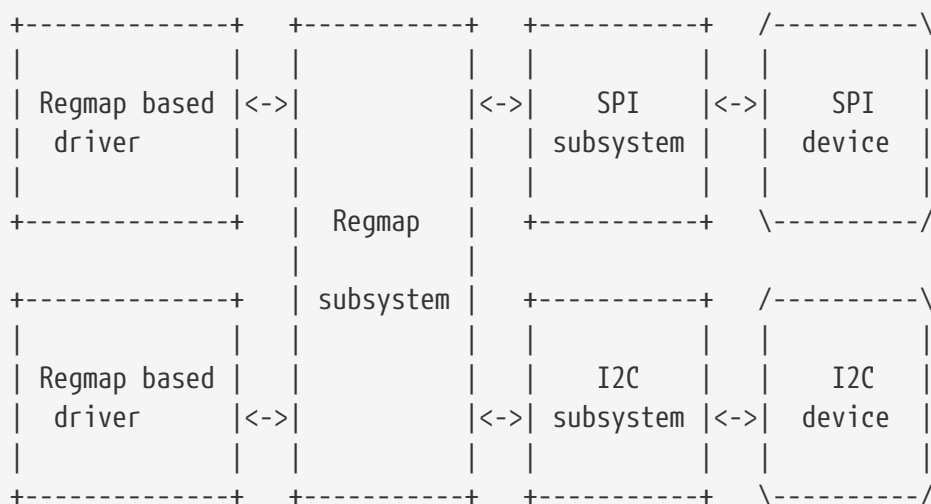
Ces deux bus simples sont utilisés depuis longtemps dans l'industrie. Historiquement, deux APIs dédiées coexistent: une pour le SPI ([linux/spi/spi.h](#)), et une pour l'I2C ([linux/i2c.h](#)). Ces deux bus ont des câblages différents, une logique d'horloge différente (synchrone vs asynchrone)...ce qui 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:

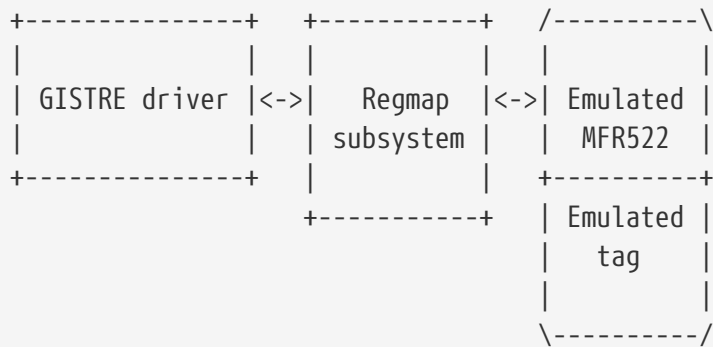
```
int regmap_read(struct regmap *map, unsigned int reg, unsigned int *val);
int regmap_write(struct regmap *map, unsigned int reg, unsigned int val);
```



## 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 deux choses: un *rootfs*, et des patches.

### 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.

Je vous explique comment l'utiliser [dans le paragraphe sur QEMU](#).

### Que contiennent les patches ?

**Mes patches sont à appliquer sur le code entier de Linux.** Comme dit plus haut, on prendra la version 4.9.

```
$> git clone https://github.com/torvalds/linux.git linux
$> cd linux/
$> git checkout v4.9
$> git am *.patch
```

### Emulateur

Un des patches vous fournit le module d'émulation dans `drivers/gistre/mfr522_emu.c`.

Un *header* avec une petite API se trouve dans `drivers/gistre/mfr522.h`, et contient une poignée de fonctions dont vous aurez besoin pour obtenir et communiquer avec l'instance de `struct mfr522_dev` qui représente le lecteur de cartes. Ce header se veut générique: si ce n'est pas un émulateur mais la vraie carte qui est derrière, on devrait pouvoir utiliser le même.

## Device Tree

Pas de matériel (même virtuel !) sans description dans le *device tree*. Un des patches modifie `arch/arm/boot/dts/versatile_ab.dts` pour rajouter le lecteur de cartes sous le bus I2C de la "Versatile PB".

### NOTE

Les plus attentifs auront remarqué que je dis que je fais d'habitude utiliser l'interface SPI du MFR522. 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. :)

## Config kernel

Qui dit compiler pour "Versatile PB", dit utiliser `arch/arm/configs/versatile_defconfig`. Pour s'assurer de compiler mon module d'émulation de base, je l'active par défaut dans ce fichier.

## Pièges de l'émulateur

Mon émulateur est très modeste. Les commandes et registres qu'il supporte sont faciles à trouver: ce sont ceux décrits dans mes *headers*. Vous recevrez de toutes façons le code `-EIO` en retour d'une opération de lecture ou écriture, si le registre visé n'est pas supporté.

Ensuite, comme vous allez utiliser son API implicitement (à la différences de symboles exportés comme vus parfois en cours), vous allez avoir besoin de dire à Linux d'insérer votre module toujours après le mien. Pour cela, rajoutez dans le *scope* global:

```
MODULE_SOFTDEP("pre: mfr522_emu");
```

## 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:

```
$> /chemin/vers/linux/scripts/checkpatch.pl --no-tree -f votre_code.c
```

### NOTE

Rappelez-vous qu'on se basera sur Linux v4.9; il faut donc utiliser la version de *checkpatch* qui va avec.

### 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. Vous y avez donc le droit, je fais cette exception au `checkpatch` de la v4.9.

## Comment sera appliqué mon patch ?

```
$> cp <votre_patch> linux/  
$> cd linux/  
$> git am <votre_patch>
```

Avant de compiler, je vérifierai que votre *driver* et mon émulateur sont bien activés dans la config noyau. Il faudra que ce soit le cas par défaut.

## Quel environnement sera utilisé pour tester mon patch ?

J'utiliserai QEMU avec mon propre noyau Linux compilé, tout comme vous. J'utiliserai Ubuntu 18.04 HWE comme machine de test, avec une chaîne de compilation construite par moi-même avec Buildroot. Elle devrait être sensiblement identique à la *toolchain arm-linux-gnueabi-gcc* disponible dans les dépôts Ubuntu (paquet *gcc-arm-linux-gnueabi*).

#### 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 la v4.9 du noyau.

## Où écrire mon driver ?

Je m'attends à ce que vous écriviez votre driver dans `drivers/gistre/`. Il faudra juste modifier le `Makefile` et le `Kconfig`.

## Mais alors, on ne compile pas un module ?

Non, cette fois-ci je vous demande de compiler votre module en *built-in* dans le kernel.

Vous devrez modifier la configuration *Kconfig* du kernel (voir premier cours) pour que votre driver en fasse partie, parce que la config de base ne le connaît pas. En résumé, il faudra pouvoir faire ça:



```
$> export ARCH=arm
$> export CROSS_COMPILE=arm-linux-gnueabi-

# Config de base pour la "Versatile PB"
$> make versatile_defconfig
# Admettons que votre option Kconfig s'appelle "CONFIG_FOO"...
$> echo "CONFIG_FOO=y" >> .config

$> make
```

#### NOTE

Je sais qu'en cours, cette partie a été survolée. Rien de compliqué, ne vous en faites pas. Inspirez-vous de [drivers/gistre/Kconfig](#) et [drivers/gistre/Makefile](#).

## 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 [drivers/gistre/mfrc522.h](#). Il vous permet de trouver le MFR522, 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 MFR522 ?

Ce lecteur de cartes expose un **jeu de commandes** accessible depuis le bus SPI/I2C émulé. Je vous laisse le soin de lire le [manuel](#) du MFR522 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.

## Comment lancer QEMU ?

Vous avez besoin du paquet "ARM" de QEMU, *qemu-system-arm* sous Ubuntu. Il sait comment émuler la carte Versatile PB.

Une fois votre Linux compilé, lancez:

```
$> qemu-system-arm \
    -M versatilepb \
    -m 128 \
    -kernel arch/arm/boot/zImage \
    -dtb arch/arm/boot/dts/versatile-pb.dtb \
    -initrd rootfs.cpio.gz \
    -serial stdio \
    -append "console=ttyAMA0,115200 initrd=/bin/bash"
```

- `arch/arm/boot/zImage` est votre noyau compilé
- `gcc-arm-linux-gnueabi` est le *device tree* qui va avec (compilé automatiquement)
- `rootfs.cpio.gz` est le *rootfs* que je vous fournis

Il y a un utilisateur *root* sans mot de passe.

## 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)

Prenons en main l'existant.

- **Clonez** les sources de Linux.
- **Patchez**-les avec mes patches.
- **Compilez** le tout.
- **Lancez** ça sous QEMU, avec mon *rootfs*.
- Dites-moi quel **message d'accueil** Linux vous donne au moment de se loguer (`dmesg`).

### Palier 1 (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:

- Avoir son option *Kconfig* dédiée, avec une dépendance sur mon émulateur (si vous vous en servez) et une **compilation conditionnelle** via cette option.
- **S'initialiser et se clore** correctement;
- A l'initialisation, **afficher "Hello, GISTRE card !"** dans les traces du kernel;

### Palier 2 (4 pts)

Préparons les échanges avec le MFR522. Vous allez faire de votre module un driver en mode

caractère simple, qui va exploiter une unique commande du MFR522. 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.
- **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 MFR522, 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
<b>mem_write:&lt;len&gt;:&lt;data&gt;</b>	<b>len</b> est le nombre d'octets que fait <b>data</b> . Si ce nombre excède la capacité du matériel (indiqué dans le manuel), le code doit "couper" pour ajuster. <b>data</b> est une suite d'octets quelconque.	Correspond au mode "écriture" de la commande <i>Mem</i> . Ecrit forcément 25 octets, en complétant avec des zéros si nécessaire. Ecrase les données précédentes.
<b>mem_read</b>	Aucun.	Correspond au mode "lecture" de la commande <i>Mem</i> . 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 <i>destructrice</i> : on ne gère pas d' <i>offsets</i> , 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
```

## Bonus 0 (1 pt)

En se basant sur le [palier 2](#), rajoutons le support pour la commande de génération de nombre aléatoire du MFRC522.

Commande	Arguments	Description
<b>gen_rand_id</b>	Aucun.	Correspond à la commande <i>GenerateRandomId</i> .

### 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
00000000 42a9 852e 9200 3990 7476 c9a9 73f2 034f
00000010 dc84 4dc5 3d40 a59d 0099
00000019
```

## Bonus 1 (2 pts)

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

## Bonus 2 (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 montrer tous les **octets lus depuis / écrits vers** le MFR522, 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
<b>debug:&lt;mode&gt;</b>	<code>mode</code> 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**.

## 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](#)".

## 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...
  - **Un patch:** Contient:
    - Un ajout à `drivers/gistre/Makefile` pour compiler votre/vos module(s) en *built-in*.
    - Un ajout à `drivers/gistre/Kconfig` pour donner l'option de configuration associée à votre module.
    - Votre/vos fichier(s) `.c` et `.h`.

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 vendredi 4 mars. 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](https://www.amazon.fr/gp/product/B0716T7R1Y/ref=ppx_yo_dt_b_asin_title_o00_s00?ie=UTF8&psc=1)
- Manuel de référence du MFR522: <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](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](https://upload.wikimedia.org/wikipedia/commons/3/39/MiFare_Byte_Layout.png)

### API regmap

- La série de patches 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 Madiou, "*Linux Device Drivers Development*", où j'ai moi-même

découvert l'API il y a quelques semaines)

## 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>