Développer son application de détection de chocs sur PEGASE3

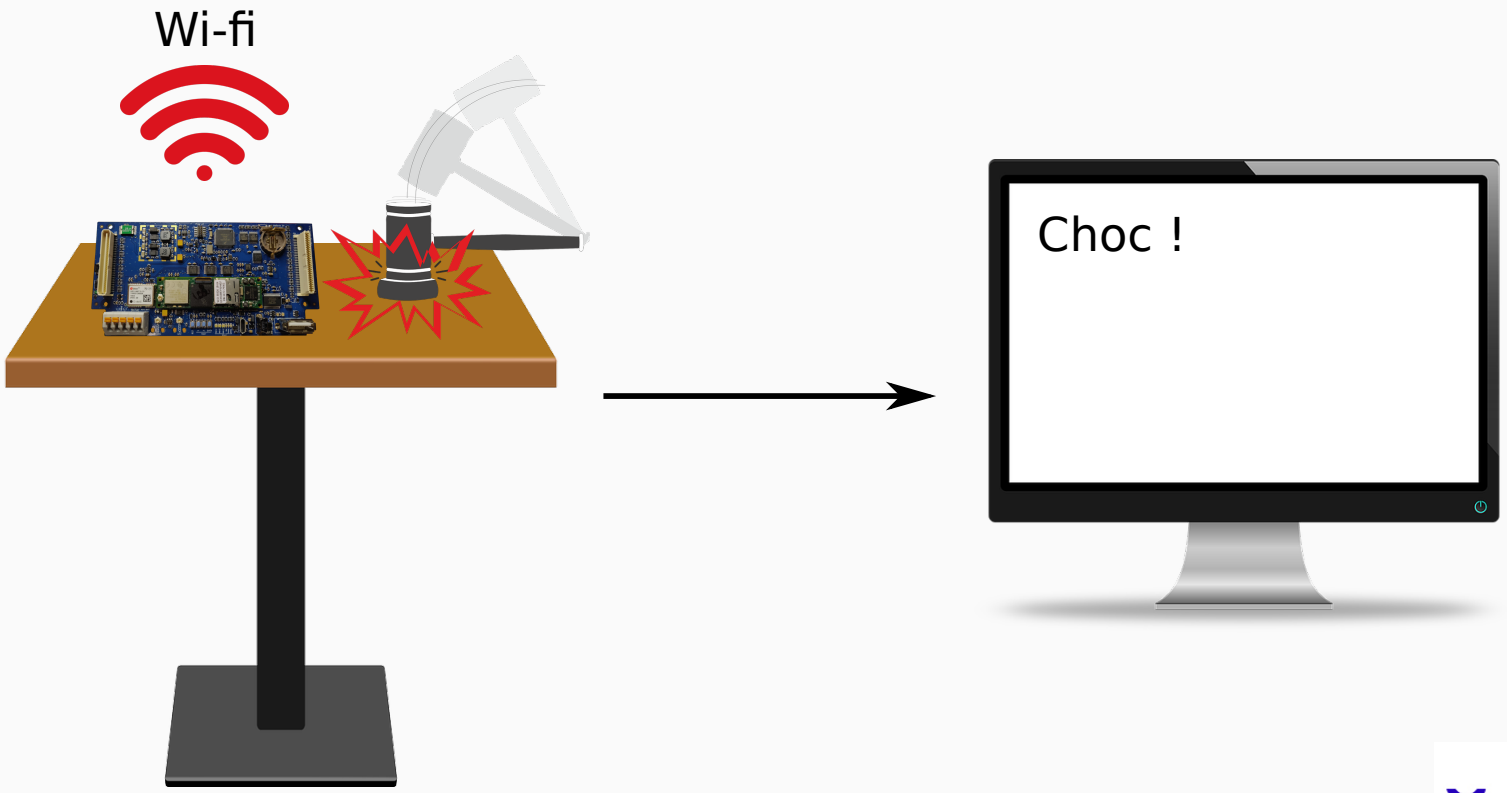
Arthur BOUCHE

28 octobre 2021

Université Gustave Eiffel

L'application

Le but de ce TP est de développer une application qui alerte un serveur lorsque la carte détecte un choc.



Université Gustave Eiffel

L'application

Cette application devra :

- Pouvoir se déployer à l'aide d'un paquet *Debian* (.deb).
- Se connecter automatiquement au WiFi.
- Se lancer automatiquement au démarrage.
- Réaliser des acquisitions d'accélération au niveau de la carte.
- Prévenir un serveur en cas d'une variation trop importante de l'accélération.

Université Gustave Eiffel

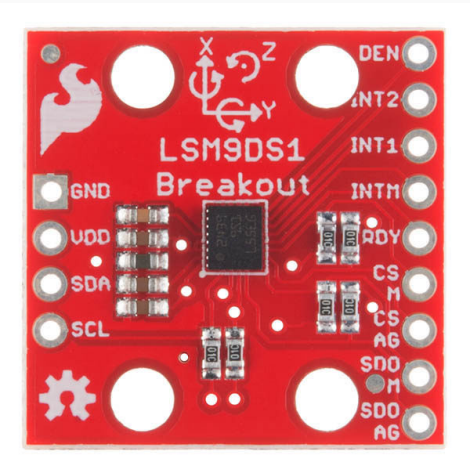
Sommaire

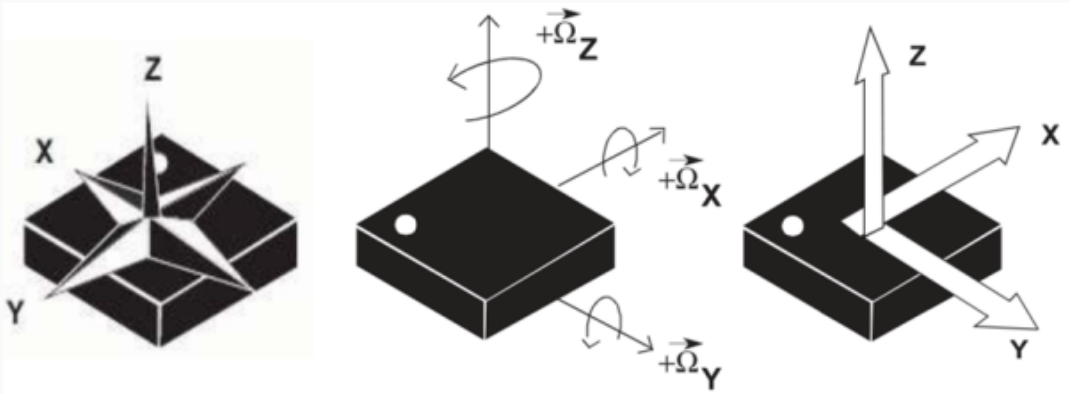
Université Gustave Eiffel

Réaliser le driver LSM9SD1

Présentation

La carte mère PEGASE 3 possède dans son environnement, un circuit LSM9SD1. Le LSM9SD1 est un système intégré comprenant un accéléromètre 3D, un gyroscope 3D, un magnétomètre 3D ainsi qu'un capteur de température.

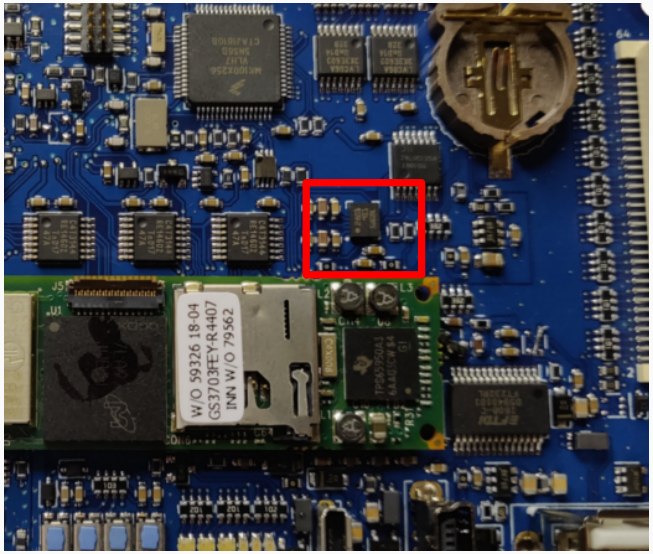




Université Gustave Eiffel

Présentation

Le LSM9SD1 communique en I2C avec la GumStix, il est sur le bus **I2C n°2** à l'adresse **0x6A**. De plus, son interruption INT1_A/G est reliée au **GPIO 93**. Nous allons donc utiliser l'accéléromètre du LSM9SD1 pour la détection de choc.



Dans un premier temps, il est nécessaire de réaliser un driver pour utiliser ce composant.

Université Gustave Eiffel

Device tree

Une première étape est nécessaire en amont du développement du driver, la déclaration du composant dans le device tree.

Qu'est ce qu'un device tree ?

En informatique, un arbre de périphériques (device tree) est une structure de données décrivant les composants matériels d'une architecture particulière afin que le noyau du système d'exploitation puisse utiliser et gérer ces composants, y compris le ou les processeurs, la mémoire, les bus et les périphériques.

Dans notre cas, le fichier **dtb** (pour device tree binary) **pegase3-mere.dtb** (`/kernel/dts/`) contient toutes les informations matérielles de la carte PEGASE 3. Nous allons donc modifier son fichier source (fichier dts, pour device tree source) pour indiquer que le composant LSM9SD1 est sur le bus I2C 2 à l'adresse **0x6A**. Une fois le device tree modifié, nous pourrons commencer la réalisation du driver.

Université Gustave Eiffel

Ajouter son device I2C dans le device tree (1/2)

Décompilez le **dtb pegase3-mere.dtb** dans un fichier **myDts.txt** (`/kernel/dts/`)

```
fdtdump pegase3-mere.dtb > myDts.txt
```

1. Retrouvez le champs "`aliases i2c2`" (nom utilisé pour le Linux kernel mais pas pour les dts) et trouver l'adresse de : `i2c@ALIASES_I2C2`.
2. Dans le nœud (aussi appelé "node") `i2c@ALIASES_I2C2` retrouvez le champs : `ti,hwmods = YYYY`.
3. Pour faire appel à ce nœud il vous faut utiliser la syntaxe suivante &YYYY.

Université Gustave Eiffel

Ajouter son device I2C dans le device tree (2/2)

Dans pegase3-mere.dts (dts pour "device tree source") ajoutez les lignes (ajoutez un device sur le bus I2C) :

```
&YYYY {
  lsm: lsm@ADDR { // ADDR correspond  l'adresse du lsm sur le bus I2C
    compatible = "ifsttar,lsm";
    reg = <ADDR>; // ADDR correspond  l'adresse du lsm sur le bus I2C
  };
};
```

1. Compilez le nouveau dts et **mettez le dtb sur la carte dans /boot/dtbs/current.**
2. **Redémarrez la carte**
3. Sur le PC décompilez le nouveau dtb (`fdtdump ... myDts.txt`)
   et vérifiez que votre nouveau nœud y est présent.

(1) Faites valider par un encadrant en montrant le nœud lsm dans myDts.txt

Université Gustave Eiffel

Activer fonction probe/remove d'un driver I2C

Dans le driver **lsm9ds1.c** (myAppDetection/driver) ajoutez la structure suivante :

```
static struct i2c_driver lsm_driver = {
  .driver = {
    .name = "lsm_driver",
    .of_match_table = of_match_ptr(lsm_of_match),
  },

  .probe = lsm_probe,
  .remove = lsm_remove,
  .id_table = lsm_id,
};
```

Cette structure est appelée par les fonctions suivantes (à mettre dans le **__init** et le **__exit**)

```
i2c_add_driver(&lsm_driver); // dans la fonction __init
i2c_del_driver(&lsm_driver); // dans la fonction __exit
```

Université Gustave Eiffel

Nous allons maintenant définir les structures et fonctions **lsm_of_match**, **lsm_probe**, **lsm_remove** et **lsm_id**.

Activer fonction probe/remove d'un driver I2C

Dans le cadre d'un driver I2C, il est nécessaire d'ajouter **2 structures :**

1. `of_device_id` : permettant de faire le lien avec le dts

```
static const struct of_device_id lsm_of_match[] = {
  { .compatible = "ifsttar,lsm" },
  {}
};
```

2. `i2c_device_id` : permettant de connaître l'ID du chip I2C. Un même driver peut être utilisé pour plusieurs composants I2C (dans notre cas il n'y en a qu'un).

```
static const struct i2c_device_id lsm_id[] = {
  { "lsm", 1 },
  {}
};
```

Lors de l'appel de la fonction `i2c_add_driver` le Linux va chercher le champs `compatible` dans la structure `of_device_id`, si le linux retrouve ce champ ainsi que **lsm** de la structure `i2c_device_id` dans le **device tree**, alors la fonction **probe sera appelée**.

Université Gustave Eiffel

Activer fonction probe/remove d'un driver I2C

Il faut maintenant définir la fonction **probe** et **remove** :

```
static int lsm_probe(struct i2c_client *client,
                     const struct i2c_device_id *id) {
  printk("lsm_probe !!!\n");
  return 0;
}
static int lsm_remove(struct i2c_client *client) {
  printk("lsm_remove\n");
  return 0;
}
```

Dans la fonction probe le paramètre :

- `struct i2c_client` correspond aux informations de la structure `of_device_id`.
- `struct i2c_device_id` correspond aux informations de la structure `i2c_device_id`.

Modifiez la fonction probe pour afficher le nom (.name) et l'ID (.driver_data) de `i2c_device_id` et l'adresse du chip (.addr) de `i2c_client`. Puis, **compilez** le driver et mettez-le sur la carte, faites `insmod lsm9ds1.ko` pour monter le driver.

Université Gustave Eiffel

(2) Faites valider par un encadrant en affichant, nom, id et adresse du chip

Lire un registre

Voici les fonctions que nous allons utiliser pour lire/écrire dans des registres en I2C.
Pour lire et écrire un octet :

```
s32 i2c_smbus_read_byte_data(struct i2c_client *client, u8 command);
s32 i2c_smbus_write_byte_data(struct i2c_client *client,u8 command,
u8 value);
```

Pour lire et écrire des blocs :

```
s32 i2c_smbus_read_i2c_block_data(struct i2c_client *client,
u8 command, u8 length, u8 *values);
s32 i2c_smbus_write_i2c_block_data(struct i2c_client *client,u8 command,
u8 length, const u8 *values);
```

Dans la datasheet du **lsm9ds1** (lsm9ds1.pdf), trouvez le registre **WHO_AM_I** permettant d'identifier le chip.

(3) Faites valider par un encadrant en affichant la valeur WHO_AM_I.

Université Gustave Eiffel

Lire la température

Coté driver :

Dans la fonction **read du driver LSM9SD1** ajoutez la lecture de la température (registre **OUT_TEMP_L** et **OUT_TEMP_H**) et remplissez le champ **.temp** de la structure **dataAccel_t** (dans lsm9ds1.h) puis renvoyez cette structure à l'aide du copy_to_user.

```
dataAccel_t data;
uint8_t temp[2];
.........// lecture des deux registres de temperature
data.temp= (temp[1] << 8) | temp[0];//temp[1] = tempH et temp[0] = tempL
copy_to_user(...);
```

Coté application :

Dans l'application testerLsm9ds1.cpp (`myAppDetection/driver/testerDriver`) faites un "open" puis un "read" toutes les secondes pour afficher la valeur de la température (après conversion).

```
Read(...)
temp = (dataToRead.temp/16.0)+25; //conversion de la temprature (float)
printf("temprature read est %f\n",temp);
```

(4) Faites valider par un encadrant, que remarquez vous ?

Lire la température

Le **LSM9SD1** est par défaut **non activé**, donc la température est fixe. Pour activer l'acquisition de la température, il est nécessaire **d'activer l'accéléromètre**.

Coté driver :

Dans le driver LSM9SD1, mettez en place un "ioctl"
≪ **IOCTL_START_ACCEL_10HZ** ≫ qui écrit **START_10HZ (0x20)** dans le **registre CTRL_REG6_XL (0x20)**

```
switch(cmd)
{
    case IOCTL_START_ACCEL_10HZ:
      // activer avec freq=10hz et + ou - 2g
      i2c_smbus_write_byte_data(client, CTRL_REG6_XL, START_10HZ);
      break;
}
```

Coté application :

Dans l'application faites appel à cette **ioctl** pour activer l'accéléromètre puis faire la lecture de la température toutes les secondes.

(5) Faites valider par un encadrant en faisant varier la température lue

Université Gustave Eiffel

Activer et lire les 3 axes (x,y,z)

Maintenant que l'accéléromètre est activé et que la lecture de la température fonctionne, nous allons **activer les axes x,y,z** et venir les **lire**.

Coté driver :

- Dans la fonction **IOCTL** du driver ajoutez un nouveau ioctl pour activer les 3 axes (x,y,z) à l'aide du registre **CTRL_REG5_XL** (voir datasheet lsm9ds1).
- Dans la fonction **READ** du driver, venir lire les registres **OUT_X_L_XL** à **OUT_Z_H_XL** puis les transmettre au **userspace** avec la même structure que pour la température.

Coté application :

Dans l'application, affichez la température ainsi que les trois axes (**double**) en **mg**.
Pour la conversion des axes, chercher dans la section "**Sensor characteristics**" du pdf du lsm9ds1.

(6) Faites valider par un encadrant en constatant les accélérations

Université Gustave Eiffel

Lire les 3 axes sur data ready

Nous avons maintenant une application fonctionnelle qui permet de lire la température et l'accélération sur les trois axes. Il reste cependant un problème, la vitesse de rafraîchissement de la valeur des axes dépend de notre application et non de la vitesse d'acquisition du LSM9SD1. La prochaine étape sera donc de lire la valeur des axes sur l'évènement de **data ready** (interruption sur le GPIO 93).

3 niveaux de modification seront nécéssaires :

- **Device tree :**
  - ajouter l'interruption sur le bon GPIO
- **Driver :**
  - Configurer le LSM9SD1 en mode data ready.
  - Ajouter la gestion d'interruption.
  - Ajouter mécanisme de "read bloquant".
- **Application :**
  - Configurer le LSM9SD1 en mode data ready.
  - Utiliser un read bloquant.

Université Gustave Eiffel

Device tree et driver : interruption

Coté device tree :

Ajoutez dans le nœud lsm I2C précédemment ajouté, une interruption sur le gpio 93. Compilez et mettez le nouveau dtb sur la carte. Aide : (YYY - 1) * 32 + XXX = 93

```
interrupt-parent = <&gpioYYY>;
interrupts = <XXX IRQ_TYPE_EDGE_RISING>; /* GPIO_93 */
```

Coté driver :

Dans la fonction **probe** et **remove** du driver ajoutez les fonctions suivantes pour la gestion de l'interruption (`lsm_irq` est la fonction d'interruption) :

```
devm_request_threaded_irq(.......);
```

Université Gustave Eiffel

Application et driver : read bloquant et LSM9SD1 en mode data ready

Read bloquant

1. Dans le driver réalisez un read bloquant débloqué sur interruption en utilisant :
   - Un mutex.
   - La fonction d'interruption.
2. Dans l'application enlevez le sleep.

LSM9SD1 en mode data ready

1. Dans le driver ajoutez un IOCTL pour activer le mode data ready sur la pin INT1_A/G
2. Dans l'application appelez cette IOCTL avant la lecture.

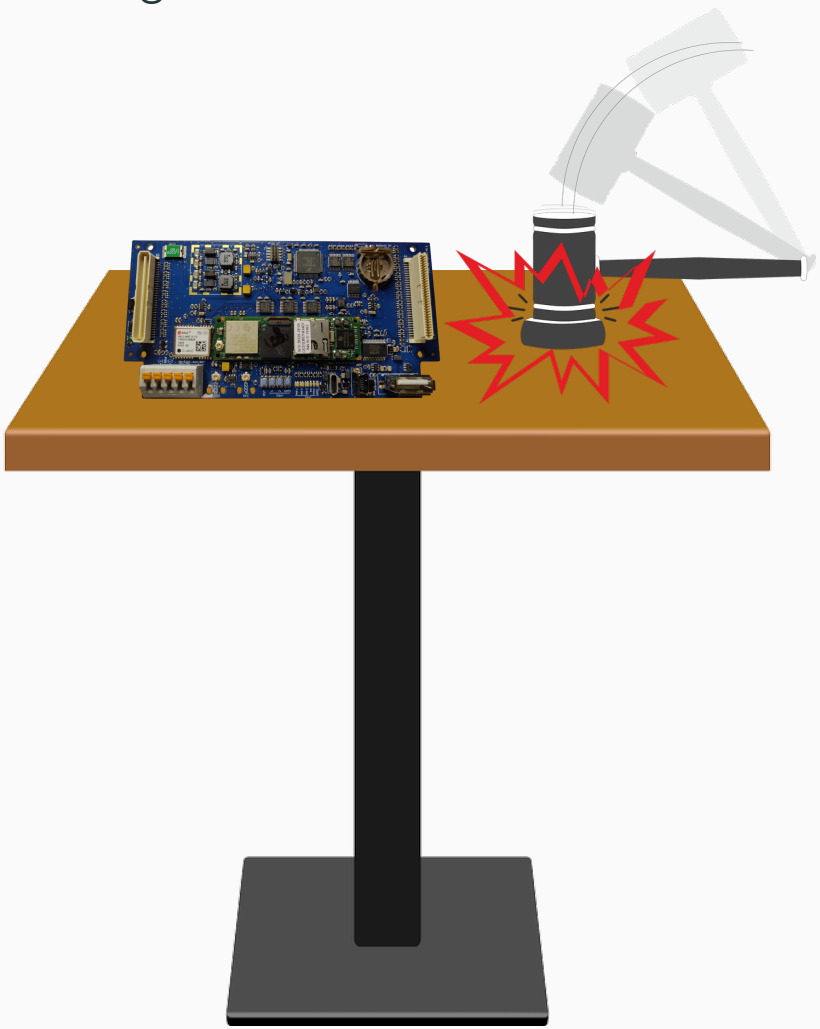
(7) Faites valider par un encadrant en constatant les accélérations à la fréquence de 10hz

Université Gustave Eiffel

Réaliser l'application de détection de choc

Détection de choc

Dans l'application, comparez la valeur des axes d'une lecture à l'autre, affichez un message si la variation d'accélération est supérieur à 0,1g.



(8) Faites valider par un encadrant en affichant le message lors d'un choc

Université Gustave Eiffel

Envoi d'un message par liaison TCP/IP

Nous voulons maintenant envoyer cette information de choc à un serveur distant. Dans le dossier myAppDetection/src, vous trouverez un programme en Qt qui permet de se connecter à un serveur distant et de lui envoyer ≪ hello ! ≫ toutes les secondes. Modifier le code pour envoyer un message personnalisé (demandez à un encadrant l'adresse et le port du serveur). Utilisez la commande "make" pour compiler le programme.

Qu'est ce que Qt

- une API orientée objet et développée en C++, conjointement par The Qt Company et Qt Project. Qt offre des composants d'interface graphique, d'accès aux données, de connexions réseaux, de gestion des fils d'exécution, d'analyse XML, etc
- par certains aspects, elle ressemble à un framework lorsqu'on l'utilise pour concevoir des interfaces graphiques ou que l'on conçoit l'architecture de son application en utilisant le mécanisme des signaux et slots par exemple.

(9) Retrouver votre message sur le serveur distant ou utiliser nc -lvp port

Université Gustave Eiffel

Envoi du signal choc par liaison TCP/IP

Modifiez maintenant le main.c pour qu'il envoi un message personnalisé lorsque que la carte détecte un choc (aidez-vous du code de testerDriver.cpp).

(10) Faites valider par un encadrant en affichant le message sur le serveur distant lors d'un choc

Université Gustave Eiffel

Automatiser le déploiement et le lancement de l'application

Déploiement : utilisation d'un paquet (.deb)

Nous avons maintenant une application fonctionnelle mais nécessitant des interventions humaine pour la lancer ou pour son déploiement, nous allons donc automatiser ceci en 2 étapes :

Déploiement : utilisation d'un paquet (.deb)

1. Le paquet Debian va permettre la compilation automatisée du driver et de l'application afin de créer un .deb.
2. le .deb va permettre de déployer automatiquement les fichiers nécessaires à l'application sur la carte.

Lancement : utilisation d'un service

1. réalisation d'un service allumant le WiFi au démarrage (si non réalisé).
2. réalisation d'un service lançant l'application lorsque le réseau est connecté.

Université Gustave Eiffel

Déploiement : utilisation d'un paquet (.deb)

Afin de faciliter le déploiement d'une application ou d'un driver, dans l'environnement Linux, nous utilisons des paquets (.rpm pour Red Hat et ses suites, .apk pour Android, .PKG pour Mac OS, Playstation, .snap pour Ubuntu. Ici, nous nous focaliserons sur .deb, qui est supporté par Debian et ses dérivées comme Ubuntu ou Linux Mint). Regardez le dossier debian dans ≪ myAppDetection ≫, vous y trouverez 5 fichiers de configuration du paquet :

- **changelog** : contient le numéro de version, de révision, de distribution et l'urgence de votre paquet.
- **compat** : contient le niveau de compatibilité de debhelper.
- **control** : fichier décrivant les informations relatives à notre paquet.
- **rules** : contient les règles utilisées par dpkg-buildpackage pour créer le paquet, assimilable à un Makefile.
- **install** : permet de configurer l'installation des fichiers.

Pour plus d'informations, allez voir ici.

Université Gustave Eiffel

Déploiement : utilisation d'un paquet (.deb)

Pour compiler le paquet faire la ligne suivante dans ≪ myAppDetection ≫

```
dpkg-buildpackage --host-arch armel --target-arch armel -us -uc -b
```

Vérifiez la bonne compilation du paquet.

Vous pouvez maintenant modifier le fichier install (dans le répertoire debian) pour automatiser l'installation de votre driver ainsi que votre application dans les bons répertoires de la cible. Ce fichier install possède une ligne par fichier installé, avec le nom du fichier (par rapport au répertoire de construction principal) suivi d'un espace puis du répertoire d'installation (par rapport au répertoire d'install). Par exemple, si un binaire src/truc n'est pas installé, le fichier install pourrait ressembler à :

```
src/truc usr/bin
```

Dans notre cas, automatisez le déploiement de votre driver dans /home/pegase/ et de votre application dans /usr/bin/

Université Gustave Eiffel

(11) Faites valider par un encadrant en installant votre paquet modifié sur la carte

Lancement : utilisant d'un service

Afin d'automatiser le lancement de l'application, nous allons utiliser un service qui est une des briques de "systemd" (plus d'info ici) l'utilisation des services remplace l'utilisation des "crontab" /"cronjobs". Dans le dossier service, créer un fichier **myapp.service** et y écrire les lignes suivantes :

```
[Unit]
Description=Lance mon application
After=xxxxxxx

[Service]
Type=simple
ExecStart=xxxxxxx

[Install]
WantedBy=multi-user.target
```

Completez la ligne **ExecStart** pour lancer votre application, elle correspond à la ligne de commande qui sera exécutée par le service (exemple "/usr/bin/binaire -parametre valeur").
Nous allons maintenant compléter la ligne « **After=** ».

Université Gustave Eiffel

Lancement : utilisation d'un service

Dans le cas de notre application, elle doit avoir internet pour réaliser la connexion TCP-IP avec le serveur. Or la ligne ≪ **After=** ≫ permet l'exécution de la commande **ExecStart** après un processus donné. Lister les processus à l'aide de la commande suivante :

```
systemctl list-units --type=target
```

Compléter la ligne **After** avec le processus correspondant à notre cas d'application. Une fois le fichier **myapp.service** complété, automatiser son installation dans le dossier /lib/systemd/system lors du déploiement du paquet.

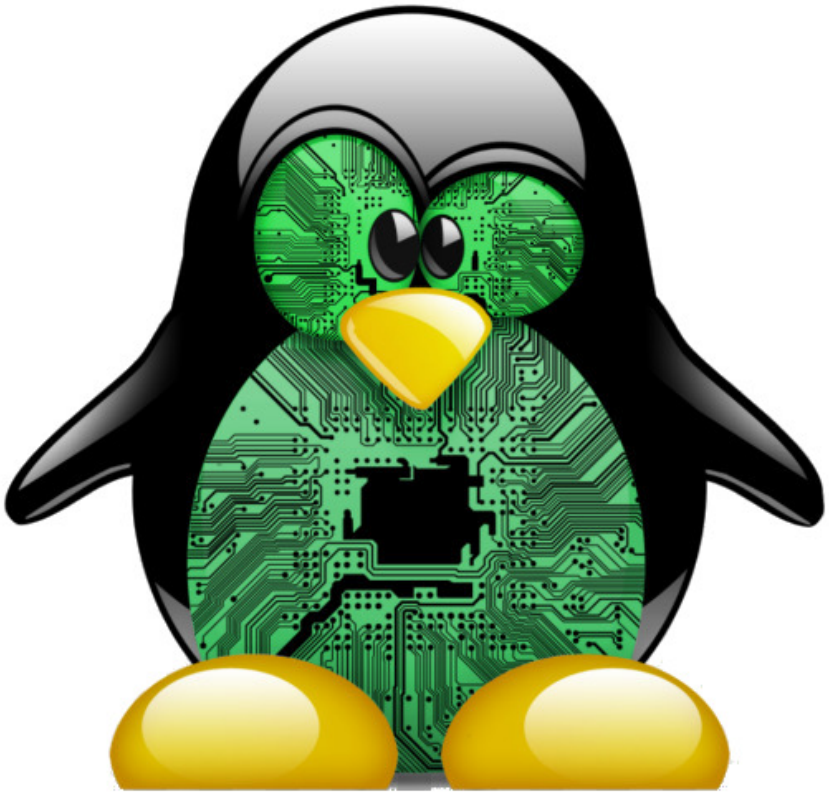
(12) Faites valider par un encadrant faisant fonctionner votre application sans intervention humaine (mise à part mettre sous tension la carte)

Université Gustave Eiffel

Bravo à vous !

Vous maîtrisez maintenant :

- Les device tree.
- La programmation de driver.
- L'utilisation de paquets Debian.
- L'utilisation de service.



Université Gustave Eiffel