Emulation de carte STM32L-Discovery avec Qemu

EID Timothée - MERCIER Michael - CLAVELIN Aurélien

RICM 4 - 2012

POLYTECH GRENOBLE

Table des matières

1 Présentation du projet

Le but de ce projet est d'ajouter le support de la carte STM32L-Discovery dans le logiciel de virtualisation Qemu. Il fait suite au projet de l'année dernière basé lui aussi sur Qemu mais portant sur la carte Stellaris Luminary Lm3s6965.

1.1 Présentation de Qemu

Qemu (Quick EMUlator), est un émulateur libre de processeur basé sur un système de translation binaire dynamique. En plus de l'émulation de processeurs, Qemu offre un grand nombre de fichiers permettant l'émulation de composants matériels, comme le processeur ARM Cortex-M3.
Pour de plus amples informations, vous pouvez vous référer à la page wikipédia de Qemu :
http ://en.wikipedia.org/wiki/QEMU

Qemu propose également une API, appelée chardev, permettant de dialoguer avec le système émulé.

1.2 Présentation de la carte STM32L-Discovery

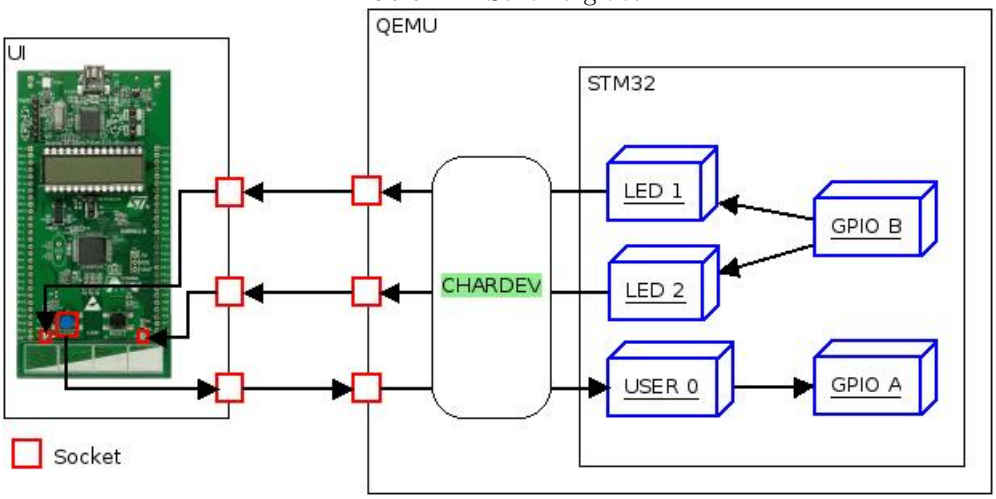
La carte STM32L-Discovery est une carte à très basse consommation permettant de découvrir la gamme STM32L des microcontrolleurs de STMicroelectronics. Elle est basée sur le microcontrôleur STM32L152RBT6 et comporte un kit STlink/V2, un écran LCD, des LEDs, des boutons pressoirs et un touchpad.
Elle est principalement composée d'un processeur ARM Cortex-M3 de 32bits, et de 6 ports GPIOs.

Figure 1 – Carte STM32L-Discovery

2 Travail réalisé

La plus grande difficulté du projet aura été de bien comprendre le fonctionnement de la carte et de Qemu. Nous avons donc passé une grande partie du temps dévolu à lire de la documentation sur la carte et le processeur, et à comprendre le fonctionnement du projet de l'an passé. Nous avons ensuite émulé différents composants de la carte, puis utilisé l'API chardev afin d'observer le fonctionnement des fichiers réalisés. Enfin, pour rendre le projet plus facile d'utilisation, nous avons réalisé une interface.

FIGURE 2 – Schéma global



2.1 Composants émulés

2.1.1 Les GPIOs

L'émulation des GPIOs se fait à l'aide du fichier stm32_gpio.c. En effet, nous avons dû recréer un fichier intégralement, car aucun de ceux fourni avec Qemu ne correspondait à l'implémentation des GPIOs de la carte STM32L-Discovery. C'est le composant principale de notre projet d'émulation de la carte STM32L-Discovery.

2.1.2 Les LEDs

L'émulation des LEDs réalisée dans le fichier stm32_led.c, sert essentiellement à connecter la sortie du GPIO à l'interface utilisateur via l'utilisation de CharDev.

2.1.3 Les Boutons

Le fichier stm32_button.c permet l'émulation de boutons en transmettant les signaux reçus par l'interface CharDev à une entrée de GPIO. C'est grossièrement l'inverse du composant LED.

2.1.4 Le RCC

Le module RCC n'a été que très partiellement implémenté pour des raisons décrites dans la section 4.2 : seule l'écriture et la lecture des registres, sans aucune verification sont fonctionnelles bien que non complètement testées.

2.2 L'interface

Nous avons réalisé une interface permettant de visualiser les composants émulés. L'observation se fait grâce à une communication via des sockets ce qui permet, entre autre, d'observer l'émulation à distance.
Nous avons choisi de développer cette interface en python qui est un langage de haut niveau, portable, et qui nous a semblé relativement simple. Nous avons utilisé la bibliothèque graphique d'origine de python Tkinter, et qui est disponible sur la plupart des distributions Unix.
L'interface est faite pour être utilisée avec python 2.7. Nous avons choisi d'utiliser cette version dans un soucis de cohérence avec Qemu qui recquiert l'utilisation de cette version.

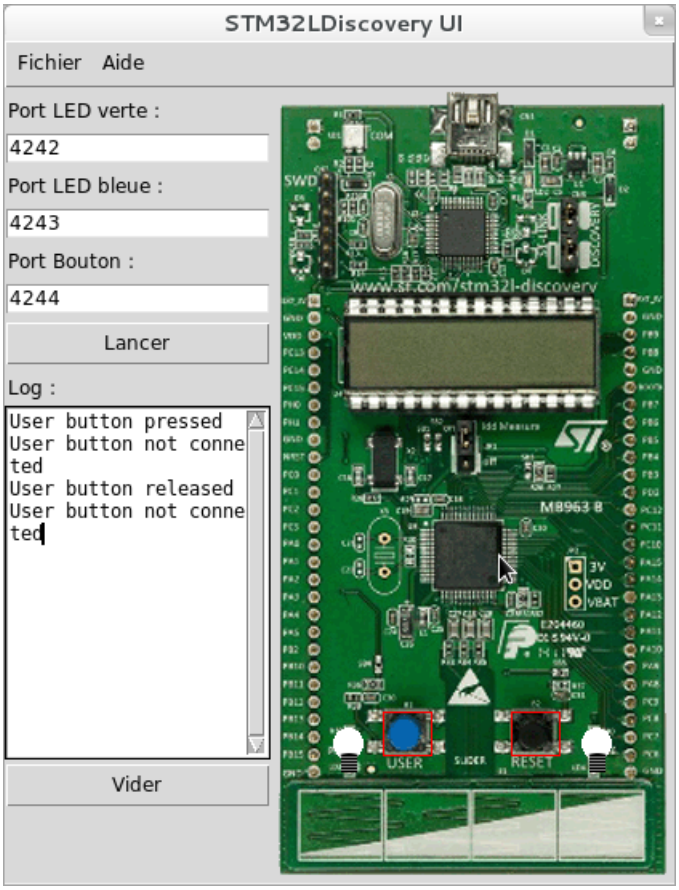


FIGURE 3 – Interface utilisateur

3 Utilisation

3.1 Qemu

3.1.1 Installation

- Récuperer la dernière version de qemu sur http ://wiki.qemu.org/Download (v1.0 lors de la rédaction de cet article)
- Décompresser le dossier Qemu-x.xtar.gz
- Configuration de compilation

Dans le dossier extrait, taper la commande :

```
./configure --target-list=arm-softmmu --python=/usr/bin/python2.7
```

La première option permet de compiler qemu uniquement pour l'émulation de programmes destinés à une architecture ARM. La deuxième option permet de forcer l'utilisation de la version 2.7 de python.

- Compiler les sources

Executer la commande :

```
make
```

Le résultat de la compilation sera placé dans un sous-répertoire en fonction des options utilisées dans la configuation (Ex : arm-softmmu)

3.1.2 Assemblage

Qemu est un programme modulaire. Chaque fichier présent dans le dossier hardware (src/hw) représente l'émulation d'un ou plusieurs périphériques. On trouve par exemple dans src/hw/ les fichiers suivants :

- pl022.c : Port série synchrone
- pl050.c : Interface clavier
- pl061.c : GPIO

Pour émuler un système complet, il faut donc le constituer composant par composant. Chaque composant est crée grâce à la fonction :

```
DeviceState *sysbus_create_simple(const char *name,
                                                target_phys_addr_t addr,
                                                qemu_irq irq)
```

ou, s'il y plusieurs irq :

```
DeviceState *sysbus_create_varargs(const char *name,
                                   target_phys_addr_t addr, ...);
```

Un composant est représenté dans Qemu par la structure suivante contenant les liens avec les autres composants et les zones de la mémoire émulée qu'il utilise :

```
struct SysBusDevice {
   DeviceState qdev;
   int num_irq;
   qemu_irq irqs[QDEV_MAX_IRQ];
   qemu_irq *irqp[QDEV_MAX_IRQ];
   int num_mmio;
   struct {
       target_phys_addr_t addr;
       target_phys_addr_t size;
       mmio_mapfunc cb;
       mmio_mapfunc unmap;
       ram_addr_t iofunc;
       MemoryRegion *memory;
   } mmio[QDEV_MAX_MMIO];
   int num_pio;
   pio_addr_t pio[QDEV_MAX_PIO];
};
```

Cette structure contient une autre structure appelée qdev gérant le lien avec les autres composants. C'est au travers de l'api définie dans qdev.h que ces liens sont définis :

```
qemu_irq qdev_get_gpio_in(DeviceState *dev, int n);
void qdev_connect_gpio_out(DeviceState *dev, int n, qemu_irq pin);
```

Remarque : L'appel à cette API devrait disparaitre dans les prochaines versions de qemu au profit d'un fichier de configuration définissant le composant ainsi que ses connexions avec les autres composants.

Un lecteur attentif aura remarqué l'utilisation de la structure de données qemu_irq dans les fonctions précédentes. Une qemu_irq est en fait la représentation d'une interruption matériel à l'intérieur de Qemu, mais elle est couramment utilisée pour envoyer des signaux entre les différents composants. Elle est de la forme suivante :

```
typedef struct IRQState *qemu_irq;
struct IRQState {
   qemu_irq_handler handler;
   void *opaque;
   int n;
};
```

3.1.3 Mémoire

Pour fonctionner, un composant peut avoir besoin d'utiliser une zone de la mémoire fournie par le système. L'adresse de cette zone est définie lors de l'appel des fonction sysbus_create. Une fonction d'initialisation dans le composant est alors chargée de contacter Qemu afin qu'il lui délègue la gestion de cette zone.

```
iomemtype = cpu_register_io_memory(readHandler, writeHandler, deviceData, NATIVE_ENDIAN);
sysbus_init_mmio(dev, 0x1000, iomemtype);
```

Où 0x1000 est la taille réservée en mémoire pour le composant.

3.1.4 Compilation et lancement du programme émulé

Cross compilation

Dans le cas où le système émulé n'a pas la même architecture que le système hôte, il est nécessaire d'utiliser un utilitaire de cross-compilation à la place de l'utilitaire de compilation standard de l'hôte.

Emulation

- Compiler les sources du programme (dans le cas ou l'architecture du système hôte est différente de celle du système cible, voir la rubrique : cross compilation)
- Lancement de Qemu (dans le cas d'une cible ARM)

```
./qemu-system-arm -M stm32l152rbt6 -nographic -kernel ./main.bin
```

Options :
- -M xxxx : Sélection de la machine émulée
- -nographic : désactive les entrées/sorties (I/O) graphiques et redirige les I/O séries vers la console
- -kernel xxxx : Programme à émuler

La liste d'options peut être retrouvée grâce à l'option –help

3.1.5 Qemu et GDB

Lancement en mode debug

Il est possible de lier Qemu à GDB. Ainsi, il sera aisé de suivre le déroulement du programme émulé.

```
./qemu-system-arm -M stm32l152rbt6 -nographic -kernel ./main.bin -S -gdb tcp::51234
```

Options :
- -S : Active le mode debug
- -gdb tcp : :<port> : Spécifie l'interface de débbogage

Remarque : Il faut également que le programme éxécuté par l'émulateur ai été compilé avec les options de débogage

Connexion avec GDB

Il suffit maintenant de connecter gdb à qemu.

```
arm-none-eabi-gdb
target remote localhost:51234
load ./main.elf
continue
```

3.1.6 CharDev

Présentation

Qemu offre un système de communication avec l'environnement extérieur : les char devices. Ce sont des flux offerts par Qemu sur lesquels un composant peut lire ou écrire. Lors du lancement de l'émulation, il sera possible de connecter ces flux à des structure informatique (telnet, socket, fichier, ...). D'un coté il faudra implémenter le chardev dans un composant de Qemu, puis de l'autre coté il faudra fournir au démarrage de l'application les connexions nécessaires aux chardev.

Implémentation

- Initialisation du chardev

```
chrdev = qemu_chr_find("idCharDev");
if (s->chr) {
    chr_add_handlers(chrdev, canReceiveHandler, receiveHandler, eventHandler, deviceData);
}
```

- Envoi de caractères dans le chardev. Le buffer est un tableau de uint8_t

```
qemu_chr_fe_write(chrdev, buffer, bufferSize);
```

- La réception des données se fait de la même manière à travers la fonction receiveHandler de type void IOReadHandler définie par la signature suivante :

```
void IOReadHandler(void *opaque, const uint8_t *buf, int size);
```

Utilisation

Il suffit de lancer Qemu avec l'option -chardev. Voici un exemple d'utilisation :

```
./src/arm-softmmu/qemu-system-arm -M stm32l152rbt6 -nographic -chardev socket,id=B,port=4242,host=loc
```

La documentation de Qemu expose les différentes options à passer pour connecter les chardev : UserDoc

3.2 Module STM32

3.2.1 Fichiers

Afin d'émuler la carte STM32L-Discovery les fichiers suivant ont été ajoutés au dossier src/hw de Qemu :

- stm32.c pour décrire la carte dans son ensemble
- stm32_gpio.c pour les GPIOs
- stm32_button.c pour les boutons
- stm32_led.c pour les LEDs
- stm32_rcc.c pour le module RCC (ce module n'est que très partiellement émulé et donc non fonctionnel)

3.2.2 Composants émulés

Processeur et Mémoire

Le processeur utilisé par la carte est un processeur Cortex-M3 r1p1 (doc) déjà supporté par Qemu. Pour initialiser celui-ci, il est nécessaire d'utiliser la fonction

```
MemoryRegion *address_space_mem = get_system_memory();
pic = armv7m_init(address_space_mem, flash_size, sram_size, kernel_filename, cpu_model);
```

où pic est un tableau de qemu_irq (voir Qemu#Assemblage) correspondant aux fils d'entrée dans le processeur. Dans notre cas les tailles des différentes mémoires sont :

- RAM : 16 KBits
- Flash : 128 Kbits

Processeur et Mémoire

L'émulation des GPIOs est gérée par le fichier stm32_gpio.c. Les GPIOs sont les connecteurs de base de la carte. Ils permettent de communiquer avec le bouton USER 0 et les LEDs.

Périphériques

L'émulation ne porte pour l'instant que sur les deux LEDs (stm32_led.c) et le bouton USER0 (stm32_button.c) qui communiquent avec l'extérieur de Qemu grâçe à chardev.

Exemple

Nous allons détailler la composition d'un composant à partir de l'exemple de stm32_gpio

- Tout d'abord il faut inclure sysbus

```
#include "sysbus.h"
```

- Les registres utilisés par le composant sont ensuite placés dans une structure :

```
typedef struct {
   SysBusDevice busdev;

   /* Registres GPIO (Reference Manual p119 */
   uint32_t mode; /* Mode */
   uint16_t otype; /* Output type */
   uint32_t ospeed; /* Output speed */
   uint32_t pupd; /* Pull-up/Pull-down */
   uint16_t ind; /* Input data */
   uint16_t outd; /* Output data register */
   uint16_t outd_old; /* Output data register */
   uint32_t bsr; /* Bit set/reset */
   uint32_t lck; /* Lock */
   uint32_t afrl; /* Alternate function low */
   uint32_t afrh; /* Alternate function high */

   qemu_irq irq_out[NB_PIN];
   unsigned char id;
} stm32_gpio_state;
```

- Cette structure est décrite sous la forme d'une VMStateDescription permettant à Qemu de l'enregistrer correctement

```
static const VMStateDescription vmstate_stm32_gpio = {
   .name = "stm32_gpio",
   .version_id = 1,
   .minimum_version_id = 1,
   .fields = (VMStateField[])
   {
       VMSTATE_UINT32(mode, stm32_gpio_state),
       VMSTATE_UINT16(otype, stm32_gpio_state),
       VMSTATE_UINT32(ospeed, stm32_gpio_state),
       VMSTATE_UINT32(pupd, stm32_gpio_state),
       VMSTATE_UINT16(ind, stm32_gpio_state),
       VMSTATE_UINT16(outd, stm32_gpio_state),
       VMSTATE_UINT16(outd_old, stm32_gpio_state),
       VMSTATE_UINT32(bsr, stm32_gpio_state),
       VMSTATE_UINT32(lck, stm32_gpio_state),
       VMSTATE_UINT32(afrl, stm32_gpio_state),
       VMSTATE_UINT32(afrh, stm32_gpio_state),
       VMSTATE_END_OF_LIST()
   }
};
```

- Une structure SysBusDeviceInfo renseigne la fonction d'initialisation, le nom, la taille de la structure des registres ainsi que la VMStateDescription.

```
static SysBusDeviceInfo stm32_gpioA_info = {
   .init = stm32_gpio_init_A,
   .qdev.name = "stm32_gpio_A",
```

```
    .qdev.size = sizeof (stm32_gpio_state),
    .qdev.vmsd = &vmstate_stm32_gpio,
};
```

- Elle est passée en paramètre à la fonction sysbus_register_withprop elle même passée en paramètre de la macro device_init

```
static void stm32_gpio_register_devices(void) {
    sysbus_register_withprop(&stm32_gpioA_info);
}
device_init(stm32_gpio_register_devices)
```

- Voici la fonction d'initialisation du composant :

```
static int stm32_gpio_init(SysBusDevice *dev, const unsigned char id) {
    int iomemtype;
    stm32_gpio_state *s = FROM_SYSBUS(stm32_gpio_state, dev);
    s->id = id;

    //Initialisation de la plage mémoire
    iomemtype = cpu_register_io_memory(stm32_gpio_readfn, stm32_gpio_writefn, s, DEVICE_NATIVE_ENDIAN)
    sysbus_init_mmio(dev, 0x24, iomemtype);

    //Initialisation des pins
    qdev_init_gpio_in(&dev->qdev, stm32_gpio_in_recv, NB_PIN);
    qdev_init_gpio_out(&dev->qdev, s->irq_out, NB_PIN);

    //Initialisation
    stm32_gpio_reset(s);
    vmstate_register(&dev->qdev, -1, &vmstate_stm32_gpio, s);

    return 0;
}
```

les structures stm32_gpio_readfn et stm32_gpio_writefn sont respectivement une CPUReadMemoryFunc et une CPUWriteMemoryFunc et contiennent les fonctions de lecture et d'écriture sur les registres déclarés dans le stm32_gpio_state. Ces fonctions sont de la forme :

```
static uint32_t stm32_gpio_read(void *opaque, target_phys_addr_t offset) {
    stm32_gpio_state *s = (stm32_gpio_state *) opaque;

    switch (offset) {
        case 0x00: /* Mode */
            return s->mode;
        case 0x04: /* oType */
            return s->otype;
        case 0x08: /* oSpeed */
            return s->ospeed;
        ...
```

4 Difficultés rencontrées et solutions apportées

Commme dit dans le paragraphe précédent, la plus grande difficulté aura été de bien comprendre le fonctionnement de la carte et de l'émulation. Au départ, nous avons employé notre temps à la lecture de la documentation, au sujet de Qemu, au sujet de la carte STML-Discovery et aussi du processeur ARM Cortex-M3.
Il nous a également fallu nous approprier un vocabulaire que nous n'avions pas (GPIOs, RCC, ...).

4.1 Programmes de test sur la carte

Pour vérifier notre émulation, il nous fallait un programme simple de test sur la carte. Cependant n'ayant aucune connaissance en éléctronique, nous avons avancé en tatonnant. Bien que la conférence avec les reprsentants de STMicroelectronics nous ai permis de comprendre un peu mieux le focntionnement de cette dernière, leur programme de base était trop compliqué pour que l'on puisse l'utiliser pour notre projet. Bien que le projet Stlink de texane sur github donne un exemple de code utilisable sur notre carte, nous n'avions pas connaissance de tous les fichiers nécessaires au focntionnement de celui-ci. Par exemple au début nous n'avions pas noté que le fichier startup codé en ARM était nécessaire. C'est seulement en tatonant à partir de l'exercice de ST que nous avons réalisé la nécessité de fournir ce fichier startup. !!Programmes de test sur la carte pour tester emulation => manque de connaissance elec... finalement utilisation du startup !!

4.2 Reset and Clock Control

Au cours du projet nous avons compris que le module Reset and Clock Control(RCC) est necessaire à l'utilisation des GPIOs. Ce module permet, entre autre, l'activation des horloges associées aux GPIOs qui enclenchent regulièrement la lecture des registres de données. Cependant, un programme n'activant pas cette horloge fonctionne correctement dans Qemu car la lecture des registres de données est directe. Ce composant a donc été crée afin de permettre la lecture et l'ecriture dans les registres de contrôles mais n'a aucune fonction réélle lors de l'émulation.

4.3 Les interruptions

Le fontionnement des interruptions dans la carte STM32 était assez obscure pour nous au démarrage du projet. Or, dans Qemu une interruption est représentée par une structure nommée **qemu_irq** qui sert à toutes les liaisons entre les composants et contient à la fois le lien et le handler d'interruption ce qui porte à confusion. Nous avons fini par mieux apréhender le problème mais nous n'avons pas eu le temps de développer les modules nécessaires au fonctionnement des interruptions.

4.4 Compréhension iterative

La compréhension à la fois de Qemu et du matériel de la carte STM32 s'est faite de manière iterative. C'est à dire qu'il à fallu lire et navigué dans le code de Qemu (peu, voir pas commenté ni documenté) en se servant de l'exemple de la carte Stellaris pour apréhender les APIs ainsi que les conceptes sous-jacents. N'ayant que quelques notions d'électronique, il à aussi été difficile de comprendre le vocabulaire des documentations techniques et le fonctionnement de la carte. Ces aprentissages se sont fait pas à pas, ce qui à conduit au développement de plusieurs versions intermédiaire de programme de test afin de comprendre un concepte pour pouvoir passer au suivant.

4.5 Adressage de la mémoire

Dans Qemu, l'adressage de la mémoire utilisée par l'implémentation existante du processeur ARM Cortex M3 est différente de l'adressage par default dans la carte STM32L-Discovery : La mémoire Flash se situe à l'adresse 0x0000 0000 pour Qemu alors qu'elle se situe à l'adresse 0x0800 0000 sur la carte. Il est cepeandant possible de configurer la carte pour que l'adressage soit conforme à Qemu à l'aide d'un cavalier ou d'un registre système.

4.6 Gestion des threads en python

Après avoir réalisé une première interface, nous nous sommes aperçus que les actions lancées par celles-ci n'etaient pas automatiquement multi-threadées. En effet les exemples que nous avons vu d'inetrface en python ne nécessitaient pas d'être multi-thread du fait de leur simplicité. Cependant pour notre application nous avons besoin de pouvoir lancer des exécutions en parallèle. La compréhension des threads en python nous a pris un certain temps. En effet, python dispose de deux façon de gérer les threads qui ne nous sont pas familières.
Nous avons finalement opté pour la méthode plus poussée "threading" qui permet de rattacher un thread à un event.

5 Bilan

Bien qu'ayant mis du temps à pouvoir commencer à coder, nous sommes globalement satisfaits de ce que nous avons réalisé. Nous avons émulé plusieurs parties essentielles de la carte (les ports GPIOs notament) et ce de façon très fléxible ce qui permettrait de facilement continuer. D'autre part l'utilisation de chardev permet une grande fléxibilité quant au moyen d'interagir avec le matériel émulé, et laisse donc le choix à l'utilisateur de réaliser son propre système d'interraction si notre interface ne lui convenait pas.

Le gros point noir aura été le temps que l'on a mis à bien comprendre le fonctionnement de Qemu ainsi que de la carte. En effet, si nous avions eu les quelques connaissances de base nécessaires au démarrage du projet, nous aurions pu émuler plus de composants.
Dans l'idée où quelqu'un prendrait la suite du projet, nous avons expliqué tout ce que nous avons compris et fait sur la page wiki de la salle air, ce qui permet de s'approprier les notions nécessaires à la compréhension du projet beaucoup plus rapidement.