

## PROJET DE MASTER 1

---

# Réalisation d'un robot explorateur à intelligence artificielle déportée : Réalisation de la télécommande sans écran

---

*Auteurs :*

Taha MENEBHI  
Thibaud LE DOLEDEC  
Thomas LEPOIX

*Superviseurs :*

Pierre AUBRY  
Rémi OUKRAT  
Zakari SAIBI

MASTER Systèmes Embarqués

E.S.T.E.I.

École Supérieure des Technologies Électronique, Informatique, et Infographie  
Département Systèmes Embarqués

11 avril 2018

## *Remerciements*

Nous souhaitons remercier toutes personnes ayant participé au développement de ce projet. Tout d'abord à nos encadrants Mr. Oukrat, Mr. Saibii et Mr. Aubry qui ont su nous guider techniquement sur tous les aspects de notre partie. Nous voulons aussi remercier Mr. Texier pour nous avoir aidé sur la partie Yocto ainsi que l'ensemble de nos collègues de Master1 et Master2 pour leurs conseils et leur temps consacré.

# Table des matières

<b>Remerciements</b>	<b>1</b>
<b>Table des matières</b>	<b>2</b>
<b>I Introduction</b>	<b>4</b>
0.1 Diagramme fonctionnel de niveau 1 . . . . .	5
0.1.1 Description des fonctions principales . . . . .	5
0.1.2 Description des signaux . . . . .	6
0.2 Diagramme fonctionnel de niveau 2 . . . . .	6
0.2.1 Description des fonctions secondaires . . . . .	6
0.2.2 Description des signaux . . . . .	6
<b>II Thomas LEPOIX - Aspects hardware et middleware de la télécommande</b>	<b>7</b>
<b>1 Introduction</b>	<b>8</b>
<b>2 Conception de l'interface électronique de la télécommande</b>	<b>11</b>
2.1 Cahier des charges de l'interface . . . . .	11
2.2 Fabrication de la carte . . . . .	15
2.3 Validation du fonctionnement de la carte . . . . .	17
<b>3 Élaboration d'un système d'exploitation pour la Raspberry Pi</b>	<b>19</b>
3.1 Vue d'ensemble du fonctionnement de yocto . . . . .	19
3.2 Cahier des charges du système d'exploitation . . . . .	21
3.3 Création d'un environnement de travail . . . . .	22
3.4 Mise en place des premiers éléments de la layer dédiée au projet . . . . .	24
3.5 Création d'une première image système . . . . .	26
3.6 Ajout de fonctionnalités diverses . . . . .	29
3.6.1 Ajout d'un compte utilisateur . . . . .	29
3.6.2 Ajout d'un nom de machine . . . . .	30
3.6.3 Ajout d'un message d'accueil . . . . .	30
3.6.4 Suppression du message d'erreur : " INIT : Id "S0" respawning too fast : disabled for 5 minutes " . . . . .	31
3.6.5 Ajout d'un écran de démarrage . . . . .	32
3.7 Parenthèse sur l'initialisation et les daemons . . . . .	33
3.8 Gestion des entrées / sorties . . . . .	34
3.9 Gestion de la connexion Wifi . . . . .	38
3.9.1 Connexion au réseau MASTER_SE . . . . .	38
3.9.2 Connexion automatique . . . . .	39
3.9.3 Prévision de l'échec de connexion . . . . .	40
3.10 Gestion du programme principal . . . . .	44

3.10.1 Test helloworld . . . . .	44
3.10.2 Programme principal . . . . .	47
<b>4 Conclusion</b>	<b>50</b>
<b>III Thibaud LE DOLEDEC - Programme de gestion des boutons</b>	<b>52</b>
<b>5 Programme de gestion des boutons</b>	<b>53</b>
5.1 Explication du fonctionnement . . . . .	53
5.2 Algorigrammes . . . . .	55
5.2.1 Fonction principale . . . . .	55
5.2.2 Fonction de récupération des valeurs des boutons . . . . .	56
5.2.3 Fonction d'interprétation de la machine d'état et d'envoi des commandes	58
<b>IV Taha MENEBHI - Connexion client-serveur et boîtier de la télécommande</b>	<b>60</b>
<b>6 Connexion client-serveur</b>	<b>61</b>
6.1 Qu'est ce qu'un socket? . . . . .	61
6.2 TCP vs UDP . . . . .	61
6.3 Utilisation des sockets sur Qt . . . . .	63
6.3.1 void QAbstractSocket ::connectToHost . . . . .	63
6.3.2 bool QAbstractSocket ::waitForConnected(int msec = XXX) . . . . .	64
6.3.3 bool QAbstractSocket ::waitForReadyRead(int msec = XXX) . . . . .	64
6.3.4 qint64 QAbstractSocket ::bytesAvailable() const . . . . .	64
6.3.5 QByteArray QIODevice ::readAll() . . . . .	64
<b>7 Design du boîtier</b>	<b>65</b>
<b>V Conclusion</b>	<b>67</b>

**Première partie**

**Introduction**

Ce projet consiste à concevoir un robot intelligent pouvant se déplacer d'un point A à un point B tout en évitant des obstacles détectés par une caméra placée au-dessus de la scène. Ce guidage se fait grâce à une télécommande avec écran tactile. Ceci dit, afin de tester le fonctionnement du robot et afin d'avoir un control manuel sur ce dernier, il a fallu concevoir une télécommande sans écran.

On nous a imposé dans le cahier des charges l'utilisation d'une Raspberry Pi 0W. Plusieurs solutions se sont offertes à nous, nous avons mis en priorité le coté pédagogique et restitution de ce que nous avons vu ou verrons en cours ainsi que l'espérance de développement de celles-ci. Nous avons donc décidé de faire une image Yocto et de développer que ce soit la connexion au serveur ou la lecture des états des GPIO en Qt.

Nous nous sommes ainsi répartis les tâches de la manière suivante :

- Taha : Prototypage électronique, connexion au serveur en Qt, design du boîtier.
- Thomas : Image Yocto, CAO électronique.
- Thibaud : Programme GPIO en Qt.

Par soucis de lisibilité et afin d'éviter toute confusion, nous avons décidé de vous présenter dans ce rapport que notre produit final. Avant d'arriver à celui-ci plusieurs prototype coté hardware et software ont été réalisés. Ces derniers seront développés durant la soutenance.

## 0.1 Diagramme fonctionnel de niveau 1

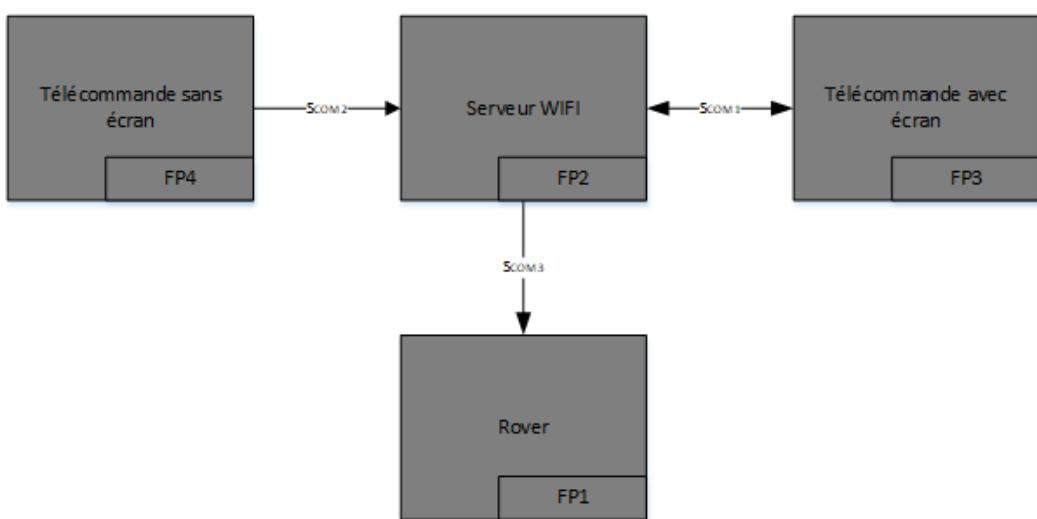


FIGURE 1 – Diagramme fonctionnel de niveau 1

### 0.1.1 Description des fonctions principales

- FP1 : Cette partie représente le rover et est chargé de pouvoir se déplacer dans toutes les directions.

- FP2 : Cette partie est chargée du traitement des informations issue de la caméra et du contrôle du rover dans le cas du mode autonome, de la transmission des consignes de directions dans le cas du mode de test ainsi que du retour utilisateur dans le mode de pilotage.
- FP3 : Cette partie permet à l'utilisateur de contrôler le rover manuellement et à distance grâce à un écran qui permet de visualiser le rover et les obstacles du dessus.
- FP4 : Cette partie assure un moyen de tester simplement le fonctionnement du rover et du serveur.

### 0.1.2 Description des signaux

- SCOM1 : Signal bidirectionnel WIFI sous forme de trame TCP permettant l'affichage du flux vidéo et l'envoi des consigne au rover.
- SCOM2 : Signal unidirectionnel WIFI sous forme de trame TCP contenant les consignes adressées au rover.
- SCOM3 : Signal unidirectionnel WIFI sous forme de trame TCP contenant les consignes adressées au rover.

## 0.2 Diagramme fonctionnel de niveau 2

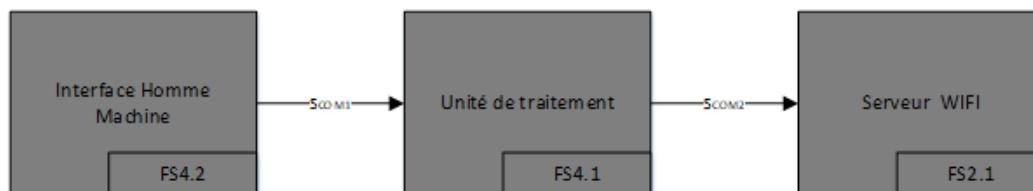


FIGURE 2 – Diagramme fonctionnel de niveau 2

### 0.2.1 Description des fonctions secondaires

- FS4.1 : Cette partie assure le traitement de l'information issue de l'interface homme machine.
- FS4.2 : Cette partie permet l'interaction entre l'utilisateur et la télécommande grâce à des boutons poussoir.
- FS2.1 : Cette partie gère les différentes connexions des télécommandes, quelles consignes envoyer au rover et le traitement de l'information issue de la caméra.

### 0.2.2 Description des signaux

- SCOM1 : Signal unidirectionnel numérique image de l'état des boutons poussoir.
- SCOM2 : Signal unidirectionnel WIFI sous forme de trame TCP contenant les consignes destinées au rover.

## **Deuxième partie**

**Thomas LEPOIX - Aspects hardware et  
middleware de la télécommande**

## Chapitre 1

# Introduction

Le présent rapport traite de la conception de la télécommande sans écran et plus précisément de l'élaboration d'une solution matérielle ainsi que du système d'exploitation permettant à la télécommande de fonctionner.

On parle des aspects hardware et middleware de la télécommande.

Pour réaliser la télécommande sans écran, le cahier des charges impose l'utilisation d'une carte Raspberry Pi Zero W et la création d'une interface ergonomique, semblable à une manette de jeu.

Par soucis de simplification, le cahier des charges dispense la télécommande d'être nomade et donc de disposer d'une alimentation embarquée. L'alimentation de la télécommande sera une alimentation classique de carte Raspberry Pi, c'est à dire un chargeur secteur - micro USB, comme un chargeur de téléphone quelconque.




---

FIGURE 1.1 – Raspberry Pi Zero W

La télécommande doit être en mesure d'ordonner au véhicule robotique des déplacements rectilignes dans les quatre directions, les quatre diagonales, ainsi que des rotations dans le sens trigonométrique et dans le sens horaire.

De plus il n'est pas question de vitesse, dans ce mode de déplacement, le robot dispose d'une vitesse unique et les commandes de déplacement sont de type tout-ou-rien.

On peut donc d'ores et déjà imaginer une interface composée de six boutons directionnels : deux de rotation et quatre de direction, les diagonales étant obtenues par combinaisons de directions primaires.

La télécommande serait alors composée de la Raspberry Pi et de l'interface représentée ci-dessous, pluggée sur la Raspberry Pi.

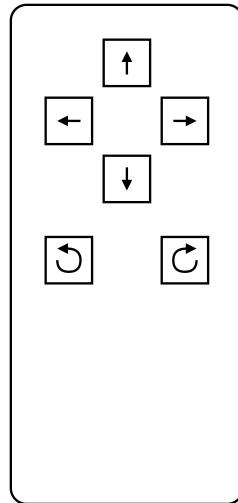


FIGURE 1.2 – Allure de la télécommande

Ci-dessous un schéma fonctionnel du système ainsi formé. Un indicateur lumineux peut être ajouté à l'interface de la télécommande pour apporter un retour visuel à l'utilisateur quant à l'état de fonctionnement de la télécommande.

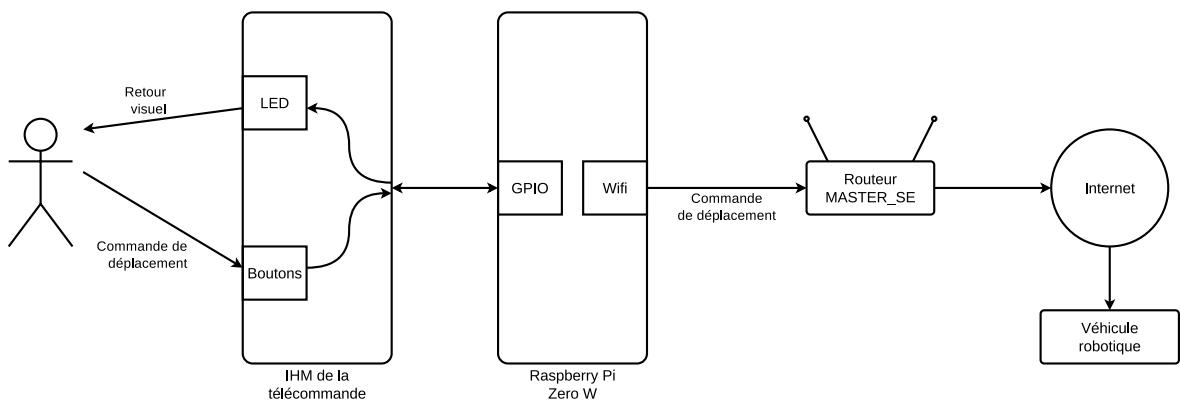


FIGURE 1.3 – Schéma fonctionnel de la télécommande au sein du projet

Quant au système d'exploitation, il n'est pas strictement nécessaire d'en utiliser un cependant, contrairement à d'autres cartes de développement comme la STM32, la Raspberry Pi ne dispose pas de HAL (Couche d'abstraction matérielle) pour faciliter le développement qui dans notre cas peut s'avérer complexe du fait de l'utilisation du réseau Wifi.

Confectionner une distribution Linux embarquée adaptée aux besoins du projet est donc un choix pertinent permettant d'économiser du temps de développement et de disposer d'un système robuste sans réinventer la roue, cela n'étant pas le but du projet.

L'outil Yocto sera utilisé pour construire une distribution Linux adéquate.

La difficulté avec Yocto réside avant tout dans l'appréhension de la démarche à adopter ainsi que dans les variables internes à l'outil dont l'utilisation nécessite de se plonger dans la documentation. C'est pourquoi le présent rapport se veut accessible, parfois aux allures de tutoriel et ne manquera pas d'illustrations en lignes de code.

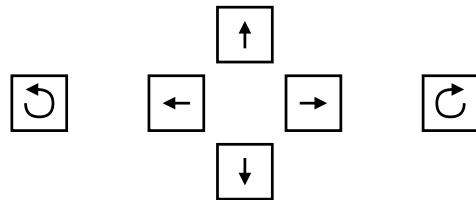
## Chapitre 2

# Conception de l'interface électronique de la télécommande

### 2.1 Cahier des charges de l'interface

Comme expliqué en introduction, la télécommande sans écran doit pouvoir ordonner au robot d'effectuer des déplacements rectilignes de type Manhattan, dans les quatre diagonales, ainsi que des rotations dans les sens horaire et trigonométrique.

Les déplacements étant tous à vitesse unique, l'interface peut être réalisée avec de simples boutons, des Joysticks ne seraient que gadget. Les déplacement en diagonale étant commandé par combinaisons de déplacement primaires.




---

FIGURE 2.1 – Les 6 déplacements commandés au robot par la télécommande sans écran

Un bouton supplémentaire peut être ajouté pour éteindre la télécommande. En effet, celle ci fonctionnant avec un système d'exploitation et un système de fichier monté en lecture/écriture, une mise sous tension sauvage pourrait corrompre le système. D'où la nécessité de stopper le système d'exploitation avant la mise hors tension par le débranchement du câble d'alimentation.

Un retour lumineux peut également être ajouté pour améliorer l'expérience utilisateur en lui fournissant une information quant à l'état de la télécommande. Par exemple le voyant pourrait :

- Être éteint lorsque la télécommande ne fonctionne pas.
- Être allumé lorsque celle ci fonctionne normalement.
- Clignoter durant la phase de connexion au réseau Wifi.

Ci dessous une représentation de l'allure de l'interface ainsi formée. Interface au format de la Raspberry Pi 0 W, qui viendrait se plugger sur celle ci.

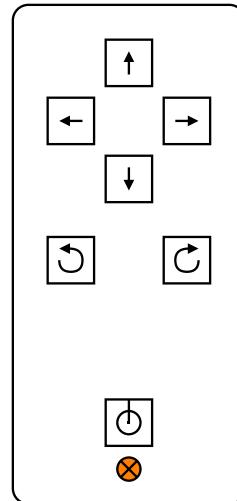


FIGURE 2.2 – Allure de l'interface à plugger sur la Raspberry Pi 0 W

Concernant la connexion entre la Raspberry Pi et son module, cela se passera par le connecteur 40 broches dont l'assignation des pins figure ci-dessous :

	Position	Power	Ground	Control	GPIO
	Wiring	BCM	Serial	PWM	Misc
Different places use different pin numbers GPIO, Wiring, and BCM have been included.					
		3.3V	1	2	5V
SDA	8	2	3	4	5V
SCL	9	3	5	6	GND
GPCLK0	4	7	4	7	GND
spi1 CS1	17	0	17	11	TXD
	27	2	27	13	RXD
	22	3	22	15	1
			3.3V	17	18
MOSI	12	10	19	14	GND
MISO	13	9	21	16	23
SCLK	14	11	23	18	5
			GND	20	24
ID_SD	30	0	DNC	27	GND
GPCLK1	5	21	5	30	25
GPCLK2	6	22	6	32	10
PWM1	13	23	13	24	SPI CS0
PWM1 misol	19	24	19	26	7
	26	25	26	36	11
			GND	38	31
				21	ID_SC
				21	PWM0
				29	16
				21	spi1 CS2
				21	mosi1
				21	sclk1

FIGURE 2.3 – Connecteur 40 broches de la Raspberry Pi 0 W

Si l'on considère une LED et sept boutons chacun utilisant une entrée/sortie (GPIO), l'utilisation de huit pin GPIO est nécessaire. En tenant compte du positionnement des GPIO et des ports d'alimentation sur le connecteur en partant de la broche 1, le connecteur de la carte doit être au minimum de  $2 \times 7$  pin.

$2 \times 8$  étant un format de connecteur plus standard (et donc moins coûteux), celui ci sera utilisé, offrant ainsi plus de souplesse lors du routage de la carte.

Pour commander une LED avec une GPIO, le port peut être directement utilisé comme source de courant à la condition que la LED ne consomme pas davantage que ce que peut générer le port sans subir de dommage.

Sur une carte Raspberry Pi, une sortie peut générer jusqu'à  $16mA$ , si plusieurs sorties sont utilisées simultanément, un courant total de  $50mA$  ne peut être dépassé.

Si l'on choisit une LED basse consommation : KPT-3216LVZGCK avec une chute de tension de  $2,65V$  pour un courant nominal de  $2mA$ , le montage est le suivant.

On ajoute une résistance de  $470\Omega$ , nécessaire pour ne pas survolter la LED, ce qui se traduirait par une augmentation du courant circulant dans la branche et possiblement des dégâts sur la LED et la Raspberry Pi.

La valeur  $470\Omega$  limite le courant dans la branche à  $1,38mA$ .

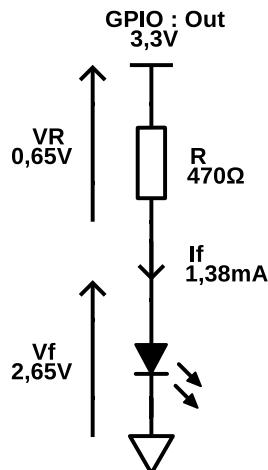


FIGURE 2.4 – Schéma électronique de la commande de la LED

Les boutons quant à eux seront câblés ainsi : L'enjeu de la résistance de pullup est d'augmenter au maximum sa valeur afin de réduire d'autant plus la consommation en courant de la télécommande. Tout en évitant la limite où sa valeur serait trop importante et donc la chute de tension à ses bornes suffisamment grande pour que les niveaux de tension de la sortie ne correspondent plus à la logique CMOS.

Le courant prélevé en entrée des GPIO ne figure pas clairement dans la documentation, toutefois il est généralement de l'ordre du  $\mu A$  sur ce genre de carte. Ce qui provoque une chute de tension de l'ordre de  $100mV$  aux bornes d'une résistance de l'ordre de  $100k\Omega$  lorsque le bouton est relâché et une consommation de l'ordre de  $33\mu A$  lors de l'appui.

Le condensateur de  $100nF$  est un condensateur de découplage, il sert à lisser le signal, notamment lors des changement d'état des boutons qui provoquent des "rebonds".

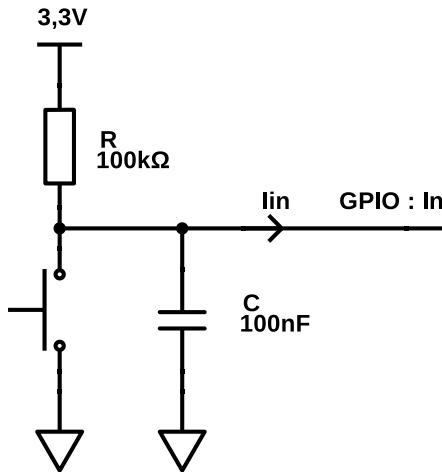


FIGURE 2.5 – Schéma électrique de l'environnement des boutons

Une dernière considération avant de réaliser la carte, ses dimensions mécaniques doivent correspondre à celles de la Raspberry Pi 0 W. Les trois éléments critiques sont :

- La longueur et la largeur de la carte.
- La position et le diamètre des trous de fixation.
- La position du connecteur.

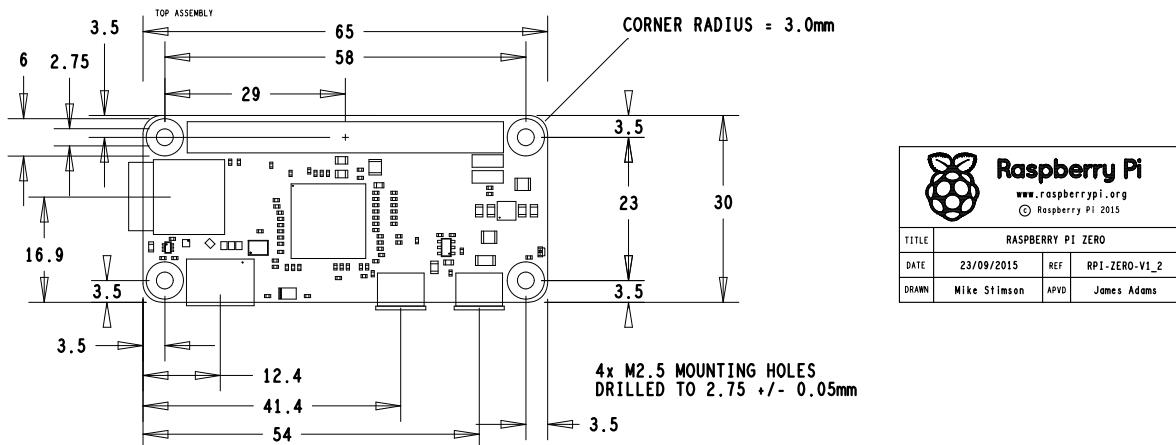


FIGURE 2.6 – Schéma mécanique de la Raspberry Pi 0 W

## 2.2 Fabrication de la carte

Composant	Valeur	Ref Constructeur	Ref FARNELL	Quantité	Prix unitaire HT (€)	Prix total HT (€)
Header	2x8 ; 2,54mm ; 7mm	76342-308LF	1098053	1	1,58	1,58
Bouton poussoir	Off-(On)	KSC521GR0HS	2320052	7	0,446	3,122
R	470Ω ; 0805	MCWR08X4700FTL	2447662	10	0,0068	0,068
R	100kΩ ; 0805	CR0805-FX-1003ELF	2333538	10	0,0392	0,392
C	100nF ; 0805	CC0805KRX7R9BB104	3019949	10	0,0463	0,463
LED	Vert ; 2mA	KPT-3216LVZGCK	2610424	1	0,316	0,316
					<b>Prix total HT (€)</b>	<b>5,941</b>

FIGURE 2.7 – Bon de commande des composants utilisés

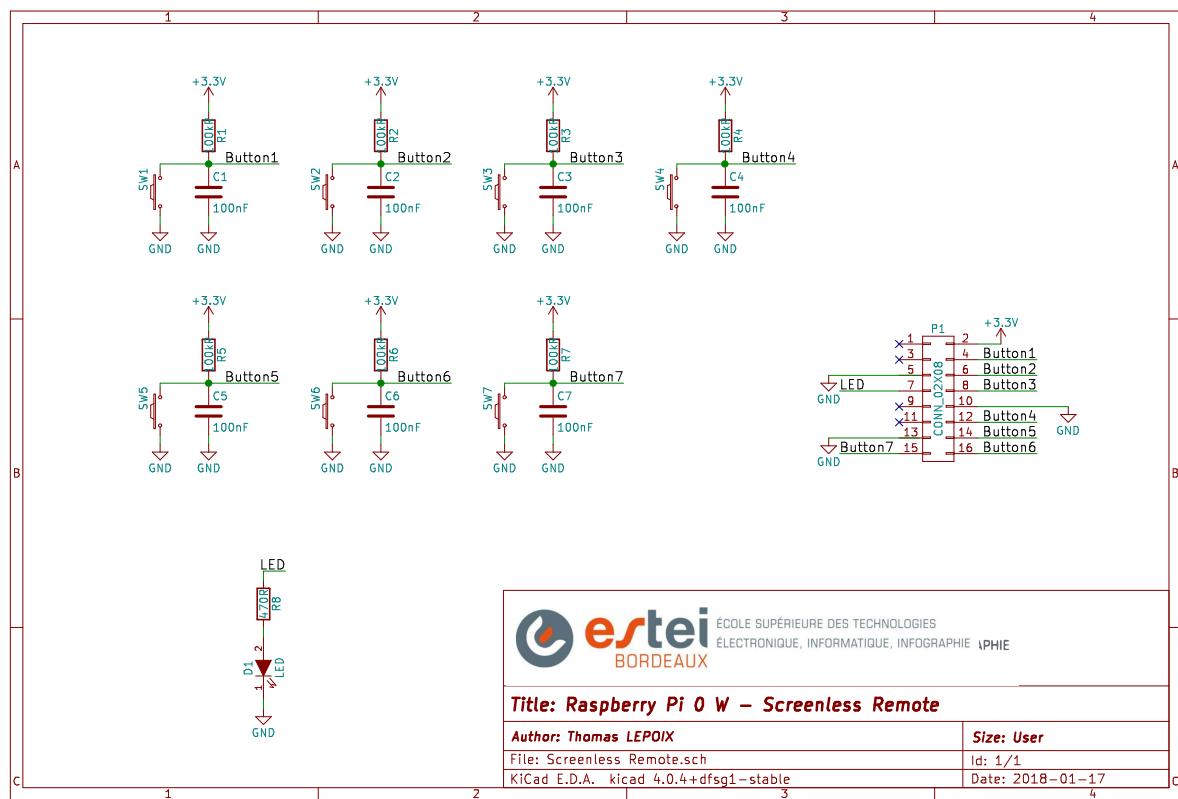


FIGURE 2.8 – Schéma électrique de la carte télécommande

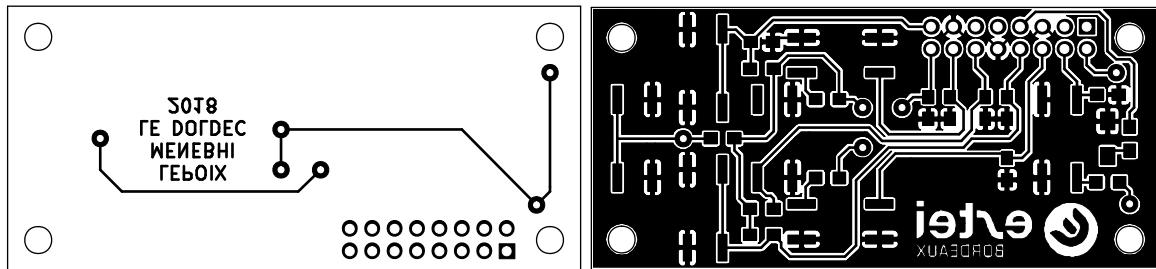


FIGURE 2.9 – Face inférieure et supérieure du typon de la carte  
(respectivement vues de dessus et de dessous)

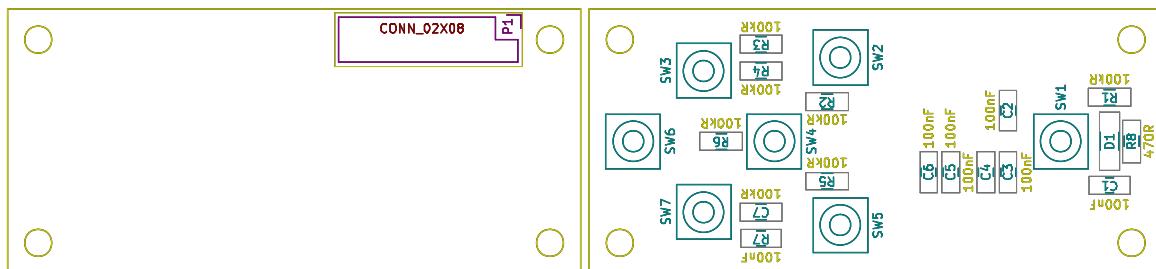


FIGURE 2.10 – Face inférieure et supérieure de l'implantation des composants  
(respectivement vues de dessous et de dessus)

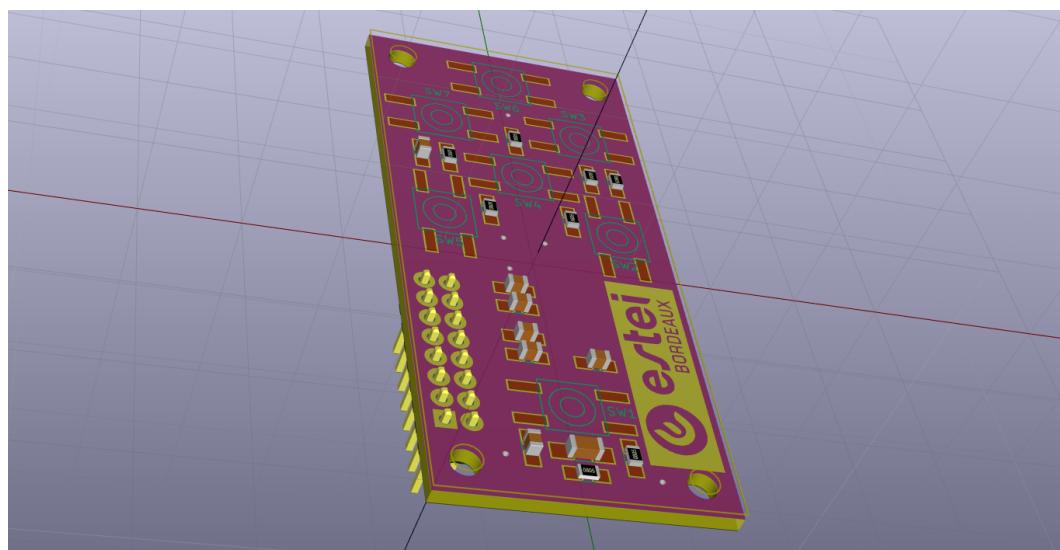


FIGURE 2.11 – Modélisation 3D de la carte

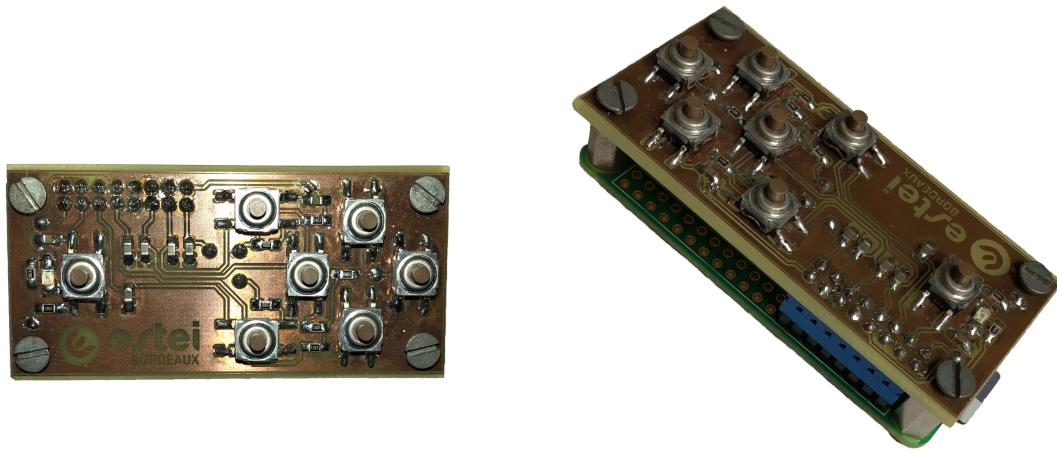


FIGURE 2.12 – Vue de la carte de dessus et de biais, pluggée sur la Raspberry Pi

### 2.3 Validation du fonctionnement de la carte

Bien que la carte soit simple il est nécessaire de valider le fonctionnement de chaque élément avant de la plonger sur la Raspberry Pi.

Pour valider le fonctionnement de la LED, on alimente celle ci en mesurant le courant passant dans la branche :

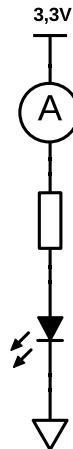


FIGURE 2.13 – Procédure de test du montage de la LED

On mesure un courant  $I_{LED} = 1,51mA$  pour une tension d'alimentation  $V_{in} = 3,35V$ . Ce courant étant inférieur à  $16mA$ , il n'y a aucun risque de fonctionnement. À l'œil l'intensité lumineuse est largement suffisante.

Pour valider le fonctionnement des boutons, on alimente la carte et on mesure la tension de sortie à l'état enfoncé et relâché de chaque bouton :

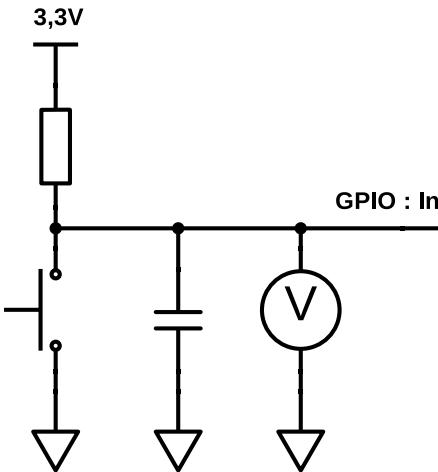


FIGURE 2.14 – Procédure de test du montage des boutons

Bouton	V <sub>in</sub> (V)	V <sub>out ↑</sub> (V)	V <sub>out ↓</sub> (mV)
↑	3,35	3,3	0,4
←	3,35	3,3	0,4
→	3,35	3,3	0,4
↓	3,35	3,3	0,4
↶	3,35	3,3	0,4
↷	3,35	3,3	0,4
☰	3,35	3,3	0,4

FIGURE 2.15 – Mesure de la tension de sortie des boutons

Logique	CMOS (V)
0	0 – 1,1
1	2,2 – 3,3

FIGURE 2.16 – Niveaux de tensions de la logique CMOS

On constate que les niveaux électriques haut et bas sont largement inclus dans les plages de tolérances CMOS, les boutons sont fonctionnels.

De plus la chute de tension aux bornes de la résistance étant de  $50\text{mV}$ , on peut déduire le courant prélevé par l'entrée de la Raspberry Pi :  $I_{in} = 0,5\mu\text{A}$ .

## Chapitre 3

# Élaboration d'un système d'exploitation pour la Raspberry Pi

### 3.1 Vue d'ensemble du fonctionnement de yocto

Yocto est en fait un projet regroupant plusieurs éléments :

- OpenEmbedded : Il s'agit d'un environnement de compilation croisée.
- Bitbake : Outil de gestion de métadonnées et d'ordonnancement de tâches utilisé par OpenEmbedded. C'est un outil comparable à GNU Make ou à Portage, le gestionnaire de paquets de la distribution GNU/Linux Gentoo.
- Poky : Terme ambigu faisant référence à la fois à une distribution GNU/Linux minimale utilisée par le projet Yocto comme point de départ et aux fichiers de métadonnées utilisés par Bitbake et OpenEmbedded pour construire cette distribution.

Bitbake, du point de vue de l'utilisateur, utilise principalement des fichiers `.bb` que l'on appelle *recipe*, "recette" en français, qui sont en quelque sorte des listes de choses à faire (compiler telle source de telle façon, copier tel fichier à tel endroit, modifier tel autre, etc.). Ces fichiers sont comparables aux Makefiles. Il existe aussi des fichiers `.bbappend` qui sont également des recipes, la différence étant qu'une recipe `.bbappend` est relative à une recipe `.bb` déjà existante qu'elle viendra "surcharger".

Certaines recipes peuvent être exécutées par Bitbake pour créer une image système. Ces recipes-images invoquent elles-mêmes d'autres recipes, images ou non et ne faisant pas nécessairement partie de la même layer, pour former un tout cohérent, une distribution.

Les recipes, ces fichiers de métadonnées sont regroupées et distribuées sous forme de *layer*, couches dont le nom est précédé du préfixe *meta*. Bien que le terme juste pour désigner ces couches soit *layer*, le terme *meta* est souvent utilisé indifféremment, par abus de langage. Il existe par exemple :

- La meta-openembedded qui regroupe entre autres de nombreux outils et fichiers de configurations utiles dans un environnement Linux.
- La meta-raspberrypi qui contient tout le support des cartes Raspberry Pi.
- La meta-qt5 qui contient les bibliothèques de Qt, qui permet de créer un kit de développement (SDK), une chaîne de compilation croisée, etc.

Bitbake utilise un dossier de travail dans lequel il place tout fichier qu'il crée, y compris l'image système qu'il générera. Ce dossier contient aussi le fichier `local.conf`. Ce fichier permet de configurer l'image sur laquelle on travaille, en particulier de choisir l'architecture de la machine cible de l'image. Concrètement ce fichier permet de configurer autant de paramètres de l'image que le fichier `image.bb` lui-même.

Il apparaît alors plusieurs façons de travailler sur une image déjà existante, par exemple une image faisant partie de la meta-raspberrypi (représentée en violet sur le schéma) :

- Modifier directement les fichiers de la meta-raspberrypi, de poky, etc. Cette méthode est à proscrire impérativement.
- Modifier `local.conf`. Le rôle de ce fichier est justement d'éviter la méthode précédente. Travailler sur ce fichier est pertinent pour du prototypage ou pour de la configuration au sens propre du terme, c'est-à-dire modifier des paramètres, ajouter des fonctionnalités présentes dans les metas utilisées, etc.
- Pour des projets plus lourds, comme celui-ci où il est question de créer des éléments nouveaux, la méthode juste consiste à créer une layer dédiée au projet et donc une nouvelle recipe-image incluant la recipe-image de la meta-raspberrypi sur laquelle on travaille.

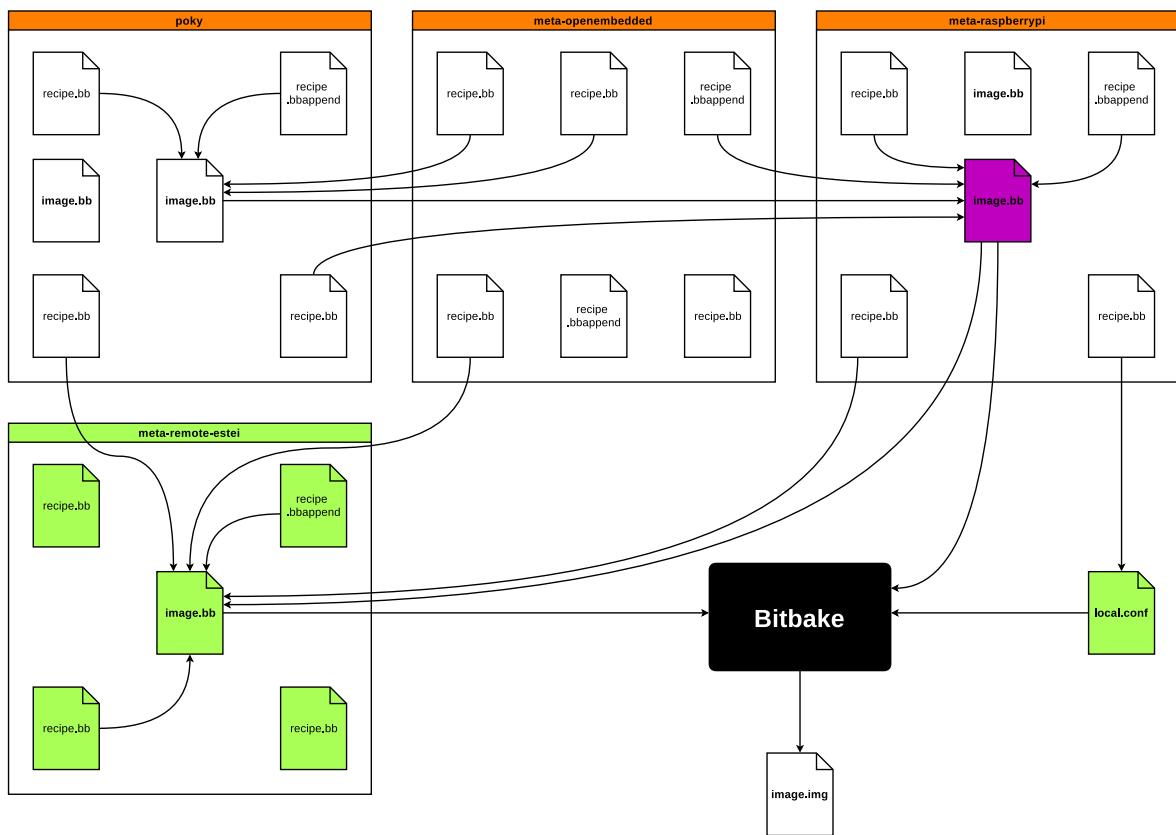


FIGURE 3.1 – Schéma d'exemple du fonctionnement de Yocto

Le schéma ci-dessus illustre les fichiers auxquels Bitbake fait appel lorsqu'on l'utilise pour créer une image système à partir d'une recipe-image. Cela par l'exécution de la commande suivante :

```
1 ~/yocto/dossier-de-travail $ bitbake nom-de-l'image
```

Les éléments représentés en vert sont les seuls que l'usage nous autorise à modifier.

Dans le cas où l'on travaille directement sur une image de la meta-raspberrypi, représentée en violet, le seul moyen dont on dispose pour modifier ou configurer notre image est le fichier `local.conf`.

Dans le cas où l'on travaille sur l'image de la meta-raspberrypi en l'incluant à une nouvelle image créée pour l'occasion, libre à nous de modifier cette nouvelle image à notre guise en créant de nouvelles recipes, en utilisant d'autres déjà existantes, etc. Cette nouvelle image ainsi que la collection de recipes créées pour le projet forment alors une nouvelle layer, dédiée au projet.

Les explications précédentes ne sont pas exhaustives, elles permettent d'appréhender le fonctionnement global de Bitbake ainsi que la démarche à adopter pour construire l'image d'une distribution Linux en utilisant Yocto.

Les layers ne sont pas constituées uniquement de recipes, il existe par exemple les classes `.bbclass` qui permettent de déporter dans un fichier unique une portion de recipe redondante qu'il suffit d'appeler par un *heritage*. Le principe est le même qu'en programmation orientée objet.

Il existe également des recipes qui permettent de construire autre chose qu'une image système. On trouve par exemple dans la meta-qt5 des recipes permettant de construire un kit de développement (SDK) ou encore une chaîne de compilation croisée utilisable ensuite par un IDE comme Qt Creator.

## 3.2 Cahier des charges du système d'exploitation

Compte tenu du cahier des charges de la télécommande et plus généralement celui du projet, un cahier des charges peut être établi pour son système d'exploitation.

Le logiciel principal de la télécommande, c'est-à-dire celui qui va surveiller l'état des boutons et en fonction envoyer des directives au robot par le réseau Wifi, est traité par d'autres étudiants. Le système d'exploitation a pour rôle de rendre la Raspberry Pi opérationnelle pour l'exécution de ce programme. Puis cette condition réalisée, de lancer automatiquement le programme principal au démarrage de la télécommande.

Le système doit donc au démarrage :

- Configurer les entrées et sorties (GPIO) de la carte.
- Se connecter au réseau Wifi, plus précisément au réseau MASTER\_SE mis en place pour le projet.
- Lancer le programme principal.
- Demeurer capable de s'éteindre à l'appui sur le bouton [OFF].

Ci-dessous une représentation du déroulement logiciel de la télécommande :

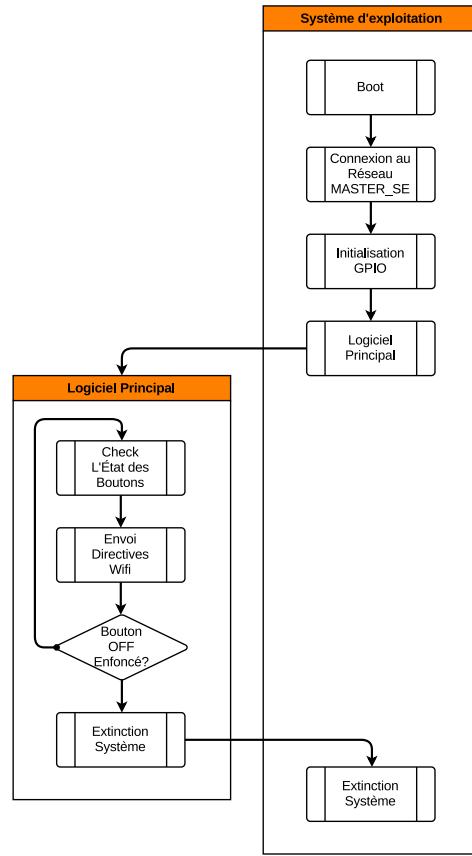


FIGURE 3.2 – Algorigramme du fonctionnement logiciel de la télécommande

On peut également ajouter quelques fantaisies en guise de signature comme un écran de démarrage ou un message d'accueil aux couleurs de l'école. Visibles uniquement un écran branché à la Raspberry Pi bien entendu.

### 3.3 Crédation d'un environnement de travail

Le fonctionnement de Yocto est certifié sur plusieurs distributions GNU/Linux. Les procédures décrites dans le présent rapport sont valables pour un système Ubuntu (testé sur Ubuntu Mate 16.10 64-bits) et peuvent varier selon les distributions.

L'utilisation de Yocto requiert quelques manipulations. Avant toute chose quelques dépendances sont à installer.

```
1 ~ $ sudo apt install gawk wget git-core diffstat unzip texinfo gcc-multilib
   ↵ build-essential chrpath
```

Puis on construit une arborescence et on récupere les métadonnées que l'on utilisera, à savoir :

- Les métadonnées de Poky.
- Les métadonnées de OpenEmbedded.
- Les métadonnées du support des cartes Raspberry Pi.
- Les métadonnées de Qt5.

```

1 ~ $ \
2 mkdir yocto/ && cd yocto/ ;\
3 git clone -b pyro git://git.yoctoproject.org/poky.git ;\
4 mkdir sources/ && cd sources/ ;\
5 git clone -b pyro git://git.openembedded.org/meta-openembedded.git ;\
6 git clone -b pyro https://github.com/aqherzan/meta-raspberrypi.git ;\
7 git clone git://github.com/meta-qt5/meta-qt5.git

```

Ensuite on initialise un dossier de travail à l'aide d'un script fourni avec Poky.

```
1 ~/yocto $ source poky/oe-init-build-env rpi-estei-build/
```

Cette commande exporte des variables dans le shell courant et est donc à répéter au début de chaque session de travail. Elle crée également le dossier `rpi-estei-build/` ainsi que son contenu, s'ils n'existent pas déjà :

```

1 ~/yocto/rpi-estei-build $ tree
2 .
3 └── conf
4     ├── bblayers.conf
5     ├── local.conf
6     └── templateconf.cfg

```

- `rpi-estei-build/` : Ce dossier de travail est le dossier dans lequel tout fichier créé par Bitbake sera placé, y compris l'image de la distribution Linux produite.
- `bblayers.conf` : Ce fichier, par le biais des lignes ci-dessous répertorie les chemins d'accès vers les métadonnées utilisées lors de la construction de notre image. Cela permet à Bitbake de disposer d'une abstraction par rapport à l'arborescence.

```

1 BBLAYERS ?= " \
2   /home/user/yocto/poky/meta \
3   /home/user/yocto/poky/meta-poky \
4   /home/user/yocto/poky/meta-yocto-bsp \
5 "

```

- `local.conf` : Ce fichier contient différents paramètres concernant l'image à créer, notamment l'architecture de la machine cible via la variable `MACHINE`.

Remarque : Lorsque Bitbake travaille, il conserve ses fichiers de travail. Cela permet d'une création d'image à l'autre, de ne pas ré-exécuter des tâches qui demeureraient inchangées. Ce qui est particulièrement appréciable lorsque des compilations entrent en jeu, cela permet d'économiser quelques heures à chaque création d'image.

En contrepartie, le dossier de travail devient rapidement très volumineux, il faut donc prévoir qu'il atteigne des tailles de l'ordre de 50Go à 100Go.

Finalement notre arborescence a l'allure suivante :

```
1 ~/yocto $ tree -L 3 --filelimit 5
2 .
```

```

3   └── poky/
4   └── rpi-estei-build
5       └── conf
6           ├── bblayers.conf
7           ├── local.conf
8           └── templateconf.cfg
9   └── sources
10    ├── meta-openembedded/
11    ├── meta-qt5/
12    └── meta-raspberrypi/

```

Il faut maintenant faire prendre connaissance à Bitbake de l'existence des couches de métadonnées autres que Poky que l'on va utiliser en ajoutant leur chemin d'accès au fichier **bblayers .conf**. Pour cela une commande de Bitbake existe et évite de la faire manuellement. Il est important de noter que l'ordre d'inclusion n'est pas anodin et peut être source d'échec de l'exécution de Bitbake. La layer que l'on va créer sera à inclure à la suite de celles ci.

```

1 ~/yocto/rpi-estei-build $ \
2     bitbake-layers add-layer ../sources/meta-raspberrypi/ && \
3     bitbake-layers add-layer ../sources/meta-openembedded/meta-oe/ && \
4     bitbake-layers add-layer ../sources/meta-openembedded/meta-python/ && \
5     bitbake-layers add-layer ../sources/meta-openembedded/meta-networking/ && \
6     bitbake-layers add-layer ../sources/meta-qt5/

```

On peut également choisir dès maintenant l'architecture de la machine cible parmi les architectures proposées dans les métadonnées Raspberry Pi. Nous utilisons une carte Raspberry Pi Zero W.

```

1 ~/yocto $ tree -L 1 sources/meta-raspberrypi/conf/machine/
2 sources/meta-raspberrypi/conf/machine/
3   ├── include/
4   ├── raspberrypi0.conf
5   ├── raspberrypi0-wifi.conf
6   ├── raspberrypi2.conf
7   ├── raspberrypi3-64.conf
8   ├── raspberrypi3.conf
9   ├── raspberrypi-cm3.conf
10  ├── raspberrypi-cm.conf
11  └── raspberrypi.conf

12 ~/yocto $ echo 'MACHINE = "raspberrypi0-wifi"' >> rpi-estei-build/conf/local.conf

```

### 3.4 Mise en place des premiers éléments de la layer dédiée au projet

L'architecture d'une layer est relativement libre, elle est organisée comme suit :

- Un dossier **conf/** contenant obligatoirement le fichier **layer.conf**. Ce fichier contient des informations essentielles à Bitbake pour comprendre et utiliser notre layer.
- Un éventuel dossier **classes**, contenant les classes **.bbclass** s'il y en a.
- Des dossiers **recipes-\*/** contenant les recipes. L'organisation des recipes est régie davantage par la coutume que par une règle stricte. Ainsi l'on retrouve dans de nombreuses layers un dossier **recipes-connectivity/** contenant les recipes liées aux connexions réseau, ou encore un dossier **recipes-core/images/** dans lequel sont stockées les recipes servant à produire une image système.

On crée donc l'arborescence de départ de notre layer dans le dossier `sources/`, avec les autres layers. Puis le fichier `layer.conf` et une première recipe-image `screenless-remote-image.bb`.

```

1 ~/yocto/sources $ \
2     mkdir -p meta-remote-estei/{conf,recipes-core/images} && \
3     cd meta-remote-estei / && \
4     touch conf/layer.conf recipes-core/images/screenless-remote-image.bb
5 .
6 .
7 └── conf
8     └── layer.conf
9 └── recipes-core
10    └── images
11        └── screenless-remote-image.bb

```

- `layer.conf` contient principalement des informations liées à l'arborescence de la layer, son contenu peut être calqué sur celui d'une autre layer, par exemple la `meta-raspberrypi`:

```

1 # We have a conf and classes directory, add to BBPATH
2 BBPATH = "${LAYERDIR}:"
3
4 # We have recipes-* directories, add to BBFILES
5 BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
6             ${LAYERDIR}/recipes-*/*/*.bbappend"
7
8 BBFILE_COLLECTIONS += "remote-estei"
9 BBFILE_PATTERN_remote-estei = "^${LAYERDIR}/"
10 BBFILE_PRIORITY_remote-estei = "5"
11
12 # This should only be incremented on significant changes that will
13 # cause compatibility issues with other layers
14 LAYERVERSION_remote-estei = "1"

```

- `screenless-remote-image.bb` sera complété au fur et à mesure de l'avancement du projet, néanmoins quelques lignes peuvent déjà être ajoutées :

```

1 DESCRIPTION = "Screenless remote image"
2 LICENSE = "MIT"
3 export IMAGE_BASENAME = "screenless-remote-image"
4 include recipes-core/images/rpi-basic-image.bb

```

La dernière ligne correspond à l'inclusion d'une image de la `meta-raspberrypi`. Plusieurs images ont été essayées comme point de départ.

Ci-dessous différentes images disponibles comme point de départ pour notre image :

```

1 ~/yocto $ tree sources/meta-raspberrypi/recipes-core/images/
2 sources/meta-raspberrypi/recipes-core/images/
3     ├── rpi-basic-image.bb
4     ├── rpi-hwup-image.bb
5     └── rpi-test-image.bb
6
7 ~/yocto $ tree -L 1 poky/meta/recipes-core/images/
8 poky/meta/recipes-core/images/
9     ├── build-appliance-image/
10    └── build-appliance-image_15.0.0.bb
11    └── core-image-base.bb
12    └── core-image-minimal.bb
13    └── core-image-minimal-dev.bb
14    └── core-image-minimal-initramfs.bb
15    └── core-image-minimal-mtdutils.bb
16    └── core-image-tiny-initramfs.bb

```

#poky met en fait à disposition de nombreuses images accessibles comme ceci

```
16 ~/yocto $ ls poky/meta*/recipes*/images/*.bb
```

À noter que l'image Raspberry Pi `rpi-basic-image.bb` inclut elle même une autre image Raspberry Pi `rpi-hwup-image.bb` qui inclut à son tour une image de Poky `core-image-minimal.bb` auxquelles elles n'ajoutent chacune que peu d'éléments :

```
1 ~/yocto $ cat sources/meta-raspberrypi/recipes-core/images/rpi-basic-image.bb
2 # Base this image on rpi-hwup-image
3 include rpi-hwup-image.bb
4
5 SPLASH = "psplash-raspberrypi"
6
7 IMAGE_FEATURES += "ssh-server-dropbear splash"

8 ~/yocto $ cat sources/meta-raspberrypi/recipes-core/images/rpi-hwup-image.bb
9 # Base this image on core-image-minimal
10 include recipes-core/images/core-image-minimal.bb
11
12 # Include modules in rootfs
13 IMAGE_INSTALL += " \
14   kernel-modules \
15 "
```

L'image `rpi-basic-image.bb` a été choisie car l'écran de démarrage "splash screen" sera utilisé, de même que le serveur ssh qui apportera à la télécommande une interface de maintenance plus légère que le branchement d'un clavier et d'un écran. Dropbear est un serveur ssh plus léger que Openssh. Quant aux modules du kernel, leur nécessité sera expliquée ultérieurement.

Maintenant que le fichier `layer.conf` existe et que son contenu est cohérent, il est possible d'ajouter la couche de métadonnées au fichier `bblayers.conf` :

```
1 ~/yocto/rpi-estei-build $ bitbake-layers add-layer ../sources/meta-remote-estei/
```

### 3.5 Crédit d'une première image système

Nous pouvons d'ores et déjà exécuter notre recipe-image fraîchement créée. Celle-ci ne consistant pour l'instant qu'en une inclusion de la recipe `rpi-basic-image.bb` de la meta-raspberrypi, cela reviendra à exécuter cette recipe.

Comme tout développement, le travail sous Yocto se déroule étape par étape, chaque élément doit être vérifié après création. La recipe-image sera donc exécutée de nombreuses fois, après chaque inclusion d'éléments nouveaux. De plus cela étant la première exécution de Bitbake dans ce dossier de travail, le processus durera quelques heures.

Il est donc judicieux de se débarrasser dès maintenant de cette tâche et par là même de vérifier la validité du socle de notre layer.

Pour cela :

```
1 ~/yocto/rpi-estei-build $ bitbake screenless-remote-image
```

Rappel : Pour que la commande fonctionne, celle ci-dessous doit impérativement avoir été exécutée au préalable, et ce à chaque session de travail c'est-à-dire dans chaque nouveau terminal.

```
1 ~/yocto $ source poky/oe-init-build-env rpi-estei-build/
```

Il arrive que certaines modifications de recipe ne soient pas détectées par Bitbake, pour y remédier, il est utile de connaître la commande suivante qui nettoie le dossier de travail. À lancer avant l'exécution de la recipe-image par Bitbake.

```
1 ~/yocto/rpi-estei-build $ bitbake -c clean screenless-remote-image
```

À l'issue de la procédure, on peut observer le *rootfs* de l'image, c'est-à-dire l'arborescence du système de fichiers, dans le dossier suivant :

```
1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
    ↪ build-essential chrpathstart-basic-image/1.0-r0/rootfs $ ls
2 bin/ boot/ dev/ etc/ home/ lib/ media/ mnt/ proc/ run/ sbin/ sys/ tmp/ usr/ var/
```

Note : Lorsque l'on relance Bitbake, il est nécessaire d'actualiser sa position dans le rootfs comme ceci.

```
1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
    ↪ build-essential chrpathstart-basic-image/1.0-r0/rootfs $ cd $PWD
```

Dans notre cas il est par exemple pertinent de vérifier la présence du splash screen ou du serveur ssh qui sont des particularités de l'image choisie :

```
1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
    ↪ screenless-remote-image/1.0-r0/rootfs $ \
2     find . | grep psplash
3 ./mnt/.psplash
4 ./etc/rc0.d/K20psplash.sh
5 ./etc/init.d/psplash.sh
6 ./etc/rc6.d/K20psplash.sh
7 ./etc/rcS.d/S00psplash.sh
8 ./etc/rc1.d/K20psplash.sh
9 ./usr/bin/psplash-write
10 ./usr/bin/psplash
11 ./usr/bin/psplash-default
12 ./usr/bin/psplash-raspberrypi
13 ./usr/lib/opkg/alternatives/psplash

14 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
    ↪ screenless-remote-image/1.0-r0/rootfs $ \
15     find . | grep dropbear
16 ./etc/rc0.d/K10dropbear
16 ./etc/rc2.d/S10dropbear
16 ./etc/rc4.d/S10dropbear
16 ./etc/init.d/dropbear
16 ./etc/rc5.d/S10dropbear
16 ./etc/rc3.d/S10dropbear
16 ./etc/rc6.d/K10dropbear
16 ./etc/rc1.d/K10dropbear
16 ./etc/dropbear
16 ./usr/sbin/dropbearmulti
16 ./usr/sbin/dropbearkey
16 ./usr/sbin/dropbearconvert
16 ./usr/sbin/dropbear
```

On observe les exécutables de psplash dans le dossier `/usr/bin/` et les scripts d'initialisation permettant de le lancer au démarrage dans `/etc/*.d`. De même pour dropbear, le serveur ssh.

L'autre dossier intéressant est celui dans lequel sont stockées les images systèmes produites par Bitbake :

```
1 ~/yocto/rpi-estei-build/tmp/deploy/images/raspberrypi0-wifi $ \
2   ls | grep screenless-remote-image
3 screenless-remote-image-raspberrypi0-wifi-20180120161249.rootfs.ext3
4 screenless-remote-image-raspberrypi0-wifi-20180120161249.rootfs.manifest
5 screenless-remote-image-raspberrypi0-wifi-20180120161249.rootfs.rpi-sdimg
6 screenless-remote-image-raspberrypi0-wifi-20180120161249.rootfs.tar.bz2
7 screenless-remote-image-raspberrypi0-wifi-20180120161249testdata.json
8 screenless-remote-image-raspberrypi0-wifi.ext3
9 screenless-remote-image-raspberrypi0-wifi.manifest
10 screenless-remote-image-raspberrypi0-wifi.rpi-sdimg
11 screenless-remote-image-raspberrypi0-wifi.tar.bz2
12 screenless-remote-image-raspberrypi0-wifi.testdata.json
```

Le fichiers dont le nom contient un numéro sont ceux produits par Bitbake, une partie de ces fichiers sont conservés au fil des exécutions de Bitbake. Ceux au nom plus épuré sont des liens symboliques vers les versions les plus récentes des véritables fichiers.

Le fichier d'extension `.rpi-sdimg` est une image système destinée à être directement copiée sur une carte microSD, qu'il ne restera qu'à insérer dans la Raspberry Pi.

La procédure est la suivante : On utilise `lsblk` pour afficher les périphériques de stockage de l'ordinateur et détecter le nom du périphérique sur lequel on va copier l'image. On démonte ensuite le périphérique avec la commande `umount` s'il ne l'est pas déjà. Enfin on copie l'image système sur le périphérique à l'aide de la commande `dd` qui réalise des copies au sens binaire du terme, c'est à dire que l'image sera copiée sur le périphérique, à l'identique, du premier au dernier bit. En comparaison la commande `cp` copie un fichier en se référant au système de fichier du périphérique de stockage pour savoir où le placer.

Remarque : Cette étape nécessite une attention toute particulière du fait de l'aspect irréversible de la copie, une copie accidentelle sur un disque système ou de données personnelles peut être dramatique.

```
1 ~ $ lsblk
2 NAME   MAJ:MIN RM    SIZE RO TYPE MOUNTPOINT
3 sdb      8:16   0 119,2G  0 disk
4  |-sdb2   8:18   0     30G  0 part /
5  |-sdb3   8:19   0     60G  0 part /media/ssd
6  `--sdb1  8:17   0    100M  0 part /boot/efi
7 sr0     11:0    1 1024M  0 rom
8 sda      8:0    0 931,5G  0 disk
9  |-sda4   8:4    0     6G  0 part [SWAP]
10 |-sda2   8:2    0     10G  0 part /tmp
11 |-sda3   8:3    0     10G  0 part /var
12  `--sda1  8:1    0    800G  0 part /home

# INSERTION DE LA CARTE SD

13 ~ $ lsblk
14 NAME   MAJ:MIN RM    SIZE RO TYPE MOUNTPOINT
15 sdb      8:16   0 119,2G  0 disk
16  |-sdb2   8:18   0     30G  0 part /
```

```

17 └─sdb3      8:19    0    60G  0 part /media/ssd
18 └─sdb1      8:17    0   100M  0 part /boot/efi
19 sr0        11:0    1 1024M  0 rom
20 sda        8:0    0 931,5G  0 disk
21 ├─sda4      8:4    0    6G  0 part [SWAP]
22 ├─sda2      8:2    0   10G  0 part /tmp
23 ├─sda3      8:3    0   10G  0 part /var
24 └─sda1      8:1    0 800G  0 part /home
25 mmcblk0     179:0  0 14,9G  1 disk
26 ├─mmcblk0p2 179:2  0 344M  1 part /media/user/b8fa876f-e670-4e09-b472-3abf3fa8259f
27 └─mmcblk0p1 179:1  0   40M  1 part /media/user/raspberrypi

28 ~ $ umount /dev/mmcblk0* ; lsblk

29 ~ $ lsblk
30 NAME      MAJ:MIN RM    SIZE RO TYPE MOUNTPOINT
31 sdb       8:16    0 119,2G  0 disk
32 └─sdb2     8:18    0   30G  0 part /
33 └─sdb3     8:19    0   60G  0 part /media/ssd
34 └─sdb1     8:17    0   100M 0 part /boot/efi
35 sr0       11:0    1 1024M  0 rom
36 sda       8:0    0 931,5G  0 disk
37 ├─sda4      8:4    0    6G  0 part [SWAP]
38 ├─sda2      8:2    0   10G  0 part /tmp
39 ├─sda3      8:3    0   10G  0 part /var
40 └─sda1      8:1    0 800G  0 part /home
41 mmcblk0     179:0  0 14,9G  1 disk
42 ├─mmcblk0p2 179:2  0 344M  1 part
43 └─mmcblk0p1 179:1  0   40M  1 part

44 ~/yocto/rpi-estei-build/tmp/deploy/images/raspberrypi0-wifi $ \
45 sudo dd \
46   if="screenless-remote-image-raspberrypi0-wifi.rpi-sdimg" \
47   of="/dev/mmcblk0" \
48   status=progress

```

## 3.6 Ajout de fonctionnalités diverses

Certaines fonctionnalités, bien que n'entrant pas dans le cahier des charges peuvent être utiles ou simplement intéressantes à ajouter à notre image. Certaines relèvent de l'administration, d'autres de la fantaisie.

### 3.6.1 Ajout d'un compte utilisateur

Par défaut les images produites sont dotées d'un compte root dont le mot de passe est inconnu. Poky permet de disposer d'un compte root dénué de mot de passe par l'option `debug - tweaks` que l'on invoque indifféremment de l'une des deux façons ci-dessous :

```

1 ~/yocto $ echo 'EXTRA_IMAGE_FEATURES = "debug-tweaks"' \
2   >> rpi-estei-build/conf/local.conf
#
# OU
#
3 ~/yocto $ echo 'IMAGE_FEATURES += "debug-tweaks"' \
4   >> sources/meta-remote-estei/recipes-core/images/screenless-remote-image.bb

```

Toutefois cette méthode est brouillon et n'est pas sécurisée, il s'agit d'ailleurs d'un outil de développement. On préfère donc définir le mot de passe root. D'autres comptes utilisateurs peuvent être ajoutés mais l'intérêt ne se présente pas ici.

On ajoute donc les lignes suivantes au fichier `screenless-remote-image` :

```

1 inherit extrausers
2 EXTRA_USERS_PARAMS = "\"
3     usermod -P estei root; "

```

Le mot de passe root est maintenant *estei*.

### 3.6.2 Ajout d'un nom de machine

le nom de machine est à l'origine défini dans une la recipe **base-files** de Poky.

```

1 ~/yocto $ cat poky/meta/recipes-core/base-files/base-files_*.bb \
2     | grep 'hostname =' \
3 hostname = "${MACHINE}"

```

Il va falloir surcharger cette recipe :

```

1 ~/yocto/sources/meta-remote-estei $ \
2     mkdir -p recipes-estei/base-files/ ; \
3     touch recipes-estei/base-files/base-files_%.bbappend

```

Contenu du fichier **base-files\_%.bbappend** :

```

1 # Set HOSTNAME
2 hostname = "estei"

```

- % dans le nom d'une recipe de surcharge se positionne là où figure un numéro de version dans le nom de la recipe d'origine. Ce caractère permet à la surcharge d'être compatible avec n'importe quelle version de la recipe.

### 3.6.3 Ajout d'un message d'accueil

Le message d'accueil est un message s'affichant dans le shell lors de la connexion d'un utilisateur. Il est contenu dans le fichier **/etc/motd** pour "Message Of The Day".

Son ajout dans l'image dépend également de la recipe **base-files**. On ajoute au contenu de la recipe la ligne suivante qui pointe vers le contenu du dossier **files/** dans lequel se trouvera le fichier **motd** :

```

1 FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
2
3 # Set HOSTNAME
4 hostname = "estei"

```

Puis on crée le dossier **files/** et le fichier **motd** dans lequel on place un logo en ASCII de l'école.

```

1 ~/yocto/sources/meta-remote-estei $ \
2     mkdir recipes-estei/base-files/files/ ; \
3     touch recipes-estei/base-files/files/motd

```

Contenu du fichier **motd** :

```

1
2
3
4     == EUSTEI ==
5
6

```

7

On observe après exécution de Bitbake la présence du fichier dans l'arborescence :

```

1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
   ↪ screenless-remote-image/1.0-r0/rootfs $ \
2   cat etc/motd
3
4
5   ——————
6   — H5TII —
7
8
9

```

### 3.6.4 Suppression du message d'erreur : " INIT: Id "S0" respawning too fast: disabled for 5 minutes "

Lorsque l'on démarre la Raspberry Pi, le message d'erreur "`INIT: Id "S0" respawning too fast: disabled for 5 minutes`" apparaît sur le terminal toutes les cinq minutes, ce qui peut être dérangeant. Peut être qu'une erreur de configuration en est à l'origine?

L'apparition de cette ligne est due à la ligne suivante du fichier `/etc/inittab`:

```
1 S0:12345:respawn:/bin/start_getty 115200 ttyS0 vt102
```

Mettre cette ligne en commentaire et redémarrer le système résout le problème.

Ce fichier est généré par la recipe de Poky `poky/meta/recipes-core/sysvinit-inittab/sysvinit-inittab_*.bb`. Par les lignes ci-dessous plus précisément :

```

1 tmp="${SERIAL_CONSOLES}"
2 for i in $tmp
3 do
4 j=`echo ${i} | sed s/\;/\g` 
5 l=`echo ${i} | sed -e 's/tty// -e 's/*;/// -e 's/;.*//'
6 label=`echo $1 | sed 's/.*\(\....\)/\1/'` 
7 echo "$label:12345:respawn:${base_bindir}/start_getty ${j} vt102"
   ↪ >> ${D}${sysconfdir}/inittab
8 done

```

Vider le contenu de la variable `SERIAL_CONSOLES` permettrait de ne pas rentrer dans la boucle qui génère la ligne. Il faut donc surcharger la recipe pour réaliser cela :

```

1 ~/yocto/sources/meta-remote-estei $ \
2   mkdir recipes-estei/sysvinit/ &&
3   echo 'SERIAL_CONSOLES = "' > recipes-estei/sysvinit/sysvinit-inittab_.bbappend

```

On observe après exécution de Bitbake que la ligne en question ne fait plus partie du fichier `/etc/inittab`:

```

1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
   ↪ screenless-remote-image/1.0-r0/rootfs $ \
2   cat etc/inittab | grep 'S0:12345'
3

```

### 3.6.5 Ajout d'un écran de démarrage

Le *splashscreen* (écran de démarrage) est une image s'affichant à l'écran lors de la phase de boot du système. En l'état actuel nous disposons du logo Yocto ou du logo Raspberry Pi.



FIGURE 3.3 – Logos Yocto et Raspberry Pi

Nous pouvons par exemple créer un écran de démarrage avec le logo de l'école agrémenté du visage de Jean-Claude PERESSINOTTO, illustre professeur de physique de l'école.



FIGURE 3.4 – Logo ESTEI & Jean-Claude PERESSINOTTO

Yocto utilise Psplash pour gérer l'écran de démarrage. Un utilitaire est également fourni pour convertir une image en header utilisable par Bitbake pour générer l'écran de démarrage. La procédure consiste donc à générer le header dans un premier temps, puis à surcharger la recette psplash :

```

1 ~ $ \
2     git clone git://git.yoctoproject.org/psplash && cd psplash ;\
3     ./make-image-header.sh ~/images/estei-logo-jc.png POKY ;\
4     ls estei*
5 estei-logo-jc-img.h

6 ~/yocto $ tree poky/meta/recipes-core/psplash/
7 poky/meta/recipes-core/psplash/
8   ├── files
9   │   └── psplash-init
10  └── psplash-poky-img.h
11   └── psplash_git.bb

12 ~/yocto/sources/meta-remote-estei $ \
13     mkdir -p recipes-estei/psplash/files/ \
14     cp ~/psplash/estei-logo-jc-img.h recipes-estei/psplash/files/psplash-poky-img.h \
15     echo 'FILESEXTRAPATHS_prepend := "${THISDIR}/files:''\'
16             > recipes-estei/psplash/psplash_git.bbappend
  
```

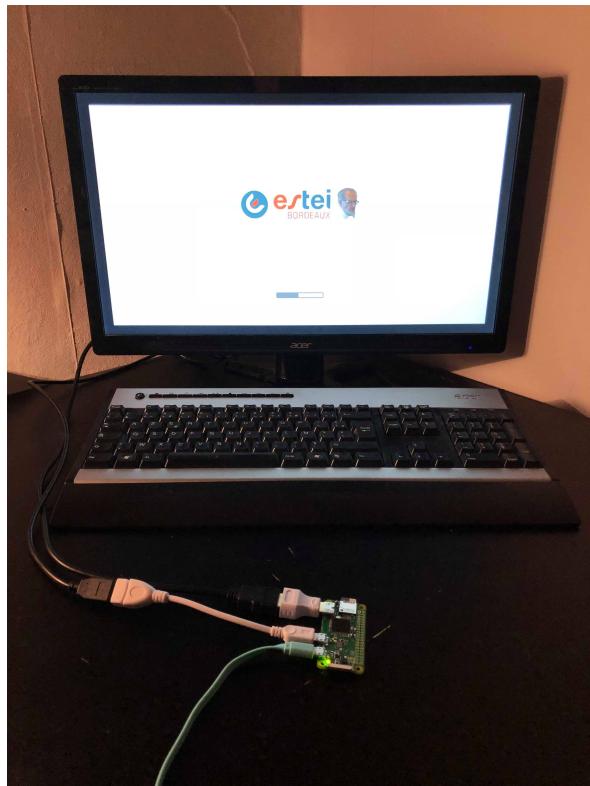


FIGURE 3.5 – Écran de démarrage de la Raspberry Pi

### 3.7 Parenthèse sur l'initialisation et les daemons

Plusieurs éléments du système d'exploitation devant se lancer au démarrage, une partie consacrée aux daemons et au processus d'initialisation s'impose.

Sur les systèmes de type Unix, un daemon est un logiciel fonctionnant en tâche de fond. Par exemple, un serveur SSH est un daemon surveillant les requêtes sur les ports attribués au protocole.

Le gestionnaire d'initialisation est un programme se lançant au démarrage du système et ayant pour but de lui-même lancer divers daemons et scripts d'initialisation. Plusieurs gestionnaires d'initialisation sont utilisés dans le monde Linux / Unix, les principaux étant :

- SysVinit : Gestionnaire d'initialisation historique d'Unix System V. Utilisé pendant de nombreuses années sur la quasi-totalité des distributions GNU/Linux.
- Systemd : Plus moderne et ayant de nombreux avantages sur SysVinit mais ne faisant pas l'unanimité au sein de la communauté car ne respecte pas le principe philosophique Unix KISS (Keep It Simple Stupid) prônant des logiciels simples consacrés à une tâche unique. Systemd a actuellement remplacé SysVinit sur la grande majorité des distributions GNU/Linux.
- OpenRC : Gestionnaire d'initialisation par défaut de la distribution Gentoo.

Par défaut Poky utilise SysVinit, celui ci sera utilisé et les suivantes explications s'appliquent à ce gestionnaire d'initialisation.

Le dossier `/etc/init.d/` contient tous les scripts d'initialisation du système. Ces scripts servent à invoquer les daemons qui ne sont autre que des exécutables quelconques (se trouvant dans les dossiers `/*bin/` et `/usr/*bin/`). SysVinit utilise des *runlevels* c'est-à-dire des niveaux de fonctionnement. Généralement le runlevel 0 correspond à l'arrêt de la machine, le 1 correspond à un mode mono-utilisateur de maintenance, le 6 correspond au redémarrage et les autres dépendent généralement de la distribution utilisée. Ici le runlevel de fonctionnement normal est le 5.

Il existe un dossier `/etc/rc*.d/` associé à chaque runlevel. Ces dossiers contiennent des liens symboliques vers les scripts de `/etc/init.d/` dont le nom est précédé de `s` ou de `k` selon que le daemon soit à démarrer (Start) ou à arrêter (Kill) dans le runlevel en question, et d'un nombre entre `00` et `99` indiquant l'ordre d'appel du script.

Les scripts d'initialisation sont prévus pour être invoqués avec en argument `start`, `stop`, `restart`, `status`, etc. Les scripts d'initialisation contiennent généralement une entête avec des informations relatives à l'initialisation, comme par exemple les runlevels où le daemon est invoqué et ceux où il est stoppé. Par exemple :

```

1 ~ $ head /etc/init.d/ssh
2 #! /bin/sh
3
4 ### BEGIN INIT INFO
5 # Provides: sshd
6 # Required-Start: $remote_fs $syslog
7 # Required-Stop: $remote_fs $syslog
8 # Default-Start: 2 3 4 5
9 # Default-Stop:
10 # Short-Description: OpenBSD Secure Shell server
11 ### END INIT INFO

```

### 3.8 Gestion des entrées / sorties

Sur Linux, l'accès aux entrées / sorties (GPIO) passe par le dossier `/sys/class/gpio/`. Il existe dans ce dossier deux fichiers `export` et `unexport` permettant d'initialiser et désinitialiser une entrée / sortie.

L'écriture d'un numéro dans le fichier `export` crée un dossier associé à la GPIO correspondante. Ce nouveau dossier contient notamment un fichier `direction` permettant de configurer la direction du port simplement par l'écriture de `in` ou `out` dans celui ci, et un fichier `value` contenant `1` ou `0` dans lequel on va lire ou écrire en fonction de la direction du port.

```

1 raspberrypi:/sys/class/gpio # ls
2 export unexport
3 raspberrypi:/sys/class/gpio # echo 2 > export tree
4 .
5   └── export
6   └── unexport
7   └── gpiochip0
8     └── gpio2
9       ├── active-low
9       └── device

```

```

10 └── direction
11 └── edge
12 └── power
13 └── subsystem
14 └── uevent
15 └── value
16 raspberrypi:/sys/class/gpio # echo in > gpio2/direction
17 raspberrypi:/sys/class/gpio # cat gpio2/value
18 1

# MISE À LA MASSE

19 raspberrypi:/sys/class/gpio # cat gpio2/value
20 0
21 raspberrypi:/sys/class/gpio # echo in > gpio2/direction
22 raspberrypi:/sys/class/gpio # echo 1 > gpio2/value

# TENSION MESURÉE : 3,33V

23 raspberrypi:/sys/class/gpio # echo 0 > gpio2/value
# TENSION MESURÉE : 0,06V

```

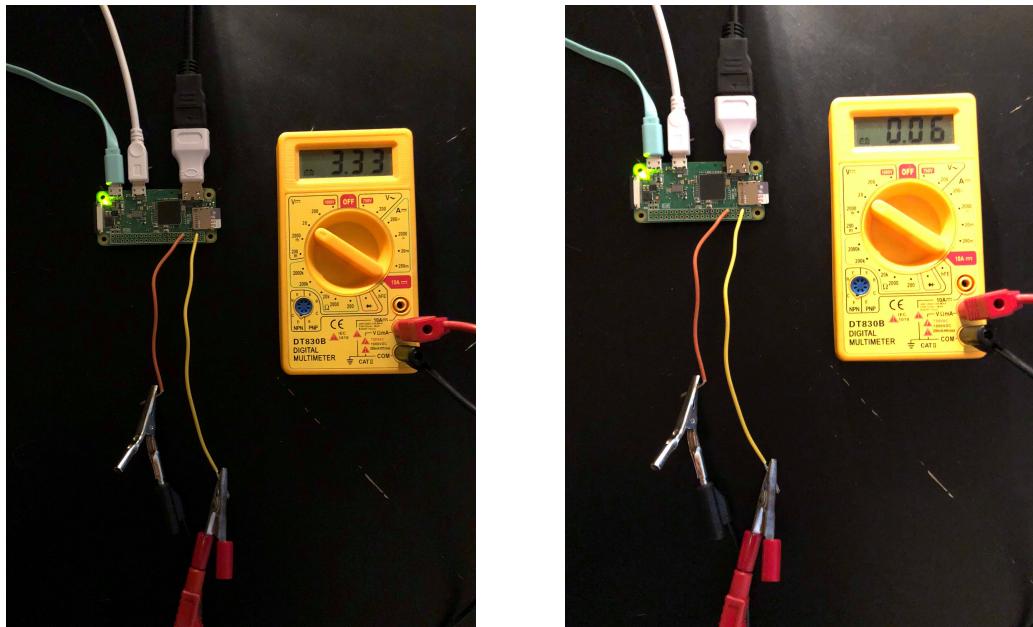


FIGURE 3.6 – Tensions mesurées lors de la manipulation ci-dessus

Dans notre cas il faut créer un script se lançant au démarrage du système pour configurer les ports associés aux boutons en entrée et celui associé à la LED en sortie. Ci-dessous l'assignation des entrées/sorties. Pour élaborer le script, on reprend la convention utilisée par les scripts d'initialisation SysVinit.

GPIO	Fonction
2	⊕
3	⊖
4	→
14	LED
17	↓
22	↑
23	←
27	⏚

FIGURE 3.7 – Assignation des entrées/sorties de la Raspberry Pi

```

1 #!/bin/sh
2 ### BEGIN INIT INFO
3 # Provides:           init-gpio
4 # Required-Start:    $remote_fs $syslog
5 # Required-Stop:
6 # Default-Start:    5
7 # Default-Stop:     0 1 6
8 # Short-Description: GPIO initialization for remote shield
9 ### END INIT INFO
10
11 case "$1" in
12     start)
13         echo -n 'Setting GPIO... '
14         echo '2' >/sys/class/gpio/export
15         echo '3' >/sys/class/gpio/export
16         echo '4' >/sys/class/gpio/export
17         echo '14' >/sys/class/gpio/export
18         echo '17' >/sys/class/gpio/export
19         echo '22' >/sys/class/gpio/export
20         echo '23' >/sys/class/gpio/export
21         echo '27' >/sys/class/gpio/export
22
23         echo 'in' >/sys/class/gpio/gpio2/direction
24         echo 'in' >/sys/class/gpio/gpio3/direction
25         echo 'in' >/sys/class/gpio/gpio4/direction
26         echo 'in' >/sys/class/gpio/gpio17/direction
27         echo 'in' >/sys/class/gpio/gpio22/direction
28         echo 'in' >/sys/class/gpio/gpio23/direction
29         echo 'in' >/sys/class/gpio/gpio27/direction
30         echo 'out' >/sys/class/gpio/gpio14/direction
31         echo 'Done'
32         ;;
33     stop)
34         echo -n 'Unsetting GPIO... '
35         echo '2' >/sys/class/gpio/unexport
36         echo '3' >/sys/class/gpio/unexport
37         echo '4' >/sys/class/gpio/unexport
38         echo '14' >/sys/class/gpio/unexport
39         echo '17' >/sys/class/gpio/unexport
40         echo '22' >/sys/class/gpio/unexport
41         echo '23' >/sys/class/gpio/unexport
42         echo '27' >/sys/class/gpio/unexport
43         echo 'Done'
44         ;;
45     restart)
46         $0 stop
47         $0 start
48         ;;
49     *)
50         echo "Usage : $0 start|stop|restart"
51         ;;
52     esac
53 exit 0

```

Pour inclure ce script dans notre système, on crée la recipe associée dans un dossier regroupant les recipes liées à la fonction première de la Raspberry Pi au sein du projet : être une télécommande.

```
1 ~/yocto/sources/meta-remote-estei $ \
2     mkdir -p recipes-project/init-gpio/files/ ;\
3     touch recipes-project/init-gpio/files/init-gpio.sh/ ;\
4     touch recipes-project/init-gpio/init-gpio.bb
```

```
1 SUMMARY = "GPIO initialization for remote shield"
2 LICENSE = "MIT"
3 LIC_FILES_CHKSUM = "file://$COREBASE/meta/COPYING.MIT;
        ↪ md5=3da9cfbc788c80a0384361b4de20420"
4
5
6 SRC_URI = " file://init-gpio.sh "
7
8 SRCREV = "$AUTOREV"
9
10 inherit update-rc.d
11
12 do_install() {
13     install -d ${sysconfdir}/init.d/
14     install -m 0755 ${WORKDIR}/init-gpio.sh ${sysconfdir}/init.d/init-gpio
15 }
16
17 INITSCRIPT_NAME = "init-gpio"
18 INITSCRIPT_PARAMS = "start 97 5 . stop 00 0 1 6 ."
```

- Le contenu de la commande `do_install()` permet d'installer le script, c'est à dire de le placer correctement dans l'arborescence (le rootfs) : on le place dans `/etc/init.d/` sous le nom `init-gpio`, dénué d'extension.
- La gestion des liens symboliques dans les dossiers `/etc/rc*.d` se fait par l'utilisation de la classe `update-rc.d` de Poky. Notamment par la variable `INITSCRIPT_PARAMS` où l'on définit l'ordre d'invocation et les runlevels associés à l'appel du script avec l'argument `start` puis `stop` c'est à dire à l'initialisation puis à la désinitialisation des entrées/sorties. On choisit ici le numéro de priorité 97 pour ne pas interférer avec les processus systèmes se situant au début de l'intervalle [00;99] tout en laissant une place aux éléments que l'on va ajouter devant être appelés ultérieurement à ce script.

Pour faire prendre connaissance de cette recipe à Bitbake, on ajoute la ligne suivante à la recipe-image `~/yocto/sources/meta-remote-estei/recipes-core/images/screenless-remote-image` :

```
1 IMAGE_INSTALL += " init-gpio"
```

On observe après utilisation de Bitbake l'installation du script :

```
1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifipoky-linux-gnueabi/
    ↪ screenless-remote-image/1.0-r0/rootfs $ \
2     find . | grep init-gpio
3 ./etc/init.d/init-gpio
4 ./etc/rc5.d/S97init-gpio
```

```

5 ./etc/rc6.d/K00init-gpio
6 ./etc/rc1.d/K00init-gpio
7 ./etc/rc0.d/K00init-gpio

```

Puis lorsque l'on démarre la Raspberry Pi, la ligne suivante est la dernière ligne affichée durant la phase de boot, avant l'affichage du prompt.

```
1 Setting GPIO... DONE
```

On observe également la configuration des entrées/sorties : toutes sont configurées en entrée à l'exception de la GPIO 14, associée à la LED.

```

1 raspberrypi:/sys/class/gpio # \
2 for i in `ls | grep "gpio[[:digit:]]"` ;\
3   do echo -n "${i} : " ;\
4   cat ${i}/direction ;\
5 done
6 gpio14 : out
7 gpio17 : in
8 gpio2 : in
9 gpio22 : in
10 gpio23 : in
11 gpio27 : in
12 gpio3 : in
13 gpio4 : in

```

## 3.9 Gestion de la connexion Wifi

### 3.9.1 Connexion au réseau MASTER\_SE

Le fichier `/etc/wpa-supplicant.conf` permet de stocker des configurations réseau Wifi.

Il existe une recipe de Poky permettant de configurer le fichier par surcharge de celle ci.

```

1 ~/yocto/sources/meta-remote-estei $ \
2     mkdir -p recipes-connectivity/wpa-supplicant/wpa-supplicant/ ;\
3     touch recipes-connectivity/wpa-supplicant/wpa-supplicant/ \
4           ↪ wpa_supplicant.conf-sane ;\
5     touch recipes-connectivity/wpa-supplicant/wpa-supplicant_%.bbappend

```

Contenu de `wpa-supplicant_%.bbappend`:

```
1 FILESEXTRAPATHS_prepend := "$THISDIR/wpa-supplicant:"
```

Configuration du réseau MASTER\_SE dans le fichier `wpa_supplicant.conf-sane` :

```

1 ctrl_interface=/var/run/wpa_supplicant
2 ctrl_interface_group=0
3 update_config=1
4
5 network={
6   ssid="Master_SE"
7   psk="U8d_4fW@"
8   proto=RSN
9   key_mgmt=WPA-PSK
10 }

```

L'utilisation de l'utilitaire wpa\_supplicant nécessite quelques dépendances et l'activation matérielle du Wifi. Tout cela se faisant par la recipe-image `screenless-remote-image` dont figure un extrait ci-dessous.

```

1 CONNECTIVITY = " \
2   linux-firmware \
3   i2c-tools \
4   python-smbus \
5   bridge-utils \
6   hostapd \
7   iptables \
8   wpa_supplicant \
9 "
10
11 DISTRO_FEATURES += "wifi"
12 MACHINE_FEATURES += "wifi"
13 IMAGE_INSTALL += " ${CONNECTIVITY} init-gpio"
```

On observe après utilisation de Bitbake la présence du fichier sur le rootfs :

```

1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
   ↪ screenless-remote-image/1.0-r0/rootfs $ \
2   cat etc/wpa_supplicant.conf
3 ctrl_interface=/var/run/wpa_supplicant
4 ctrl_interface_group=0
5 update_config=1
6
7 network={
8   ssid="Master_SE"
9   psk="U8d_4fW@"
10  proto=RSN
11  key_mgmt=WPA-PSK
12 }
```

Maintenant, au démarrage de la carte, une commande `ifup wlan0` suffit à la connecter au réseau MASTER\_SE. Il est alors possible depuis un ordinateur de lui envoyer un ping ou même une requête SSH.

### 3.9.2 Connexion automatique

Le fichier `/etc/network/interfaces` permet de configurer les interfaces lancées de façon automatique. On configure ce fichier la recipe de Poky `init-ifupdown` que l'on va surcharger :

```

1 ~/yocto/sources/meta-remote-estei $ \
2   mkdir -p recipes-connectivity/init-ifupdown/files/; \
3   touch recipes-connectivity/init-ifupdown/files/interfaces
4   touch recipes-connectivity/init-ifupdown/init-ifupdown_.bbappend
```

Contenu de `init-ifupdown_.bbappend` :

```
1 FILESEXTRAPATHS_prepend := "${THISDIR}/files:"
```

Contenu de `interfaces` :

```

1 # /etc/network/interfaces -- configuration file for ifup(8), ifdown(8)
2
3 # The loopback interface
```

```

4 auto lo
5 iface lo inet loopback
6
7 # Wireless interfaces
8 auto wlan0
9 iface wlan0 inet dhcp
10    wireless_mode managed
11    wireless_essid any
12    wpa-driver wext
13    wpa-conf /etc/wpa_supplicant.conf

```

Désormais la Raspberry Pi se connecte au réseau et se fait attribuer une adresse IP par le serveur DHCP dès la phase de boot.

### 3.9.3 Prévision de l'échec de connexion

Il est tout à fait possible que pour une raison ou une autre la tentative de connexion au réseau échoue. Il est donc judicieux de prévoir cette éventualité.

Comme l'unique fonctionnalité de la Raspberry Pi est de servir de télécommande et que cela dépend intégralement de la connexion au réseau, il est pertinent de lancer une nouvelle tentative de connexion si la précédente échoue, et ce perpétuellement.

Et c'est ici qu'interviendra la LED en étant témoin de l'état de connexion de la Raspberry Pi :

- Clignotement lors d'une tentative de connexion.
- Allumée lorsque la connexion au réseau MASTER\_SE est opérationnelle.

```

1 ~/yocto/rpi-estei-build/tmp/work/raspberrypi0_wifi-poky-linux-gnueabi/
   ↳ screenless-remote-image/1.0-r0/rootfs $ \
2     tree etc/rc5.d/
3 etc/rc5.d/
4     └── S01networking
5     ├── S02dbus-1
6     ├── S10dropbear
7     ├── S15mountnfs.sh
8     ├── S20hostapd
9     ├── S20syslog
10    ├── S97init-gpio
11    ├── S99rmnologin.sh
12    └── S99stop-bootlogd

```

Si l'on observe l'ordre d'appel des scripts du runlevel 5 (celui du fonctionnement normal), on remarque que le script **networking** effectuant la première tentative de connexion est appelé en premier.

La façon de faire la plus simple, serait d'effectuer les éventuelles tentatives supplémentaires à la fin de la phase d'initialisation, après l'appel de **init-gpio** pour avoir accès à la LED.

Note : Le script réitérant les tentatives de connexion devra être bloquant afin que le programme principal de la télécommande ne se lance pas tant que la connexion au réseau n'est pas établie.

Le script n'ayant pas de condition d'arrêt et pouvant se dérouler indéfiniment, dans le cas où l'on allumerait la télécommande hors de portée du réseau Wifi par exemple, il serait avisé de permettre un arrêt de celle ci par un appui sur le bouton [OFF] (normalement géré par le programme principal) pendant que la carte retente de se connecter.

Ci-dessous le script `resetnet` ainsi qu'un algorigramme décrivant son fonctionnement.

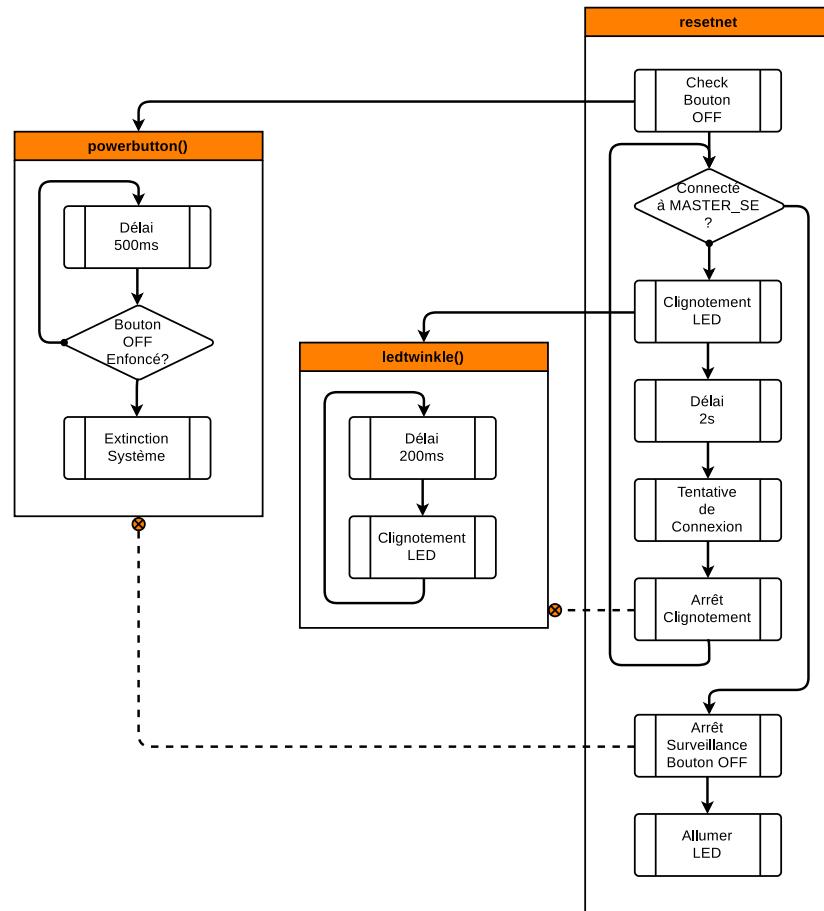


FIGURE 3.8 – Algorigramme du script réitérant les tentatives de connexion : `resetnet`

```

1 #!/bin/sh
2 ### BEGIN INIT INFO
3 # Provides:           resetnet
4 # Required-Start:    $remote_fs $syslog
5 # Required-Stop:
6 # Default-Start:     5
7 # Default-Stop:
8 # Short-Description: Loop until connection with MASTER_SE is set
9 ### END INIT INFO
10
11 LED='/sys/class/gpio/gpio14/value'
12 OFF='/sys/class/gpio/gpio2/value'
13
14 ledtwinkle() {
15     while [ true ]
16     do
17         echo '1' > ${LED}
18         sleep 0.2s

```

```

19         echo '0' > ${LED}
20         sleep 0.2s
21         done ; }
22
23 powerbutton() {
24     while [ true ]
25     do
26         [ "$(cat ${OFF})" = '0' ] && poweroff
27         sleep 0.5s
28         done ; }
29
30 case "$1" in
31     start)
32         powerbutton & powerbuttonPID=$!
33         echo '0' > ${LED}
34
35         while [ -z "$(ifconfig wlan0 | grep -w 'inet')" ]
36         do
37             echo 'Connection failed'
38             sleep 1s
39             ledtwinkle & ledtwinklePID=$!
40             ifdown wlan0
41             sleep 1s
42             echo 'Try to touch MASTER_SE network...'
43             ifup wlan0
44             kill ${ledtwinklePID}
45             echo '0' > ${LED}
46             done
47
48         echo '1' > ${LED}
49         echo 'Connection to MASTER_SE : DONE'
50         kill ${powerbuttonPID}
51         ;;
52     *)
53         echo "$0 isn't a daemon"
54         echo "Usage : $0 {start}"
55         ;;
56     esac
57 exit 0

```

On crée ensuite la recette associée à ce script :

```

1 ~/yocto/sources/meta-remote-estei $ \
2     mkdir -p recipes-connectivity/resetnet/files/ ;\
3     touch recipes-connectivity/resetnet/files/resetnet.sh
4     touch recipes-connectivity/resetnet/resetnet.bb

```

Contenu de la recette `resetnet.bb` :

```

1 SUMMARY = "Retry to connect to MASTER_SE"
2 LICENSE = "MIT"
3 LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;
4     ↳ md5=3da9cfbcb788c80a0384361b4de20420"
5
6 SRC_URI = " file://resetnet.sh "
7
8 SRCREV = "${AUTOREV}"
9
10 inherit update-rc.d
11
12 do_install() {
13     install -d ${D}${sysconfdir}/init.d/
14     install -m 0755 ${WORKDIR}/resetnet.sh ${D}${sysconfdir}/init.d/resetnet
15 }
16
17 INITSCRIPT_NAME = "resetnet"
18 INITSCRIPT_PARAMS = "start 98 5 ."

```

- On installe le script dans `/etc/init.d`. Il sera appelé dans le runlevel 5 avec le niveau de priorité 98.

On ajoute la recipe à notre image :

```
1 IMAGE_INSTALL += " ${CONNECTIVITY} init-gpio resetnet"
```

On observe les messages de boot en démarrant la Raspberry Pi hors de porté du réseau Wifi. Plusieurs messages systèmes apparaissent, puis une ligne provenant du script `init-gpio`, puis une succession de lignes relatives à la connexion Wifi se répétant indéfiniment :

```
1 Setting GPIO... DONE
# RÉPÉTITION DES LIGNES SUIVANTES
2 Connection failed
3 Try to touch MASTER_SE network...
4 Successfully initialized wpa_supplicant
5 udhcpc (v1.24.1) started
6 Sending discover...
7 Sending discover...
8 Sending discover...
9 No lease, forking to background
```

Et si l'on met à la masse l'entrée associée au bouton [OFF] (GPIO2), on observe le système d'exploitation s'éteindre normalement.

Quant au clignotement de la LED, on mesure à l'analyseur logique la tension sur la sortie associée :

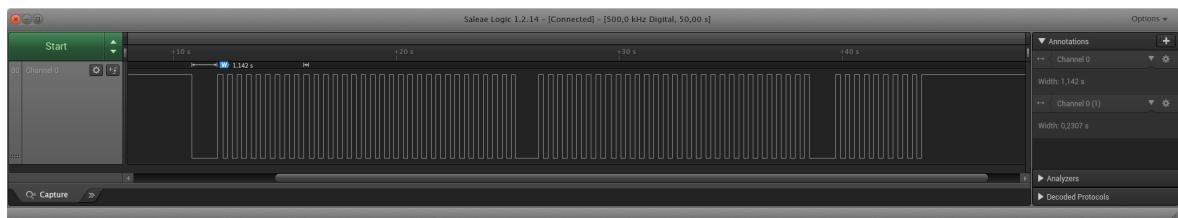


FIGURE 3.9 – Mesure à l'analyseur logique de la tension commandant la LED durant la recherche du réseau MASTER\_SE

On observe un état bas de 1s au début de chaque cycle ainsi que des états hauts et bas de 200ms lors du clignotement. La sortie cesse de clignoter au milieu d'un cycle et passe à l'état de repos (haut) lorsque l'on met l'entrée associée au bouton [OFF] à la masse. Il s'agit d'un arrêt du système.

Le fonctionnement du script est validé.

## 3.10 Gestion du programme principal

### 3.10.1 Test helloworld

En vue d'intégrer le programme principal de la télécommande, réalisé par d'autres étudiants, des tests préliminaires ont été faits avec un programme simple, de type *helloworld*.

Arborescence de la layer associée au programme *helloworld* :

```
1 ~/yocto/sources/meta-remote-estei $ \
2     tree recipes-project/helloworld/
3 recipes-project/helloworld/
4     ├── files
5     │   ├── helloworld.pro
6     │   ├── helloworld.sh
7     │   └── main.cpp
8     └── helloworld.bb
```

Ci-dessous un programme simple en C++ affichant un message toutes les 2s :

```
1 #include <iostream>
2 #include <unistd.h>
3 using namespace std;
4
5 int main() {
6     while(1) {
7         cout<<"helloworld : TEST MESSAGE"<<endl;
8         usleep(2000000);
9     }
10    return 0;
11 }
```

La meta-qt5 offre une procédure de compilation simplifiée pour les programmes écrits en Qt. Le programme principal utilisera les librairies Qt. Celui ci n'en utilise aucune mais peut être assimilé à un programme Qt.

Il suffit en fait d'ajouter la ligne suivante à notre recipe pour compiler et installer le programme.

```
1 inherit qmake5
```

Bitbake va alors scanner la liste des fichiers inclus dans la recipe à la recherche d'un **.pro**, sorte de MakeFile propre à Qt. Ce fichier est créé automatiquement si l'on code en utilisant un IDE tel QtCreator. Deux lignes sont toutefois à ajouter au fichier **.pro**, il s'agit d'indiquer où dans l'arborescence le programme compilé sera installé. Dans notre cas dans **/usr/bin/**, dossier où sont normalement installés les programmes destinés à être utilisés par l'utilisateur et ne faisant pas partie du cœur de Linux.

Contenu du fichier **helloworld.pro** :

```
1 TEMPLATE = app
2 TARGET = helloworld
3 INCLUDEPATH += .
4
5 SOURCES += main.cpp
6
7 target.path = /usr/bin
8 INSTALLS += target
```

Contenu total de la recette `helloworld.bb`. Celle ci doit faire deux choses, tout en n'oubliant pas d'inclure tous les fichiers utilisés :

- Compiler et installer le programme C++ en `/usr/bin/helloworld` par l'utilisation de `inherit qmake5`.
- Installer le script d'initialisation associé en `/etc/init.d/helloworld` avec le niveau de priorité 99 pour intervenir après les scripts `init-gpio` et `resetnet`. La méthode est expliquée plus en détail dans les chapitres concernant les deux scripts précédemment cités.

```

1 SUMMARY = "Helloworld test"
2 LICENSE = "MIT"
3 LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;
        ↳ md5=3da9cfbcb788c80a0384361b4de20420"
4
5 SRC_URI = " \
6     file://helloworld.sh \
7     file://helloworld.pro \
8     file://main.cpp \
9     "
10
11 SRCREV = "${AUTOREV}"
12 S = "${WORKDIR}"
13 PR = "r0"
14 DEPENDS = " qtbase"
15
16 inherit update-rc.d
17 inherit qmake5
18
19 do_install_append() {
20     install -d ${D}${sysconfdir}/init.d/
21     install -m 0755 ${WORKDIR}/helloworld.sh ${D}${sysconfdir}/init.d/helloworld
22 }
23
24 INITSCRIPT_NAME = "helloworld"
25 INITSCRIPT_PARAMS = "start 99 5 . stop 00 0 1 6 ."

```

On ajoute la recette à notre image en prenant soin de désactiver le script `resetnet` pour un débug plus aisné, ce script monopolisant le terminal tant que la connexion au réseau n'est pas établie :

```
1 IMAGE_INSTALL += " ${CONNECTIVITY} init-gpio helloworld"
```

Le script d'initialisation devra lancer le programme `helloworld` dans un processus distinct puis se terminer afin de permettre au processus d'init de se terminer et au prompt d'apparaître. Il serait également intéressant de rediriger les messages produits par le programme vers une console différente de celle où est affichée le prompt (`tty1`). Afficher les messages sur le terminal principal aurait pour effet de bloquer celui ci et ne pas les afficher du tout n'aurait aucun intérêt ici.

On choisit donc de rediriger les messages affichés par le programme vers la console `tty3` accessible par la combinaison de touches `[ctrl] + [alt] + [F3]`. La console `tty2` sera attribuée au programme principal.

```
1 start-stop-daemon -S -b -C -q -x ${DAEMON} > /dev/tty3
```

La ligne ci-dessus permettrait de réaliser cela avec la commande `start-stop-daemon` dont l'usage est recommandé pour gérer ses daemons, cependant la version de `start-stop-daemon` inclue dans Busybox ne reconnaît pas l'option `-c` qui permet de préserver le flux de sortie du daemon lorsque celui ci est lancé en arrière plan. On préfère donc utiliser une syntaxe plus classique :

```
1 ${DAEMON} > /dev/tty3 &
```

Contenu de `helloworld.sh` :

```
1#!/bin/sh
2### BEGIN INIT INFO
3# Provides:                      helloworld
4# Required-Start:      $remote_fs $syslog
5# Required-Stop:       $remote_fs $syslog
6# Default-Start:        5
7# Default-Stop:        0 1 6
8# Short-Description:   Helloworld test
9## END INIT INFO
10
11NAME='helloworld'
12DESC='Test program'
13DAEMON="/usr/bin/helloworld"
14
15case "$1" in
16    start)
17        echo -n "Starting ${DESC} : ${NAME}... "
18        #start-stop-daemon -S -b -C -q -x ${DAEMON} > /dev/tty3
19        ${DAEMON} > /dev/tty3 &
20        echo "Done"
21        ;;
22    stop)
23        echo -n "Stopping ${DESC} : ${NAME}... "
24        start-stop-daemon -K -q -x ${DAEMON}
25        echo "Done"
26        ;;
27    restart)
28        $0 stop
29        $0 start
30        ;;
31    status)
32        start-stop-daemon -T -q -x ${DAEMON}
33        case "$?" in
34            0)
35                echo "Le programme ${NAME} est en cours d'exécution."
36                ;;
37            1)
38                echo "Le programme ${NAME} n'est pas en cours d'exécution
↪ et le fichier PID existe."
39                ;;
40            3)
41                echo "Le programme ${NAME} n'est pas en cours d'exécution."
42                ;;
43            4)
44                echo "Impossible de déterminer l'état du programme ${NAME}."
45                ;;
46        esac
47        ;;
48    *)
49        echo "Usage : $0 {start|stop|restart|status}"
50        exit 1
51        ;;
52    esac
53 exit 0
```

Note : Les codes relatifs à l'état de fonctionnement du programme sont donnés dans le manuel de **start-stop-daemon**.

Lorsque l'on démarre la Raspberry Pi, le splashscreen se termine et un prompt apparaît. Si l'on appuie sur [ctrl] + [alt] + [F3] on voit effectivement toutes les 2s une nouvelle ligne **helloworld : TEST MESSAGE** apparaître.

La gestion de ce simple programme est validée, le système d'exploitation est prêt pour l'intégration du programme principal qui ne sera qu'une adaptation de la procédure à un programme au code source plus complexe.

### 3.10.2 Programme principal

L'intégration du programme principal est basée sur le travail effectué précédemment avec un programme helloworld.

Le code source est cette fois ci composé de trois fichiers :

- main.cpp
- MyTcpSocket.h
- MyTcpSocket.cpp

Arborescence de la layer associée au programme principal :

```
1 ~/yocto/sources/meta-remote-estei $ \
2     tree recipes-project/remote/
3 recipes-project/remote/
4     ├── files
5     │   ├── main.cpp
6     │   ├── MyTcpSocket.cpp
7     │   ├── MyTcpSocket.h
8     │   ├── remote.pro
9     │   └── remote.sh
11    └── remote.bb
```

Contenu de **remote.pro** :

```
1 -----
2 #
3 # Project created by QtCreator 2018-01-31T09:10:03
4 #
5 -----
6
7 QT += core network
8
9 QT -= gui
10
11 TARGET = remote
12 CONFIG += console
13 CONFIG -= app_bundle
14 INCLUDEPATH += .
15
16 TEMPLATE = app
17
```

```

18 SOURCES += main.cpp \
19     MyTcpSocket.cpp
20
21 HEADERS += \
22     MyTcpSocket.h
23
24 target.path = /usr/bin
25 INSTALLS += target

```

Contenu de la recipe `remote.bb`:

```

1 SUMMARY = "Main Program"
2 LICENSE = "MIT"
3 LIC_FILES_CHKSUM = "file:///${COREBASE}/meta/COPYING.MIT;
        ↪ md5=3da9cfbc788c80a0384361b4de20420"
4
5 SRC_URI = " \
6     file://remote.sh \
7     file://remote.pro \
8     file://main.cpp \
9     file://MyTcpSocket.h \
10    file://MyTcpSocket.cpp \
11    "
12
13 SRCREV = "${AUTOREV}"
14 S = "${WORKDIR}"
15 PR = "r0"
16 DEPENDS = " qtbase"
17
18 inherit update-rc.d
19 inherit qmake5
20
21 do_install_append() {
22     install -d ${D}${sysconfdir}/init.d/
23     install -m 0755 ${WORKDIR}/remote.sh ${D}${sysconfdir}/init.d/remote
24 }
25
26 INITSCRIPT_NAME = "remote"
27 INITSCRIPT_PARAMS = "start 99 5 . stop 00 0 1 6 ."

```

Le script d'initialisation est semblable à celui du helloworld à la différence que la redirection du flux de sortie du programme se fait vers le terminal tty2. Un délai a également été ajouté avant l'invocation du programme car sans lui, la première tentative de connexion du programme au serveur échoue systématiquement bien que le système d'exploitation soit connecté au réseau MASTER\_SE.

Contenu du script `remote.sh`:

```

1#!/bin/sh
2### BEGIN INIT INFO
3# Provides:          remote
4# Required-Start:    $remote_fs $syslog
5# Required-Stop:     $remote_fs $syslog
6# Default-Start:    5
7# Default-Stop:     0 1 6
8# Short-Description: Core screenless-remote program
9## END INIT INFO
10
11 NAME='remote'
12 DESC='Core screenless-remote program'
13 DAEMON='/usr/bin/remote'
14
15 case "$1" in
16     start)

```

```
17     echo -n "Starting ${DESC} : ${NAME}... "
18     sleep 2s
19     ${DAEMON} > /dev/tty2 &
20     echo "Done"
21     ;;
22 stop)
23     echo -n "Stopping ${DESC} : ${NAME}... "
24     start-stop-daemon -K -q -x ${DAEMON}
25     echo "Done"
26     ;;
27 restart)
28     $0 stop
29     $0 start
30     ;;
31 status)
32     start-stop-daemon -T -q -x ${DAEMON}
33     case "$?" in
34         0)
35             echo "Le programme ${NAME} est en cours d'exécution."
36             ;;
37         1)
38             echo "Le programme ${NAME} n'est pas en cours d'exécution
↪ et le fichier PID existe."
39             ;;
40         3)
41             echo "Le programme ${NAME} n'est pas en cours d'exécution."
42             ;;
43         4)
44             echo "Impossible de déterminer l'état du programme ${NAME}."
45             ;;
46         esac
47     ;;
48 *)    echo "Usage : $0 {start|stop|restart|status}"
49     exit 1
50     ;;
51     esac
52     ;;
53 exit 0
```

## Chapitre 4

# Conclusion

Voici donc l'arborescence complète de la layer yocto dédiée au projet :

```

1 ~/yocto/sources/meta-remote-estei $ tree
2 .
3   ├── conf
4   │   └── layer.conf
5   ├── recipes-connectivity
6   │   ├── init-ifupdown
7   │   │   ├── files
8   │   │   └── interfaces
9   │   │       └── init-ifupdown_%.bbappend
10  │   ├── resetnet
11  │   │   ├── files
12  │   │   └── resetnet.sh
13  │   └── resetnet.bb
14   └── wpa-supplicant
15     ├── wpa-supplicant
16     │   └── wpa_supplicant.conf-sane
17     └── wpa_supplicant_%.bbappend
18   └── recipes-core
19     └── images
20       └── screenless-remote-image.bb
21   └── recipes-estei
22     ├── base-files
23     │   ├── base-files_%.bbappend
24     │   └── files
25     │       ├── motd
26     │       └── motd.tmp
27     ├── psplash
28     │   ├── files
29     │   └── psplash-poky-img.h
30     └── psplash_git.bbappend
31   └── sysvinit
32     └── sysvinit-inittab_%.bbappend
33   └── recipes-project
34     ├── helloworld
35     │   ├── files
36     │   │   ├── helloworld.pro
37     │   │   ├── helloworld.sh
38     │   │   └── main.cpp
39     │   └── helloworld.bb
40     ├── init-gpio
41     │   ├── files
42     │   │   └── init-gpio.sh
43     │   └── init-gpio.bb
44     └── remote
45       ├── files
46       │   ├── main.cpp
47       │   ├── MyTcpSocket.cpp
48       │   ├── MyTcpSocket.h
49       │   └── remote.pro
50       └── remote.sh
51       └── remote.bb

```

Ainsi que le code complet de la recipe-image `screenless-remote-image.bb` tel qu'il devra être pour la version finale du système d'exploitation :

```
1 DESCRIPTION = "Screenless remote image"
2 LICENSE = "MIT"
3 export IMAGE_BASENAME = "screenless-remote-image"
4 include recipes-core/images/rpi-basic-image.bb
5
6 inherit extrausers
7 EXTRA_USERS_PARAMS = "\n    usermod -P estei root; "
8
9
10 CONNECTIVITY = " \
11     linux-firmware \
12     i2c-tools \
13     python-smbus \
14     bridge-utils \
15     hostapd \
16     iptables \
17     wpa-supplicant \
18 "
19
20 DISTRO_FEATURES += "wifi"
21 MACHINE_FEATURES += "wifi"
22 IMAGE_INSTALL += " \
23     ${CONNECTIVITY} \
24     init-gpio \
25     resetnet \
26     remote \
27 "
```

La layer est disponible en libre accès sur internet à l'adresse suivante :  
<http://github.com/thomaslepoix/meta-remote-estei/>

La partie électronique de la télécommande est fonctionnelle et son système d'exploitation également. Cette partie du projet a donc été menée à terme.

Une amélioration possible serait d'initialiser l'état des entrées/sorties avec l'utilisation du device tree au lieu d'un script bash.

### **Troisième partie**

**Thibaud LE DOLEDEC - Programme  
de gestion des boutons**

## Chapitre 5

# Programme de gestion des boutons

L'objet de cette partie est de réaliser un programme permettant la récupération des actions sur les boutons, leurs conversion en commandes et l'envoi de celles-ci aux serveur afin de piloter le rover.

### 5.1 Explication du fonctionnement

Dans un premier temps il est nécessaire d'initialiser les gpio qui se trouve dans le répertoire "/sys/class/gpio" grâce à deux script: export et unexport. Un "echo N > export" où N est le numéro de la gpio souhaité, crée un répertoire gpioN dans "/sys/class/gpio" dans lequel se trouve deux fichiers texte "value" et "direction", contenant la valeur de la gpio(1/0) et sa direction(in/out). L'arborescence suivante montre ce répertoire:

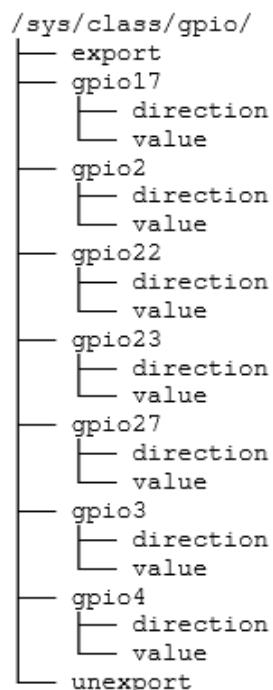


FIGURE 5.1 – Arborescence du répertoire gpio

Le tableau suivant nous montre le numéro de la gpio, son rôle ainsi que le chemin permettant d'y accéder:

numéro de la gpio	rôle de la gpio	chemin d'accès
22	haut	/sys/class/gpio/gpio22/value
17	bas	/sys/class/gpio/gpio17/value
4	droite	/sys/class/gpio/gpio4/value
23	gauche	/sys/class/gpio/gpio23/value
3	rotation droite	/sys/class/gpio/gpio3/value
27	rotation gauche	/sys/class/gpio/gpio27/value
2	power	/sys/class/gpio/gpio2/value

Les tâche à réaliser sont donc:

- Récupération de valeurs des boutons
- Conversion des valeurs en commandes
- Envoi des commandes au serveur

Afin de concevoir ce programme j'ai décidé de créer une machine d'état représentant l'état des boutons. L'organisation de la machine d'état est la suivante:

0	1/0	1/0	1/0	1/0	1/0	1/0	1/0
inutilisé	power	rotation gauche	rotation droite	gauche	droite	bas	haut

Une première fonction récupère la valeur de chaque boutons sous forme de chaîne de caractères et la convertie en un booléen qui est stocké dans la machine d'état à l'endroit correspondant au bouton actionné. Une deuxième fonction utilise la machine d'état afin de générer les commandes à envoyer au serveur, elle traite les boutons suivant un certain ordre de priorité. Le premier bouton à être testé est le bouton "power", qui déconnecte la télécommande du serveur et l'arrête, les suivants sont les boutons contrôlant la rotation du rover à droite et à gauche, car ces trois boutons ne peuvent être utilisé en combinaison avec d'autres. Enfin les 4 derniers boutons (haut/bas/droite/gauche) sont testés car il peuvent être combinés par exemple:

- haut/doite : déplacement en diagonale vers l'avant sur la droite
- haut/gauche : déplacement en diagonale vers l'avant sur la gauche
- bas/droite : déplacement en diagonale vers l'arrière sur la droite
- bas/gauche : déplacement en diagonale vers l'arrière sur la gauche

Il existe également deux combinaisons spéciales:

- haut/bas : déconnections du serveur
- droite/gauche : connections au serveur

## 5.2 Algorigrammes

### 5.2.1 Fonction principale

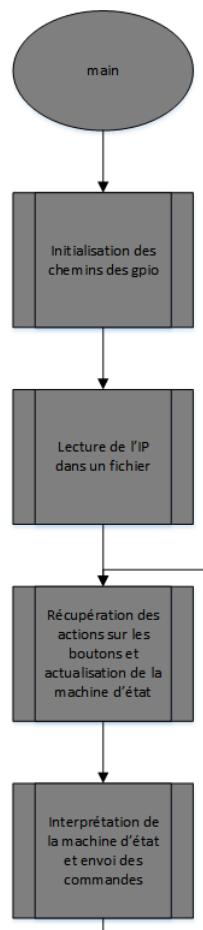


FIGURE 5.2 – Algorigramme de la fonction principale

Dans un premier temps la fonction d'initialisation affecte à chaque membre du tableau le chemin d'une gpio. Dans un second temps une fonction vient lire le fichier dans lequel est définis l'IP du serveur et configurer le programme d'envoi TCP/IP. Vient ensuite une boucle infini contenant deux fonctions: la première permet la récupération de la valeur des boutons et l'actualisation de la machine d'état, la deuxième permet l'interprétation de la machine d'état en commandes et leurs envoi.

### 5.2.2 Fonction de récupération des valeurs des boutons

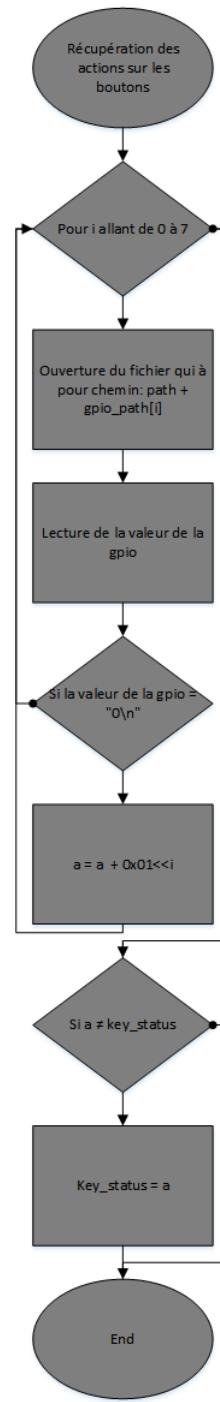


FIGURE 5.3 – Algorigramme de la fonction de récupération des valeurs des boutons

La fonction de récupération des valeurs des boutons s'organise de la façon suivante:

Premièrement une boucle permet d'ouvrir les fichiers "value" de nos gpio dans l'ordre que nous avons défini à savoir l'ordre de la machine d'état, pour ce faire la variable "i" de la boucle est utilisée pour indexer le tableau contenant les chemins des gpio(gpio\_path[]).

Par exemple le bit 0 de la machine d'état correspond au bouton "haut" donc le chemin vers le fichier "value" du bouton "haut" se trouve dans le membre 0 du tableau "gpio\_path[]".

Ainsi si la valeur contenu dans le fichier "value" est égale à "0\n" (bouton activé/logique inversée) on ajoute à une variable "a" 0x01 décalé de "i".

A la fin de la boucle si notre variable "a" est différente de la machine d'état alors on actualise la machine d'état sinon on ne fait rien.

### 5.2.3 Fonction d'interprétation de la machine d'état et d'envoi des commandes

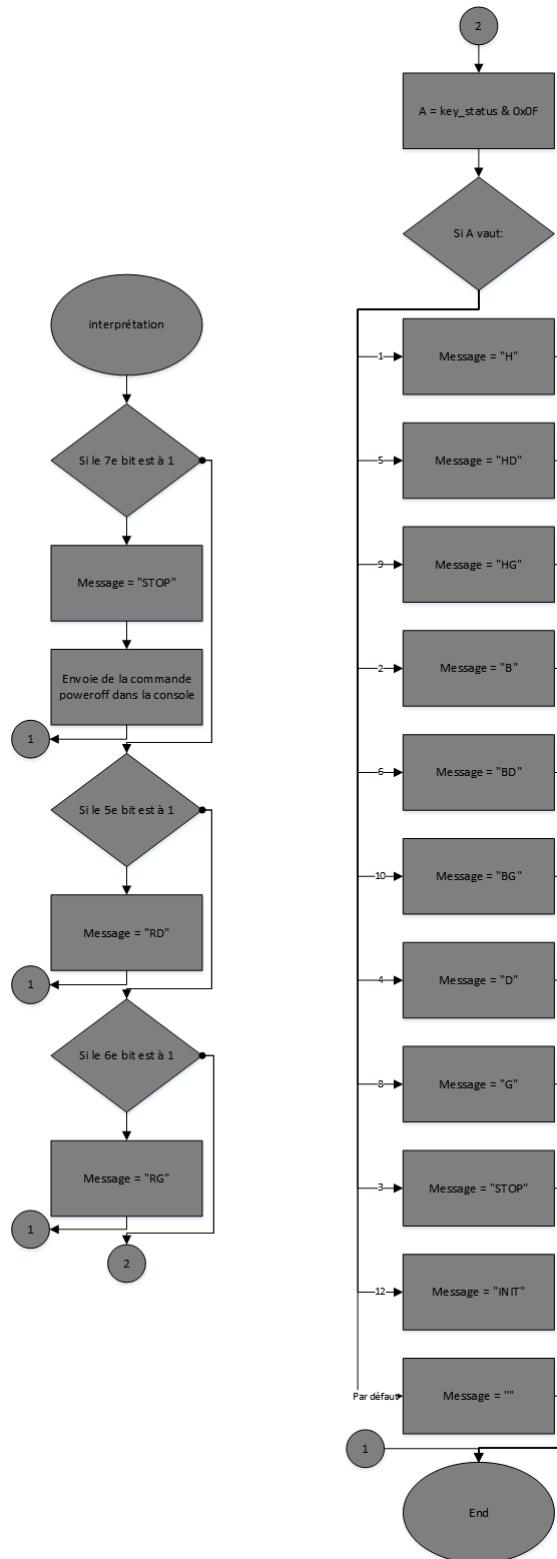


FIGURE 5.4 – Algorigramme de la fonction d'interprétation de la machine d'état et d'envoi des commandes

La fonction d'interprétation de la machine d'état et d'envoi des commandes peut être divisée en deux parties.

La partie de gauche gère l'appui sur les boutons qui ne peuvent être combinés, à savoir le bouton "power" ainsi que les boutons "rotation droite" et "rotation gauche". Le premier bouton à être testé est le bouton "power", si il est enclenché la télécommande envoie une commande de déconnexions au serveur puis s'éteint. Le deuxième bouton à être testé est le bouton "rotation droite", si il est actionné la télécommande envoie au serveur la consigne de rotation vers la droite. Enfin le troisième bouton à être testé est le bouton "rotation gauche", si il est actionné la télécommande envoie au serveur la consigne de rotation vers la gauche.

La partie de droite permet la gestion des boutons pouvant être combinés, cette partie n'est exécutée seulement si aucun des trois boutons évoqué précédemment n'est actionnés.

La première étape est un masquage de la machine d'état qui permet de ne tester uniquement les 4 derniers bits, cette opération est affecté à une variable "A". La variable "A" s'ordonne de la façon suivante:

0	0	0	0	1/0	1/0	1/0	1/0
inutilisé	power	rotation gauche	rotation droite	gauche	droite	bas	haut

N'ayant plus que les bits de poids faible variant, les valeurs de cette variable peuvent aller de 0 à 15, cela facilite l'interprétation des appuis sur les boutons car il ne reste que 16 cas au lieu de 128. Les cas possibles sont les suivants:

gauche( $2^3$ )	droite( $2^2$ )	bas( $2^1$ )	haut( $2^0$ )	valeur de "A"	commande envoyée
0	0	0	1	1	"H"
0	0	1	0	2	"B"
0	1	0	0	4	"D"
1	0	0	0	8	"G"
0	1	0	1	5	"HD"
1	0	0	1	9	"HG"
0	1	1	0	6	"BD"
1	0	1	0	10	"BG"
0	0	1	1	3	"STOP"
1	1	0	0	12	"INIT"
0	0	0	0	0/7/11/13/14/15	""

## **Quatrième partie**

**Taha MENEABI - Connexion  
client-serveur et boîtier de la  
télécommande**

## Chapitre 6

# Connexion client-serveur

### 6.1 Qu'est ce qu'un socket?

Un socket est considéré comme un point de terminaison d'une communication bidirectionnelle entre un client et un serveur. Le serveur et le client sont liés par un numéro de port TCP afin de pouvoir identifier la demande ou pas de partage de données. Le serveur est lié à un numéro de port bien spécifique et va se mettre à l'écoute d'un client qui demande une connexion.

Un socket utilise un descripteur de fichier standard (clé abstraite pour accéder à un fichier). Afin d'acquérir le descripteur de fichier pour la communication réseau, il faut appeler un descripteur de socket qu'on va utiliser comme interface de communication en utilisant `send()`/`recv()` socket API. Les ports vont ainsi servir à différencier plusieurs point de terminaison sur une adresse réseau. De plus, le numéro de port n'est pas important que pour le serveur/client mais peut être interprété par tous les autres périphérique de l'infrastructure réseau. En particulier les firewalls qui sont configurés pour comparer les paquets en fonction de leur port.

C'est la paire de socket qui va nous permettre de définir les deux points de terminaison soit :

- L'adresse IP du client
- Le numéro de port du client
- L'adresse IP du serveur
- Le numéro de port du serveur

On ne peut lier qu'une IP avec un port en utilisant le même protocole de communication sinon on aura un conflit de port où plusieurs programme vont tenter de se connecter au même numéro de port en utilisant la même adresse et le même protocol.

Dans notre cas on a bien différencier le port utiliser par la télécommande et la voiture.

Afin d'établir la connexion entre la Rpi et le serveur il a fallu créer une socket connection. Ainsi ces deux machines vont avoir les informations respectives comme expliqué au dessus. Il a ainsi fallu créer deux objets de type socket un pour le serveur et un pour la Rpi afin d'établir la communication c'est à dire l'envoi de bits des deux directions.

### 6.2 TCP vs UDP

Il existe différents types de socket qui vont déterminer la structure de la couche transport. Les plus usuels sont les stream sockets que nous avons utilisés et les datagram sockets.

TCP Segment Header Format												
Bit #	0	7	8	15	16	23	24	31				
0	Source Port				Destination Port							
32	Sequence Number											
64	Acknowledgment Number											
96	Data Offset	Res	Flags		Window Size							
128	Header and Data Checksum				Urgent Pointer							
160...	Options											

UDP Datagram Header Format								
Bit #	0	7	8	15	16	23	24	31
0	Source Port				Destination Port			
32	Length				Header and Data Checksum			

FIGURE 6.1 – Format de trames TCP et UDP

## Stream Sockets

Le socket de type stream fournit une communication bidirectionnelle fiable comparable à lorsqu'on appelle une personne au téléphone. Un côté initialise la communication et une fois que la connection est établie n'importe lequel des interlocuteurs peut communiquer. De plus, on est sûr que le message est arrivé à destination. Les stream sockets utilisent un protocole de transmission contrôlé (TCP) c'est à dire que les données sont transmises sous forme de paquet sans erreur et séquentielles. Que ce soit les serveurs web, mail ainsi que les applications clients, toutes utilisent le protocole TCP et les stream socket afin de communiquer.

## Datagram Sockets

Communiquer avec les datagram socket revient au même qu'envoyer une lettre puis appeler pour savoir si la lettre est arrivée. À l'inverse des stream sockets, communiquer en utilisant un datagram socket est unidirectionnel et non fiable. Ce n'est pas une réelle connection.

Afin d'établir une socket de connexion côté rpi (client) on a besoin de :

- Créer une socket avec l'appel de socket()
- Connecter la socket à l'adresse du serveur en appelant connect()
- Envoyer et recevoir les données, la plus simple des méthodes est en appelant read() et write()

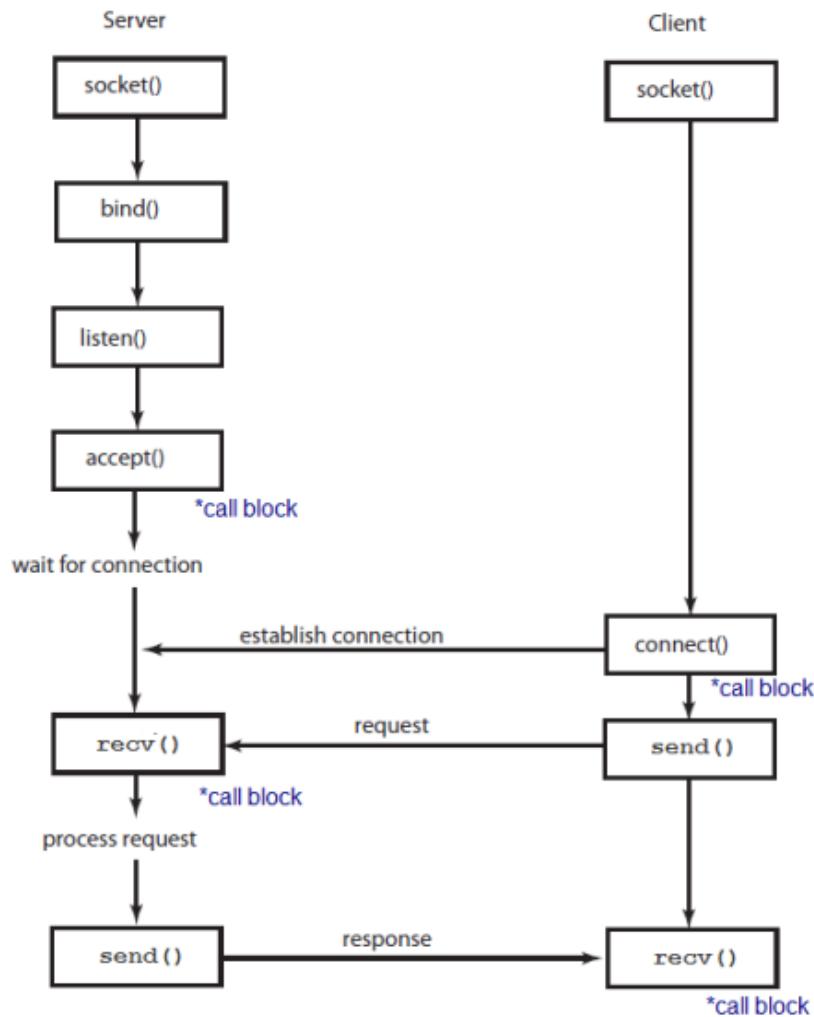


FIGURE 6.2 – Illustration du fonctionnement du socket

## 6.3 Utilisation des sockets sur Qt

QTcpSocket est une classe qui va nous fournir un TCP Socket. TCP est l'abréviation de "Transmission Control Protocol" ce qui se traduit littéralement par protocole contrôlé de transmission. C'est un protocole qui va nous permettre de contrôler le flux ainsi que la connection qui est très favorisé pour une transmission de données en continu. QTcpSocket est une sous classe de QAbstractSocket qui pose les fonctionnalités de base de tout type de socket. Tout d'abord il a fallu ajouter le module réseau à notre fichier projet .pro : QT += network. Viens la création de la classe MyTcpSocket qui hérite de QObject.

### 6.3.1 void QAbstractSocket::connectToHost

Cette fonction va permettre d'établir la connection au hostName sur le quint16 port. Le NetworkLayerProtocol nous permet de choisir le port du réseau (IPv4 ou IPv6).

La socket est ouverte en fonction de l'OpenMode et va commencer par entrer en état HostLookupState qui va identifier l'hostName. Si la vérification est positive, un hostFound() est émis et notre QAbstractSocket entre en mode connection c'est à dire: ConnectingState. Enfin

si la connection est établie QAbstractSocket rentre en état ConnectedState et émet connectected(). A n'importe quel moment, la socket peut émettre un error() qui signifie qu'une erreur est survenue. L'hostName peut être une adresse IP comme dans notre cas ou un nom de type "toto.com".

### **6.3.2 bool QAbstractSocket::waitForConnected(int msec = XXX)**

Permet d'attendre que notre socket soit connectée sur une durée XXXms. Si la connection est établie, cette fonction va retourner true, sinon false. Une fois que false est retournée par celle ci, nous pouvons appeler error() afin de determiner la cause de l'erreur.

### **6.3.3 bool QAbstractSocket::waitForReadyRead(int msec = XXX)**

Cette fonction hérite de QIODevice::waitForReadyRead(). Elle est en état bloquée jusqu'à ce que de nouvelles données sont disponible pour la lecture et que readyRead() est émis. Cette fonction sera active durant XXXms. Elle retournera true si readyRead est émis et qu'il y'a de nouvelles données disponible pour la lecture, sinon false.

### **6.3.4 qint64 QAbstractSocket::bytesAvailable() const**

Cette fonction hérite de QIODevice::bytesAvailable(). Elle retourne le nombre d'octets en attente de lecture.

### **6.3.5 QByteArray QIODevice::readAll()**

Cette fonction permet de lire toutes les données disponibles et les retourne en QByteArray.

## Chapitre 7

# Design du boîtier

Afin de créer le boîtier, j'ai choisis la solution OnShape pour différentes raisons: La première étant le fait que c'est une solution gratuite qui nécessite pas d'installation de logiciel. De plus, ce site a été utilisé pour faire le socle du Robot et était donc à la portée d'autre étudiants, on en a ainsi profité pour apprendre à l'utiliser.

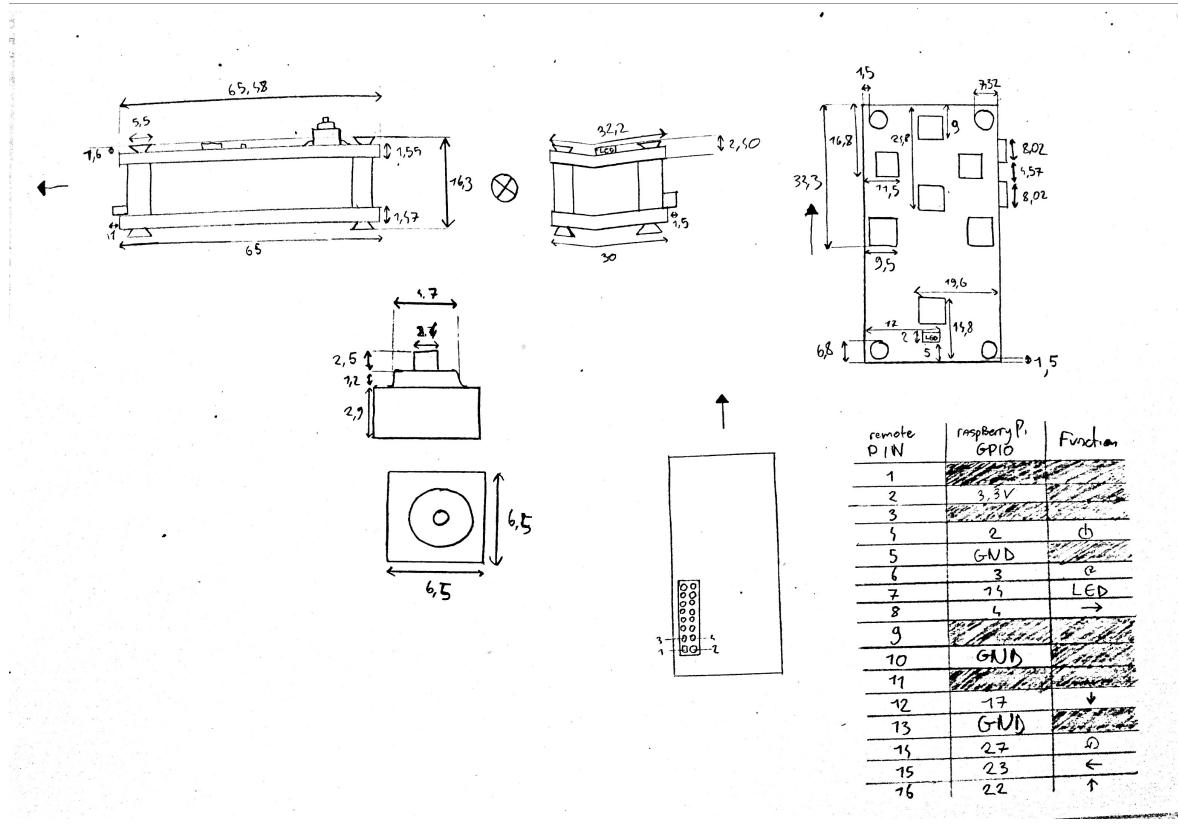


FIGURE 7.1 – Schéma des dimensions de la carte pluggée sur la Raspberry Pi

Tout d'abord, on a commencé par créer le sketch qui est la vue de dessus des contours de notre socle avec les écritures ainsi que les supports des vis. Ensuite viens la fonction extrude qui va nous permettre de creuser ou de d'élargir des parties du sketch.

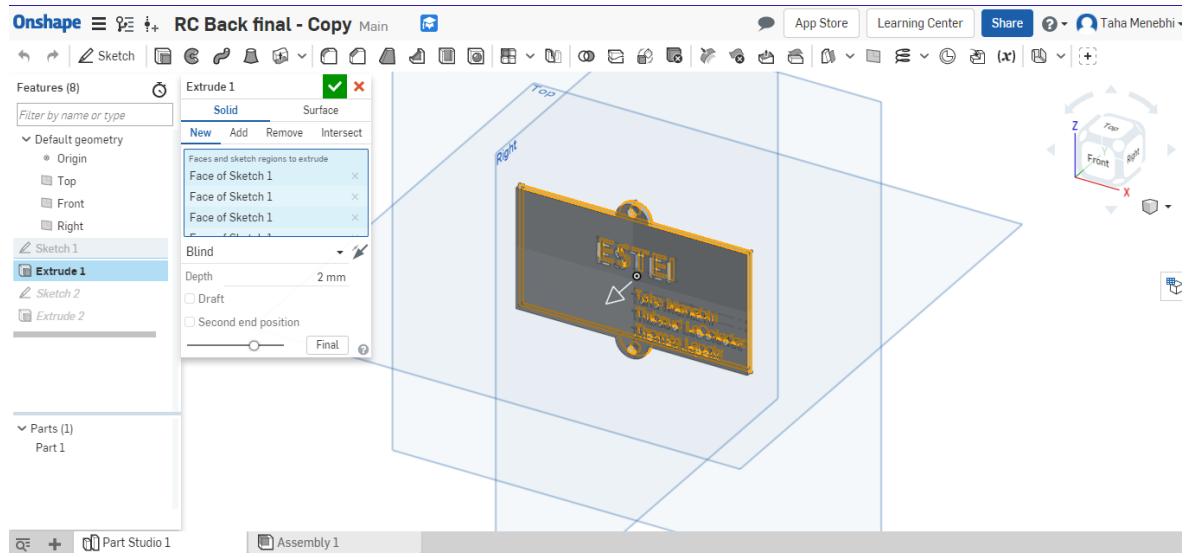


FIGURE 7.2 – OnShape : Face inférieure du boîtier

Enfin pour la partie dessus de notre couvercle, afin que les bords de ce dernier ne soient pas pointu mais arrondis, nous avons utilisé l'outil Fillet qui nous permet d'avoir cette forme plus pratique et confortable comme montré ci dessous.

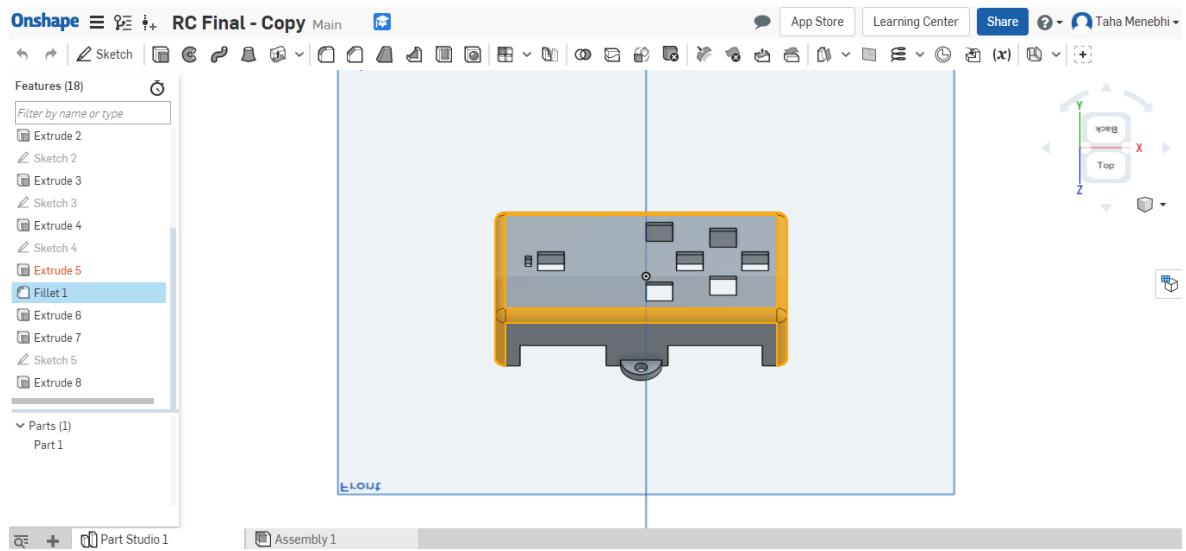


FIGURE 7.3 – OnShape : Face supérieure du boîtier

## **Cinquième partie**

## **Conclusion**

À l'heure actuelle le montage électronique de la télécommande ainsi que son système d'exploitation sont opérationnels. L'ajout du programme principal au système d'exploitation est un succès.

Une phase d'intégration a été entamée avec l'équipe chargée de la réalisation du serveur et n'a pour l'instant pas été menée à terme.

Une amélioration possible serait d'initialiser l'état des entrées/sorties avec l'utilisation du device tree au lieu d'un script bash.

Une seconde amélioration serait de réaliser un programme gérant les entrées/sorties avec les interruptions du système d'exploitation plutôt qu'une boucle perpétuelle lisant le contenu des fichiers de `/sys/class/gpio/`.

Une troisième amélioration pensée lors de la phase d'intégration avec le serveur serait de déporter l'adresse IP du serveur jusqu'à présent notée en dur dans le code source du programme client dans un fichier, par exemple `/etc/remote_ip.conf`. Cela permettrait de s'affranchir de devoir reconstruire l'image pour modifier cette adresse, et par là même d'alléger la phase d'intégration et de rendre la télécommande davantage configurable.