YNOV

PROJET DE MASTER EN PARTENARIAT AVEC THALES

Intégration du driver IMX219 sur une OpenRex Basic

Auteur :
Alan AIT-ALI
Martin LAPORTE
Clément AILLOUD
Romain PETIT

Superviseurs :
Patrick PIQUART
David COUÉ

Rapport final présentant le travail effectué sur l'intégration du driver IMX219 sur une OpenRex Basic

au sein du

Département Aéronautique & Systèmes Embarqués

22 février 2018

YNOV

Résumé

YNOV
Département Aéronautique & Systèmes Embarqués

Master Systèmes Embarqués

Intégration du driver IMX219 sur une OpenRex Basic

par Alan AIT-ALI, Martin LAPORTE,
Clément AILLOUD & Romain PETIT

Actuellement, les casques de réalité augmentée pour pilote migrent vers le domaine civil. Thales a réalisé un prototype fonctionnant avec une Raspberry Pi et la Raspberry Pi Camera pour l'acquisition d'images ou de flux vidéo. Thales a fait appel aux étudiants du campus Ynov pour participer à l'amélioration de ce prototype.

L'objectif du projet est de réaliser le portage du driver de la caméra sur un autre système d'exploitation que Raspberry Pi. Ce système devra donc être capable faire des captures d'image et un flux vidéo de la caméra depuis une autre carte, l'Openrex-IMX6-Quad car pour des raisons internes à l'équipe de développement Thales-LUCY.

Avec Yocto, nous avons généré dans un premier temps, un système d'exploitation fonctionnel sur la carte cible (kernel Linux 3.14 puis 4.1). Par la suite, nous avons essayé de faire fonctionner des codes existants en les adaptant à la carte Openrex. N'y arrivant pas, nous avons alors entrepris la rédaction d'un driver de la caméra. Actuellement, nous avons réussi à établir une connexion I2C entre la caméra et l'IMX6 Openrex en utilisant le port CSI-2.

Notre travail étant inachevé, ce rapport permet de partager nos connaissances acquises lors du déroulement du projet. Il est nécessaire pour une bonne compréhension d'être introduit à l'usage de Yocto.

Nowadays augmented reality pilot's helmet move to the civil domain. At the moment Thales group prototyped one working with a Raspberry Pi and it's associated raspberry camera v2 for picture and video capture. Thales called ynov campus students for applications at being part of the prototype improvement.

Our aim is to build an Raspberry Pi camera, picture and video capture compatible, operating system on l'Openrex-IMX6-Quad. Indeed for Thales-LUCY team internal reasons the initial Raspberry Pi board won't be used in the MVP project state.

In a first hand we built a Yocto-project generated target board operating system (kernel Linux 3.14 then 4.1). In a second hand we tried to port existing source codes to the Openrex board. Unable to succeed, we undertook the camera driver redaction from scratch. We already connected the camera module with both CSI and I2C communication protocols as the mobile industry processor interface (MIPI) standard describes.

As our work in unfinished, the present report hand our state of knowledge and achievement over. Linux operating system Yocto generation usage knowledge is needed for it's good understanding.

Remerciements

En premier lieu, nous tenons à remercier l'équipe de Thales qui a bien voulu nous faire confiance pour porter un de leur projet et pour la bienveillance dont ils ont fait preuve lorsque les voies empruntées n'étaient pas les bonnes.

Nous remercions également l'équipe enseignante de notre formation sans qui ce projet n'aurait pas pu être, plus particulièrement Monsieur Pierre-Jean TEXIER pour ses conseils et son soutien technique.

Nous voudrions ensuite remercier Monsieur Patick PIQUART et Monsieur David COUE pour nous avoir permis de participer au projet au sein du groupe Ynov, ainsi que de leurs appuis en termes de gestion de projet.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Ce rapport est un rapport d'activité sur le travail effectué sur le rojet proposé par Thales. Ce rapport a été écrit par les 4 élèves participants au projet. Pour un souci de notation il nous a été demandé de signaler quelle partie a été traité par quel élève. Afin de ne pas rendre le rapport illisible en expliquant à chaque fois qui c'est chargé de faire telle ou telle partie nous avons ajouté des annotations en bas de page.

1.1 Projet professionnel

Les technologies d'affichage à tête haute ou HUD (head up display) apparaissent de plus en plus dans le commerce grand public (voitures, lunettes, chirurgies...). Il y a quelques années Thales a développé un casque à visée tête haute pour les pilotes militaires. Ce système permet de projeter des informations sur la vitre du casque du pilote sans gêner la vue réelle en arrière-plan. Cela évite aux avions de faire un premier passage pour repérer visuellement la cible avant de passer à l'action. À présent, la cible est balisée à vue dès l'arrivée de l'avion. Récemment Thales s'est lancé en interne à adapter ce produit pour des applications civiles.

Ce projet, appelé LUCY, a passé l'étape du proof of concept (POC). Aujourd'hui les ingénieurs travaillent sur son amélioration en minimum viable product (MVP) pour commencer à équiper le système de potentiels clients et à déterminer plus précisément les besoins de ceux-ci.

1.2 Projet étudiant

Notre travail au sein du projet professionnel consiste à améliorer la méthode permettant de détecter et suivre d'orientatier la tête du pilote par rapport au cockpit. En particulier lors de la phase de calibration, où il faut ajuster la ligne d'horizon sur la visière du pilote. Pour assurer la compatibilité des drivers, la preuve de faisabilité (proof of concept POC) était initialement constituée d'une caméra Raspberry-Pi (v2) contrôlée par une carte Raspberry Pi. Au passage au MVP les responsables du projet ont pris la décision de changer de carte tout en gardant la caméra Raspberry Pi. Notre rôle dans le projet est de préparer la carte Openrex pour l'acquisition d'images et de flux vidéo en mettant en place un driver compatible.

1.2.1 Cadre du projet étudiant

Le projet étudiant a pour but de permettre à la carte mère (single board computer, SBC) Openrex-basic avec microcontrôleur (system on chip, SOC) IMX6S, de contrôler le module (system on module, SOM) "Raspberry Pi Camera v2", de la fondation Raspberry Pi, pour réaliser des captures d'image et de flux vidéo.

Afin que le code applicatif existant sur la raspberry puisse être porté sur l'Openrex, on proposera donc un système d'exploitation (operating system, OS) avec la même version de kernel (4.1) que celle mise en place sur le POC.

Pour écourter la normalisation du produit, le système ne devra en aucun cas être en interaction avec les équipements avioniques, déjà présents sur l'appareil.

Chapitre 2

Gestion de projet

2.1 Organisation de l'équipe

2.1.1 Membres du projet

Nom / Prénom	Rôle
Patrick PIQUART	Scrum Master : Responsable de l'organisation interne et de la communication avec le client
David COUÉ	Product Owner : Responsable de la communication avec le client
Alan AIT-ALI	Développeur
Martin LAPORTE	Développeur
Clément AILLOUD	Développeur
Romain PETIT	Développeur

TABLE 2.1 – Membres du projet

2.1.2 Organisation externe

À l'initiative de ce projet, il s'est tenu une présentation pour les sections Master 1 et Master 2 de la formation. À l'issue de celle-ci, M. David Coué et M. Patrick Picard ont formé un groupe de 4 étudiants. Initialement, M. Patrick Picard a pris le rôle de "Scrum Master", c'est avec lui que s'est faite la "kick-off review". Une réunion a ensuite eu lieu pour que le groupe de travail puisse bien cerner les enjeux et qu'il soit en accord sur les méthodes de suivi de projet. Matérialisé par un "backlog product", cette méthode a permis d'avoir une bonne communication interne et de suivre l'avancement des différentes tâches. Pour la communication avec le client, le groupe d'étudiant devait impérativement passer par le "Product Owner" ou le "Scrum Master" pour faire remonter les informations.

Quelques temps après la "kick-off review", le groupe de travail rencontra les ingénieurs en charge du projet. À l'occasion de cet entretien fut exposé l'état de l'art ainsi que des avis sur certains points techniques.

L'emploi du temps du projet et celui du Scrum Master n'étant pas compatible, les "daily review" se sont tenues quasi-quotidiennement jusqu'à fin décembre entre les 4 étudiants. Face à la désinformation du "Scrum Master" et des clients, un changement de méthode nous a été conseillé en janvier. La daily fut alors remplacée par un rapport d'avancement. Cette méthode a permis de renouer le lien entre le travail fournit par les étudiants et le client. Cela a fait prendre une toute nouvelle direction au projet, malheureusement un peu tard car le projet devra s'arrêter le 25 février.

La gestion du partage des tâches au sein de l'équipe s'est auto-organisée selon les tâches à développer. On ne pourra pas attribuer de rôle bien précis à chacun car en fonction de l'avancée du projet les rôles se sont intervertis.

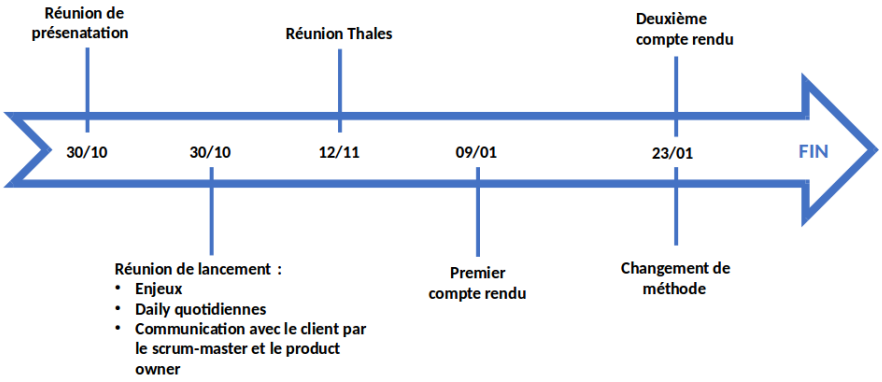

FIGURE 2.1 – Chronologie du projet

2.1.3 Organisation interne

À l'initialisation du projet, nous avons décidé de nous séparer en deux groupes. Le premier groupe portera son étude depuis la couche haut niveau. À l'inverse, le second groupe partira du bas niveau (modèle OSI). Pour mieux appréhender les différents problèmes et axes de développement dans leur ensemble.

Le binôme Clément et Romain recherchaient à faire le lien depuis les couches applicatives vers le kernel. Le binôme Alan et Martin au contraire cherchaient à établir en priorité les couches basses pour ensuite les rendre compatibles avec le kernel. Pour résumer les deux binômes n'avaient pas le même point de départ tout en ayant le même kernel comme point d'arrivée. Clément et Romain sont donc chargés de comprendre l'utilisation de Gstreamer et de la couche V4L2 ; Martin et Alan de rendre les drivers compatibles avec notre système.

Afin de se concentrer sur l'option la plus prometteuse (c.f. imx219 - Nvidia-Tegra Chromium-Os) nous nous sommes lancés dans l'analyse du code C du pilote, donc de son fonctionnement interne. Le code réparti entre chacun, nous avons cherché à comprendre les utilités des structures et de l'organisation du code. Nous avons poursuivi les appels aux fichiers systèmes, en mutualisant les informations oralement. Grâce à cette organisation agile nous avons pu préciser la source du problème avant d'en chercher la solution.

Par la suite Romain et Alan se sont lancés dans le second objectif d'évolution. Ils ont compilé un noyau linux récent, afin de précéder le portage des sources spécifiques à la carte de développement (board support package, BSP) Openrex de la distribution. En raison du peu de résultats et des nouvelles pistes données par le client, le groupe s'est orienté vers l'étude d'un driver existant. Chacun est chargé de comprendre et modifier les fonctions pour les rendre compatibles avec notre camera.

En clair, le groupe s'est organisé de façon à progresser le plus rapidement possible sur une même piste. Au maximum, deux pistes différentes étaient étudiées en parallèle. Lorsque le travail pouvait être divisé, chaque binôme s'occupait d'en prendre une partie pour ensuite tout remettre en commun.

2.2 Diagramme de Gantt

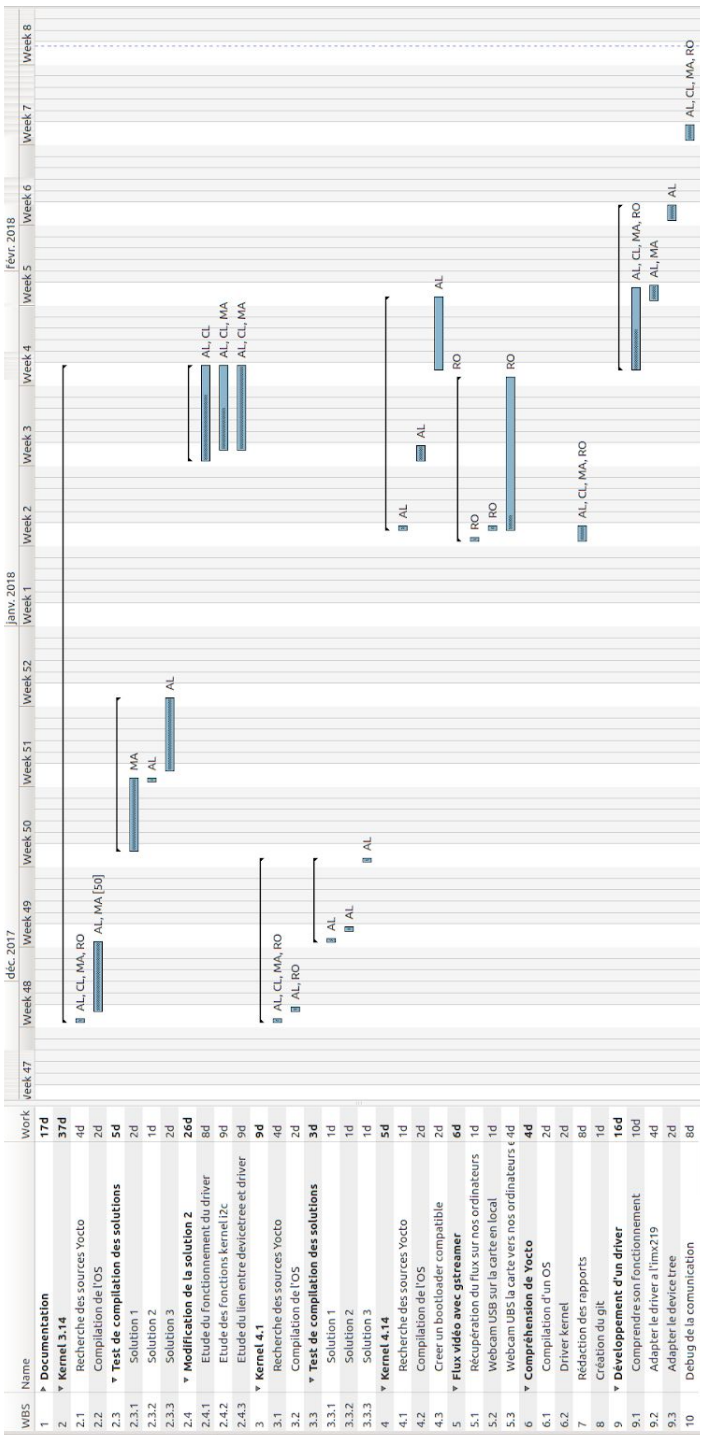

FIGURE 2.2 – Avancement détaillé du projet

2.3 Outils utilisés

2.3.1 Versionning des codes

GitHub est un service de versioning ainsi qu'un service web d'hébergement. Il permet de stocker toutes les sources d'un projet en différenciant par des versions. En effet, si nous effectuons des modifications corrompant tout le projet, il est possible de récupérer la version fonctionnelle si elle a été versionnée.

GitHub est trés utilisé dans le monde professionnel puisqu'il permet à plusieurs dizaines de personnes à travailler simultanément sur un projet, par exemple le système d'exploitation Linux sur GitHub mis à jour par des centaines de contributeurs.

Une fonctionnalité importante aussi est le système de "branches". Si l'on souhaite séparer certaines parties indépendantes d'un projet on utilise ce système. Cela permet de travailler sur les mêmes fichiers en parallèle sans que les modifications apportées par les autres ne posent de problèmes, à condition que les lignes concernées soient différentes. Le versioning d'un projet s'effectue avec les quelques commandes principales ci-dessous.

Commandes

Clone : Crée un dépôt local sur l'ordinateur depuis un dépôt en ligne

Add : Ajoute les fichiers ou dossiers dans l'index que nous voulons versionner sur le GitHub

Commit : Transfère le contenu de l'index vers le répertoire local ; commet la version. il est possible de rajouter un commentaire avec l'option -m

Push : Pousse les fichiers et dossiers contenus dans le répertoire local vers le dépôt en ligne après la commande commit

Pull : Actualise la branche locale sur l'ordinateur depuis un dépôt en ligne

Branch : Créé une nouvelle branche

Checkout Change de branche

Diff : Affiche les différences de fichiers entre le contenu local et le contenu du dépôt en ligne

Voic un shéma résumant graphiquement toutes les commandes ci-dessus :
Il existe sur internet de nombreuses recommandations pour entretenir un dépôt git propre. Étant débutants du principe, nous nous sommes concentrés sur l'aspect fonctionnel.
Ci-dessous, l'adresse de notre GitHub :
https://github.com/Alanaitali/meta-openrexpicam

2.3.2 Communication

A l'aube du projet quand nous avons choisi nos méthodes de communication, nous avons décidé d'utiliser le logiciel (et hébergeur) de dialogue instantané nommé Slack. D'une part nous avions déjà défini que nous nous verrions deux jours hebdomadairement d'autre part nous avions déjà choisi de dialoguer avec l'équipe Thales en passant par notre superviseur par mail qui relayerai les requêtes. Enfin comme précisé ci-dessus, un git était à l'œuvre pour les échanges de code.

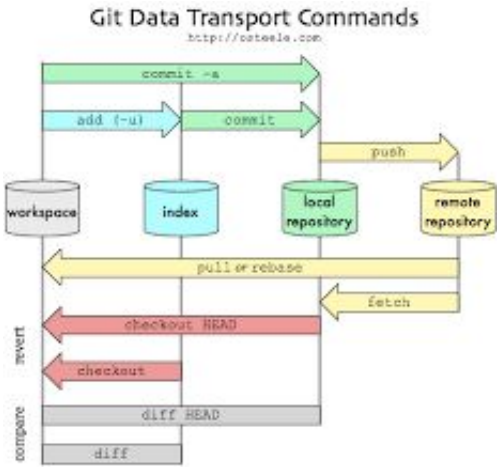

FIGURE 2.3 – Résumé des commandes git

Avantages de Slack

- Interface très complète de dialogue en groupe
- Capacité de rechercher parmi les messages
- Conversations publiques/privées, en sous-groupe... etc
- Accessible depuis un navigateur, et application pour ordinateur et smartphone de tous types, utile pour être notifié.
- Gestions séparées des projets pour éviter de s'éparpiller sur un autre contenu (orientation professionnelle)
- Appel à des applications en ligne possible (calc, github, google_drive ...)

Inconvénients de Slack

- Gestion des logins indépendantes entre les projets (contre intuitif)
- Applications mobiles et ordinateur non-native et donc plus consommatrices
- Interface nouvelle avec des capacités restées inexploitées (planning, fils de discussions sur un message)

2.3.3 Yocto

OpenEmbedded, était à son commencement en 2003, un projet de la société du même nom, rejointe plus tard par OpenZaurus. Avant son arrivée, l'outil massivement employé était Buildroot, celui-ci étant principalement prévu pour construire des systèmes de fichiers et non des distributions voire des SDK GNU-Linux.

En 2010 OpenEmbedded devient un "lab workgroup" de la fondation Linux avec 22 entreprises collaborant entre-elles. En 2011, lors de son rachat par Intel, le projet est nommé Yocto. Celui-ci a pour but de faciliter la conception de systèmes Linux avec une empreinte mémoire minime et la compilation croisée. Il permet théoriquement de développer une distribution spécifique aux besoins d'un utilisateur indépendamment de la cible et du poste de développement.

Enrichi par un grand nombre d'entreprises tel que NXP ou Texas instrument, le projet Yocto s'est développé et est maintenant maintenu autour de deux blocs :

- Bitbake : outil de construction dérivé du gestionnaire de paquet "portage" par l'équipe du projet OpenEmbedded. Bitbake active les différents ingrédients utiles à la compilation des recettes.
- OpenEmbedded-core : les sources de base sous forme de métadonnées pour la génération d'un système GNU/Linux base, la distribution poky.

Yocto peut être rendu compatible avec de très nombreuses combinaisons de SBC et SOM grâce à une gestion open-source et à la méthode de développement incrémental des recettes qu'il instaure. Il permet de générer une distribution complète (image, bootloader, SDK, rootfs, device-tree...), à partir d'assemblages de métadonnées. À l'intérieur des méta-données nous trouvons des recettes et des bouts de recettes dépendants d'une recette mère. Une recette correspond à un arbre de compilation.

Bitbake se charge alors de parcourir (fetch) les sources pour recomposer la recette à travers le fichier .bb et les fichiers .bbappend dans un arbre de compilation, puis exécute la compilation. Cela fait, il installe tous les binaires dans une même image. L'un des principaux inconvénients de Yocto, est le besoin de disposer d'un grand espace disque (environ 50 Gb). La première fois, Yocto a conservé plusieurs états des tâches. Ainsi, tout ce qui n'a pas été modifié ne sera pas exécuté à nouveau par Bitbake, on gagne alors un temps précieux en échange de l'espace mémoire (shared-state cache).

Poky

Poky c'est la distribution de référence générée par Yocto, elle est maintenue pour être compilable sur toutes les machines cibles officielles et se compose :

- d'un bootloader (U-boot)
- d'un Kernel Linux, et d'applications (GNU compilant pour la plupart)
- d'un device tree (fichier binaire .dtb) qui peut être interprété dans /sys/ par des drivers
- d'un système de fichiers partant du répertoire racine dit rootfs
- d'éventuels modules et drivers

Poky contient concrètement les lignes de codes nécessaires à l'obtention d'une distribution.

Bitbake

Bitbake est un outil de compilation de sources à partir de répertoires locaux et en ligne. Il décompose chaque phase de compilation en : do_fetch, do_unpack, do_patch, do_configure, do_compile, do_install, do_package. Une caractéristique essentielle de Bitbake est sa capacité à gérer les interruptions dans la compilation et de pouvoir reprendre la compilation là où il l'avait laissé, à un tâche prête. Les étapes do_fetch et do_unpack respectivement, téléchargent et décompressent les sources vers le répertoire de travail (variable $WORKDIR). Pour cela ils passent par un répertoire intermédiaire de téléchargement ($DL_DIR), ces sources peuvent être téléchargées à la demande par une commande, telle que celle ci-dessous, pour le packet

zlib. En effet, pour des problèmes de connexion réseau, il peut être nécessaire de déclencher manuellement le téléchargement des paquets non récupérés.

```
user@poky : /fsl-community-bsp/build$ Bitbake -c fetch zlib
```

Les étapes do_configure et do_compile correspondent à la préparation de l'arborescence de compilation et à la compilation même (précompilation, ln, as...) des recettes qui composeront l'image. Bitbake sous-traite la préparation et la compilation sur des commandes comme autotools, cmake, scon, qmake ou encore un script « ./configure », make, make install ; ce choix étant laissé aux rédacteurs du paquet.

Alors que la commande make se contente de placer le fichier Makefile pour ordonner les compilations, Bitbake apporte plus de dynamisme. Les ordres d'une compilation Bitbake sont décentralisés sur 3 formats de fichiers. Principalement les .bb sont les fichiers « recette-mère » qui seront parcourus par Bitbake et déclencheront la compilation (ou non) des sources en présence. Les fichiers .bbappend (recette-fille) permettent de compléter un fichier .bb, et les fichiers .bbclass permettent d'indiquer a Bitbake de prendre en compteles .bb et .bbappend. L'intérêt étant de classer les recettes mères et filles dans des dossiers (métadonnées ou meta) par fonctionnalité et non par dépendance de compilation. En constant développement, le "projet Yocto" se compose de poky, Bitbake, ainsi que l'ensemble des métadonnées mises à disposition par la communauté.

2.4 Caméra Raspberry Pi v2

FIGURE 2.4 – Raspi Cam v2

La Raspberry Pi Caméra (V2) est la dernière version de la gamme Raspberry. Cette caméra dispose d'un microcontrôleur imx219 développé par Sony. C'est un composant d'acquisition d'images associé à un système optique et à quelques composants passifs. Pour donner une notion de ses capacités, ce microcontrôleur est capable de réaliser une capture vidéo 1080p à 60 images/s.

Dans le document suivant nous parlerons toujours du driver imx219 pour désigner le driver de la caméra Raspberry Pi. La communication avec l'imx219 se fait à travers l'utilisation de l'I2C sur port CSI2 disposant d'une connexion D-phy 2 ou 4 Lanes et d'une clock pour le transfert du flux vidéo. Dans notre cas, nous travaillons avec la configuration 2 lanes. La connexion utilise donc 6 broches avec un doublet (une lane) de pistes en communication half-duplex et les 2 autres doublets en simplex.

L'utilisation de l'interface I2C permet de configurer les registres de l'imx219 et de récupérer le flux vidéo. la communication se fait à une fréquence comprise entre 11 MHz et 27 MHz.

Enfin la présence de la broche XCLR permet le reset du composant. De la documentation technique est accessible pour l'imx219, cependant nous n'avons pas trouvé la schématique de la carte caméra Raspberry .

2.5 Interface caméra MIPI CSI-2

L'alliance MIPI compte plus de 250 entreprises aujourd'hui mais parmi les 6 contributeurs initiaux (02/2004) on compte ARM limited, NXP Semiconductors et OmniVision Technologies AG. Ce dernier est également producteur de contrôleur-caméra. Nous reparlerons plus tard du composant ov5640 car celui-ci est déjà présent sur la plateforme openrex au kernel 4.1.

L'interface processeur des industriels du mobile standardise les communications avec tous lespériphériques habituels environnant le processeur d'un smartphone. Pour accéder à une caméra l'interface prévoit une communication par le protocole CSI ou CSI-2 et une liaison par les couches physiques C-phy et D-phy (M-phy étant en développement). Ce genre de connexion s'effectue avec 2 à 4 lanes de données et une lane d'horloge. Une lane est un même signal différentiel à haute fréquence. Dans le standard D-phy, une lane d'information est transportée sur deux pistes électroniques (en différentiel). Dans le standard C-phy, 2 lanes d'information peuvent transiter sur 3 pistes électroniques (doublement différentielles), mais nous ne nous intéresserons pas au C-phy.

Cette illustration présente les différentes "lane" du block MIPI :

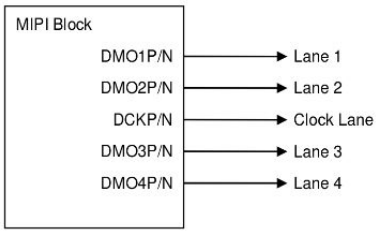

FIGURE 2.5 – Différents signaux du MIPI

L'imx6s Openrex supporte le mode M-phy 2 lanes. Les deux broches de données et la broche de la clock seront respectivement les broches (DMO1P/N, DMO2P/N, DCKP/N) du port CSI-2.

2.6 Carte de développement OpenRex

L'Openrex est une carte de développement conçue par Fedevel et produite par Voipac. Elle intègre un « système on chip » IMX6 disposant de un ou quatre coeurs suivant les versions. La schématique ainsi que le routage de ces deux cartes sont opensources et disponible sur le site de http://www.imx6rex.com/open-rex/.

Notre objectif est de porter le driver imx219 de la Raspberry sur l'IMX6S Openrex. Cette dernière possède un port CSI-2 identique au port de la Raspberry Pis. De plus, ce port peut accueillir des niveaux de tensions LVDS, nous avons donc vérifié qu'elle était configurée sur le port CSI-2 comme indiqué par la figure X. Le routage du connecteur correspond donc à la figure XX ? On y retrouve les couples de pistes lane D0, lane D1 et la CLK0, les deux broches de communication I2C SCL et SDA, les deux GPIO.

FIGURE 2.6 – Carte OpenRex Basic

2.7 Video for Linux 2 (V4L2)

V4L2 est la seconde version de l'API V4L. Elle est utilisée par des périphériques vidéo (caméras, écrans) contenue dans l'espace utilisateur d'un système linux. Elle est aussi faite de composants audio contrôlés par l'API Alsa. Elle permet de manipuler une très grande variété de périphériques en utilisant les mêmes fonctions. En général, il s'agit de périphériques I2C, mais si ce n'est pas le cas, une structure (v4l2_subdev) a été créée pour fournir au driver une interface en adéquation avec celle des sous-périphériques.

Principe d'usage :

- Ouvrir le périphérique
- Modifier les propriétés de l'appareil (résolution, luminosité, ...)
- Sélectionner le format de donnée
- Récevoir / Envoyer les données
- Ferme le périphérique

Les drivers V4L2 sont implémentés comme des modules kernels, ils sont chargés automatiquement ou manuellement (suivant les drivers) lors de la détection du périphérique par le système. Ils s'exécutent dans le kernel-space c'est-à-dire sous le kernel.

La commande ci-dessous est une surcouche de la commande insmod. insmod charge simplement un module, tandis que modprobe charge ses modules dépendants également.

```
root@poky : # modprobe <driver_name>
```

Suite au chargement du module, le gestionnaire de périphériques "udev" va créer un fichier dans le répertoire /dev, permettant ensuite l'accès aux périphériques. Selon le choix au développement, une arborescence de contrôle plus poussée peut apparaître dans /sys.

2.7.1 Package V4L-utils

Aujourd'hui le package v4l-utils implémente V4L2 dans une distribution , V4L devenant obsolète.

2.7.2 Principales commandes de contrôle

v4l2-ctl : outil de contrôle utilisé en ligne de commande

v4l2-dbg : outil permettant l'accés aux registres des périphériques v4l2

q4l2 : interface graphique v4l2 utilisant les commandes de contrôle de l'API

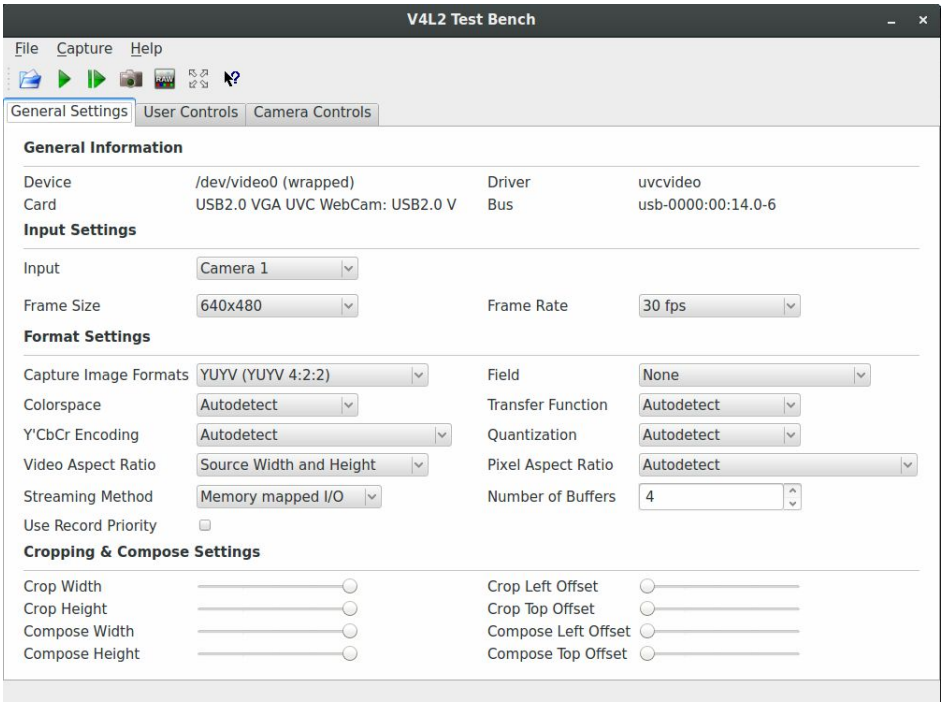

FIGURE 2.7 – Application graphique de V4L2

L'API V4L2 met à disposition le package "v4l-utils". Ce package fournit une interface graphique et des commandes de contrôle permettant la configuration d'un périphérique.

Comme il est illustré sur la figure 2.7, cette interface permet la configuration de plusieurs paramètres disponibles sur une majorité des périphériques vidéo.

Paramètres généraux : sélection du périphérique d'entrée, configuration du format des données et dimensionnement de l'affichage.

Utilisateur : Configuration du rendu vidéo

Caméra : Configuration de l'exposition lumineuse

Pour manipuler l'ensemble de ces paramètres, les différentes structures de v4l2 doivent être correctement instanciées dans le driver par les fonctions C, configurant la caméra.

Structures V4L2

v4l2_subdev

```
1  struct v4l2_subdev
2  {
3      #if defined(CONFIG_MEDIA_CONTROLLER)
4      struct media_entity entity;
5      #endif
6      struct list_head list;
```

```
7         struct module * owner;
8         bool owner_v4l2_dev;
9         u32 flags;
10        struct v4l2_device * v4l2_dev;
11        const struct v4l2_subdev_ops * ops;
12        const struct v4l2_subdev_internal_ops * internal_ops;
13        struct v4l2_ctrl_handler * ctrl_handler;
14        char name[V4L2_SUBDEV_NAME_SIZE];
15        u32 grp_id;
16        void * dev_priv;
17        void * host_priv;
18        struct video_device * devnode;
19        struct device * dev;
20        struct device_node * of_node;
21        struct list_head async_list;
22        struct v4l2_async_subdev * asd;
23        struct v4l2_async_notifier * notifier;
24        struct v4l2_subdev_platform_data * pdata;
25    };
```

Cette structure permet de gérer le multiplexage audio et vidéo de sous-périphériques (capteurs et contrôleurs de caméra).

i2c_client

```
1     struct i2c_client
2     {
3         unsigned short flags;
4         unsigned short addr;
5         char name[I2C_NAME_SIZE];
6         struct i2c_adapter * adapter;
7         struct device dev;
8         int irq;
9         struct list_head detected;
10        #if IS_ENABLED(CONFIG_I2C_SLAVE)
11        i2c_slave_cb_t slave_cb;
12        #endif
13    };
```

Cette structure donne accès au bus i2c pour établir la communication et interagir avec le périphérique. Pour protéger en écriture cette configuration, il est conseillé d'utiliser la fonction v4l2_set_subdevdata(). Elle permet de stocker le pointeur de cette structure dans les données privées de v4l2_subdev.

Initialisation de v4l2_subdev

Déclaration des fonctions d'initialisation dans la structure v4l2_subdev_core_ops

```
1     static struct v4l2_subdev_core_ops imx219_subdev_core_ops =
2     {
3     .s_power = imx219_s_power,
4     };
```

Implémentation des fonctions permettant l'initialisation des paramètres du flux vidéo :

```
static struct v4l2_subdev_video_ops imx219_subdev_video_ops =
{
.s_stream = imx219_s_stream,
.cropcap = imx219_cropcap,
.g_crop = imx219_g_crop,
.s_crop = imx219_s_crop,
.enum_mbus_fmt = imx219_enum_mbus_fmt,
.g_mbus_fmt = imx219_g_mbus_fmt,
.try_mbus_fmt = imx219_try_mbus_fmt,
.s_mbus_fmt = imx219_s_mbus_fmt,
.g_mbus_config = imx219_g_mbus_config,
};
```

On crée donc la structure du périphérique (imx219.c) :

```
struct <chipname>_state
{
    struct v4l2_subdev sd;
};
```

Cette structure doit contenir la structure v4l2_subdev pour donner un accés direct à la configuration du sous-périphérique (interface i2c).

Initialiser le sous-périphérique i2c :

```
v4l2_i2c_subdev_init(&state->sd, client, subdev_ops);
```

Faire le lien entre la structure i2c et v4l2_subdev :

```
struct i2c_client *client = v4l2_get_subdevdata(sd);
struct v4l2_subdev *sd = i2c_get_clientdata(client);
```

Instancier la structure pour ajouter la configuration du périphérique au kernel :

```
struct v4l2_subdev * v4l2_i2c_new_subdev(struct v4l2_device *
   v4l2_dev, struct
i2c_adapter * adapter, const char * client_type, u8 addr,
   const unsigned short *
probe_addrs);
```

Charge la configuration du flux vidéo du port CSI-2 définit par l'utilisateur.

```
v4l2_subdev_video_ops->s_stream()
```

Mise sous tension du port CSI-2 :

```
v4l2_subdev_core_ops->s_power()
```

2.8 GStreamer

GStreamer est un framework multimédia développé en C et porté sur plusieurs autres systèmes d'exploitation que GNU/Linux comme Android, OS X, iOS ou encore Windows. Ce projet débuta en Juin 1999 et fut implémenté dans l'environnement bureautique GNOME en Juillet 2012.

GStreamer utilise principalement des tubes (pipeline) inter-connectés ainsi le type d'un flux traversant un tube est connu des autres de plus ce framework est capable de gérer des fichiers audio et vidéo (capture, encodage, streaming, écoute, affichage).

GStreamer est basé sur des plugins qui améliorent son développement et ajoute des fonctionnalités comme l'encodage/décodage géré par le plugin FFMPEG.

GStreamer propose une fonctionnalité de streaming en local ou par un réseau, cette dernière passe par les protocoles UDP et TCP de la couche IP.

Le principe de ce framework repose sur l'association d'éléments reliés par des pipelines cependant l'entrée d'un élément doit être compatible avec la sortie de l'élément précédent. L'ordre des éléments est donc très important lorsqu'on écrit la commande, voici un schéma expliquant l'ordre des éléments :

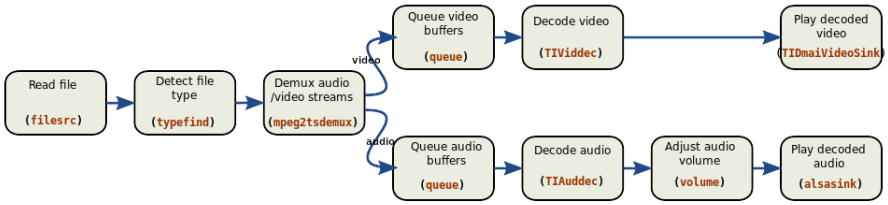

FIGURE 2.8 – Étapes de GStreamer

La première étape indique la source de la commande GStreamer. Cela peut être un fichier ou bien une caméra par le biais de V4L2. Ensuite, le système détecte le type du fichier source et différencie un fichier audio d'un fichier vidéo. Nous nous intéresserons plus sur la partie vidéo qu'audio puisque l'objectif du projet est la capture du flux vidéo, pour cela nous allons expliquer les commandes que nous utilisions :

```
user@poky : $ gst-launch-1.0 v4l2src ! 'image/jpeg,width=1280,height=720, framerate=30/1' ! imxvpudec ! imxipuvideotransform ! imxeglvivsink sync=false
```

Si l'on souhaite visualiser le flux vidéo d'un appareil V4L2 avec la synchronisation désactivé en 30 FPS et d'une résolution de 1280x720 pixels avec un format d'image JPEG, cette commande est adaptée. De plus celle-ci utilise une synchronisation EGL ainsi qu'un décodage V4L2.

```
user@poky : $ gst-launch-1.0 v4l2src ! 'image/jpeg,width=1280,height=720, framerate=30/1' ! v4l2sink sync=false
```

Cette commande permet d'afficher le flux vidéo de V4L2 en 30 FPS avec une taille de 1280x720 pixels. Si l'on souhaite juste faire une capture photo, l'image sera au format JPEG. V4L2sink est utilisé pour afficher le flux vidéo de l'appareil V4L2 en désactivant la synchronisation.

```
user@poky : $ gst-launch-1.0 v4l2src device=/dev/video0 ! 'image/jpeg,
width=1280,height=720, framerate=30/1' ! v4l2sink sync=false
```

Cette commande est la même que la première, la seule différence vient de la source. Celle-ci prend l'appareil vidéo numéro 0 en entrée qui correspond à la caméra.

Chapitre 3

Travail réalisé

3.1 Génération d'une méta-donnée

Pour réaliser une métadonnée nous avons lancé la commande suivante :

```
user@poky : $ yocto-layer create-layer your_layer_name
```

Elle nous assiste lors de la génération d'une grande partie de l'architecture standardisée commune aux metas.

Nous avons ensuite comparé notre meta à la meta-skeleton, le squelette de base permettant de doter notre système d'exploitation d'un programme helloworld.

Nous aurions préféré partir de la commande ci-dessous seulement celle-ci n'est accessible avant la version 2.4 du projet.

```
user@poky : $ bitbake-layer create-layer your_layer_name
```

Nous rajoutons cette meta à la liste des objets à compiler par la commande suivante.

```
user@poky : $ bitbake-layer add-layer your_layer_name
```

Après utilisation voici à quoi la meta ressemble depuis l'interface de développement atom.

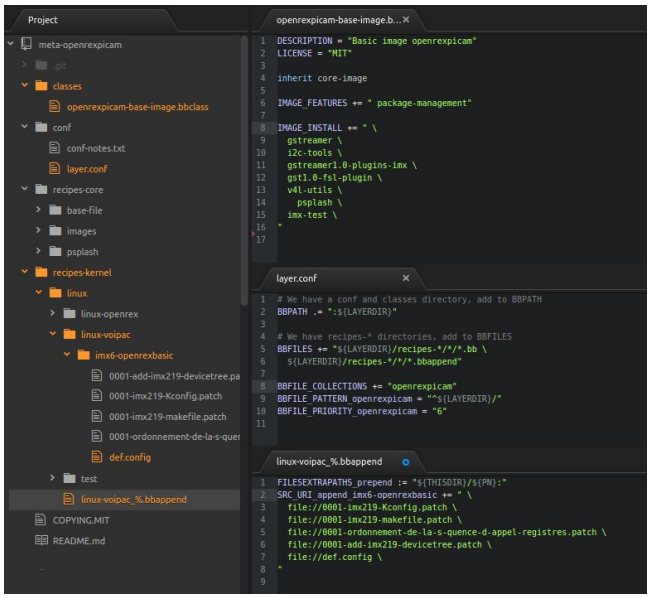

FIGURE 3.1 – Architecture d'une méta-donnée

Au début de la création de notre métadonnée nous souhaitions pouvoir reconstituer notre espace de travail aisément et cela est possible grâce aux deux dépôts fsl-community-bsp-platform et fsl-community-bsp-base de Freescale. Cependant il est nécessaire de les modifier afin qu'ils correspondent à aux fichiers nécessaires à notre meta-donnée.

Le dépôt fsl-community-bsp-platform contient un fichier "default.xml" que l'on appelle un fichier manifest, celui-ci permet de définir tous les dépôts contenant les fichiers et dossiers nécessaires à la création de notre image Yocto.

Le second dépôt fsl-community-bsp-data créera notre fichier "bblayers.conf" avec toutes les métadonnées nécessaires à la compilation ainsi que le fichier "setup-environment" qui concevra notre fichier local.conf.

3.1.1 fsl-community-bsp-platform

Nous avons repris le "default.xml" utilisé par Frescale incluant un kernel 4.1 (sur ce lien)

Cela a permis de connaître les dépôts ainsi que leurs branches afin de réaliser notre propre fichier manifest. Nous avons rajouté les liens vers notre méta-donnée ainsi que notre dépôt fsl-community-bsp-data grâce à ces deux lignes :

```
1     <remote fetch="git://github.com/petit-romain" name="romain"/>
2     <remote fetch="git://github.com/Alanaitali" name="equipe"/>
```

On a donné l'ordre de récupérer le fichier "setup-environment" permettant de sourcer notre environnement de travail en le plaçant dans le dossier sources/base :

```
1     <project remote="romain" revision="master"
          name="fsl-community-bsp-base" path="sources/base">
2         <copyfile dest="setup-environment"
              src="setup-environment"/>
3     </project>
```

Enfin on précise la branche sur laquelle nous développons notre meta ainsi que son emplacement dans notre dossier de travail (sources/meta-openrexpicam) :

```
1     <project remote="equipe" revision="develop/porting"
          path="sources/meta-openrexpicam"/>
```

3.1.2 fsl-community-bsp-data

Ce dépôt recopiera automatiquement notre fichier "bblayers.conf" puisqu'il contient le contenu du fichier :

```
1     BBLAYERS = "␣\
2 ␣␣␣␣${BSPDIR}/sources/poky/meta␣\
3 ␣␣␣␣${BSPDIR}/sources/poky/meta-Yocto␣\
4 ␣␣␣␣\
```

```
5  ␣␣␣␣${BSPDIR}/sources/meta-openembedded/meta-oe␣\
6  ␣␣␣␣${BSPDIR}/sources/meta-openembedded/meta-multimedia␣\
7  ␣␣␣␣\
8  ␣␣␣␣${BSPDIR}/sources/meta-fsl-arm␣\
9  ␣␣␣␣${BSPDIR}/sources/meta-fsl-arm-extra␣\
10 ␣␣␣␣${BSPDIR}/sources/meta-fsl-demos␣\
11 ␣␣␣␣\
12 ␣␣␣␣${BSPDIR}/sources/meta-openrexpicam␣\
13 ␣␣␣␣"
14     BBLAYERS += "${BSPDIR}/sources/meta-fsl-arm-voipac"
```

Pour le fichier "setup-environment", nous n'avions rien à modifier cependant nous avons ajouté une signature ainsi que les images compilables de notre méta-donnée :

```
1     Welcome to our project !
2     You can now run 'bitbake <target>'
3     Common targets are :
4         - openrexpicam-base-image
5     Signed-off-by: Alan Ait-Ali, Romain Petit, Clément Ailloud &
          Martin Laporte
```

3.2 Génération d'un OS

3.2.1 Préparation de l'environnement

Grâce à l'environnement de travail Yocto nous avons pu rapidement mettre en place un système d'exploitation fonctionnel sur la cible Openrex. En effet les équipes de Voipac et Fedevel ont mis à disposition le support de la carte (BSP) afin que la distribution GNU/Linux maintenue par Freescale pour les processeurs imx6 fonctionne sur les Openrex. L'ensemble de sources fsl-community-bsp supporte donc la carte Openrex-imx6q. Cette officialité permet, une fois les sources correctement téléchargées et ordonnées, de compiler une image par la commande :

```
MACHINE=imx6s-openrex bitbake core-image-base
```

3.2.2 Préparation de notre meta-donnée

Le projet Yocto permet aussi d'améliorer l'image générée. Cette amélioration suit le principe des codes ouverts à l'amélioration et fermés aux modifications. En effet comme représenté sur la figure 3.2, lors de la compilation des recettes (100 Mo) nécessaires à la core-image-base, le programme bitbake a puisé dans des sources distantes pour composer un dossier de sources à compiler. Grâce à celles-ci bitbake forme un répertoire général des sources du noyau (18Go), bitbake compile ensuite la core-image-base et la place dans :

$BUILDDIR/tmp/deploy/images/NOM_DE_L'IMAGE :imx6-openrexbasic

Mais pour modifier cette image il n'y a pas besoin de toucher aux recettes des meta-données initiales (100Mo sur la figure 3.2). Pour modifier l'image on intervient d'abord au niveau du dossier des sources (18Go) on apporte nos améliorations. De ces modifications on réalise un patch différenciant l'état d'origine et l'état actuel

des fichiers voulus. Enfin on vient placer ce patch dans notre propre meta (1Mo) à côté des meta-données originales. En visionnant cette meta et les patchs qui s'y trouvent on versionne donc l'ensemble du projet depuis l'état d'origine fixé par la communauté freescale.

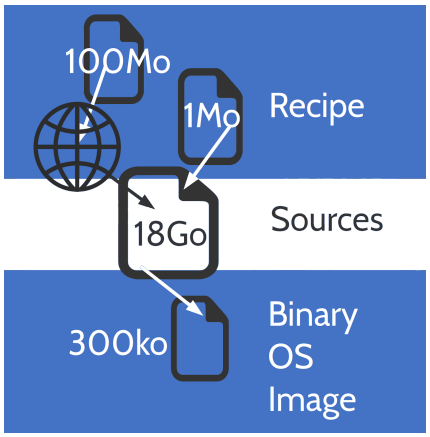

FIGURE 3.2 – Architecture d'une recette

3.3 Implémentation de code dans le BSP

Le coeur du projet vise à ajouter du code dans le bsp déjà existant, notamment au sein du kernel. Pour ce faire nous modifions le dossier de compilation (18Gb ci-dessus). Comme son nom l'indique, /git/ est versionné par git et nous permet de tirer un patch de nos modifications.

Ensuite, on applique des patchs modifiant les sources du kernel provenant du git de voipac ou fedevel. le patch généré est inclu aux sources de notre recette linux-voipac_%.bbappend. La recette s'ajoutera à la fin de la recette linux-voipac lors de sa prise en compte par bitbake.

La recette linux-voipac_%.bbappend a pour rôle de surcharger la recette se trouvant dans la meta-fsl-arm-voipac. Elle porte le même nom que la recette a surcharger suivi d'un _% qui permet de s'affranchir de la version.

Dans l'extrait ci-dessous on peut voir que la recette utilise les sources du git voipac.

3.3.1 Extrait de linux-voipac-4.1.bb

```
1 SRCBRANCH = "4.1-2.0.x-imx-rex"
2 LOCALVERSION = "-Yocto"
3 SRCREV = "ab5923c9613a97ede4da92a933842e771283d463"
4 KERNEL_SRC ?= "git://github.com/voipac/linux-fslc.git;protocol=git"
5 SRC_URI = "${KERNEL_SRC};branch=${SRCBRANCH}␣file://defconfig"
```

Notre recette permet d'ajouter les fichiers se trouvant dans imx6-openrexbasic et inscrit dans la recette comme on le voit ci-dessous.

3.3.2 Extrait de linux-voipac_ %.bbappend

```
1     FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"
2     SRC_URI_append_imx6-openrexbasic += "␣\
3 ␣␣␣␣␣␣␣␣file://0001-imx219.patch␣\
4 ␣␣␣␣␣␣␣␣file://defconfig␣\
5 ␣␣␣␣"
```

Quand nous modifions le code du dossier de compilation, le patch se réfère à l'état dans lequel la communauté des bsp freescale l'a laissé. Or dans le fichier $BUILDDIR/conf/bblayer.conf notre meta-openrexpicam est placée directement en suivant des meta nécessaires pour atteindre l'état laissé par freescale. Nous sommes donc sûrs que le patch généré est cohérent avec l'arborescence de compilation sur laquelle bitbake l'appliquera.

3.4 Implémentation des supports de compilation

Ajouter une nouvelle caméra a notre BSP, demande la création d'un driver. Pour qu'il soit utilisable il nous faut signaler au compilateur que nous voulons ajouter à notre kernel le support pour la caméra. Pour paramétrer la compilation on passe par l'outil menuconfig qui permet 3 options :

- compilation du driver et chargement en module
- compilation du driver et chargement en statique
- pas de compilation du driver

Les fichiers qui permettent de paramétrer le menuconfig sont situés au même endroit que les fichiers source.c des pilotes sous le nom de Kconfig.

L'ajout de notre caméra dans menuconfig se fait par le code suivant :

```
1     config MXC_CAMERA_IMX219_MIPI
2     tristate "Sony␣imx219␣camera␣support␣using␣mipi␣(raspicam␣v2)"
3     depends on !VIDEO_MXC_EMMA_CAMERA && I2C
```

depends on : Définit les modules à activer pour que l'option soit visible

tristate : Définit l'affichage dans le menuconfi

Config : Définit le nom de la variable de compilation

Sur la figure ci-dessus on peut remarquer un M juste devant les drivers, cela signifie qu'ils sont compilés en module. Nous avons choisi cette option afin de pouvoir décharger (rmmod) et recharger (modprobe) le driver sans devoir redémarrer le kernel. Le résultat du menuconfig est un fichier texte .config ($BUILDDIR/tmp/work/imx6_openrexbasic-poky-linux-gnueabi/linux-voipac/4.1-r0/build/ex emple.config) ou généralement defconfig. Dans l'extrait du defconfig ci-dessous on peut observer que l'ajout et le chargement en module du pilote a été pris en compte.

```
1     CONFIG_VIDEO_MXC_IPU_CAMERA=y
2     CONFIG_MXC_CAMERA_OV5640=m
3     CONFIG_MXC_CAMERA_OV5642=m
```

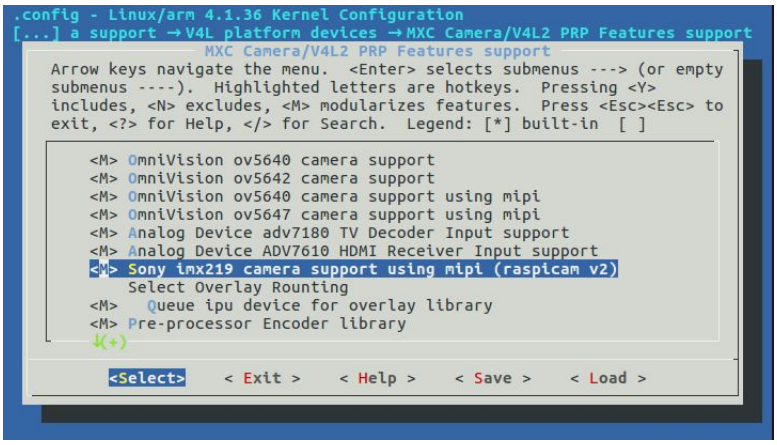

FIGURE 3.3 – Driver dans le menuconfig

```
4   CONFIG_MXC_CAMERA_OV5640_MIPI=m
5   CONFIG_MXC_CAMERA_OV5647_MIPI_INT=m
6   CONFIG_MXC_TVIN_ADV7180=m
7   CONFIG_MXC_TVIN_ADV7610=m
8   CONFIG_MXC_CAMERA_IMX219_MIPI=m
9   CONFIG_MXC_IPU_DEVICE_QUEUE_SDC=m
10  CONFIG_MXC_IPU_PRP_ENC=m
11  CONFIG_MXC_IPU_CSI_ENC=m
```

- CONFIG_MXC_CAMERA_IMX219_MIPI : Objet du kernel
- m : Ordre de compilation et d'installation en module

Le menuconfig permet de configurer la compilation, mais il n'est pas directement lié à celle-ci, c'est le Makefile qui se charge de relier les informations contenues dans le .config au compilateur. Comme le Kconfig, le Makefile se trouve dans le répertoire des sources : $BUILDDIR/tmp/work/imx6_openrexbasic-poky-linux-gnueabi/recette/4.1-r0/git/drivers/media/Selon le driver

Selon le driver, ses sources peuvent prolonger ce chemin vers "platform/mxc/-capture/" ou "/i2c/.". Le nom de la recette, lui dépends du kernel compilé. La recette "linux-openrex" est utilisée pour les kernels v3 et "linux-voipac" pour les kernels v4.

```
1   +imx219_camera_mipi-objs := imx219_mipi.o
2   +obj-$(CONFIG_MXC_CAMERA_IMX219_MIPI) += imx219_camera_mipi.o
```

- imx219_camera_mi-objs : nom du fichier .c à compiler
- CONFIG_MXC_CAMERA_IMX219_MIPI : variable de compilation fourni par le .config
- imx219_camera_mipi.o nom du driver compilé

Au lancement de bitbake le driver pourra alors être compilé et chargé par le kernel.

3.5 Utilsation de drivers existants

3.5.1 Présentation d'un driver

Un driver Linux est un programme binaire qui s'exécute dans le kernel-space. Un driver utilise les ABI kernel pour interagir avec son environnement. Le code d'un driver s'appuie donc sur les API kernel. Dans le cas du système d'exploitation linux, celles-ci sont rédigées en langage c. C'est pourquoi il est nécessaire que le compilateur du driver compile parfaitement le langage c. Dans notre cas comme dans la majorité, le driver sera écrit en c. Une particularité des codes de driver provient de l'absence de main(), celui-ci est remplacé par des fonctions init() et exit(). Init() permet le chargement du driver dans le kernel. Si le driver est compilé en statique, il est chargé (linked and locked) au démarrage et exit() est exécuté lors de l'extinction du système. Si le driver est compilé en module ce sont les fonctions insmod et rmmod qui appelleront les init et exit du fichier ".ko"

3.5.2 Utilisation de l'existant

Dans un premier temps, n'étant pas habitués à manipuler des drivers, nous avons cherché à utiliser des sources en croisant les architectures requises. Pour agir dans notre meta, nous avons eu recours à une série de patchs comme montrés ci-dessous.

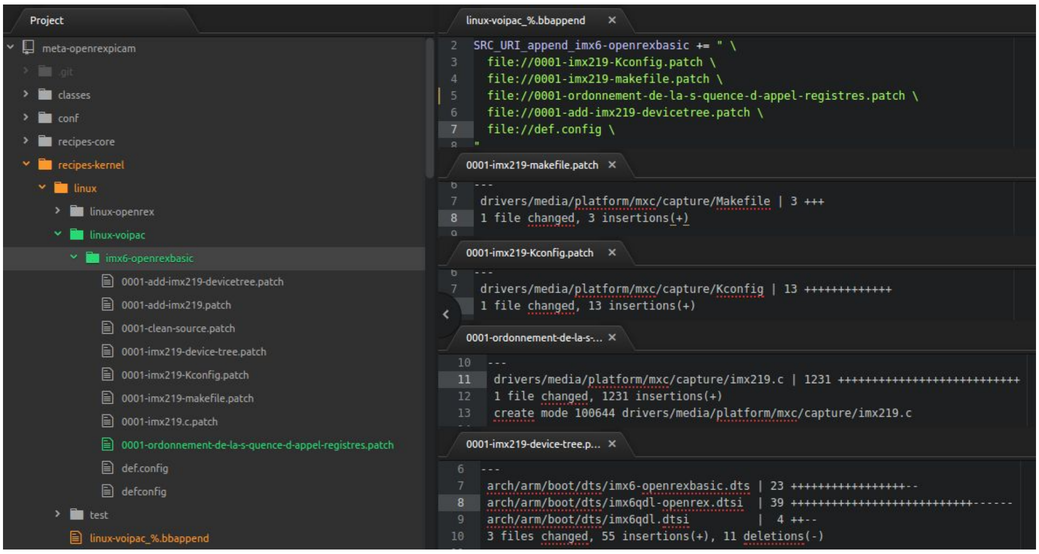

FIGURE 3.4 – Patchs du kernel 3.14

Compilation par SDK

Dans un premier temps, nous nous doutions qu'il y aurait des erreurs de compilation. Nous avons préféré décorréler les erreurs de compilation de nos propres erreurs. Nous avons alors essayé de compiler ces fichiers, en dehors de la meta depuis la cross-toolchain générée par Yocto. Nous utilisons la variable $CC sélectionnant le cross compilateur, cette variable est affectée lorsque l'on "source" le sdk généré par :

```
#génération du sdk bitbake openrexpicam-base-image -c populate_sdk #environement du sdk source /opt/poky/2.0.3/environment-setup-cortexa9hf-vfp-neon-poky-linux-gnueabi #aperçu de CC CC=arm-poky-linux-gnueabi-gcc -march=armv7-a -marm -mthumb-interwork -mfloat-abi=hard -mfpu=neon -mtune=cortex-a9 –sysroot=/opt/poky/2.0.3/sysroots/cortexa9hf-vfp-neon-poky-linux-gnueabi #première compilation « out-of-tree » $CC imx219.c
```

Cette compilation fait appel à des bibliothèques contenues dans les arborescences de compilation de différentes recettes.

```
BUILDDIR=/home/diag/workspaceThales/YOCTO/fsl-community-bsp/build-imx6rex.com IMX6S_DIR=$BUILDDIR/tmp/work/imx6s_openrex-poky-linux-gnueabi #libs linux/*.h DIRLINUX=$IMX6S_DIR/linux-openrex/3.14-r0/git/include DIRLINUX2=$IMX6S_DIR/u-boot-openrex/v2015.10+gitAUTOINC+7d8ddd7de7-r0/git/include DIRLINUX3=$IMX6S_DIR/core-image-minimal/1.0-r0/sdk/image/opt/poky/2.0.3/sysroots/cortexa9hf-vfp-neon -poky-linux-gnueabi/usr/include #libs asm/*.h DIRASM=$BUILDDIR/tmp/work/x86_64-nativesdk-pokysdk-linux/nativesdk-linux-libc-headers/4.1-r0/linux-4.1 /arch/arm/include/ DIRASM2=$IMX6S_DIR/linux-openrex/3.14-r0/build/arch/arm/include/generated DIRASM3=$IMX6S_DIR/u-boot-openrex/v2015.10+gitAUTOINC+7d8ddd7de7-r0/git/arch/arm/include/ #nouvelle commande de compilation « out of tree » $CC imx219.c -I$DIRLINUX -I$DIRASM -I$DIRASM2 -I$DIRLINUX2 -I$DIRLINUX3 -I$DIRASM3
```

Nous avons inclus 6 dossiers de bibliothèques en option puis les nouvelles dépendances étaient inexistantes dans le dossier contenant l'arborescence de compilation. Donc nous nous sommes mis à chercher une autre solution. Après avoir parlé à notre professeur de Linux embarqué des soucis de compilation que nous rencontrions nous nous sommes mis à compiler le driver en ajoutant ses sources dans l'arborescence (in-tree).

Compatilation out-of-tree

Pour cette première compilation de code source dans Yocto nous avons tout d'abord rédigé une recette comme on aurait rédigé un makefile. Un intérêt est de pouvoir compiler nos sources sans avoir à recompiler le kernel entier. Seulement cette méthode ne résout pas le problème de gestion des dépendances. En effet même si Yocto compile avec la même configuration le kernel et ce module, Yocto interprètera les recettes comme deux compilations différentes et les exécutera dans des répertoires dissociés. les includes du module seront incapables de trouver plus automatiquement que de manière out-of-tree leurs bibliothèques kernel (linux/example.h).

Compatilation in-tree

La compilation de ces sources a été effectuée in-tree sur la version v3.14 du kernel linux-openrex (branche develop/Smart). La compilation ayant fonctionné, elle a été testée et déclarée image non bootable. Comme à ce moment là les sources de type chromium os étaient plus avancées, cette piste fut abandonnée.

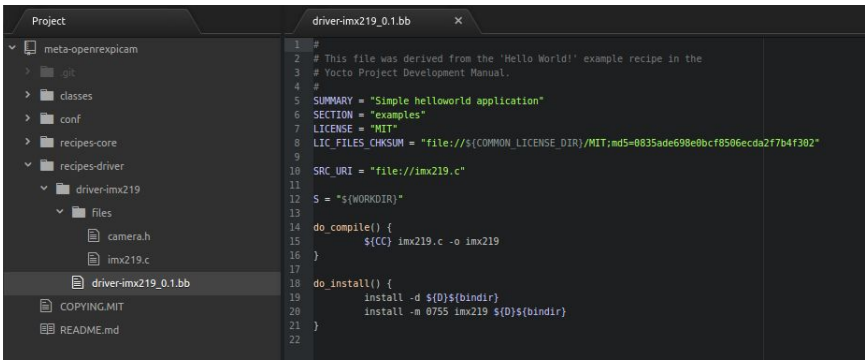

FIGURE 3.5 – Arborescence de la compilation out-of-tree

3.5.3 imx219 - Hummingboard

Origine des sources

Comme nous n'avions pas encore manipulé un driver par son code source nous sommes rapidement allés demander des conseils aux professeurs renseignés. Principalement M P.-J. Texier qui nous gardera à l'oeil le long de ce projet. Dans un mail à notre équipe M Texier nous a proposé de s'inspirer du driver disponible à l'adresse suivante. Il s'agit du git "Russell King's ARM Linux kernel tree" mais surtout d'un patch qui ajoute le driver d'imx219 et configure le makefile et kconfig pour sa compilation. L'intérêt principal étant la plateforme cible, une "hummingboard" portant un soc imx6dl, différent mais proche de notre imx6s. De plus sur ce même kernel tree, on peut trouver les fichiers device-tree correspondants au driver pour la configuration en question (source, dts).

Lors des rapports intermédiaires présentés à l'équipe de Thales nous parlions de ces sources sous le nom de "solution 3"

Compilation in-tree

Sur les conseils de M Texier, nous n'avons pas cherché à réaliser de compilation out-of-tree. Nous avons donc directement porté le driver sur notre carte ainsi que les structures device-tree.

3.5.4 imx219 - Nvidia-Tegra Chromium-Os

Chromium OS est "un système d'exploitation visant à renforcer la sécurité des utilisateurs qui passent une majorité de leur temps sur le web". Concrètement cet os est un système GNU/Linux et correspond à la branche en développement libre de "Google Chrome os". Développé par par une filiale de Google, cet os est entre autre maintenu par Nvidia sur ses processeurs Tegra. Ceux-ci sont destinés à des applications dans le domaine des smartphones ; ces processeurs respectent très probablement à la lettre les contraintes MIPI.

Origine des sources

Le driver en question provient du git officiel de chromium os de nvidia pour les cartes tegra, sur le kernel 3.14. il est accessible à ce lien . Lors des rapports intermédiaires présentés à l'équipe de Thales nous parlions de ces sources sous le nom de "solution 2".

Compatibilité avec le device tree

Lorsque l'on essayait de charger le driver imx219, nous obtenions l'erreur suivante

```
imx219 1-0064 : IMX219 : missing platform data !
```

Après analyse du driver l'erreur provient de la fonction probe du driver. Cette erreur est survenue dû à une mauvaise compatibilité entre le driver imx219 et le gestionnaire I2C, pour cela nous avons ajouté une compatibilité avec le device-tree de cette façon :

```
1  static const struct of_device_id imx219_of_match [] =
2  {
3      { . compatible = "␣sony␣,␣imx219␣" , . data = 0 } ,
4      {}
5  };
6  MODULE_DEVICE_TABLE ( of , imx219_of_match ) ;
```

Ensuite nous avons créé notre propre structure I2C pour l'imx219 afin d'implémenter la compatibilité entre le driver et l'I2C de cette manière :

```
1  static struct i2c_driver imx219_i2c_driver =
2  {
3      . driver =
4          {
5              . name = "␣imx219␣" ,
6              . of_match_table = of_match_ptr(imx219_of_match) ,
7          },
8      . probe = imx219_probe ,
9      . remove = imx219_remove ,
10     . id_table = imx219_id ,
11 };
```

De plus nous avons enlevé la condition qui nous procurait l'erreur précédente afin de tester dans un premier temps si la compilation s'effectuait correctement :

```
1  if (! ssdd)
2  {
3      dev_err(& client->dev, "IMX219:missing␣platform␣data␣!\␣
           n");
4      return -EINVAL ;
5  }
```

Après avoir modifié le driver nous obtenons une nouvelle erreur :

```
imx219 1-0064 : Error -19 getting clock
i2c 1-0064 : Driver imx219 requests probe deferral
```

```
1     if (! ssdd)
2     {
3         dev_err(& client->dev, "IMX219:missing␣platform␣data␣!\␣
              n");
4         return -EINVAL ;
5     }
```

Nous avons repris la même stratégie que précédemment en enlevant la condition générant l'erreur :

```
1     if(IS_ERR(priv->clk))
2     {
3         dev_info(&client->dev,"Error␣%ld␣getting␣clock␣\␣n",
4         PTR_ERR(priv->clk));
5         return -EPROBE_DEFER ;
6     }
```

Enfin la capture d'écran ci-dessous permet de justifier que le module se charge bien au lancement de notre OS :

```
root@openrexpicam : # i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00 : - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
10 : 10 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
20 : - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
30 : - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
40 : 40 - - - - - - - - - - - - - - 48 - - - - - - - - - - - - - -
50 : uu - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
60 : - - - - - - - - - - uu - - - - - - - - - - - - - - - - - - - -
70 : - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
root@openrexpicam : # lsmod
Module                  Size            Used by
mxc_v4l2_capture        25109           0
ipu_bg_overlay_sdc      5242            1 mxc_v4l2_capture
ipu_still               2312            1 mxc_v4l2_capture
ipu_prp_end             5872            1 mxc_v4l2_capture
ipu_csi_enc             3743            1 mxc_v4l2_capture
v4l2_int_device         2913            2 ipu_csi_enc, mxc_v4l2_capture
ipu_fg_overlay_sdc      6068            1 mxc_v4l2_capture
imx219                  7716            1
mxc_dcic                6543            0
galcore                 225000          0
evbug                   1871            0
```

Une fois ceci effectué nous voulions essayer de corriger les instructions conditionnelles de sécurité générant les erreurs afin de pouvoir charger correctement le driver. Une fois le driver chargé, nous n'avons pas réussi à effectuer la liaison entre

v4l2 et le driver imx219. Par manque de temps et de connaissances sur l'environnement v4l2, nous nous sommes tous concentré sur la nouvelle solution proposé par Thalès qui leur semblait plus pertinente.

3.5.5 imx219 - Raspberry Pi v2

Origine des sources

En premier lieu nous sommes partis des sources reçues dans les premiers mails. Ce driver lie le processeur Allwinner A80 (ARM Cortex-A15/A7) du sbc Raspberry Pi à un composant pilote vidéo imx219. Ces deux fichiers Linux V4L2 Driver (imx219.c et camera.h) onts été rédigés par "Chomoly" et sont téléchargeables à cette adrese.

Lors des rapports intermédiaires présentés à l'équipe de Thales nous parlions de ces sources sous le nom de "solution 1".

Chronologiquement, ces fichiers sources ont été la première piste explorée c'est pourquoi leur compilation n'a pas été immédiatement réussie. Nous avons essayé d'obtenir un module binaire par trois manières. D'abord, une compilation grâce à un sdk yocto, puis une compilation out-of-tree, externe à la recette du kernel et nous sommes presque parvenus à nos fins avec une compilation in-tree. Ci-dessous, la description des pratiques à éviter pour le projet d'un driver.

Portage du BSP Voipac sur un kernel 4.14

Lors de nos recherches pour les compilations ci-dessous, nous avons trouvé un driver imx219 fonctionnel sur un kernel 4.14 donc l'idée était de porter notre méta-donnée vers un kernel 4.14 afin de réutiliser le driver sur cette version de kernel. Nous sommes partis de la même structure que les versions de kernel 3.14 et 4.1 qui contient un dossier avec le defconfig et les différents patchs nécessaires à la compilation ainsi qu'un fichier .bb.

```
user@poky $ : tree sources/meta-fsl-arm-voipac/recipes-kernel/linux
|__linux-voipac-3.14
|_____defconfig
|__linux-voipac_3.14.bb
|__linux-voipac-4.1
|_____defconfig
|__linux-voipac_4.1.bb
|__linux-voipac-4.14
|_____0001-imx6s-6q-add-initial-support.patch
|_____defconfig
|__linux-voipac_4.14.bb
```

Ensuite nous avons modifié la recette du kernel 4.1 dans une nouvelle recette pour un kernel 4.14 avec les modifications suivantes :

```
1     SRCBRANCH = "4.14.x+fslc"
2     LOCALVERSION = "-Yocto"
3     SRCREV = "${AUTOREV}"
4     KERNEL_SRC ?=
          "git://github.com/Freescale/linux-fslc.git;protocol=git"
```

Cependant lors de la phase de boot nous obtenons l'erreur suivante :

```
reading boot-imx6-openrexbasic.scr
** Unable to read file boot-imx6-openrexbasic.scr **
```

Nous pouvons conclure qu'il y a une incompatibilité entre le bootloader et le kernel que nous avons modifié. Pour que notre image puisse se lancer correctement, il est nécessaire de modifier le bootloader.

3.6 Développement d'un driver

Des caméras utilisant le MIPI CSI sont implémentées de base dans le kernel, l'OV5640 en est un bon exemple. Il s'agit d'un capteur vidéo développé par Omnivision qui se trouve être ressemblant à l'imx219 au niveau du protocole de communication avec la carte mère. Le pilote déjà implémenté servira alors de base pour la création d'un driver imx219.

3.6.1 Organisation d'un driver MIPI/CSI

On dénombre 3 sections principale dans ce genre de pilote : La première contient en majeure partie du code lié au fonctionnement du composant. On y retrouve des structures contenant les registres à configurer, les différents modes d'acquisitions d'image tel que la résolution, la gestion des régulateurs, les signaux de démarrage et d'extinction, et d'autres caractéristiques. Cette section étant spécifique au périphérique, c'est ici que seront modifiés un grand nombre de fonction. En effet les registres et les séquences d'initialisation sont différentes entre les deux composants, il est donc nécessaire de les adapter. La deuxième est constituée de fonctions de contrôle et d'initialisation propres à V4L2. Elles permettent d'enregistrer et d'utiliser l'imx219 en tant qu'appareil V4L2. La dernière partie est relative à la gestion de la communication i2c, les fonctions classiques d'un périphérique i2c sont présentes tel que ov5640_probe, ov5640_remove, ov5640_init et ov5640_clean sans oublier la structure reliant les fonctions au système ov5640_i2c_driver.

Intégration au device tree

Le device tree est un sous-ensemble composé de plusieurs fichiers configurant toutes les liaisons entre le matériel et le logiciel. Il permet d'utiliser des drivers conçus pour linux. On le retrouve dans la partition boot qui est accessible en lecture/écriture, cela permet de le modifier facilement. Nous pouvons donc tester les drivers de façon rapide et efficace.

La configuration du driver imx219 :

extrait de openrexbasic.dts

```
1     &i2c2 {
2 /* Raspberry Pi camera rev 2.1 */
3 camera: imx219_mipi@64{
4 compatible = "sony,imx219";reg = <0x64>;
5 clocks = <&clks IMX6QDL_CLK_DUMMY>;
6 clock-names = "csi_mclk";
7 DOVDD-supply = <&reg_1p8v>;
8 AVDD-supply = <&reg_2p8v>;
9 DVDD-supply = <&reg_1p5v>;
```

```
pwn-gpios = <&gpio7 6 GPIO_ACTIVE_HIGH>;
csi_id = <1>;
mclk = <24000000>;
mclk_source = <0>;
pinctrl-names = "default";
pinctrl-0 = <&pinctrl_imx219>;
};
```

extrait de imx6qdl-openrex.dtsi

```
    &mipi_csi
    {
        ipu_id = <0>;
        csi_id = <1>;
        v_channel = <1>;
        lanes = <2>;
        mipi_dphy_clk = <0x28>;
        status = "okay";
    };

    pinctrl_imx219: imx219_grp
    {
        fsl,pins = <MX6QDL_PAD_SD3_DAT2__GPIO7_IO06 0x00017059>;
    };
```

l'extrait ci-dessus représente la déclaration du driver imx219 dans le device tree.

- compatible = "sony,imx219" : correspond au nom associé dans la structure imx219_i2c_driver sous la variable .name
- reg =<0x64> : correspond à l'adresse i2c du périphérique
- clocks = <&clks IMX6QDL_CLK_DUMMY> : lien vers l'horloge à utiliser
- clock-names = "csi_mclk" : nom de l'horloge
- pwn-gpios = <&gpio76GPIO_ACTIVE_HIGH> : gpio qui contrôle l'allumage
- csi_id = <1> : identifiant vers la structure CSI
- mclk = <24000000> : fréquence de l'horloge
- mclk_source = <0> : source de l'horloge en accord avec la structure v4l2_cap_1
- pinctrl-0 = <&pinctrl_imx219> : lien vers la déclaration des gpios

3.6.2 Validation du driver

Au démarrage du driver, la première fonction exécutée est imx219_probe. Comme son nom l'indique, elle a pour rôle de sonder certaines parties du composant comme la broche de démarrage (pwn-gpio) , l'horloge (mclk), le csi_id etc... En d'autres termes, elle vérifie si les paramètres donnés par le device tree sont cohérents. Pour s'assurer de la bonne communication avec le composant, Sony a prévu un registre contenant l'ID de la caméra. Une des vérifications de débogage de la fonction probe est de lire ce registre et de tester sa valeur. Or lors du chargement du driver dans le kernel une erreur relative a l'ID apparaissait.

Afin de pouvoir relier les erreurs logicielles émises par le driver à des erreurs de manipulation de l'imx219 interprétables la datasheet, nous avons monitoré la communication I2C entre l'Openrex et l'imx219.

Lecture du bus I2C

Dans un premier temps, nous avons réalisé plusieurs lectures à l'oscilloscope, puis de manière bien plus efficace nous avons utilisé un analyseur logique relié à un PC. Grâce à ce montage, nous avons pu lire le bus i2c tout au long de la phase de boot, mais aussi, nous avons pu interpréter les trames logiciellement en hexadécimal via l'application "Saleae Logic".

FIGURE 3.6 – Trame I2C

Sur le bus I2C, on a pu dénombrer 6 périphériques adressés 0x14, 0x15, 0xC8, 0xC9, 0x90 et 0x91. Nous nous sommes intéressé principalement à 0xC8 et 0xC9 vraisemblablement le SOM caméra et son interlocuteur (Openrex). On trouve dans les communications qui lui sont adressées ce premier échange. dont nous ne sommes pas sûr de l'interprétation.

- Setup Read to [0xC9] + ACK
- 0x04 + NAK
- Setup Write to [0xC8] + ACK
- 0x00 + ACK
- 0x01 + ACK
- Setup Read to [0xC9] + NAK

Puis apparait le début de la structure de communication écrite dans le driver. Elle permet de déverrouiller les configurations du fabricant mais n'est pas émise dans sa totalité et s'interrompt au 3ème registre.

- Setup Write to [0xC8] + ACK
- 0x30 + ACK
- 0xEB + ACK
- 0x05 + ACK
- Setup Write to [0xC8] + ACK
- 0x30 + ACK
- 0xEB + ACK
- 0x0C + ACK
- Setup Write to [0xC8] + ACK
- 0x30 + ACK
- 0x0A + ACK
- 0xFF + ACK
- Setup Write to [0xC8] + NAK

Chapitre 4

Conclusion

4.1 Bilan technique

Mise en place d'un environnement de compilation	Terminé
Développement d'un OS bootable sur l'OpenRex	Terminé
Driver imx219	En cours

TABLE 4.1 – Conclusion du projet

Au cours du projet, nous somme tombés sur plusieurs impasses qui nous ont permis d'apprendre les différentes façons de compiler un driver avec Yocto.

De part notre faible connaissance en driver Linux, nous ne pensions pas avoir les capacités techniques requises pour développer nous-même un driver. Notre travail c'est donc orienté vers trois drivers imx219 existants. Le BSP Openrex étant compatible avec le kernel 3,14 et 4,1 nous étions limités en ressources. Deux des drivers n'était pas compatible avec notre kernel, nous avons alors cherché à déterminer quelles bibliothèques étaient responsable de cette incompatibilité. Malheureusement, les versions étaient trop éloignées pour imaginer patcher toutes les bibliothèques utiles au fonctionnement des drivers.

Une dernière solution était de rendre compatible le driver avec notre kernel, après l'avoir rendu compatible avec notre device tree nous avions un segmentation fault lors du chargement du kernel. Simultanément nous portions le bsp de l'openrex sur un kernel 4.14 qui n'a pas pu être testé suite à une erreur survenue avec le bootloader.

Face à ces multiples échecs et un échange avec le client, nous commencions à rédiger notre propre driver en se basant sur ceux déjà inclus dans le kernel. Le driver est maintenant compilé et configuré par le device tree cependant il nous est impossible de lire ou d'écrire dans un registre de la carte. Notre travaille s'achève donc sur ce point.

Techniquement nous avons acquis un bagage de connaissances concernant l'usage des couches applicatives v4l2 nécessaire à la capture d'images de l'environnement Yocto. À l'issue de ce rapport, nous pouvons nous concentrer sur le développement du code en langage C.

4.2 Bilan de suivi de projet

Dès le commencement du projet, nous somme partis en méthode agile, notre groupe de travail a su s'auto-organiser et a perduré jusqu'à la fin du temps imparti.

Nous avons rapidement et facilement réussi à répartir le travail en fonction des compétences de chacun, des obstacles matériels et logistiques rencontrés.

Malgré les différences de niveaux initials dûes au passif technologique de chacun, chaque individu à apporter son utilité. En revanche, si le côté, communication et adaptabilité de la méthode agile est respecté, le lien avec le client quant à lui a été négligé.

C'est en partie dû à la séparation physique du scrum-master et du groupe puis au manque d'outils mise en place pour faciliter ce rapprochement. Une communication plus efficace avec l'équipe de Thales nous aurait évité par exemple de prolonger trop longtemps la piste des drivers existants.

4.3 Conclusion

Nous n'avons pas pu répondre complètement à la demande de Thales, qui est actuellement entrain de développer le driver avec des résultats encouragent. Face au obstacle notre groupe a toujours cherché à progresser en allant de plus en plus loin dans le raisonnement technique. Bien quinachevé, cette expérience reste une des plus enrichissantes de notre année. Étant soucieux d'apporter notre pierre à l'édifice nous laissons avec ce rapport un environnement de développement Yocto optimisé pour compiler un OS compatible avec l'Openrex et un guide d'utilisation et de développement en annexe.

YNOV

PROJET DE MASTER EN PARTENARIAT AVEC THALES

Guide d'utilisation et de développement de distribution GNU/Linux avec Yocto

---

Auteur :
Alan AIT-ALI
Martin LAPORTE
Clément AILLOUD
Romain PETIT

Superviseurs :
Patrick PIQUART
David COUÉ

Rapport final présentant le travail effectué sur l'intégration du driver IMX219 sur une OpenRex Basic

au sein du

Département Aéronautique & Systèmes Embarqués

22 février 2018

Table des matières

4.4 Procédure initiale

Cette partie sera une explication des différentes commandes ainsi qu'une description de la méthodologie choisie pour travailler avec Yocto.

4.4.1 Gérer son espace mémoire

Tout d'abord sachons que le manuel Yocto conseille de réserver 50 Go afin de pouvoir travailler dans de bonnes conditions. Dans notre cas nous avons une partition de 100 Go incluant Ubuntu14 (5 Go), à terme notre répertoire de travail sera suffisamment agrandi pour combler les 40 Go. Pour ne pas trop encombrer son espace de stockage on peut définir dans build/local.conf la variable

```
1  INHERIT += "rm_work" //effacera les fichiers intermédiaires au
       fur et a mesure
2  RM_OLD_IMAGE = "1" //remplacera les images par la plus récente.
```

Attention ces variables vous feront perdre des données, rm_work par exemple libérera de l'espace (16Go) mais rendra impossible le débogage de la compilation. La suppression des images précédentes est conseillée quand on est amené à en générer beaucoup. Une image pèse quelques centaines de Mo.

4.4.2 Préparation de l'environnement

Packages requis pour une compilation Yocto :

- Git 1.8.3.1 ou plus
- Tar 1.27 ou plus
- Python 3.4.0 ou plus

```
user@poky : $ sudo apt-get install gawk wget git-core diffstat unzip texinfo
gcc-multilib  build-essential chrpath socat cpio python python3 python3-pip
python3-pexpect  xz-utils debianutils iputils-ping libsdl1.2-dev xterm repo
```

4.4.3 Téléchargement des sources Freescale

La première étape consiste à télécharger tous les exécutables et métadonnées nécessaires pour créer l'image de base de l'OpenRex. En premier lieu il est nécessaire de récupérer la commande repo via les dépots officiels ou téléchargement :

```
sudo aptitude install repo sudo aptitude update sudo aptitude upgrade
```

ou

```
mkdir bin curl http ://commondatastorage.googleapis.com/git-repo-
downloads/repo > bin/repo chmod a+x bin/repo PATH=$PATH :bin
```

Ensuite il faut créer le répertoire de travail :

```
mkdir fsl-community-bsp && cd fsl-community-bsp

Yocto version 2.0

Création du répertoire du fichier manifest :
```

```
mkdir -p ./repo/local_manifests/
```

Télécharger l'ensemble des sources externes de ce projet par la commande

```
repo init -u https ://github.com/Freescale/fsl-community-bsp-platform -b jethro
```

De plus il Noter le manifest pour pouvoir télécharger/installer/décompresser les sources produites par Fedevel.

```
cat > .repo/local_manifests/imx6openrex.xml « EOF <?xml version="1.0" encoding="UTF-8"?> <manifest> <remote fetch="git ://github.com/FEDEVEL" name="fedevel"/> <project remote="fedevel" revision="jethro" name="meta-openrex" path="sources/meta-openrex"> <copyfile src="openrex-setup.sh" dest="openrex-setup.sh"/> </project> </manifest> EOF
```

Enfin, il est nécessaire de télécharger/installer/décompresser ces fichiers :

```
repo sync
```

Yocto version 2.1

```
repo init -u git@github.com/petit-romain/fsl-community-bsp-platform.git
```

Enfin, il est nécessaire de télécharger/installer/décompresser ces fichiers :

```
repo sync
```

4.4.4 Compilation (Yocto 2.0)

Créer et sourcer l'environnement pour que la compilation inclut la métadonnée Openrex.

```
source openrex-setup.sh
```

Préparer l'environnement et construire l'image. Toutes Les commandes ci-dessous seront à effectuer pendant l'usage courant, elles ne font plus vraiment partie de l'initialisation du projet.

```
MACHINE=imx6s-openrex source setup-environment build-dir bitbake core-image-base
```

4.4.5 Compilation (Yocto 2.1)

```
DISTRO=poky MACHINE=imx6-openrexbasic source setup-environment build-dir bitbake openrexpicam-base-image
```

4.4.6 Déploiement de l'image (Yocto 2.0)

Installer l'image sur la carte-sd

```
umount /dev/YourSDCard
gunzip -c tmp/deploy/images/imx6s-openrex/core-image-base-imx6s-
openrex.sdcard.gz > tmp/deploy/images/imx6s-openrex/core-image-
base-imx6s-openrex.sdcard sudo dd if=tmp/deploy/images/imx6s-
openrex/core-image-base-imx6q-openrex.sdcard of=/dev/YourSDCard umount
/dev/YourSDCard
```

A présent insérez la carte sd dans l'OpenRex et démarrez, vous devriez voir l'autoboot apparaître grâce à la communication série.

4.4.7 Déploiement de l'image (Yocto 2.1)

```
umount /dev/YourSDCard gunzip -c tmp/deploy/images/imx6-
openrexbasic/core-image-base-imx6-openrexbasic.sdcard.gz > build-
openrex/tmp/deploy/images/imx6-openrexbasic/core-image-base-imx6-
openrexbasic.sdcard sudo dd if=tmp/deploy/images/imx6-openrexbasic/core-
image-base-imx6-openrexbasic.sdcard of=/dev/YourSDCard umount
/dev/YourSDCard
```

Si vous obtenez l'erreur "Error : bootmmc not defined " , redémarrez l'OpenRex et interrompez le boot de celle-ci afin de lancer la commande ci dessous :

```
setenv bootmmc "run findfdt; mmc dev $mmcdev; if mmc rescan; then if run
loadbootscript; then run bootscript; else if run loadimage; then run mmcboot;
else run netboot; fi; fi; else run netboot; fi;\0"; saveenv; reset;
```

4.4.8 Etat initial

Une fois la procédure faite, on doit retrouver les fichiers suivant dans le dossier fsl-community-bsp :

fsl-setup-release.sh : script déjà utilisé à la premère préparation de l'environnement.

setup-environment : est utilisé pour sourcer les variables d'environnement utiles lors de la création de l'image.

Build-dir : contient tous les fichiers générés lors des compilations. On y trouve :

bitbake.lock : sémaphore temporaire assurant qu'une seule instance de bitbake accède aux fichiers

cache :

conf :

sstate-cache : avantage majeur de Yocto sur des compilation standard, dans ce répertoire sont stockées les états de compilation intermédiaire de Yocto. Pour vider la shared state cache on lance :]

```
bitbake <recette> -c cleansstate.
```

tmp : dossier principal de compilation, tmp/work contient les sources des recettes et tmp/deploy les images

sources : contient l'ensemble des métas nécessaires à la création de l'image. On y trouve :]

meta-freescale : configure l'image pour un processeur Freescale.

meta-fsl-arm : configure l'image pour un cortex arm Freescale.

meta-fsl-arm-extra : configure plus encore l'image pour un cortex arm Freescale.

meta-fsl-bsp-release : configure l'image pour une carte officielle à processeur Freescale.

meta-fsl-arm-voipac : surcouche des metas précédentes pour l'Openrex.

meta-openembedded : contient les meta des drivers materiel tel que le bluetooth et les bibliothèques utiles a l'utilisation d'un driver par exemple le python.

meta-fsl-demos : des démonstrations Freescale.

meta-browser : permet d'avoir mozilla ou bien chrome sur l'image.

poky : contient la base du build system et les metadata de base relatif a la distribution.

base : une base pour configurer son dossier build pour les meta Freescale.

meta-qt5 : elle nous permet d'utiliser du code Qt par l'OpenRex en utilisant un SDK. (adresse pour le clonage :https ://github.com/meta-qt5/meta-qt5.git)

4.4.9 Ajout de notre méta (Yocto 2.0)

meta-openrexpicam : la méta qui va définir notre distribution. (https ://github.com/Alanaitali/meta-openrexpicam.git) Si vous utilisez Yocto 2.0, notre meta doit simplement être copiée dans le dossier sources puis indiquée dans le bblayers.conf comme expliqué dans « Usage courant » ci-dessous. (follow me)

```
cd $BUILDDIR/../sources/. git clone https ://github.com/Alanaitali/meta-openrexpicam.git
```

4.5 Usage courant

Préparation de l'environnement L'environnement de développement a été créé avec fsl-setup-release.sh, mais pour chaque terminal de travail, il faudra sourcer l'environnement à nouveau. C'est-à-dire ajouter des variables d'environnement qui seront nécessaire lors de la compilation. La commande suivante source l'environnement et dans le même temps nous lui indiquons le nom de la machine dans notre cas « imx6s-Openrex » que l'on retrouve dans les sources téléchargées. De cette manière nous donnons des paramètres qui seront pris en compte lors de la compilation avec le « bitbake ». Cette assignation de machine sera inscrite dans le local.conf de manière plus pérenne.

Yocto 2.0 :

```
MACHINE=imx6s-openrex source setup-environment build-dir bitbake core-image-base
```

Yocto 2.1 :

```
DISTRO=poky MACHINE=imx6-openrexbasic source setup-environment build-dir
```

4.5.1 Vérification de l'image demandée

Avant de lancer la compilation nous devons indiquer le chemin d'accès et le nom des métadonnées à prendre en compte pour la compilation. Ces paquets sont aussi appelés couches (layers), ils sont référencés dans le fichier bblayer.conf dans le dossier /build-dir/conf. Pour inclure une nouvelle meta à la compilation il faut écrire son chemin sur bblayers.conf.

```
  POKY_BBLAYERS_CONF_VERSION = "2"

  BBPATH = "\${TOPDIR}"
  BSPDIR := "\${@os.path.abspath(os.path.dirname(d.getVar('FILE', ␣
     True)) ␣+␣'/../..')}"
  BBFILES ?= ""

  BBLAYERS = "␣\
␣␣\${BSPDIR}/sources/poky/meta␣\
␣␣\${BSPDIR}/sources/poky/meta-poky␣\
␣␣\
␣␣\${BSPDIR}/sources/meta-openembedded/meta-oe␣\
␣␣\${BSPDIR}/sources/meta-openembedded/meta-multimedia␣\
␣␣\
␣␣\${BSPDIR}/sources/meta-fsl-arm␣\
␣␣\${BSPDIR}/sources/meta-fsl-arm-extra␣\
␣␣\${BSPDIR}/sources/meta-fsl-demos␣\
␣␣\
␣␣\${BSPDIR}/sources/meta-openrexpicam␣\
␣␣"
  BBLAYERS += "\${BSPDIR}/sources/meta-fsl-arm-voipac"
```

Afin de configurer l'environnement de travail de bitbake on rédige un fichier local.conf. Attention à ce que le nom de machine (imx6-openrexbasic pour Yocto 2.1 et imx6s-openrex) soit en accord avec la recette utilisée.

```
  MACHINE ??= 'imx6-openrexbasic'
  DISTRO ?= 'poky'
  PACKAGE_CLASSES ?= "package_rpm"
  EXTRA_IMAGE_FEATURES ?= "debug-tweaks"
  USER_CLASSES ?= "buildstats␣image-mklibs"
  PATCHRESOLVE = "noop"
  BB_DISKMON_DIRS = "\
␣␣STOPTASKS,\${TMPDIR},1G,100K␣\
␣␣STOPTASKS,\${DL_DIR},1G,100K␣\
␣␣STOPTASKS,\${SSTATE_DIR},1G,100K␣\
␣␣STOPTASKS,/tmp,100M,100K␣\
␣␣ABORT,\${TMPDIR},100M,1K␣\
␣␣ABORT,\${DL_DIR},100M,1K␣\
␣␣ABORT,\${SSTATE_DIR},100M,1K␣\
␣␣ABORT,/tmp,10M,1K"
  PACKAGECONFIG_append_pn-qemu-native = "␣sdl"
  PACKAGECONFIG_append_pn-nativesdk-qemu = "␣sdl"
  CONF_VERSION = "1"

  DL_DIR ?= "\${BSPDIR}/downloads/"
  ACCEPT_FSL_EULA = "1"
```

Par soucis de personnalisation la distribution du projet peut être renommée et le nombre de threads configurés en ajoutant :

```
1  DISTRO ?= 'name'
2  BB_NUMBER_THREADS?="4"
3  PARALLEL_MAKE?="-j4"
```

4.5.2 Compilation

L'étape compilation prend bien moins de temps lors de l'usage régulier (près de 24 fois plus rapide). Pendant cette étape nous avons un grand nombre d'informations qui s'affichent sur le terminal, les plus importantes sont les warnings et les erreurs de compilation qu'il est nécessaire de comprendre puis interpréter pour résoudre.

```
bitbake openrexpicam-base-image
```

Une fois cette étape terminée on peut retrouver toutes une arborescence de dossier et de fichiers dans le dossier de construction (/build-dir), ces derniers seront au cœur du fonctionnement de notre appareils. ils sont spécifiques à notre carte, les modules, les capteurs qu'elle comporte. Le fruit de la compilation de l'image contient le root-filesystem le kernel le bootloader et le device-tree en un seul fichier compressé. /build-dir/tmp/deploy/images/imx6-openrexbasic doit contenir :

1 image sous format .sdcard.gz

1 image sous format .rootfs.ext4

1 image sous format .rootfs.manifest

1 image sous format .rootfs.sdcard