Master 2 SDI Parcours Architecture et Conception des Systèmes Intégrés
Université Pierre et Marie Curie
2008-2009

Développement d'un driver pour capteur CMOS OV7620 sur carte ARMadeus APF9328

Fouad Farid HADDAD

Laboratoire Informatique de L'Ecole Polytechnique, CNRS
Ecole Polytechnique
91120 Palaiseau CEDEX

Tuteur de stage : M. James REGIS
Responsable de formation : M. Julien Denoulet

Remerciements

Je tiens tout d'abord à remercier Monsieur James Regis pour m'avoir fait confiance pour ce stage et pour son aide durant de ces quatre mois et demi. j'éspère que le travail réalisé ainsi que ce mémoire seront à la hauteur de vos espérances.

Un très grand merci à Monsieur Jean Michel Freidt, ingénieur dans la société Senseor, hébergé par l'institut, pour m'avoir épaulé et pour avoir apporté son opinions qui m'a été plus que bénéfique. De même, je remercie infiniment Thibault Retronaz, doctorant au sein de l'institut FEMTO-ST de Besançon pour avoir été omniprésent durant ces quatre mois et demi ainsi que pour avoir mis à ma disposition ses travaux effectués sur l'apf9328.

Je suis plus que reconnaissant envers ces deux personnes qui ont su m'accueillir dans leur locaux à Besancon durant trois jours, bien qu'il n'étaient pas obligé de le faire.

Merci également à toute l'équipe Armadeus System : Julien Boibessot, Fabien Marteau, Nicolas Colombain, concepteurs de la carte, pour leur assistance, leur patience. La pertinence de leurs propos et leur expérience m'ont été rentables dans les moments de doute.

Merci à l'équipe présente sur le salon Armadeus, sur IRC (Internet Relay Chat), qui ont montré un intérêt pour le travail que j'ai fourni.

Merci à Alexandre Gordien, stagiaire au LIX durant le mois de juillet, pour m'avoir fourni un document très précieux à mes yeux concernant la rédaction de document Latex.

Je tiens également á remercier Monsieur Thomas DACHY pour nous avoir accordé confiance en nous prêtant du materiel de mesure (oscilloscope).

Et enfin, merci à mes camarades et amis Romain Paul et Mustapha Khorchid, compagnons de stage. J'ai pu passer mes quatre mois et demi de stage avec vous, votre aide et votre sympathie m'a permis de ne pas me décourager dans les moments les plus durs, heureusement que vous étiez là, ça aurait été vraiment dur tout seul.

Table des matières

1 Présentation du stage

1.1 Introduction

Dans le cadre de L'obtention e mon diplôme de Master Science de l'ingénieur parcours ACSI (Architecture et Conception des Systèmes Intégrés) de l'université Pierre et Marie Curie, j'ai effectué mon stage dans un laboratoire de recherche de l'Ecole Polytechnique de Palaiseau, le Laboratoire d'informatique (LIX). L'objet de mon stage a été d'interfacer un capteur CMOS avec une carte incluant un système d'exploitation Linux, le cœur de ce travail était de réaliser un driver en C permettant de récupérer les image de ce capteur. Les contrainte pour la réalisation de ce système étaient de faire un système le plus intégrable possible, j'entends par là, un système qui ne fait pas appel à des composants annexe autre que la carte du système et le capteur.
Par souci d'intégration mais aussi de consommation, nous nous sommes fixé ces contraintes de le but d'inclure le module complet (carte + capteur) dans un système robotisé. L'une des contraintes qui s'est opposé a moi est notamment la maitrise de la carte et de son système d'exploitation Linux, ayant peut d'expérience sur ce système d'exploitation et surtout, n'ayant pas de connaissance en programmation de driver, ce sujet de stage est l'excuse pour que je puisse me familiariser avec ces deux notions. D'autant plus que la carte fournie est une carte incluant deux plateformes différentes : un processeur de type ARM9 et un FPGA conçu par Xilinx, ces deux composants font appel aux deux principales dominantes de ma formation : architecture des microprocesseurs ainsi que la conception à base de plateforme FPGA.
Les grandes lignes de ce projet sont, dans un premier temps, la maitrise de la carte, notamment la communication entre ces deux plateformes, dans un second temps la maitrise du noyau Linux, la maitrise du fonctionnement du capteur CMOS et dans un dernier l'interfaçage des ces deux modules sera une application qui illustrera le travail effectué. Mon travail a donc été orienté en ce sens et ce mémoire va le présenter.

1.2 Présentation du LIX

Situé au coeur du Centre de Recherche sur le campus de l'école Polytechnique à Palaiseau, le Laboratoire d'Informatique de l'École Polytechnique (plus communément appelée X, et le laboratoire, le LIX) est une UMR X-CNRS d'une centaine de membres dont une moitié de doctorants et une quarantaine de permanents équitablement répartis entre le CNRS, l'INRIA et l'X. Les activités du LIX se regroupent en trois grands domaines : algorithmique, réseaux, et méthodes formelles. Au sein du laboratoire, on compte 6 projets INRIA (Institut National de Recherche en Informatique et Automatique) installés au LIX depuis la création du Pôle Commun de Recherche en Informatique du Plateau de Saclay, et une équipe commune avec le CEA LIST (Laboratoire des Intégration et Systèmes) qui préfigure les cohabitations futures entre des équipes. Enfin, le LIX abrite la Chaire « Systèmes Industriels Complexes » financée par Thalès. Le LIX s'intéresse tout particulièrement au domaine des communications en réseau, afin de se doter dans ce domaine de compétences globales relatives au traitement du signal, au chiffrement et au routage des communications, à la distribution et à la mobilité des calculs ainsi qu'à la sécurité et à l'ingénierie des protocoles. Au cœur de ces recherches, dont le but est de mettre au point des systèmes de communications fiables et efficaces, l'accent est mis sur les réseaux mobiles, qui deviendront une composante incontournable des systèmes embarqués futurs.

Le laboratoire est fortement impliqué dans les enseignements de l'École Polytechnique, et aussi dans plusieurs Master de la région parisienne. Il a des relations contractuelles avec des organismes publics. Au cours des dernières années, le LIX s'est attaché à développer de nombreuses collaborations avec le monde industriel : Thalès, Microsoft Research, Hitachi Labs, et la NASA. Le LIX compte a ce jour dix équipes qui regroupent les activités du laboratoire :

- Bioinformatique structurale : Modélisation et prédiction de la structure des protéines transmembranaires Détermination et comparaison de la structure secondaire des ARN
- Génomique comparative :
  - Comparaison de mini satellites
  - Statistiques sur les motifs
  - Recherche de sites de N-myristoylation
- Imagerie médicale :
  Modélisation et lecture automatisée de radiographies
- Algorithmique et Optimisation : Il s'agit la d'une équipe qui travaille sur des problèmes d'algorithmique, d'optimisation et de recherche opérationnelle. Les travaux portent à la fois sur des problèmes théoriques (analyse de complexité, comptage de solutions, théorie de l'ordonnancement et satisfaction de contraintes).
- Modèles Combinatoires :
  Les Thèmes principaux de cette équipe sont les suivants : énumération, algorithmique des

graphes et structures de données.

- MAX : Modélisation Algébrique :
  Cette équipe a pour but de travailler sur les question d'algorithmique, d'automatique algébrique et d'étude algébrique des systèmes dynamiques.
- Cryptologie : Les activités de cette équipe concernent toute la chaîne entre les théorèmes et les programmes qui ont vocation la performance.
- Hipercom : High Performance Communication Il s'agit la d'une équipe spécialisée dans les réseaux de télécommunication, plus particulièrement les réseaux AD HOC et sans fils.
- LogiCal : Le but des recherches menées dans le projet est de construire des systèmes de traitement de démonstrations mathématiques, c'est-à-dire des systèmes capables d'opérer des traitements divers sur des connaissances mathématiques.
- ParSifal : Cette équipe à pour objectif de développer et d'exploiter la théorie sur les spécifications et la vérification des calculs.
- Comète :
  L'équipe Comète est une équipe qui s'est familiarisé avec les systèmes fortement distribués se composant des dispositifs divers et spécialisés, fournissant des services et des applications différentes.
- MeAZI : L'équipe MeAZI est une équipe spécialisée dans les systèmes se caractérise fondamentalement par son comportement temporel entrées-sorties.

1.3 Présentation de l'association Armadeus Project

« Armadeus Project »(anciennement Armadeus) est une association sans but lucratif basée à Mulhouse, qui a pour but le développement et la production de systèmes embarqués entièrement configuré et programmé via des outils libres. Armadeus Systems est une société basée sur cette association qui a été crée afin de proposer des solutions logicielles et matérielles ayant un rapport performances/prix parfaitement adapté aux contraintes d'aujourd'hui. L'équipe d'Armadeus Systems est composée d'Ingénieurs en logiciel et en électronique ayant plusieurs années d'expérience dans les domaines de l'industrie (R&D), du médical (ultrasons) et des systèmes de communication. Notamment Julien Boibessot, Principal Ingénieur Software, et Fabien Marteau, ingénieur Hardware. A noter également qu'Armadeus Systems est agrée au titre du CIR (Crédit Impôt Recherche). Cela permet à ses clients de déclarer le montant des opérations de R&D éligibles au CIR.
Armadeus Systems développe et produit des systèmes embarqués Linux (open source) , faible coût alliant petite taille, basse consommation et connectivité étendue. A ce jour, Armadeus Systems existe grâce à l'esprit du logiciel libre et est membre de la Freescale Alliance ainsi que de Rhénatic. Freescale étant un leader mondial depuis dix ans dans les microcontrôleurs embarqués, Rhénatic est un réseau regroupant les entreprises du secteur des nouvelles technologies de Mulhouse et de ses environs.

Aujourd'hui, Armadeus Systems est une SARL avec un capital de 60 000 euros et a développé principalement deux cartes : l'apf9328 et l'apf27 ainsi que quatre kits de développement associées à ces cartes. De l'esprit du logiciel libre est né la devise de Armadeus Systems : « Une vision nouvelle de l'embarquée ».

2 Présentation des outils de développement

2.1 Outils matériels

2.1.1 L'APF9328

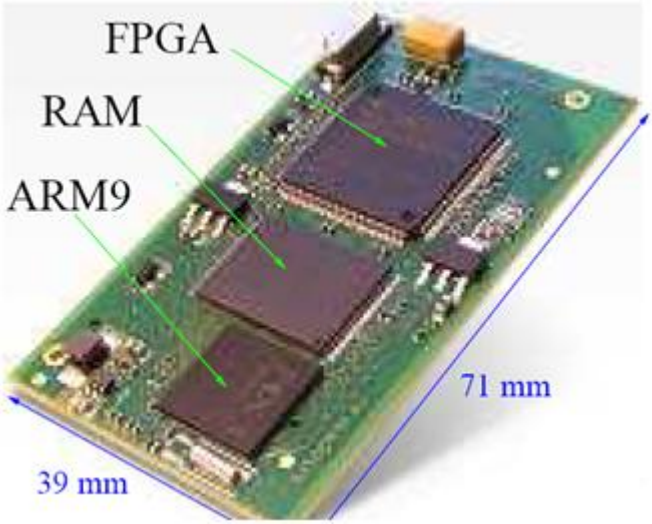
L'APF9328 est une carte à microprocesseur de taille réduite bénéficiant d'un rapport coût/performance extrêmement compétitif. Elle est équipée d'un microprocesseur MC9328MXL(de la famille i.MXL), concu par Freescale, ARM9 (ARM920T) à 200MHz, de SDRAM 8 ou 16 Mo 32 bits à 100 MHz, de FLASH 4, 8 ou 16 Mo 16 bits à 100 MHz, d'un port Ethernet 10/100Mbits, et d'un FPGA Spartan 3 Xilinx Spartan3 avec 200k portes (FPGA 50k et 400k en option).

Ce dernier est vu comme un coprocesseur par le ARM9 elle est facilement intégrable dans un système embarqué grâce notamment à ses régulateurs et ses convertisseurs de niveau (RS232/USB). Cette carte est équipée d'un système d'exploitation embarqué Linux 2.6.2x (ou supérieur). A titre indicatif, debugger externe tel qu'un JTAG n'est requis. Une simple liaison RS232 est suffisante pour des développements "faibles coûts". Un debugger GDB Linux est disponible. Du point de vue de l'alimentation, cette carte à besoin de 3.3V à plus ou moins 5% et à une consommation typique : 300mW (sans FPGA). Elle est équipée d'une gestion d'énergie permettant de réduire la consommation jusqu'à 30mA.

Au niveau conversion, l'apf9328 est équipée d'un DAC 2 canaux 10bits 400kHz ainsi que d'un ADC 8 canaux 10bits 10MHz. Concernant son environnement de fonctionnement, elle fonctionne correctement à des températures allant jusqu'à 60^{circ}C sans faire appel à un refroidissement particulier et à un taux d'humidité allant de 5 à 90%. Cette carte gère une série de périphériques qui rend ainsi cette carte adaptable à un large spectre d'application, voici les liste des périphériques gérées par cette carte :

- 1 x RS232 (RX/TX)
- 1 x RS232 niveaux TTL (RX, TX, CTS, RTS)
- 1 x I2C

- 2 x SPI
- 1 x SSI (Port série synchrone rapide)
- 1 x USB 1.1 esclave avec convertisseur de niveau
- 1 x réseau 10/100 Mbits sans connecteur RJ45
- 1 x SD/MMC
- 1 x RTC sans batterie
- 1 x PWM résolution 16bits
- 2 x Timer 32 bits avec fonctions de "capture/output compare"
- 1 x Watchdog. ajustable entre 0.5s et 64s (pas de 0.5s)
- Contrôleur vidéo LCD jusqu'à 640x512 pixels (64K couleurs)
- CSI (interface capteur vidéo CMOS)
- jusqu'à 90 Entrés/Sorties (GPIO)
- port de debug JTAG (ICE)



2.1.2 L'APF9328 DEVLight

La carte APF9328DevLight est une plateforme de développement faible coût idéale pour l'expérimentation et la vérification d'applications simples. Elle permet d'accéder à la plupart des fonctionnalités de la carte APF9328 (figure 2.1.2)tout en offrant deux zones de brochage.

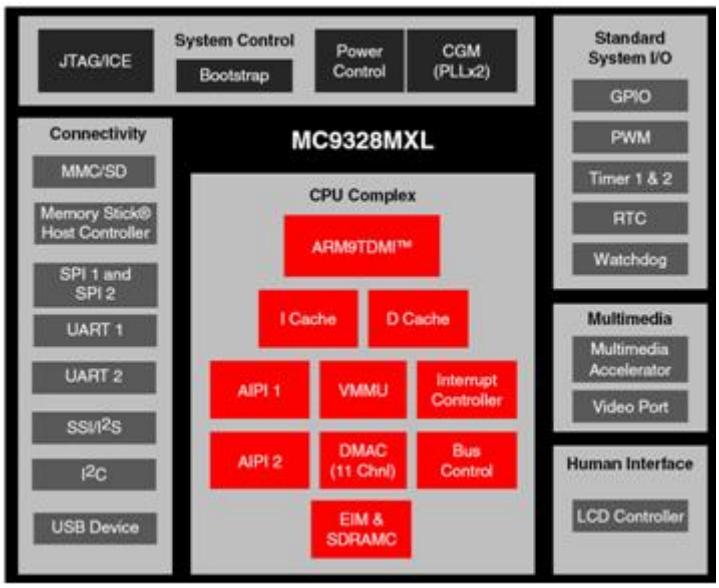
1. Alimentation :
   - Tension d'entrée 5 à 9V DC protégée par fusible réarmable
   - Régulateur passif 3.3V / 1A max
   - Bouton de reset
2. Spécificités :
   - 2 zones de prototypage équipée de trous aux pas de 2.54 mm
   - Indication des noms des connexions de la carte APF9328
   - Nouveau sur version V2 : Bus LCD pré-routé pour la connexion rapide d'un écran (pas de 2.54mm)
3. Connecteurs :
   - Ethernet (RJ45) avec transformateur d'isolement intégré
   - USB esclave (type B)
   - DB9 Male (RS232)
   - Jack 2.5mm pour l'alimentation
   - 2 connecteurs Hirose pour la carte APF9328
   - Cavalier de sélection du mode de démarrage
   - Nouveau sur version V2 : Slot pour carte micro SD
4. Mécanique
   - Dimensions : 100mm (3.9") x 75mm (3")

2.1.3 Le Processeur MC9328MXL

Le processeur MC9328MXL fait partie de la série i.MX de Freescale. Ce sont des processeurs mobile pour des applications multimédia. La série i.MX était précédemment connue comme la famille DragonBall MX, la cinquième génération des DragonBall. Le MC9328MXL à une fréquence de fonctionnement de 150 à 200 MHz.

La famille de processeurs i.MX est basée sur un noyau ARM9, le MC9328MXL est basée sur une architecture faible consommation d'ARM920T TDMI. Cette famille de processeurs inclut un certains nombres de périphériques que l'on retrouve en accord avec les besoins du marché actuel, on les directement disponible sur l'apf9328, Il fournit également une possibilité de gestion de puissance afin de répondre de la façon la plus optimale aux problèmes de portabilité.

Il intègre également des modules du type contrôleur USB, un contrôleur d'affichage à cristaux liquides (LCD), et un contrôleur de cartes MMC/SD, afin de fournir une expérience riche en multimédia. Le packging du MC9328MXL est de type Mold Array Process-Ball Grid Array (MAPBGA) à 256 contacts. Le schéma 1 montre le schéma fonctionnel du processeur MC9328MXL.



Le processeur de la famille i.MXL est implanté dans des applications de hautes technologies tel que des téléphones intelligents, des navigateurs de Web, des lecteurs MP3 audio numériques ou encore des ordinateurs portables, et des applications de transmission de messages.

2.1.4 Le FPGA Xilinx Spartan3

Le Spartan 3 est un FPGA conçu par Xilinx, l'une des plus grandes entreprises spécialisées dans le développement et la commercialisation de composants logiques programmables.

Il s'agit d'un FPGA de grande série composé de 200 milles portes logiques, il a été conçu sur les bases d'un autre FPGA du même constructeur, le Virtex II, qui est un FPGA haute performance. Ce circuit programmable sera une interface très intéressante entre le monde logiciel et le monde matériel de la carte mère apf9328, ce qui élargie davantage le spectre d'application de cette carte.

2.2 Outils logiciels

2.2.1 Linux embarqué

Comme cela a été dit précédemment, la carte apf9328 à un système d'exploitation Linux embarqué sur le processeur MC9328MXL, l'une des expériences importantes de ce stage est de pourvoir se servir d'un système d'exploitation embarqué et de programmer un microprocesseur à un niveau haut, chose totalement nouvelle pour moi. Nous allons donc décrire dans un premier temps ce qu'est Linux et les différentes partie composant ce système d'exploitation.

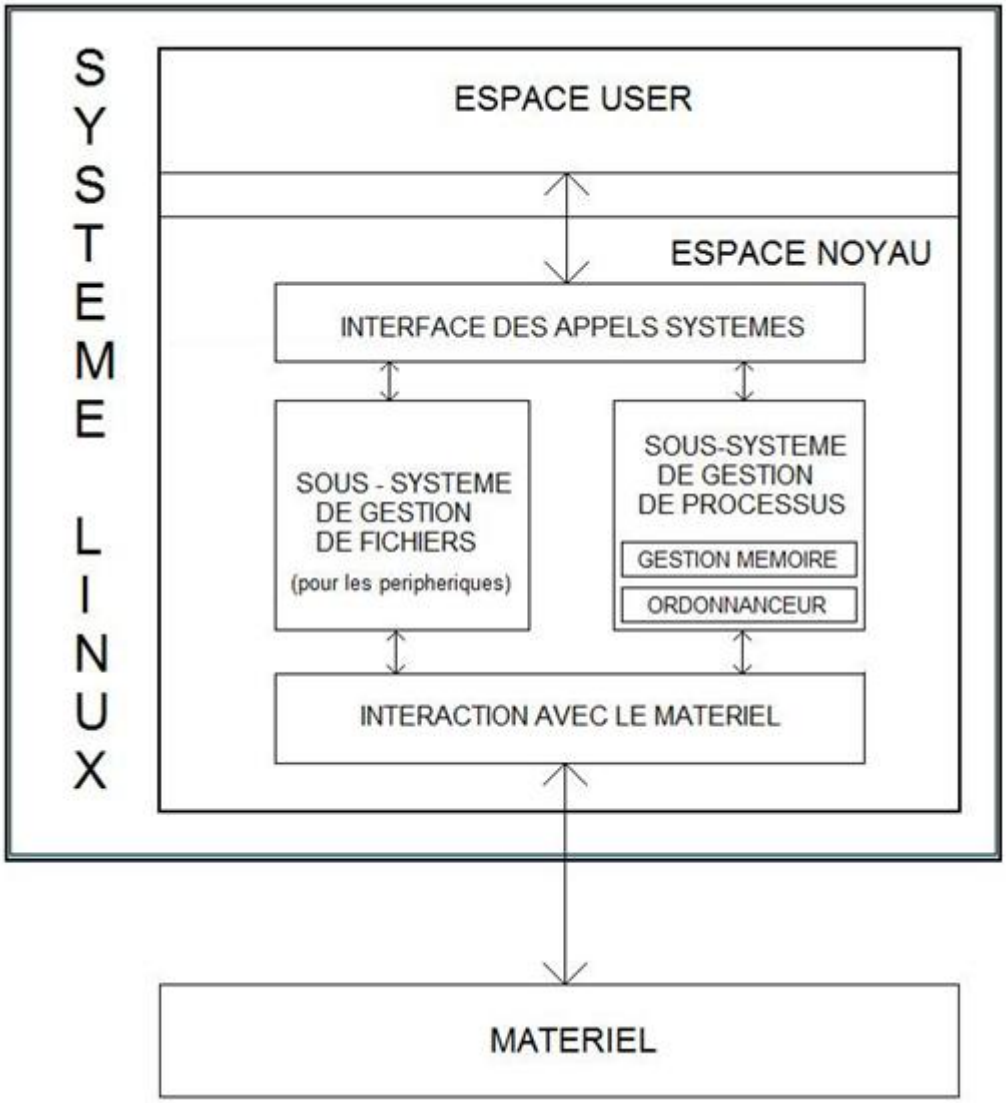
Linux, à la base, n'est pas le système d'exploitation du même nom mais le noyau en question. En effet Linux est un noyau de système d'exploitation de type UNIX. Le noyau Linux est un logiciel libre développé par une large communauté de contributeurs. Au niveau structurel, un système d'exploitation (Linux ou autre) est composée de deux espaces : l'espace user (utilisateur) et l'espace noyau.

Le noyau est le cœur du système, c'est lui qui s'occupe de fournir aux logiciels une interface pour utiliser le matériel. Le noyau Linux a été développé par Linus Torvalds au début des années 1990 pour des ordinateurs compatibles, d'architecture x86. Depuis, il a été porté sur nombre d'architectures dont m68k, PowerPC, ARM, StrongARM, , Alpha, SPARC, MIPS, etc. Il peut être au cœur autant d'un ordinateur personnel que d'un système embarqué tel un téléphone portable ou un assistant personnel. Un noyau gère les ressources de l'ordinateur et permet aux différents composants, matériels et logiciels, de communiquer entre eux.

En tant que partie du système d'exploitation, le noyau fournit des mécanismes d'abstraction du matériel, notamment de la mémoire, du (ou des) processeur(s), et des échanges d'informations entre logiciels et périphériques matériels. Le noyau autorise aussi diverses abstractions logicielles et facilite la communication entre les processus (voir la figure suivante). Il assure entre autre :

- la communication entre les logiciels et le matériel ;
- la gestion des divers logiciels d'une machine (lancement des programmes, ordonnancement,...).
- la gestion du matériel (mémoire, processeur, périphérique, stockage,...).

En l'occurrence, la version Linux 2.6.27.3 installée sur l'MC9328MXL va gérer tout les ports de ce dernier. La programmation de niveau haut en C sous Linux fait abstraction de la programmation longue et bas niveau des registres spécifiques au processeur MC9328MXL.



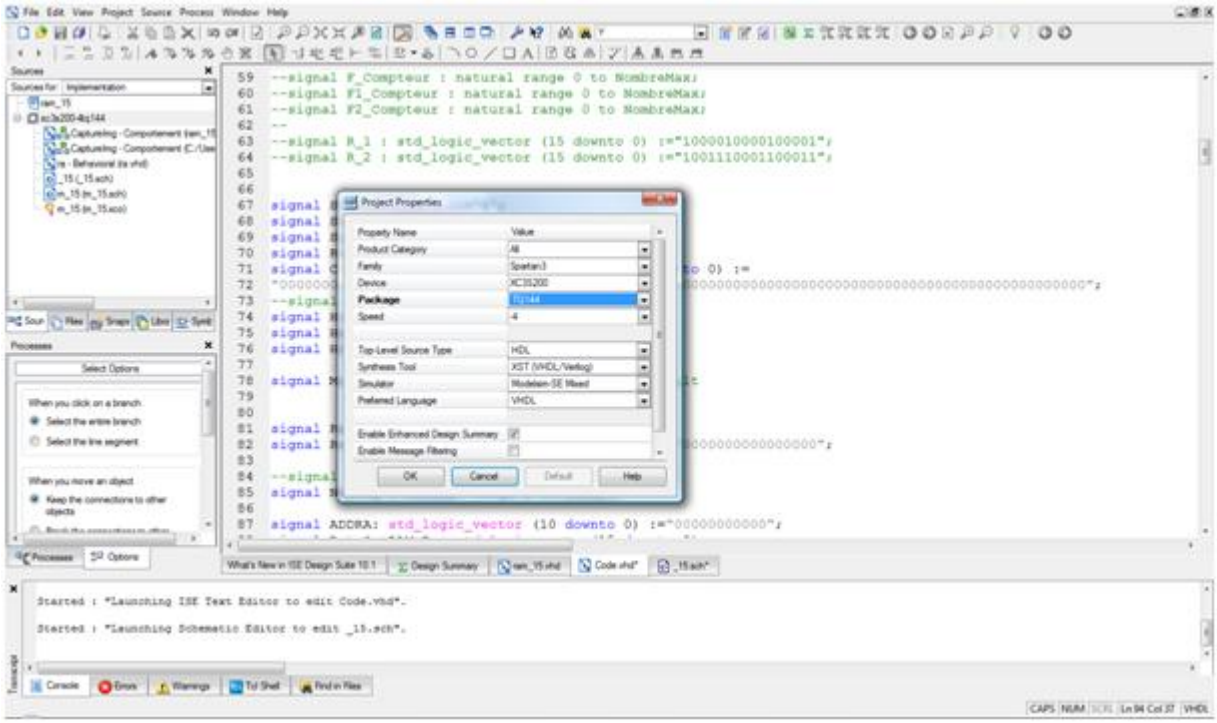
2.2.2 Installation de Xilinx ISE

Afin de pour programmer le FPGA, il est nécessaire d'avoir le logiciel Xilinx ISE fourni par le constructeur, la licence disponible sur le site http ://www.xilinx.com pour GNU/Linux dans le temps. Comme tout logiciel distribué sous GNU/Linux, cet outils de synthèse est distribué sous la forme d'une archive compressée qu'on se contentera de désarchiver au bon endroit, puis de rendre les binaires associés exécutables. La procédure d'installation est décrite sur le wiki de Armadeus :

`http://www.armadeus.com/wiki/index.php?title=ISE_WebPack_installation_on_Linux.`

Le FPGA sera, comme cela à été dit précédemment, une interface entre le monde logiciel du processeur MC9328MXL de l'apf9328 et le monde matériel, par exemple la gestion d'un périphérique externe, surtout ci celui offre un principe de fonctionnement séquentiel, sera gérable plus facilement sur le FPGA Spartan 3 que sur le processeur. Bien qu'il soit techniquement possible d'installer une plateforme de développement et de simulation VHDL complètement libre sous licence GNU, il est toujours nécessaire de recourir à Xilinx ISE, en effet ce logiciel permet de générer un fichier binaire d'une part, en adéquation avec la description d'un composant VDHL

`(composant_VHDL.bin)`



2.2.3 Installation de GTKterm

GTKterm est programme de communication adaptée pour le port série (c'est l'équivalent de l'HyperTerminal sous Windows). Il permet notamment de se connecter sur le port console d'un modem, d'un switch, d'un routeur. Tout les matériels possédant un port console peuvent être configurés par GTKterm. Ce programme permet par exemple de configurer un switch pour lui assigner une adresse IP, d'activer le protocole ce communication sur le switch, de configurer un routeur lors de son premier démarrage.

Dans notre cas, étant donné que l'apf9328 est composée d'un port série, GTKterm sera l'interface entre l'utilisateur et la carte pour tout type de communication : configuration de la carte, chargement des programmes, message d'erreur, résultats.

Le souci est que mon compte crée ne permet pas d'assurer un protocole UUCP. Le protocole UUCP (Unix to Unix Copy Protocol) est un ensemble de programmes qui permettent à deux systèmes d'échanger des fichiers et d'exécuter des commandes sur une machine distante, en l'occurrence la carte apf9328, en passant un câble série. Il a fallut donc créer un compte utilisateur ayant les propriété UUCP, ce compte à été nommé « arm », ensuite, par ssh, on se connecte sur ce compte, GTKterm a été installé sur ce compte.

Au lancement de GTKterm, il est nécessaire de configurer les paramètres sur le port série : ttyS0 pour sélectionner le port série 0, une vitesse de transfert de données à 115200 bits/secondes, une Parité à « None », Bits à 8, Bits de stop à 1 et enfin Contrôle de flux à None :

2.3 Mise à jour de la carte APF9328 et chargement du système d'exploitation

Afin de pouvoir se servir de la carte apf9328, il est nécessaire d'installer une toolchain spécifique à celle-ci, une toolchain étant l'ensemble des outils nécessaires à la compilation de binaires pour une architecture donnée. Cette toolchain comprend notamment :

- Le système d'exploitation GNU/Linux 2.6.2
- Le cross-compiler spécifique à l'apf9328, Arm-linux-gcc indispensable à la compilation des application en C s'exécutant sur le MC9328MXL
- Les librairies associées afin de fournir les outils de base ainsi que diverses librairies annexes (tel que imx-reg.h qui permet de programmer le MC9328MXL à l'état bas) répondant aux besoins du développeur

Installer GNU/Linux sur l'apf9328 est une expérience assez gratifiante en terme de système d'exploitation embarqué, il m'est paru nécessaire de la résumer, il sera notamment intéressant de comprendre que pour la construction d'un système embarqué sur une carte comme l'apf9328, il est nécessaire de générer certains fichiers binaires.
Nous avons pris la dernière release disponible pour cette carte, la release Armadeus 3, adaptée au carte mère apf9328 et apf27. Ce qui différencie cette version de la précédente, la 2.3, c'est notamment Linux kernel (2.6.29), U-Boot (1.3.4), le GCC (4.2.1), Busybox (1.12.1) ainsi qu'une version de buildroot plus récente.

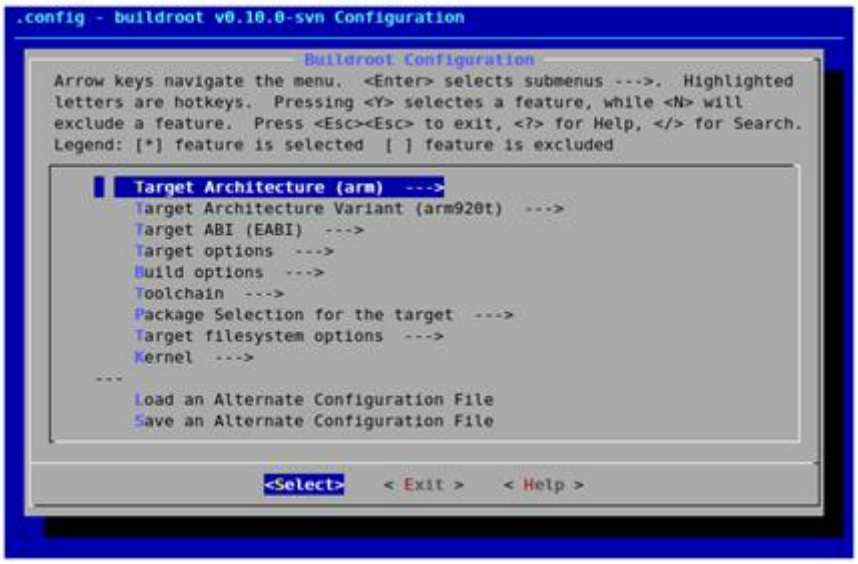
Cette version est obtenu via la commande suivante :

```
git clone git://armadeus.git.sourceforge.net/gitroot/armadeus armadeus
```

Une fois le dossier reçu et décompressé dans notre partition, la configuration se lance grâce à la commande suivante :

```
make apf9328_defconfig
```

Ce qui lance automatiquement le menu de configuration de la carte :



Ce menu, standard à toute les cartes Armadeus, permet de choisir les options nécessaires pour notre carte comme par exemple la version du noyau voulu, le type de coeur de processeur (ARM920T) ou encore la taille de la RAM. Il est aussi possible de rajouter des outils comme par exemple un cross compiler C++.

Une fois la configuration faite, il est nécessaire de quitter le menu et de lancer le processus de construction («build process ») afin de générer les fichiers nécessaires à l'installation du système d'exploitation sur la carte apf9328. Cette construction se lance via la commande « make » , à titre indicatif, la génération de ces fichiers binaires met environ une heure et demi. Nous retrouvons ces fichiers dans buildroot/binaries/apf_9328/ :

- apf9328-u-boot.brec (image apf9328 Brecord)
- apf9328-u-boot.bin (image U-Boot file to be used with U-Boot)
- apf9328-linux.bin (image Linux à utiliser avec U-Boot)
- apf9328-rootfs.arm.jffs2 (image FileSystem/RootFS to use with U-Boot, see RootFS flashing)
- apf9328-rootfs.arm.tar (pour un (rebootage) RootFS sur NFS ou bien sur MMC)

L'ensemble des outils nécessaires à la compilation de binaires pour l'apf9328 est disponible dans : buildroot/toolchain_build_armv4vt.

Ensuite, il est nécessaire de charger le dossier contenant le fichiers binaires à la configuration de la carte cible dans un répertoire publique tftpboot.
Sur la console de la carte, via GTKterm, on accède au BIOS de la carte. Le BIOS sera le « chargeur d'amorcage » du système d'exploitation à installer. il est nécessaire de configurer le réseau sur cette carte afin de lui indiquer le chemin pour récupérer les fichiers image.
Dans un premier temps, on utilise la commande setenv afin d'indiquer, d'une part, au système

armadeus l'adresse IP du réseau sur lequel sont chargé les fichiers image, et réciproquement d'indiquer à la cible l'adresse du réseau, et d'autre part, d'indiquer à la cible le chemin dans lequel se trouve le binaire necessaire à la construction du système d'exploitation sur l'apf9328 :

```
BIOS> setenv ipaddr 192.168.0.10
 BIOS> setenv serverip 192.168.0.2
 BIOS> setenv rootpath "/tftpboot/apf9328-root"
```

`Ipaddr` étant l'adresse de la carte cible (apf9328) et servip, tout naturellement l'adresse de du réseau. La commande printenv permet de vérifier que la configuration du reseau à bien été faite :

```
BIOS> printenv
bootcmd=run jffsboot
bootdelay=20
...
gatewayip=192.168.0.1
netmask=255.255.255.0
ipaddr=192.168.0.10
serverip=192.168.0.2
```

la commande saveenv permet de sauvegarder le réseau sur lequel la carte a été connectée :

```
BIOS> saveenv
```

Avant d'installer le système d'exploitation sur l'apf9328, il est nécessaire de rendre la carte *bootable* au démarrage, c'est le rôle du U-boot et du fichier binaire apf9328-u-boot.bin, ce fichier s'installe via cette commande :

```
BIOS> tftpboot 08000000 apf9328-u-boot.bin
```

Une fois U-boot installé, il est necessaire d'installer le noyau Kernel sur l'apf9328 :

```
BIOS> run update_kernel
```

Une fois que ces étapes ont été faite, la carte-mère apf9328 est prête à être utilisée : une fois redémarrée et au bootage de la carte, nous apercevons les informations suivantes :

```
BIOS> boot
```

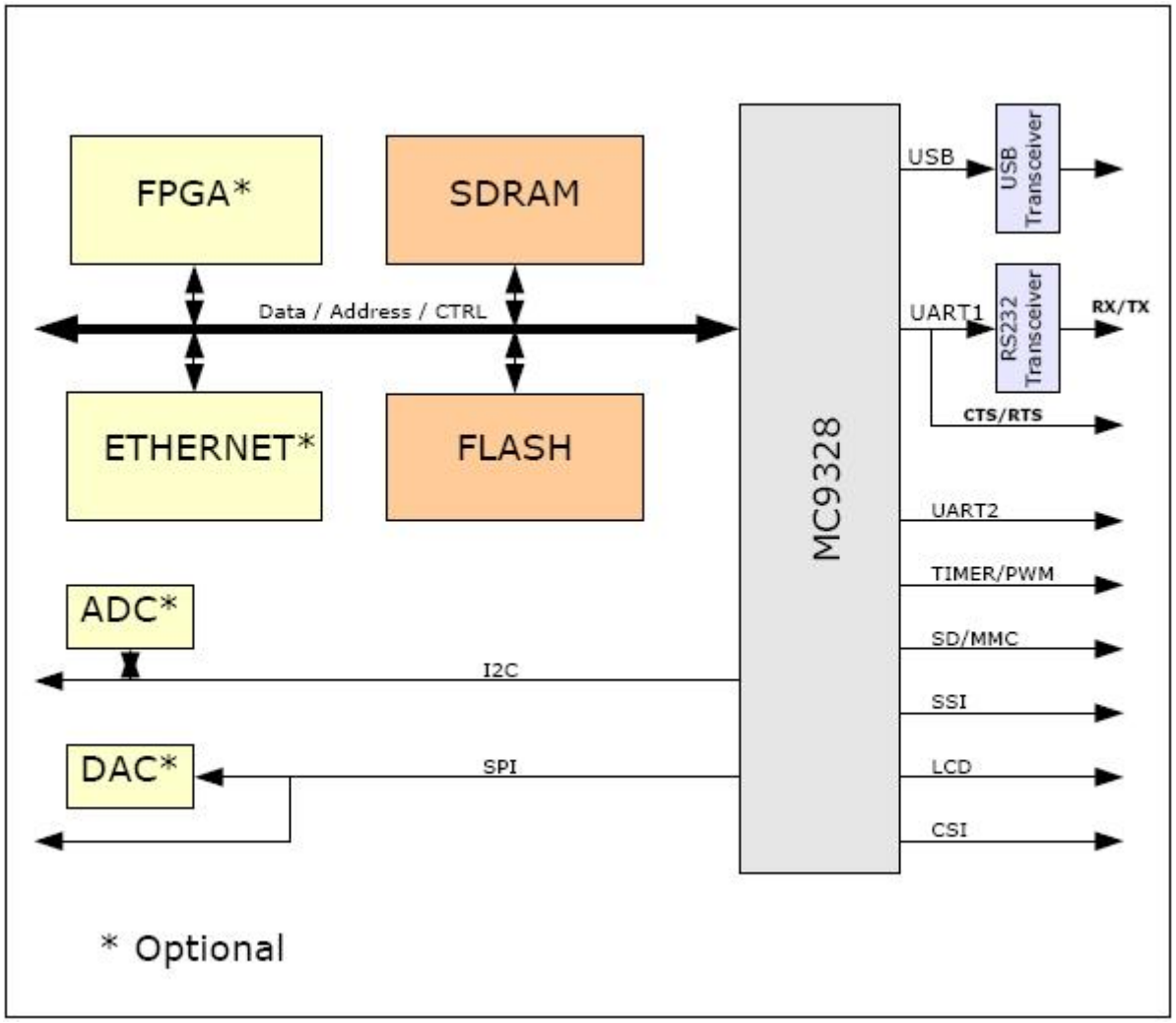
```
## Booting kernel from Legacy Image at 100a0000 ...
   Image Name:   Linux-2.6.29.4
   Created:      2009-07-30  15:06:49 UTC
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    1687332 Bytes =  1.6 MB
   Load Address: 08008000
   Entry Point:  08008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK

OK

Starting kernel ...

Uncompressing Linux..................................... done, booting the kernel.
Linux version 2.6.29.4 (haddad@icsla.lix.polytechnique.fr) (gcc version 4.2.1)
#1 PREEMPT Thu Jul 30 17:06:47 CEST 2009
CPU: ARM920T [41129200] revision 0 (ARMv4T), cr=c0007177
CPU: VIVT data cache, VIVT instruction cache
Machine: Armadeus APF9328

[..............]

Welcome to the Armadeus development environment.

armadeus login:
```

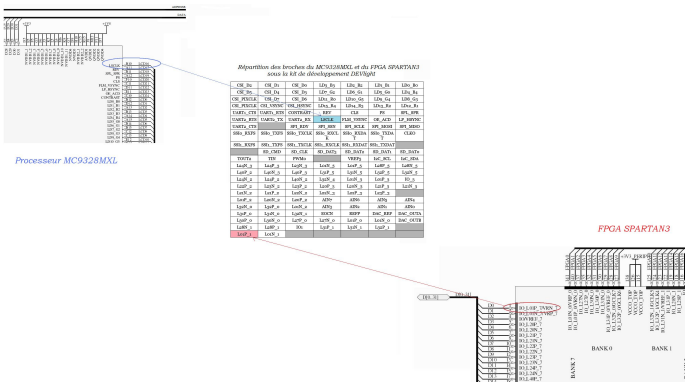
3 Premier pas sur l'APF9328 DEVlight

3.1 Le Brochage

Les deux principaux composants de l'apf9328 sont connectés par trois bus : un bus d'adresse, un bus de données et un bus de contrôle.
En effet, le FPGA est câblé de tel sorte que le MC9328MXL le voit comme un coprocesseur/composant externe. Les accès sont donc fait par le bus d'adresse et de donnée de l'i.MX lorsque l'on accède à l'adresse à laquelle est mappé le FPGA, ce dernier est donc mappé en mémoire du MC9328MXL entre les adresses $0x12000000$ et $0x12FFFFFF$, le choix d'écrire ou bien lire sur le FPGA se fait via la configuration des Chip Select au niveau du processeur.

Pour assurer la communication entre les deux composants, il faut, coté ARM9, écrire en mémoire à une adresse donné dans le processeur et coté FPGA décoder les adresses afin de lire le bus de donnée. Les IP communicantes avec le processeur doivent être présentée comme une suite de registres dans lesquels on peut lire et écrire.





Afin d'accéder depuis l'extérieur aux différentes broches des deux composants, il est nécessaire de se référer à la documentations technique du kit de développement DEVlight ainsi qu'aux schémas électriques de la carte apf9328, ces deux documentations étant fournies par le constructeur (cf. `http://www.armadeus.com/_downloads/apf9328/hardware/`). En effet, sur les schémas électriques fournies par les constructeur sont reportées les noms des broches des composants, cela nous facilite grandement le mappage, surtout pour les assignements des broches pour les IP du

FPGA : le numéro des broches pour l'assignement des entrées / sorties d'une IP sur Xilinx ISE sont directement reportées sur le composant (cf. chapitre 3.2 Allumage d'une LED). Prenons l'exemple de deux broches : admettons que l'on veuille brancher un composant externe sur la broche D0 (bit de poids faible du bus Data venant du MC9328MXL, câblé sur le FPGA), on doit se référer au schéma électrique de l'endroit où est mappé D0, il s'avère que D0 est mappé sur la broche `L01P_1` du FPGA, il suffit de se projeter dans la documentation du kit de développement DEVlight pour voir où est physiquement placé la broche `L01P_1`. Autre broche, une des broches du processeur, la broche `LSCLK` (nécessaire à la configuration d'un écran LCD) il suffit de la retrouver sur le kit de développement DEVlight, la figure suivante illustre ces propos :

3.2 Allumage d'une LED

3.2.1 Communication entre le processeur ARM9 et le FPGA SPARTAN3

Comme pour tout nouveau circuit embarqué que nous découvrons, notre premier objectif est de contrôler une diode électroluminescente (LED) connectée à une broche du FPGA. Il s'agit aussi d'une opportunité pour se familiariser avec les broches du FPGA ainsi que les ports de communication du MC9328MXL reportés sur la carte de développement DevLight V1, Cette expérience nous permettra aussi d'apprendre les mécanismes de programmation du FPGA et du processeur MC9328MXL depuis la version de GNU/Linux.

Dans un premier temps, Nous allons faire clignoter la LED par le seul intermédiaire du FPGA. Dans un second temps, nous ferons intervenir le MC9328MXL afin de contrôler la fréquence de clignotement de la LED, cette seconde expérience nous permettra de valider la communication entre le FPGA et le processeur.

Dans un premier temps, nous avons donc connecter la LED à deux broches du FPGA et implémenter un programme permettant de faire clignoter la LED à une fréquence de 1 Hz, nous avons synthétiser le code suivant sur Xilinx ISE :

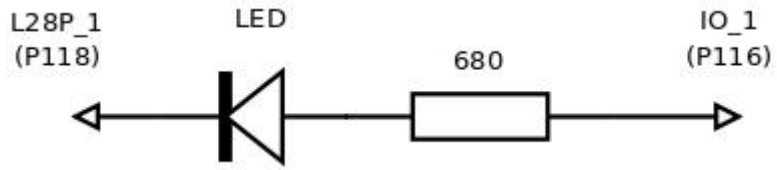
```
library IEEE;
 use IEEE.STD_LOGIC_1164.ALL;
 use IEEE.numeric_std.all;

 entity Clk_div_led is
```

```
    Port (
           Clk             : in  std_logic;
           led_cathode     : out std_logic;
           led_anode       : out std_logic
    );
end Clk_div_led;

architecture RTL of Clk_div_led is
  constant max_count : natural := 96000000;
begin

    led_cathode <= '0';

    -- compteur de 0 à max_count
    compteur : process(Clk)
        variable count : natural range 0 to max_count;
    begin
        if rising_edge(Clk) then
            if count < max_count/2 then
                led_anode    <='1';
                count := count + 1;
            elsif count < max_count then
                led_anode    <='0';
                count := count + 1;
            else
                count := 0;
                led_anode    <='1';
            end if;
        end if;
    end process compteur;
end RTL;
```

L'idée de ce code est de diviser la fréquence de l'apf9328 par elle-même afin d'obtenir 1 Hz sur la LED. Physiquement, la LED est connectée aux Broches `L28P_1` et `IO1` de la carte de développement DEVlight.

Ces broches sont raccordées respectivement aux broches P118 et P116 du FPGA Spartan3,

la résistance de 680 ohms permet de réguler le courant arrivant dans la diode.



Cet assignement de broches se fait via fichier d'extension ucf (User Constraints File). Ce fichier fait correspondre le numéro d'une broche à sa désignation. Pour connaitre le numéro de la broche, et sa position sur la carte DEVlight, il faut recourir à la documentation technique de l'apf9328 de celle de la DEVlight, dans notre cas, nous avons le ficher ucf suivant :

```
# Clock at 96MHz
NET "Clk" LOC = "P55";
NET "Clk" TNM_NET = "Clk";
TIMESPEC "TS_Clk" = PERIOD "Clk" 10 ns HIGH 50 %;
# LED
NET "led_cathode" LOC = "P118"| IOSTANDARD = LVCMOS33 ;
NET "led_anode"   LOC = "P116"| IOSTANDARD = LVCMOS33 ;
```

Remarque : il y a deux possibilités de créer un fichier ucf, soit via l'éditeur de texte de Xilinx ISE mis à disposition à cet effet ou bien utiliser Xilinx PlanAhead, qui est un logiciel distribué par Xilinx qui permet d'assigner les broches sur une architecture Xilinx de façon graphique.

Ces deux fichiers (VHD et UCF) permettent de générer les fichiers binaires nécessaires à la programmation du FPGA. La phase de génération de fichier de programmation (Generate Programming File) crée deux types de fichiers binaires : les fichier .bit et .bin. Ces deux fichiers permettent de programmer le FPGA, à la différence que le fichier .bit permet de programmer le FPGA d'une façon permanente : le FPGA reste programmé même au rebootage de l'apf9328.

L'implantation du ficher binaire, `Composant_VHDL.bin`, dans le FPGA se fait sur la console GNU/Linux de la carte apf9328 par la commande suivante :

```
dd if=Composant_VHDL.bin of=/dev/fpgaloader
```

Pour que cette commande fonctionne, il est primordial de charger un module noyau qui prend en compte la commande précédente :

```
modprobe fpgaloader
```

Fpgaloader est un driver, développé par l'équipe ArmadeusSystem, qui permet de charger notre IP dans le FPGA depuis la console GNU/Linux de l'apf9328. La commande « modprobe » permet donc de charger un module du noyau. Le noyau nous retourne le message «fpgaloader v0.9 » ce qui nous annonce que le module à bien été chargé. La première version du logiciel Armadeus que nous avions à notre disposition n'avait pas le bon module fpgaloader adequate à notre carte.

Une fois cette première étape validée, nous sommes passé à la deuxième étape qui consiste à étudier la communication entre le FPGA. Nous rappelons que le FPGA et le processeur MC9328MXL est relié au FPGA par trois bus, un de données, un d'adresse et un de contrôle. L'idée, pour bien comprendre la communication entre le FPGA et le MC9328MXL, est de faire transiter une information sur le bus de donnée depuis le MC9328MXL vers le FPGA, nous avons repris et modifié l'IP précédente pour cela.

Nous avons rajouté des process sensibles aux signaux venant du MC9328MXL :

```
process (RD, WR, Clk, CS, Addr)

end process;
```

Il scrute ainsi les événements sur les signaux de contrôle CS (chip select) et WR (mode écriture), ainsi que le bus d'adresses du processeur Addr. Afin de voir si le processeur veut communiquer avec le FPGA au moyen du bus de données Data, il est nécessaire d'avoir sur 0 sur WR et CS, actifs à l'état bas. La condition suivante

```
if WR = '0' and CS = '0' and Addr = "0000000000100" then--Linux ecrit
LED <= Data(0);
else LED <= '0';
```

L'état du bit de poids faible du bus de données (`LED<=Data(0);`) est recopie sur la broche contrôlant la LED. Ce qui nous permet, à une adresse donnée sur le bus Addr, de contrôler la led via le bus de données Data. Il ne faut pas oublier non plus de modifier le fichier ucf, il faut rajouter l'assignement des bus de données, d'adresse et de contrôle, ces trois bus etant mappé physiquement de facon permanente sur l'apf9328, les fichiers de contrainte à venir pour le FPGA, pour un projet incluant une communication avec le processeur, auront toujours les assignements suivants :

```
#PACE: Start of PACE I/O Pin Assignments

NET "Addr<0>"  LOC = "P21"  ;          NET "Data<3>"  LOC = "P5"  ;
NET "Addr<1>"  LOC = "P23"  ;          NET "Data<4>"  LOC = "P6"  ;
NET "Addr<2>"  LOC = "P24"  ;          NET "Data<5>"  LOC = "P7"  ;
NET "Addr<3>"  LOC = "P25"  ;          NET "Data<6>"  LOC = "P8"  ;
NET "Addr<4>"  LOC = "P26"  ;          NET "Data<7>"  LOC = "P10"  ;
NET "Addr<5>"  LOC = "P27"  ;          NET "Data<8>"  LOC = "P11"  ;
NET "Addr<6>"  LOC = "P28"  ;          NET "Data<9>"  LOC = "P12"  ;
NET "Addr<7>"  LOC = "P30"  ;          NET "Data<10>"  LOC = "P13"  ;
NET "Addr<8>"  LOC = "P31"  ;          NET "Data<11>"  LOC = "P14"  ;
NET "Addr<9>"  LOC = "P32"  ;          NET "Data<12>"  LOC = "P15"  ;
NET "Addr<10>" LOC = "P33";            NET "Data<13>"  LOC = "P17"  ;
NET "Addr<11>" LOC = "P35";            NET "Data<14>"  LOC = "P18"  ;
NET "Addr<12>" LOC = "P36";            NET "Data<15>"  LOC = "P20"  ;
NET "Data<0>"  LOC = "P1"  ;           NET "WR"  LOC = "P46"  ;
NET "Data<1>"  LOC = "P2"  ;           NET "RD"  LOC = "P47"  ;
NET "Data<2>"  LOC = "P4"  ;           NET "CS"  LOC = "P44"  ;
```

L'IP précédente à été pensée d'une façon a ce que, lorsque l'on envoi une valeur sur le bit de poids faible du bus de donnée à l'adresse 0x04, cela puisse contrôler l'allumage de la led.

Armadeus fournit un programme qui assure l'envoi de données à une adresse voulue, ce programme s'appelle fpgaregs.c. Cet exemple illustre la programmation sous GNU/Linux et utilise une fonction de mappage de la mémoire afin de placer les valeurs appropriées sur les bus d'adresse et de données, dans notre cas, il suffit de taper la commande suivante depuis la console GNU/Linux de l'apf9328 pour allumer la led :

```
fpgaregs 4 1
```

La commande suivante permet à l'inverse d'éteindre la led :

```
fpgaregs 4 0
```

IL est important de comprendre le fonctionnement du programme fpgaregs.c, ce dernier ne conviendrait pas pour des applications plus importantes, comprendre le processus de transfert de données entre le fpga et le processeur nous permettrait décrire un driver pour une application future.

Depuis l'espace utilisateur, le mappage de cette zone mémoire ainsi que son exploitation de se fait de la façon suivante :

```
int MEM = open("/dev/mem", O_RDWR | O_SYNC);
```

MEM est ce que l'on appelle en programmation en C sous GNU/Linux un fichier de description. Chaque ouverture de device se fait par l'intermédiaire d'un fichier de description. Dans notre ca, MEM permet d'ouvrir un device de type mémoire (/dev/mem) ayant les propriété de lecture / ecriture.

```
mappage_FPGA = mmap(0, MAP_SIZE=4096, PROT_READ | PROT_WRITE,\
MAP_SHARED, MEM, FPGA_BASE_ADDR=0x12000000);
```

Cette commande ci permet le mappage en question. La fonction mmap demande la projection en mémoire de `MAP_SIZE` octets commençant à la position `FPGA_BASE_ADDR` (adresse à laquelle le FPGA est mappée avec le processeur) depuis le fichier de description MEM. Les paramètres `PROT_READ` et `PROT_WRITE` sont des options de protection de la zone mémoire, elles doivent etre en accords avec le fichier de description MEM.

```
*(u16*) (mappage_FPGA + AddrxDI) = DataxD;
```

Cette commande permet ainsi une ecriture de la donnée, transitant sur le bus Data, en mémoire. A l'inverse, une lecture de la mémoire pour la placer sur le bus Data donne ceci :

```
DataxD = *(u16*) (mappage_FPGA + AddrxDI);
```

Remarque, La même action peut s'obtienir depuis un module noyau, sauf que le mappage de la mémoire se fait via la fonction ioremap :

```
mappage_FPGA = (void *) ioremap(FPGA_BASE_ADDR, 4096) ;
*(u16*) (mappage_FPGA + AddrxDI) = ; // ecriture
DataxD = *(u16*) (mappage_FPGA + AddrxDI); // lecture
```

4 Présentation du capteur CMOS OV7620

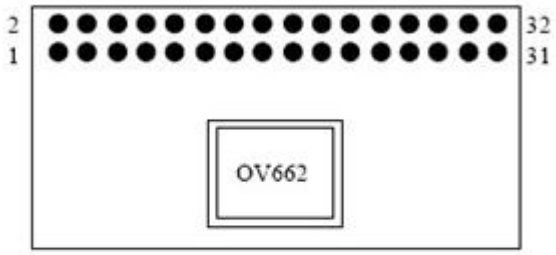
4.1 le capteur CMOS OV7620

Le capteur OV7620 est un capteur CMOS distribué par Lextronic, il est monté sur un circuit imprimé sous la nomenclature C38A. D'une résolution de 330 000 pixels (640x480), l'OV7620 n'est pas concurrent des capteurs montés sur les appareils photo vendu actuellement sur le marché mais il est suffisant pour se familiariser avec les concepts de base de l'acquisition d'images.

Une première étape, avant d'afficher les images sur un écran, serait une restitution d'une image sur un fichier adéquate au format graphique portable pixmap (fichier ppm).

Techniquement, cette mini caméra comporte 32 broches, comme le décrit la figure ci-dessous. La configuration de base de l'OV7620 offre les données de la luminance sous 256 niveaux de gris différents sur le bus Y ainsi que la chrominance sur 256 niveaux différents sur le bus UV (configuration 16 bits YUV).

1~8	Y0~Y7	Digital output Y Bus.
9	PWDN	Power down mode
10	RST	Reset
11	SDA	I^2C Serial data
12	FODD	Odd Field flag
13	SCL	I^2C Serial clock input
14	HREF	Horizontal window reference output
15	AGND	Analog Ground
16	VSYN	Vertical Sync output
17	AGND	Analog Ground
18	PCLK	Pixel clock output
19	EXCLK	External Clock input (remove crystal)
20	VCC	Power Supply 5VDC
21	AGND	Analog Ground
22	VCC	Power Supply 5VDC
23~30	UV0-UV7	Digital output UV bus.
31	GND	Common ground
32	VTO	Video Analog Output (75Ω monochrome)



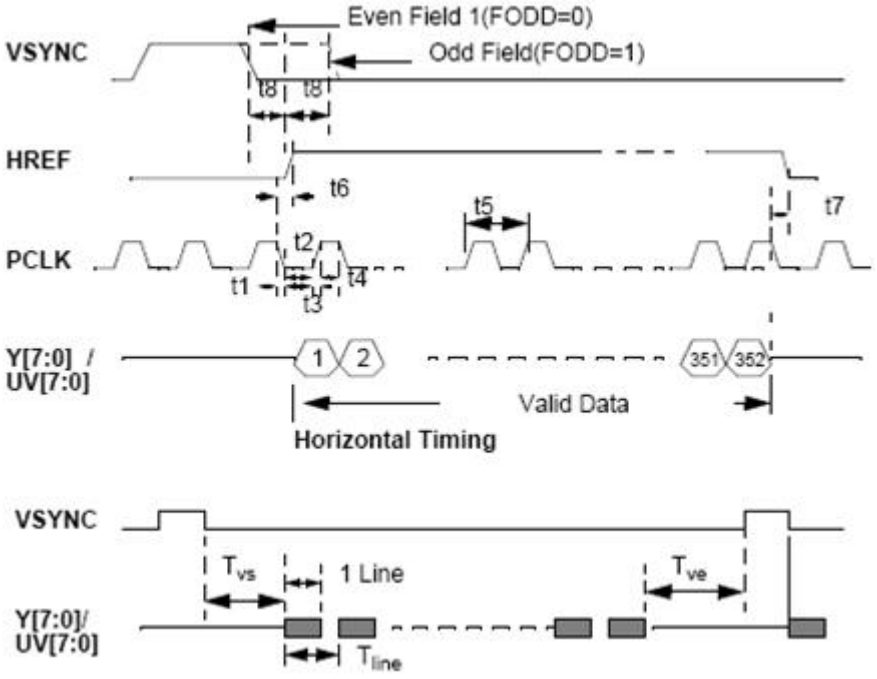
PCB Layout (Top side)

4.1.1 Principe de fonctionnement

Le principe de fonctionnement de cette caméra se base sur trois signaux : VSYN, Href et PCLK.

L'horloge PCLK passe à l'état haut lorsque les données Y et UV d'un pixel sont disponibles sur leurs bus de sortie respectifs. HREF est à l'état haut durant toute une ligne, soit 640 pixels.

La sortie Vsync passe à l'état haut pendant un certain temps lorsque les données pour les 480 lignes d'une image ont été envoyée par la caméra. Les timings suivants illustrent le principe de fonctionnement du capteur CMOS.



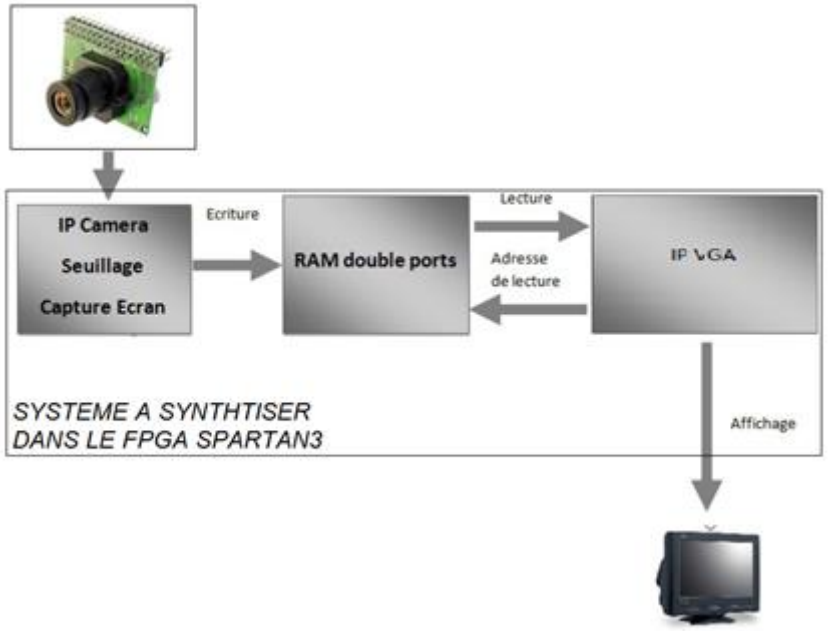
Nous désirons ainsi aborder une application plus complexe que le contrôle d'une LED, l'interfaçage de l'OV7620 avec la carte apf9328DEVlight serait un application qui serait une belle illustration de l'utilisation de la carte apf9328.
Nous avons abordé deux méthodes afin de récupérer les images du capteur OV7620. La première est la méthode vu en cours cette année sur le principe de synthétisation d'une mémoire tampon dans le FPGA afin de stocker l'image, cette méthode est une méthode purement hardware qui ne fait pas appel au processeur. Le but serait de stocker dans une RAM les valeurs de ces pixels afin de restituer une image sur un écran VGA.

La deuxième méthode fait intervenir le processeur couplé au FPGA, l'objectif d'utiliser un FPGA comme coprocesseur dédié à cette tache est de limiter le temps d'occupation du processeur pour l'acquisition d'images par une sollicitation uniquement lorsque la RAM synthétisée dans le FPGA est pleine. Nous allons décrire ces deux méthodes en détail.

5 Travaux proposés pour l'interface

5.1 Une solution FPGA seul

La première méthode pour récupérer les pixels du capteur CMOS est une méthode qui utilise uniquement le FPGA. Le principe de la reconstitution d'image est celle déjà connu : l'idée est de générer une mémoire tampon, une mémoire tampon suffisamment « grande » afin de pouvoir stocker une image, soit les 326 688 pixels de l'OV7620.
Si on se fit à la configuration de base de ce capteur CMOS, l'OV7620 nous délivre la chrominance sur le bus UV et la luminance sur le bus Y, chacun de ses bus étant codés sur 8 bits, l'une de nos première expérimentation serait de gérer uniquement le bus Y afin de restituer une image. Techniquement, l'idée serait de prendre comme mémoire tampon un mémoire de type RAM à deux ports : le premier port, composé de deux bus (adresses, données) et d'une entrée recevant une horloge, serait destiné pour la caméra et le second pour un éventuel écran VGA qui permettrait ainsi d'afficher l'image.



Ce système se compose de deux IP : la première IP gère les signaux de synchronisation de la camera Href, Vsync et l'horloge Pclk, elle écrit en RAM seulement lorsque Href est à l'état haut et remet la RAM à zéro (réinitialise le bus d'adresse) lorsque Vsync passe à l'état haut.

La seconde IP est une IP qui gère la norme d'affichage VGA, que cette norme affiche à une fréquence de balayage de 25 MHz, le capteur CMOS envoie ses données à une fréquence de 27 MHz, il n'est donc pas possible d'afficher une image « directement » sur un écran, d'où l'utilité de stocker l'image (phase d'écriture) dans une mémoire tampon à la fréquence Pclk (pixel clock) et de récupérer cette image (phase de lecture) à une fréquence qui celle de la norme VGA. La fréquence nécessaire pour la norme VGA peut être générée en divisant par 8 la fréquence du processeur ARM920T, qui est donc de 200 MHz.

Afin que ce système puisse fonctionner, il est nécessaire de placer entre l'IP de sortie et l'écran un convertisseur numérique analogique vidéo, notre choix technologique s'est tourné vers le TDA8702, conçu par Philips Semiconductor. Ce convertisseur ne convertie que 8 bits en un signal analogique pour l'écran soit qu'une seule couleur, nous avons donc besoin de trois composant comme celui-ci afin de convertir les trois couleurs RGB (rouge, vert, bleu) pour l'écran.

SYMBOL	PIN	DESCRIPTION
REF	1	voltage reference (decoupling)
AGND	2	analog ground
D2	3	data input; bit 2
D3	4	data input; bit 3
CLK	5	clock input
DGND	6	digital ground
D7	7	data input; bit 7
D6	8	data input; bit 6
D5	9	data input; bit 5
D4	10	data input; bit 4
D1	11	data input; bit 1
D0	12	data input; bit 0
V_{CCD}	13	positive supply voltage for digital circuits (+5 V)
V_{OUT}	14	analog voltage output
$V_{\overline{OUT}}$	15	complementary analog voltage output
V_{CCA}	16	positive supply voltage for analog circuits (+5 V)

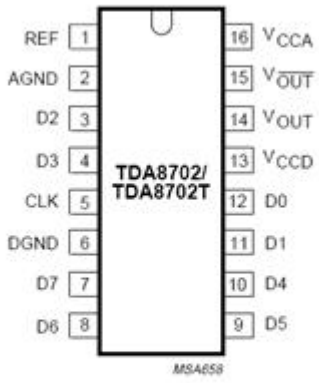


Fig.2 Pin configuration.

Notre choix technologique s'est tourné vers ce composant notamment car il avait une bande passante de 30 MHz, cette bande passante couvre la fréquence de lecture de la RAM, ce qui nous permet de dire que ce composant fonctionnera normalement. D'autre part, le TDA8702 s'alimente en 5V tout comme l'apf9328, il y a donc aucun souci d'alimentation.

Afin de stocker les 326 668 pixels d'une image, il est nécessaire de synthétiser une RAM de 326 668 octets afin de stocker la luminance. Techniquement, il est nécessaire d'avoir une RAM ayant un bus d'adresse pouvant compter jusqu'à 326 688 soit un bus de 19 bits, L'idée est donc de générer une RAM à deux ports ayant des bus d'adresse de 19 bits et un bus de données de 8 bits, la RAM deux ports est disponible dans les bibliothèques Xilinx et paramétrable via le module CORE Generartor & Architecture Wizard sous Xilinx ISE. Il est cependant impossible de générer une telle IP sur le Spartan3 de part le fait que ce dernier ne soit composée que de 200 000 portes. Techniquement, une porte est consommée générer une unité de stockage soit un octets, ainsi, la RAM maximale synthétisable dans une telle architecture serait une RAM de 200 Ko, ce qui remplirait entièrement le FPGA et le rendrait donc inutilisable pour notre système.

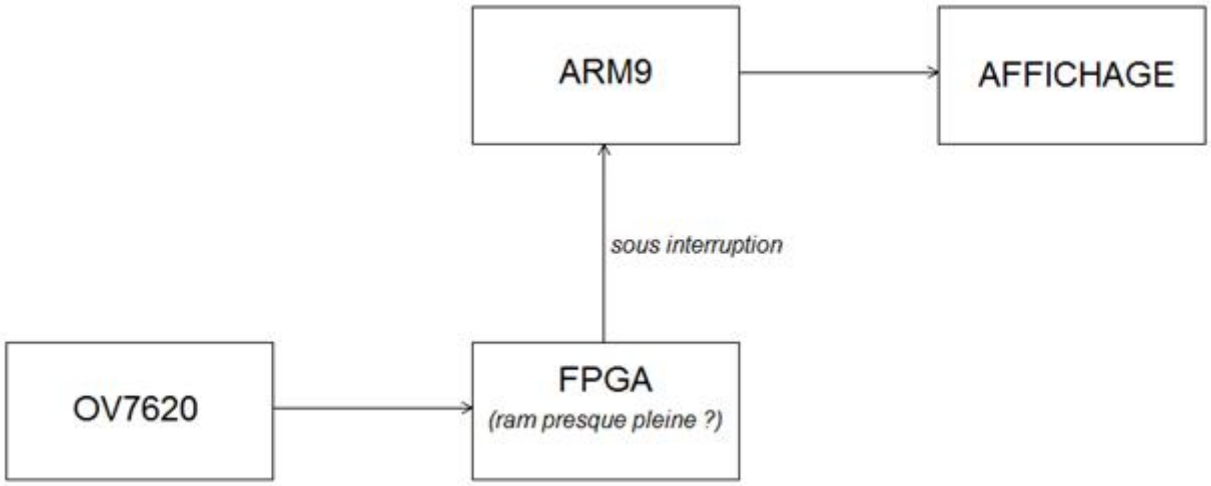
La solution serait donc de prendre une RAM externe de 330 Ko capable de gérer fréquence pour l'écriture et la lecture. Cette méthode demanderait un composant externe supplémentaire d'autant plus qu'elle ne permettrait de stocker que la luminance, pour la chrominance, il faudrait une seconde RAM deux ports ce qui rendrait cette méthode pas très pratique et peu rentable.

5.2 Une solution processeur et FPGA comme co-processeur

Une méthode moins onéreuse et plus lucrative en terme de connaissance que celle présentée précédemment peut être mise en oeuvre sur l'apf9328 afin de récupérer une image du capteur CMOS OV7620. Cette méthode fait appel au processeur MC9328MXL. Cette méthode est beaucoup plus intéressante d'un point de vue personnel étant donné que pour élaborer cette méthode, il est nécessaire de programmer le MC9328MXL. Le MC9328MXL se programme de deux façons différentes : soit à un niveau bas, en passant donc par les registres internes du processeur, soit à un niveau haut en faisant appel aux différentes fonction du système embarqué installé sur le MC9328MXL, nous allons ainsi décrire la méthode retenue pour récupérer une image et en même temps, nous allons voir le méthode à suivre pour programmer le processeur de notre système embarqué d'un point de vue « haut niveau de programmation ».

L'idée maintenue pour récupérer les images du capteur OV7620 est de passer par le processeur. La gestion des signaux du capteur CMOS se fera toujours par l'intermédiaire du FPGA Spartan3 , nous maintenons l'idée de synthétiser une RAM dans le FPGA, mais une RAM de petite taille pour les raisons que nous avons cités précédemment. Cette RAM aura pour objectif l'idée de mémoriser une partie de l'image, le processeur interviendra lorsque la RAM est presque pleine : le FPGA déclenchera un processus d'interruption au processeur qui permettra de « vider » la RAM

dans le processeur. Le processeur. Le processeur s'avisera d'afficher l'image.



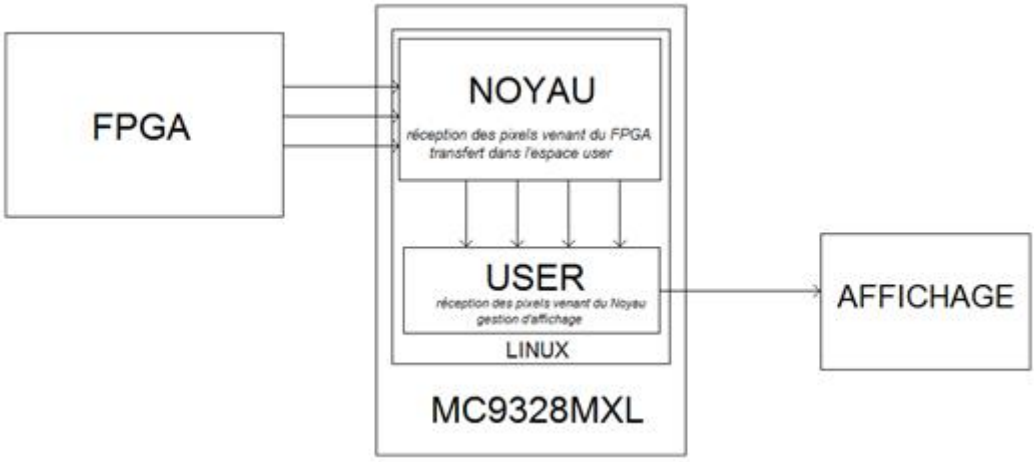
Comme cela a été dit précédemment, l'interruption pour le transfert des pixels du FPGA vers le processeur est déclenché lorsque la RAM est presque pleine. On ne peut pas attendre que la RAM soit entièrement pleine pour la vider étant donné que le FPGA écrit constamment dedans. Attendre ainsi la dernière adresse de la RAM pour la vider nous permettrait de récupérer les premiers pixels écrits en début de RAM bien que la fréquence de transfert vers le processeur soit supérieur à la fréquence de d'écriture de la RAM. Nous déclenchons donc l'interruption à une adresse donnée de la RAM : Le FPGA déclenche une interruption sur la ligne d'interruption TIM1 du processeur soit à une adresse de la RAM supérieure à 0x6EB, soit lorsque l'image est complète.

La valeur 0x6EB à été calculée d'une façon à ce que le temps de lecture corresponde exactement au temps qu'il faut à la caméra pour finir de remplir la mémoire, le calcul est détaillé dans le chapitre concernant la partie VHDL. Le signal TIM1 étant relié à la broche du FPGA sur laquelle est branchée la ligne d'interruption du processeur MC9328MXL.

Une fois arrivé dans le processeur, les pixels sont gérés par les deux parties du système d'exploitation : noyau et user. Les parties « user » et « noyau » ont deux fonctionnalités bien distincts : la partie chargée de récupérer les pixels venant de la RAM du FPGA est le noyau.

La partie « user » permet l'affichage de l'image restituée et, si besoin est, de configurer la caméra. Il y a un mécanisme de transfert à comprendre entre les deux parties, ayant pas d'expérience dans le domaine, nous nous sommes appuyé les travaux effectués par une équipe de recherche d'un laboratoire de L'ENSAMM de Besançon. Cette méthode limite le temps d'occupation du processeur puisque celui-ci n'est sollicité que chaque fois que la RAM du FPGA est presque pleine, elle est moins onéreuses et plus facile à embarquer que la première méthode étant donné que nous n'avons pas de

composant externe (RAM externe, convertisseur video) .
Et, d'un point de vue personnel, Cette méthode est d'autant plus lucrative en terme de connaissance, étant donné que je n'es pas de connaissance et d'expérience en programmation à haut niveau d'un processeur.

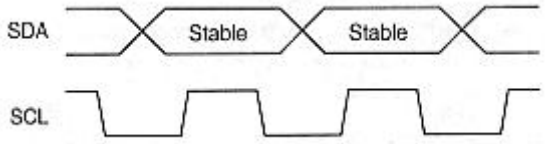


5.2.1 Rappel du fonctionnement du bus I^2C

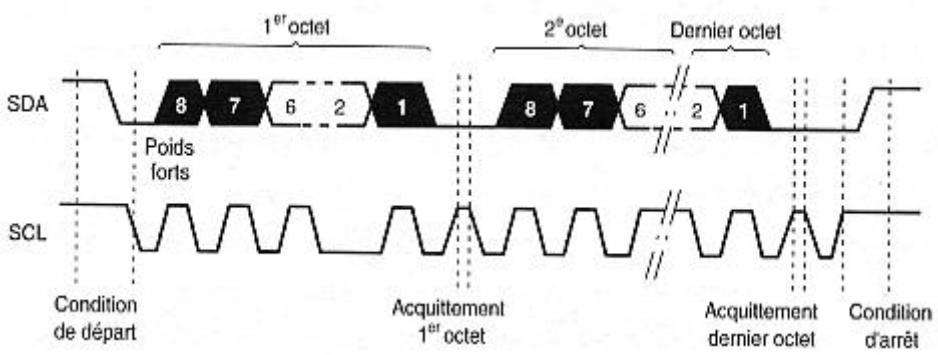
Afin de configurer les registres de la caméra, il est nécessaire de passer par le bus I^2C, étant donné que nous somme novice et que nos connaissances sont faibles concernant la théorie de ce bus série, il nous a semblé nécessaire de faire une recherche sur le bus I^2C, nous avons synthétisé cette recherche : Le bus I^2C, dont le sigle signifie Inter Integrated Circuit ce qui donne IIC et par contraction I^2C, a été proposé initialement par Philips mais est adopté de nos jours par de très nombreux fabricants. C'est un bus de communication de type série. Le bus I^2C qui n'utilise que deux lignes de signal permet à un certain nombre d'appareils d'échanger des informations sous forme série avec un débit pouvant atteindre 100 Kbits/s voir 400 Kbits/s pour les versions les plus récentes. Voici les points forts du bus I^2C :

- C'est un bus série bifilaire utilisant une ligne de données appelée SDA (Serial DAta) et une ligne d'horloge appelée SCL (Serial CLock) ;
- Les données peuvent être échangées dans les deux sens (lecture/écriture).
- le bus est multi-maîtres
- Chaque client I^2C dispose d'une adresse codée sur 7 bits. On peut donc connecter simultanément 128 clients (ou esclaves) d'adresses différentes sur le même bus.
- Un acquittement est généré pour chaque octet de donnée transféré.
- Le bus peut travailler à une vitesse maximum de 100 Kbits/s (ou 400 Kbits/s) étant entendu que son protocole permet de ralentir automatiquement l'équipement le plus rapide pour s'adapter à la vitesse de l'élément le plus lent, lors d'un transfert.

– Le nombre maximum d'esclave n'est limité que par la charge capacitive maximale du bus qui peut être de 400 pF. Ce nombre ne dépend donc que de la technologie des circuits et du mode câblage employés ;
– Les niveaux électriques permettent l'utilisation de circuits en technologies CMOS, NMOS ou TTL.
– Cette figure résume le principe fondamental d'un transfert à savoir : une donnée n'est considérée comme valide sur le bus que lorsque le signal SCL est à l'état haut. L'émetteur doit donc positionner la donnée à émettre lorsque SCL est à l'état bas et la maintenir tant que SCL reste à l'état haut (nous verrons des chronogrammes plus précis dans un instant).

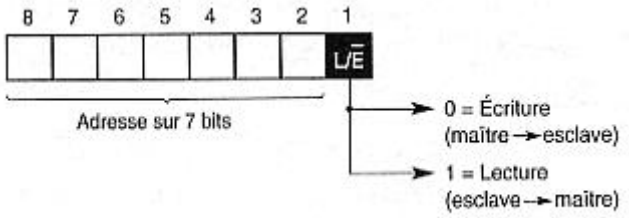


Une condition de départ est réalisée lorsque la ligne SDA passe du niveau haut au niveau bas alors que SCL est au niveau haut. Réciproquement, une condition d'arrêt est réalisée lorsque SDA passe du niveau bas au niveau haut alors que SCL est au niveau haut.



Bien que nous soyons en présence d'un bus série, les données sont envoyées par paquets de huit, même si un octet regroupe en fait huit bits indépendants. Le bit de poids fort est envoyé le premier. Chaque octet est suivi par un bit d'acquittement de la part du destinataire et l'ensemble du processus fonctionne comme indiqué sur la figure.
La figure ci-dessus montre tout d'abord une condition de départ, générée par le maître du bus à cet instant. Elle est suivie par le premier octet de données, poids forts en tête. Après le huitième bit, l'émetteur qui est aussi le maître dans ce cas, met sa ligne SDA au niveau haut c'est à dire au repos mais continue à générer l'horloge sur SCL. Pour acquitter l'octet, le récepteur doit alors forcer la ligne SDA au niveau bas pendant l'état haut de SCL qui correspond à cet acquittement.
Quand au format de transmission, comme cela à été dit précédemment, chaque périphérique esclave

à une adresse, cette adresse est codée sur 8bits. Cette figure montre le contenu du premier octet qui est toujours présent en début d'échange. Ses sept bits de poids forts contiennent l'adresse du destinataire du message ce qui autorise 128 combinaisons différentes



Le bit de poids faible indique si le maître va réaliser une lecture ou une écriture. Si ce bit est à zéro le maître va écrire dans l'esclave ou lui envoyer des données. S'il est à un, le maître va lire dans l'esclave c'est à dire que le maître va recevoir des données de l'esclave.
Dernière chose, les niveaux de tensions des signaux SCL et SDA sont régulé par les niveaux de tension de l'alimentation du processeur gérant le bus I^2C en question. Dans notre cas, il s'agit du processeur qui gère le bus I^2C, celui s'alimente en 3,3V. Ainsi les niveaux des signaux SDA et SCL sont à 3,3V. Dans le cas où un périphérique esclave est censé recevoir sur ses entrées des niveaux de tension différents de ceux de du bus I^2C, il est nécessaire de faire une adaptation de niveaux, dans un sens ou bien dans l'autre.

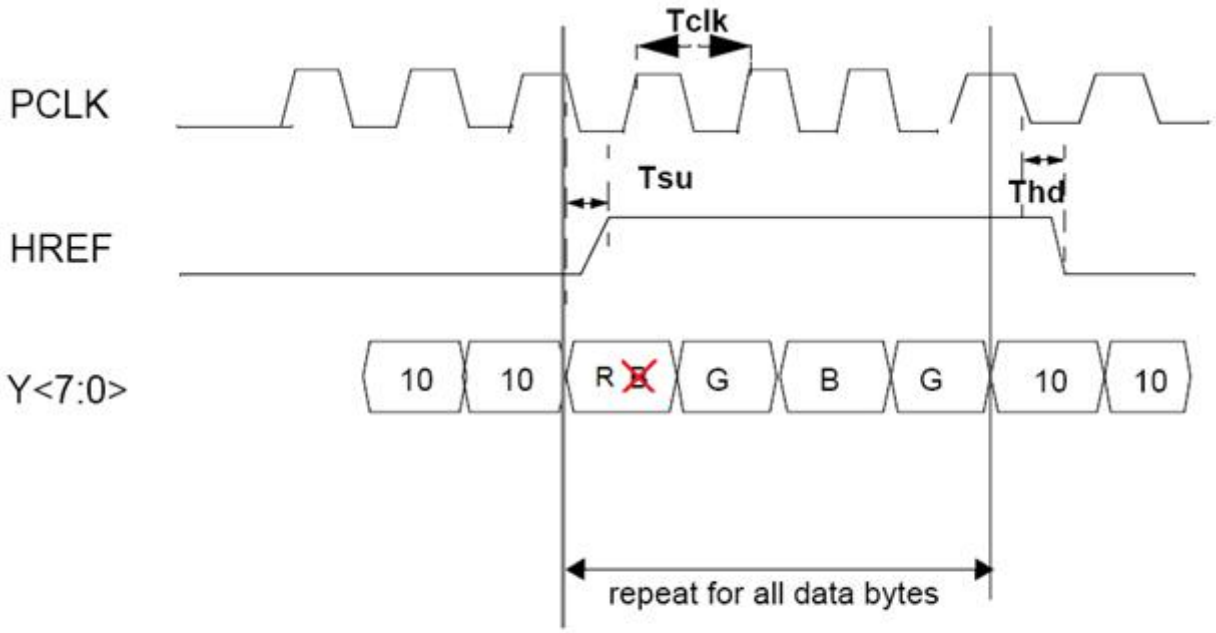
5.2.2 Configuration de L'OV7620 via le bus I2C

Afin de récupérer une image couleur , il est nécessaire, comme cela a été dit précédemment, de gérer les deux bus de la caméra, UV et Y, ce qui représente une gestion de 2 octets.

Mais, d'après la documentation de l'OV7620, il est possible de basculer la caméra en mode RGB , 8 bits (Red, Green, Blue) où la chrominance et la luminance sont réceptionnées sur un seul et unique bus, sur le bus Y ce qui représente une gestion numérique deux fois moins importante. Physiquement, cela représente huit connections à gérer, ce qui facilite d'autant plus, à long terme, la notion d'une éventuelle intégration du système. Cette configuration, envoie donc une valeur de couleur codée sur 8 bits à chaque front descendant de Pclk.

Il est important de remarquer des à présent qu'une erreur de documentation nous a été signalé par l'équipe du laboratoire FEMTO ST de Besançon concernant le mode 8 bits du capteur CMOS OV7620 que nous rectifions ici (voir figure ci-dessous).

En effet contrairement à ce qui semble être indiqué dans la documentation, la sortie en mode 8 bits est une alternance de valeurs RG et BG (et non une ligne complète de BG suivi d'une ligne complète de RG tel que nous interprétons la fig. 1.3 (deuxième figure) de la documentation technique de l'OV7620). Cette erreur à été confirmé de façon expérimentale par l'équipe de FEMTO ST, nous allons donc fier à leur correction.



Pixel Data 8 bit Timing
PCLK rising edge latch data bus

FIG 1.3 Pixel Data Bus (RGB Output)

La séquence envoyée par le capteur CMOS respecte plus ou moins la norme RGB : sur une image complète, cette norme impose 59% ?de vert, 11% ?de bleu et 31% de rouge étant donné que l'œil humain à une sensibilisé plus importante au vert qu'aux deux autres couleurs. Ce capteur envoie, sur une séquence de quatre pixels caméra (un pixel camera étant une valeur codée sur huit bits représentant une couleur), deux pixels codants le verts, un pixel codant le rouge, et un pixel codant le bleu. Ce qui donne, sur une image complète, 50% de vert, 25% de rouge et 25% de bleu.

Une telle configuration sur ce capteur se fait, comme cela à été dit précédemment, via le bus I2C la configuration des registres de la caméra se fait dans l'espace « user » du processeur. Il s'avère que Linux fourni un outils (device) permettant l'utilisation du périphérique I2C. Cet outils est disponible dans le dossier /dev du système d'exploitation Linux de L'apf9328, la compréhension de la

configuration de ce périphérique d'un point de vue logiciel nous permettra de comprendre la notion de programmation « haut niveau » du processeur MC9328MXL. Nous allons detailler les diefferentes étapes permettant la configuration de ce périphérique.
Il faut dans un premier temps admettre la notion de fichier sous Linux, en effet, ce système d'exploitation voit la plupart de ses outils comme des fichiers, des fichiers de descriptions (fd, file descriptor en anglais) qui assure la configuration du périphérique souhaité, ce fichier est une variable de type int, dans notre programme de configuration, nous l'avons appelé i2c. Ce fichier (variable) va servir dans un premier temps à ouvrir l'outils permettant l'utilisation du périphérique I2C dans un second temps, la configuration dynamique de cet outils. Pour cela, il est nécessaire de manier deux fonctions standards au système Linux : open(), ioctl() :
Open() : il s'agit de la première opération à effectuer sur le fichier. En effet avant de pouvoir effectuer la moindre action sur le driver il convient de demander l'autorisation au noyau. Par exemple si le driver I2C n'est pas préalablement chargé, le noyau vous refusera l'accès au fichier /dev/i2c0, ce qui nous donne donc dans notre programme :

```
i2c = open("/dev/i2c-0",O_RDWR);
```

Le paramètre `O_RDWR` permet de lire et écrire sur ce périphérique, dans notre cas, on aurait pu se contenter d'un paramètre de type`O_WR`(Output Write) étant donné que le capteur CMOS ne nécessite que des phases d'écriture.
Ioctl() : comme nous l'avons vu précédemment, le bus I2C permet d'avoir plusieurs périphériques connectés au bus, il convient donc de préciser l'adresse de l'esclave, autrement dit du capteur OV7620, sur le bus I2C. La fonction ioctl() sert ici à préciser d'une part l'adresse de destination de la caméra, d'autre part que cette camera est un esclave sur le bus I2C, notez la valeur hexadécimale 0x703 est précisé dans le manuel de l'outils I2C (man i2c-0 sous Linux) afin de « faire comprendre » au bus que la camera est son esclave. L'adresse de la caméra est 0x60, notez aussi que cette valeur n'est pas donnée par le constructeur dans la documentation de l'OV7620, elle à été trouvé dans divers codes, sans cette valeur, la caméra n'est pas configurable. Cette fonction donne donc en C, pour notre cas, ceci :

```
#define I2C_SLAVE  0x703
#define C38A 0x60

ioctl(i2c, I2C_SLAVE, C38A);
```

Une fois ce paramétrage effectué, il est nécessaire de configurer les registre de la caméra afin de la passer en RGB, ces registres sont notamment décrits dans la documentation technique de l'OV7620. Nous avons repris la fonction write(fd,*p,val) qui permet d'écrire une valeur « val »

dans le fichier associé au descripteur fd (dans notre cas, i2c) depuis le pointeur pointé par *p, nous l'avons modifié pour lui rajouter des options liées au support I2C, notamment cette fonction affiche un message d'erreur lorsque la configuration ne s'effectue physiquement sur le capteur. La fonction ioctl() nous retourne -1 dans le cas où elle a échouée et 0 si la fonction à réussi son travail, sachant cela, nous avons rajouté dans le corps de notre fonction d'écriture ceci :

```
if ( ioctl( fd, I2C_RDWR, &rdwr ) < 0 ){  // dans le cas ou la synchronisation
                                           // du registre n'à pas été effectuée
printf("Mauvaise synchronisation I2C \n");
return -1;
}

if ( ioctl( fd, I2C_RDWR, &rdwr ) == 0 ){ // dans le cas ou la synchronisation
                                        //du registre à été effectuée (voir man ioctl)
printf("OK \n");
return 0;
}
```

Nous en avons profité pour renommer cette fonction « `ecriture_registre`». Ainsi, pour configurer par exemple le registre 0x12 à la valeur 0x80 (ce qui crée un reset sur le capteur, voir la documentation en annexe) , il suffit d'écrire `ecriture_registre(i2c, 0x12, 0x80 );`
Afin de passer la caméra en mode RGB, 8bits, il suffit de modifier deux registres : les registres 0x12 et 0x13, les registres COMMON Controls. Le registre 0x12, d'après la documentation, permet de passer la caméra en RGB, codé sur 8 bits, il faut notamment passer le 3eme bit (COMMA3) à 1 afin d'avoir un mode RGB (raw data signal), ce bit restant à 0, la caméra serait en YUV, la configuration de base étant 0x24, elle devient 0x2C. De même pour le registre 0x13, celui-ci permet de passer la caméra d'un mode 16 bits à un mode 8 bits grâce à son 5eme bit (COMMB5) : en mettant ce bit a 1, la donnée est obtenu sur 8 bits, sur le bus Y donc, et les donnée couleurs (RGB) et chrominance sont ainsi multiplexé que ce bus (cf. p.30 de la documentation technique de l'OV7620), la valeur de base de ce registre étant 0x01, nous devons mettre 0x21 afin d'avoir le mode voulu.
En lisant plusieurs documents sur ce type de capteurs et pour en avoir parlé avec des gens ayant expérimenté leur interfaçage avant moi, il s'avère que la configuration I2C d'un tel capteur ne fonctionne pas si on ne reconfigure pas tous les registres, même si c'est dans le but de leur redéfinir leur valeur par default, d'autant plus qu'il faut respecter un certain ordre, une certaine séquence bien précise, qui n'est pas fournie par le fabricant. Nous avons trouvé cette séquence à respecter dans divers code paramétrant le capteur OV7620 et nous avons modifié les registres 0x12 et 0x13 afin d'avoir le mode RGB, 8bits. Cette séquence est bien particulière d'autant plus que nous avons une redéfinition successive et à plusieurs reprises d'un même registre à des moments particulier,

notez que la version 100 Kpixels, la OV6620, n'impose pas une telle séquence : il est possible de modifier seul les registres voulus. Cette séquence est fournie en annexe de ce rapport.
Une fois ceci fait, la fonction `confguration_i2c()` est complète, il est notamment conseillé de finir par une fonction « close(i2c) » qui est la fonction opposée l'opération open. Il permet de signaler au noyau et au driver que l'application ne souhaite plus utiliser le périphérique I2C, bien que le noyau Linux le fera automatiquement.
Le support I2C est ensuite compilé dans le noyau fourni par défaut avec la carte Armadeus. La compilation de cette fonction se fait avec un cross-compiler spécifique aux cartes ArmadeusSystem, nommé « arm-linux-gcc », celui-ci génère les fichier.o nécessaire pour exécuter notre application sur notre carte, ce cross-compiler est disponible dans le chemin qui suit : `armadeus/buildroot/build_armv4vt/staging_dir/usr/bin/`.

Remarque : la compilation se fait depuis un ordinateur annexe, et le transfert des fichiers.o se fait via une commande « wget » qui permet le téléchargement avancé de fichiers sur des réseaux et sur Internet. Ainsi, en tapant, depuis la console de notre carte, `wget http://129.104.11.29/~haddad/confguration_i2c`, on reçoit le fichier.o nécessaire à l'exécution de notre programme, 129.104.11.29 étant l'adresse IP de notre ordinateur annexe. Depuis le répertoire courant, il suffit de saisir `./confguration_i2c` afin d'exécuter le programme, Il est important de noter que le programme ne s'exécute pas étant donné que nous n'avons pas le permissions d'exécution sur chaque fichier objet reçu sur la carte, il est nécessaire d'attribuer le droit d'exécution sur chaque fichiers reçu sur la carte, pour cela il faut utiliser la fonction chmod et donc saisir la commande suivante : `chmod +x configuration_i2` afin de changer les permissions d'accès, l'option « +x » autorise uniquement l'exécution (pas la lecture, ni d'écriture autorisée sur ce fichier).
Il s'avère notamment que la configuration des registres via l'I2C ne fonctionne pas, en effet le message « mauvaise synchronisation I2C » s'affiche à chaque tentative de configuration d'un registre, ce qui signifie que, soit la requête I2C sur ioctl() n'as pas fonctionné, soit le problème serait plutôt matériel à default d'être logiciel (voir les deux problèmes en même temps !).
Un Outils disponible sur GNU/Linux nous permet de vérifier si le souci vient du logiciel, il s'agit de l'outils I2cdetect. L'outils I2cdetect permet notamment de déterminer quel sont les périphériques esclaves placés sur le bus I2C, en saisissant la commande i2cdetect 0, cet outil scanne le bus I2C de numéro 0. Tel que le montre la figure suivante, on voit qu'un périphérique esclave d'adresse 0x38 est connecté sur le bus, il s'agit là du DAC de la carte fourni (cf. fig xxxx brochage). Si le système d'exploitation avait détecté un périphérique esclave d'adresse 0x60 (l'OV7620), nous aurions vu 60 apparaitre à sa position. Etant donnée que ce n'est pas le cas, on en conclut que le capteur CMOS n'a pas été détecté.

```
# ./i2cdetect 0
```

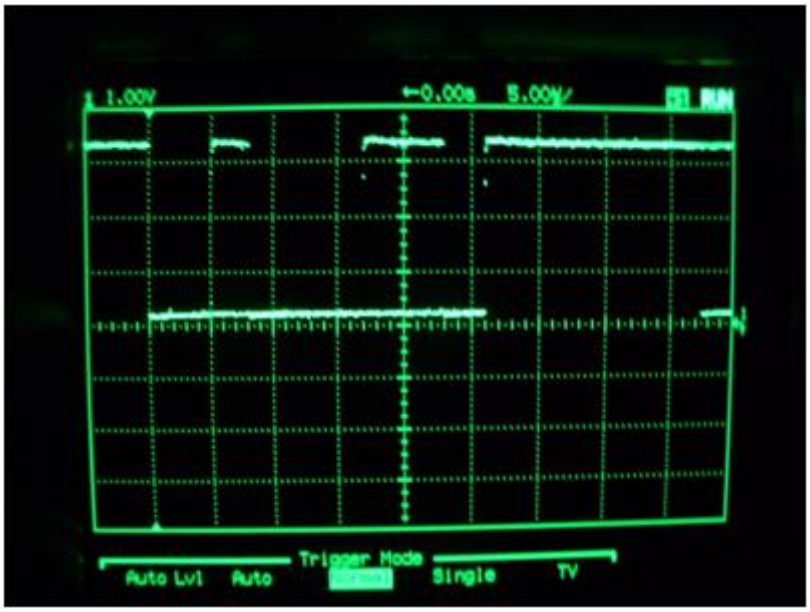
```
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- 38 -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
#
```

Un autre outils nommé i2cset permet, quant a lui, de configurer les registre « à la main » un par un, en saisissant ainsi i2cset 0 0x60 0x12 0x80 (configuration du registre 12 à la valeur 80, pour une adresse esclave 0x60 sur le port I2C-0), la console nous affiche un message d'erreur annonçant que la configuration ne peut être faite étant donné qu'il ne trouve pas d'esclave à l'adresse 0x60.

```
# ./i2cset 0 0x60 0x12 0x80
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will write to device file /dev/i2c-0, chip address 0x60, data address
0x12, data 0x80, mode byte.
Continue? [Y/n] y
Error: Write failed
#
```

Nous avons pus notamment emprunter un oscilloscope à un laboratoire voisin afin de verifier si, lors du lancement de notre programme, l'horloge SCL était bien présente. Lors du lancement, nous visualisons le signal suivant :

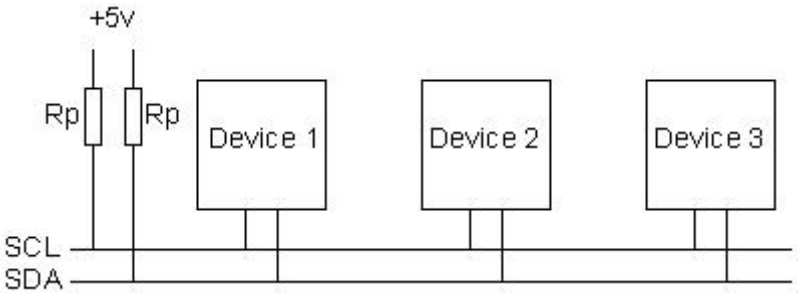
Ce qui nous permet donc d'en déduire que l'ouverture du périphérique */dev/i2c-0* à bien été effectué : SCL a une fréquence de 100 KHz (100 Kbits/s) pendant la phase de configuration du DAC, au delà de cette phase, le signal reste à 3,3V, valeur d'alimentation du processeur. De même pour SDA, la donnée lue correspond à celle du DAC :



Le fait que le signal reste à 3,3V par la suite nous permet de nous poser une question : le capteur OV7620 s'alimente en 5V, faudrait il des niveaux logiques à 5V afin que la configuration I2C puisse se faire ?

En effet, si on se réfère au chapitre « rappels sur le bus I2C » il est dit notamment qu'il est primordial que tous les niveaux logiques d'une communication I2C soient équivalent, le cas échéant,

la communication en se fera pas. Une éventuelle piste serait donc de mettre soit des résistances de pull up sur l'I2C afin de passe les signaux de 3,3V à 5V, soit de prendre un composant « adaptateur de niveaux » spécifique au bus I2C qui permet également de passer le bus I2C d'un niveau à un autre souhaité.



Des résistances de 10Kohms peuvent être placé sur Rp afin d'assurer cette adaptation. Faute de temps et de moyen, nous laissons la configuration YUV,16bits au dépend de passer le capteur en RGB,8bits. Nous prendrons cependant que la luminance (Y) afin de vérifier si les différentes parties composant notre projet fonctionne.

5.2.3 Traitement des données brutes

La partie user, comme cela à été dit précédemment, permet de reconstruire l'image dans un fichier de type ppm, d'une part, et d'autre part, il lance une requête de réception d'image au noyau.

Mais avant de pouvoir reconstruire l'image, il faut d'abord savoir réceptionner les pixels mémorisés dans le noyau. Pour cela, le module crée un périphérique qui permet le transfert entre le module noyau et l'espace user. Nous avons nommé ce périphérique « mydriver », ce périphérique fera donc office de « charnière » entre les deux espaces et donc il va permettre de passer les pixels. Afin de créer cette charnière, il suffit d'ouvrir ce périphérique via un fichier de description avec la commande open(), comme cela à été fait avec la configuration de l'I2C.

```
int file = open("/dev/mydriver", O_RDWR);
```

Nous déclarons également un tableau de type char (un octet, la taille d'un pixel) réceptionnant les pixels et de longueur 326 688 (taille complète d'une image). Nous ouvrons aussi un fichier de type ppm via un fichier de type File qui pointe sur une fichier ppm,la commande pour ouvrir un fichier est la fonction fopen(), « w » nous permet de préciser qu'il s'agira d'effectuer seulement une écriture :

```
FILE *fppm;
fppm=fopen("/tmp/image.ppm", "w");
```

Comme cela à été dit précédemment, c'est ce programme qui lance une requête au noyau afin que ce dernier active le transfert des pixels depuis le FPGA, cette requête est faite via la commande suivante :

```
ioctl(file, STARTC, 0);
```

Coté, noyau, nous avons la constante STARTC qui configure la requete via le bout de code suivant :

```
case STARTC :
{

if(arg==1){
Partie_FPGA=0;
enable_irq(IRQ_GPIOA(1));}
else
{disable_irq(IRQ_GPIOA(1));

ecriture_fpga(REQ_I_ADDR, arg);  // lancement de la requête, cette
                                 // commande écrit 1 sur le bus de donnée
                                 // à l'adresse FPGA_ADDR + REQ_I_ADDR.
            break;
}
```

La constante `REQ_I_ADDR` est initialisée de façon arbitraire à l'adresse 0x04. Puis, coté FPGA, cette adresse est décodée de cette façon :

```
else if WRxBI = '0' and CSxBI = '0' then --Linux veut écrire sur le FPGA
  case AddrxDI is

when "0000000000100" =>--0x04
Request_Image<=Data(0); //
```

Une fois ceci fait, nous avons un certain nombre de configuration à faire afin que le transfert de pixels depuis le noyau vers l'espace user puisse se faire, pour cette configuration, nous allons nous

servir du fichier de description « file » de notre périphérique charnière « mydriver », ce transfert est crée principalement grâce à deux fonctions particulier :

```
ioctl(file,IMG,NULL);
```

Cette fonction permet de se connecter dans le noyau : elle fait appel à la fonction « `mon_ioctl` » dans le noyau, pour IMG, on affiche un message que la connexion est bien faite .

```
read(file, &Mon_Image, img_size);
```

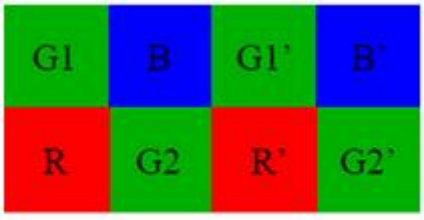
Cette fonction copie ce qui est contenu dans img dans le tableau `Mon_Image`, la variable img est une variable du noyau réceptionnant les pixels. Le contenu de cette variable est transféré dans l'espace « User », le tableau `Mon_Image` récupère les données par l'intermédiaire de la fonction read().

Une fois reçue, dans le tableau `Mon_Image`, on peut faire ce que l'on veut de l'image : l'afficher, effectuer un traitement particulier, appliquer un filtre... Nous décidons de l'afficher dans un fichier de type portable pixmap (ppm), ce fichier permet une reconstruction rapide et simple de l'image. Il s'agit là d'un simple fichier texte contenant la valeur de chaque pixel. Nous ecrivons dans ce fichier via la fonction suivante :

```
for(j=0;j<480;j++){
   for(i=0;i<640;i++){

   fprintf(fppm %u\n", Mon_Image[j*664+i]);
   }
 }
```

Le fichier est ainsi crée dans le dossier /tmp de la carte. Ce fichier est transferé sur notre session et le fichier .ppm est reconstruit grâce à l'outil GNU/Linux Gthumb (gthumb image.ppm).

Il faut savoir que l'OV7620, d'un point de vue d'une représentation spatiale, respecte ce que l'on appelle un filtrage de Bayer. Sur une partie de l'image, elle envoie ceci :



organisation spatiale

Dans le cas où le capteur est configuré en RGB, 8 bits, les pixels ci-dessus sont envoyée de façon séquentielle sur le bus Y Ainsi, pour l'organisation si dessus, le capteur nous envoie la séquence suivante :

Sachant que la caméra possède un filtrage de Bayer, il est nécessaire de refaire une réorganisation des pixels : quatre pixels capteur (rg1bg2), correspondent à deux pixels image (rg1b, rg2b). Ce qui nous permet ainsi d'écrire la fonction suivante dans notre fichier .ppm :

```
Char Mon_Image [img_size*2];

  for(j=0;j<480;j++){
    for(i=0;i<640;i++){

fprintf(fppm,"%u %u %u\n", Mon_Image[j*640*2+4*i+2], Mon_Image[j*640*2+4*i+1],
Mon_Image[j*640*2+4*i]);

//création d'un pixel image

fprintf(fppm,"%u %u %u\n", Mon_Image[j*640*2+4*i+2], Mon_Image[j*640*2+4*i+3],
Mon_Image[j*640*2+4*i]);

//création d'un second pixel image

    }
  }
```

Notez que pour restituer la résolution de la camera sur une image réelle, soit 640x480, avec ce mode, il est nécessaire de percevoir deux fois plus de pixels (vu qu'avec 4 pixels caméra on fait un pixel pour l'image) de la caméra, d'où une taille deux fois plus importante pour le tableau contenant l'image : pour faire une ligne de 640 pixels, il faut percevoir 1280 pixels de la caméra pour créer cette ligne, d'où l'offset 640*2 présent dans notre fonction.

5.2.4 le module Noyau

Le module noyau permet de réceptionner les pixels venants de l'OV7620 lorsqu'une interruption est déclenchée sur TIM1. J'ai aucune notion en programmation d'un module noyau, au

travers cette application, j'ai appris à créer un module noyau. Nous avons trouvé un document très intéressant sur la création de drivers sur Linux (donné en annexe). Il faut tout d'abord savoir qu'un driver en C ne contient pas de main, il est composé de deux fonctions principales : `module_init` et `module_exit`.
La fonction `module_init()` doit s'occuper de préparer le terrain pour l'utilisation de notre module (allocation mémoire, initialisation matérielle...). La fonction `module_exit()` doit quant à elle défaire ce qui a été fait par la fonction `module_init()` . Un driver Linux est aussi composé des fonctions de base qui permettent l'écriture, la lecture au sein du driver, (`my_read_function, my_write_function`) mais aussi de fonctions permettant l'initialisation du driver pour, par exemple une éventuelle allocation mémoire ou encore une fonction de fermeture (qui desalloue). Ces fonctions sont appelées lors des appels systèmes (open, read...) du côté utilisateur. Elles sont données dans une structure :

```
static struct file_operations fops = {
read : my_read_function,
write : my_write_function,
open : my_open_function,
release : my_release_function
};
```

Ainsi, par exemple, la sollicitation de la fonction read() du coté utilisateur fait appel à la fonction `my_read_function` coté noyau.
Il faut associer ce driver au fichier « mydriver » ouvert dans le programme utilisateur (cf. chapitre « traitement des données brutes ») pour cela, nous devons le préciser dans la fonction `module_init()` avec la commande suivante :

```
int ret;
ret = register_chrdev(major, "mydriver", &fops);
```

La fonction `register_chrdev()` permet d'associer les fonctions de base du module noyau à notre fichier charnière « mydriver », ainsi nous pouvons écrire et lire au travers de « mydriver ». fops est un pointeur vers la structure fops décrite précédemment. La variable major est le nombre majeur, déclaré au prealable, ce nombre majeur a pour but d'identifier notre fichier charniere. Dans notre cas, nous avons pris 250 pour le nombre majeur.
La fonction `module_init()` permet aussi d'allouer une mémoire tampon à partir de 0x12000000, adresse à laquelle est mappée le FPGA, afin d'effectuer les transferts :

```
FPGA_Addr = (char *)ioremap(0x12000000,4096) ;
```

On en profite aussi pour configurer la fonction d'interruption :

```
initialisation_interrupt();
```

Initialisation de cette interruption se fait principalement grace au gestionnaire d'interruptions `request_irq` via cette ligne de code :

```
request_irq(IRQ_GPIOA(1), interruption, SA_INTERRUPT, "TIM1", NULL);
```

« interruption » est la fonction qui s'exécutera dans le cas où TIM1 passe à l'état bas(interruption). Dans cette fonction nous faisons notamment le vidage de la RAM tampon synthétisée dans le FPGA et le remplissage de la variable img1, tableau tampon entre le noyau et le user, :

```
static int interruption(int irq, void *dev_id, struct pt_regs *regs)
{
static int i;
static unsigned short reception;
/* Fonctionnement En lecture RAM directe*/
for(i=0;i<256;i++){
reception=lecture_ram(i);
img1[Indice_Tableau]=reception;

}
return 0;
}
```

La fonction `lecture_ram` fait notamment ceci :

```
unsigned short lecture_fpga_ram(unsigned short addr){
static unsigned long pixel;
pixel= *(unsigned long*)(FPGA_Addr+addr);
return pixel;
}
```

Cette fonction a été adaptée et inspirée du programme fpgaregs.c permettant le transfert de données entre le FPGA et le processeur, nous avons notamment à la ligne 111 de ce programme cette ligne qui effectue principalement la lecture au sein du FPGA :

```
value = *(unsigned short*)(ptr_fpga+(address));
```

Il est tres important de parler de la fonction « `my_function_read` » qui copie les donnée dans l'espace utilisateur notamment grace à la fonction de base, définie dans la bibliothèque ¡linux/fs.h¿ `copy_to_user` :

```
static ssize_t my_read_function(struct file *file, char *buf, size_t count, loff_t *ppos)
{

int lus = 0;
int i;//,j;
printk(KERN_DEBUG "read: demande lecture de %d octets\n", count);
lus = count;
i=copy_to_user(buf, (unsigned char *)img1 , count);
*ppos += lus;
printk(KERN_DEBUG "read: %d octets reellement lus\n", lus);
printk(KERN_DEBUG "read: position=%d\n", (int)*ppos);
return lus;
    return 0;
}
```

Coté utilisateur, comme cela à été stipulé dans le chapitre « traitement des donnée brutes », il suffit de tapper la commande suivante afin de recuperer les données dans le tableau `Mon_Image` :

```
read(file, &Mon_Image, img_size);
```

Les deux fonctions read() et `my_read_function()` sont associées l'une à l'autre de chaque coté des espaces du système d'exploitation.
Quant à la fonction `my_open_function` alloue de la mémoire au tableau tampon img, on lui alloue notamment une taille de 326 688 octets, de quoi stocker une image complète du capteur OV7620.

```
img1=(unsigned short *)kmalloc(326688 *sizeof(unsigned short), GFP_KERNEL);
```

Etant associée à open() coté user, lorsqu'on saisit la commande suivante, on crée le tableau img de taille 326 688 octets :

```
file = open("/dev/mydriver", O_RDWR);
```

Comme cela a été dit précédemment, la fonction release() s'associe à close() et désalloue les tampons et « desinitialise » toutes les initialisations faites dans le noyau (notamment pour l'interruption) .
Ainsi, lorsqu'une requête d'image est faite depuis le programme utilisateur, le module noyau alloue la mémoire au tableau tampon qui réceptionne l'image (dans le noyau), initialise l'interruption et attend un événement sur celle-ci afin de lire les pixels venant de la RAM du FPGA et de remplir avec ces pixels le tableau tampon img dans le noyau. L'ensemble décrit noyau-user crée ce que l'on appelle un driver.
La compilation d'un tel programme ne se fait pas via un cross-compiler comme gcc, il est necessaire de faire ce que l'on appel un makefile. Un makefile est un fichier incluant les cibles nécessaires à la génération du module noyau, il indique le chemin pour les libraires. Une fois le makefile fait, la construction du module se fait via la commande make. Une fois généré, nous avons le fichier monmodule.ko (kernel object), ce fichier est transferé sur la carte apf9328 via la commande wget, décrite précédemment. Voici le Makefile :

```
# Makefile for the Armadeus GPIO drivers
# Part executed when called from kernel build system:
ifneq ($(KERNELRELEASE),)

obj-$(CONFIG_ARMADEUS_GPIO_DRIVER) += MonModule.o
#MonModule-objs := core.o

# Part executed when called from standard make in this directory:
# (preferably use Makefile in parent directory)
else

ARMADEUS_BASE_DIR=../../../..
include $(ARMADEUS_BASE_DIR)/Makefile.in

KDIR    := $(ARMADEUS_LINUX_DIR)
PWD := $(shell pwd)

# Armadeus custom drivers common targets:
include ../Makefile.in

endif
```

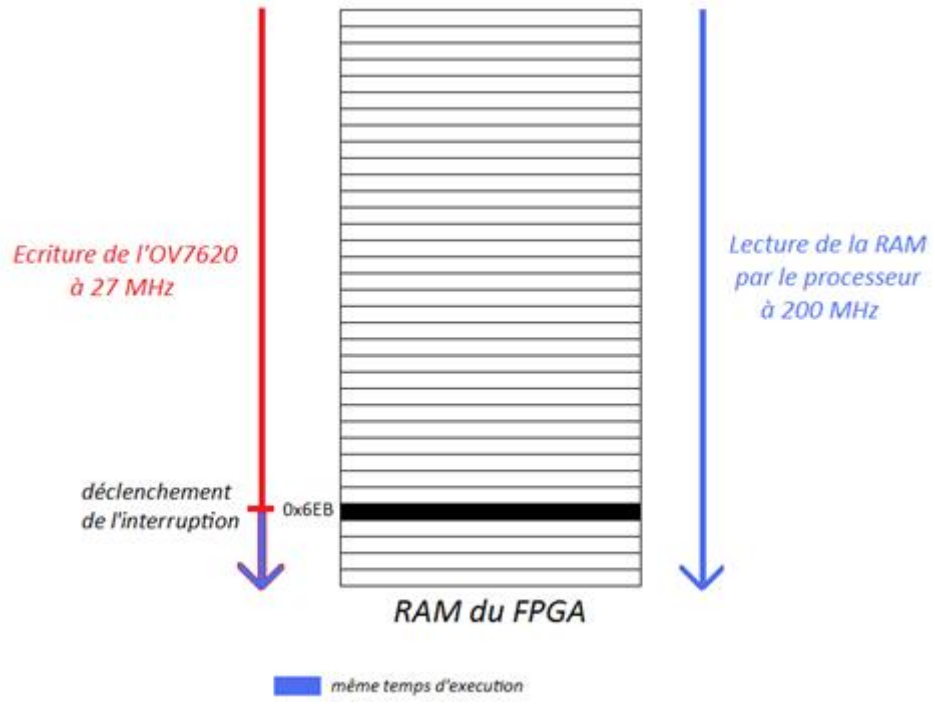
Pour lancer un tel programme, nous nous sommes servi de la commande « insmod » (insert module » qui permet comme son nom l'indique d'insérer le module en fond de tache :

```
insmod MonModule.ko
```

5.2.5 Synchronisation des signaux du capteur CMOS via le FPGA

Coté FPGA, nous allons d'une part gérer la synchronisation des signaux de la caméra en fonction des timings fournies dans la documentation technique, d'autre part, nous allons synthétiser la RAM allant réceptionner une partie de l'image.
Comme cela à été présenté précédemment, nous allons déclencher le transfert des pixels vers le processeur par interruption. Nous déclenchons cette interruption dans le cas où l'image est complète (Vsync à l'état haut) ou bien dans le cas ou on ateint une valeur bien de la RAM.
Cette valeur est calculée d'une façon à ce que le temps de lecture corresponde exactement au temps qu'il faut à la caméra pour finir de remplir la mémoire : le processeur lit à une fréquence de 200 MHz et le capteur CMOS, quant à lui, écrit à une fréquence de 27 MHz, soit le processeur lit 7.4 fois plus rapidement que le capteur écrit, la RAM choisie est un composant standard Xilinx qui comporte 2048 valeurs possibles pour le bus d'adresse (bus de 11 bits), autrement dit, pour parcourir les 2048 adresses possibles, le processeur les parcourra 7.4 fois plus rapidement que l'OV7620. Lorsque le processeur aura lu les 2048 valeurs, le capteur, quant à lui, aura écrit dans 277 valeurs d'adresse seulement.

L'idée est donc de déclencher la lecture du processeur (interruption) lorsqu'il ne reste que 277 pixels à écrire, soit à la 1771eme (2048-277) valeur du bus d'adresse, soit en hexadécimale 0x6EB. Cette valeur reste néanmoins théorique, un éventuel ajustement expérimentale est fait en fonction de l'image obtenu.

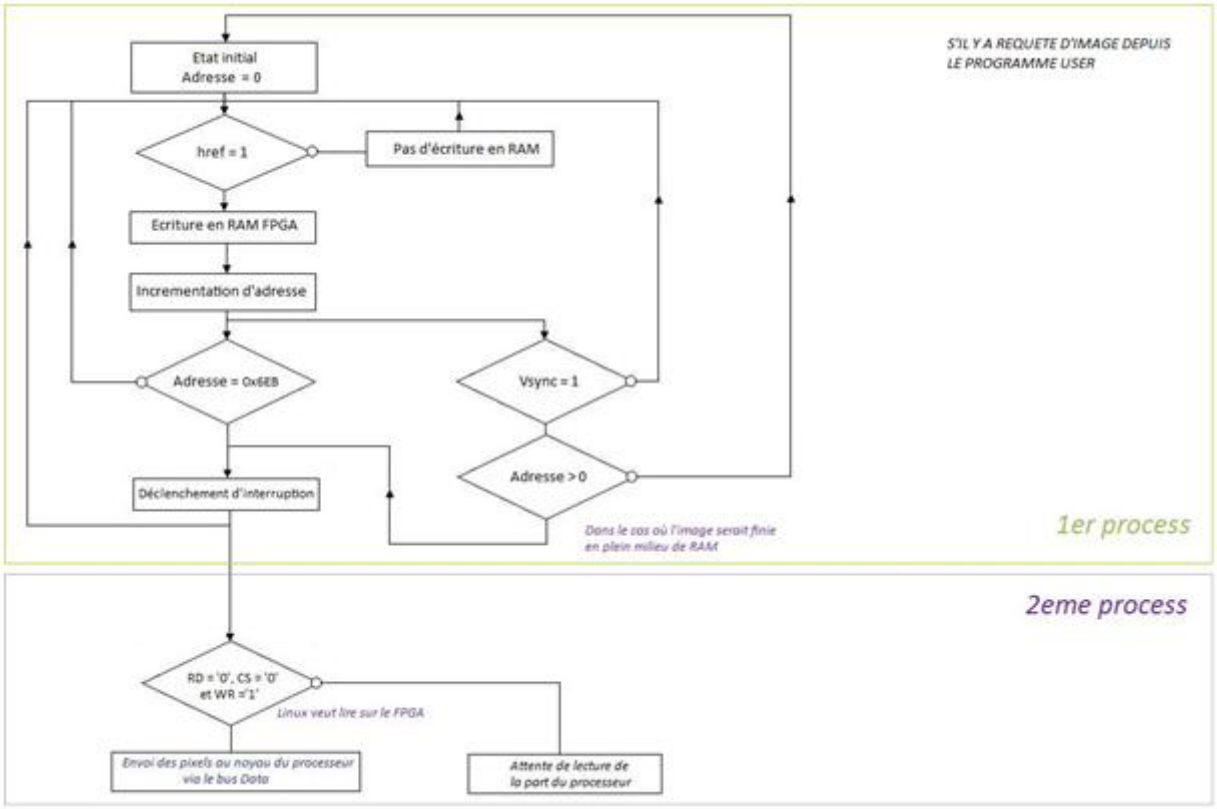


L'interruption est déclenchée lorsque nous atteignons cette valeur ou bien lorsque l'image est complète (Vsync = 1). La ligne d'interruption TIM1 est active à l'état bas. Physiquement, la ligne TIM1 est reliée au FPGA, ce qui nous facilite grandement la tache, d'un point de vue du VHDL, ce déclenchement se résume à écrire ceci :

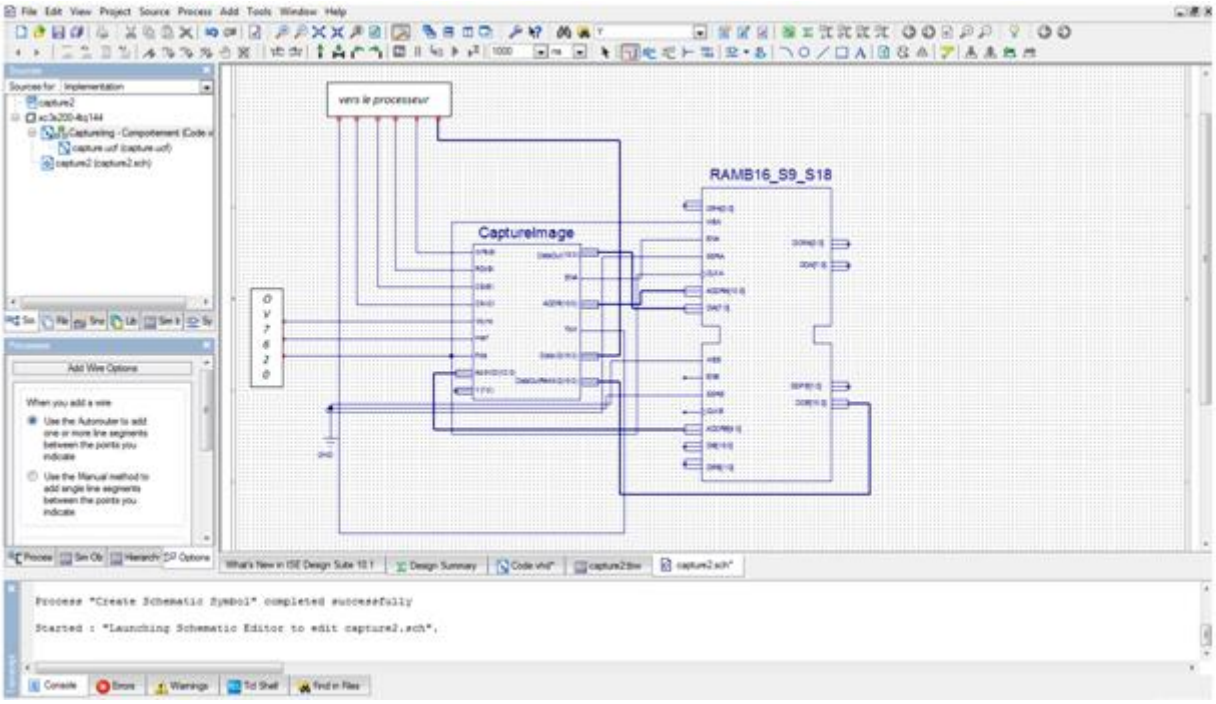
```
if  (addr>"10111110000"   or (Vsync='1' and addr>"0")) then
-- Condition de déclenchement d'interruption : déclenche si 0x6EB atteint ou
bien fin d'image(avec une RAM FPGA pas vide)
       ITproc<='0';
-- déclenchement sur etat bas sur ITproc si addr = 0x6EB ou bien si Image complète
    else
        ITproc<='1';
end if;
```

L'architecture de notre programme VHDL se compose de deux process : le premier permet de gérer les timings de la camera, de déclencher l'interruption et d'envoyer les pixels à la RAM (sensible sur Pclk). Le second process permet l'écriture des pixels dans le noyau du processeur, ce dernier process est sensible sur les signaux de contrôle du processeur (WR, RD, CS) et la donnée réceptionnée (sortant de la RAM), il envoie la donnée au processeur lorsque ce dernier est en phase de lecture (WR = 1, RD = 0 et CS = 0, les signaux de contrôle du processeur sont actif à l'état bas). Dans notre système, il s'agit du programme s'exécutant depuis l'espace utilisateur qui se charge, d'une part de recevoir les pixels en dernier lieu, et d'autre part d'envoyer une requête pour la réception

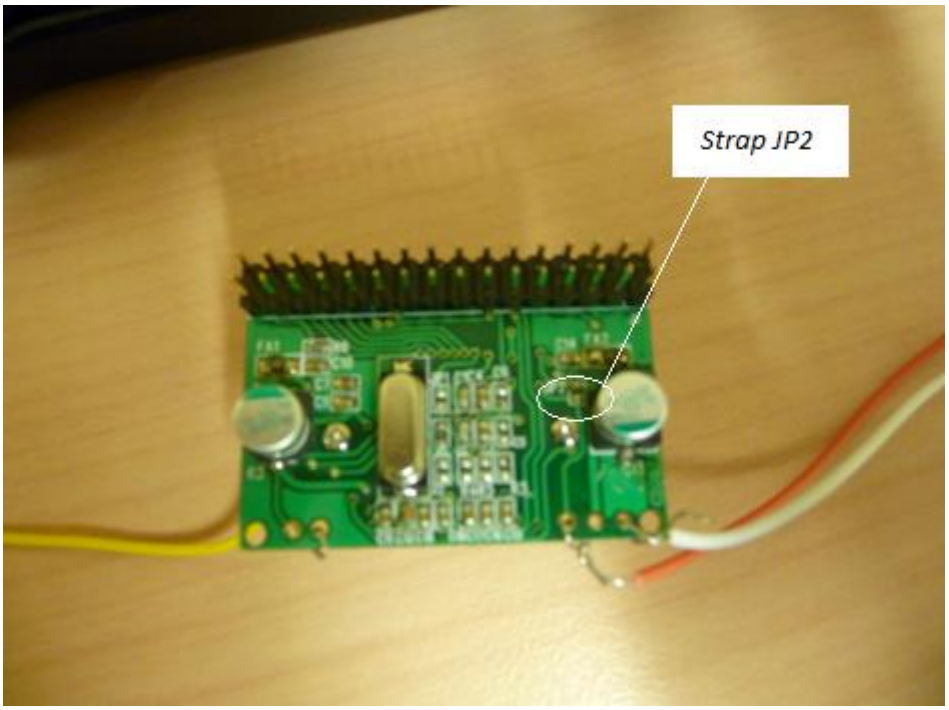
d'une image. S'il y a requête, alors le noyau s'avise de lancer le processus de transfert des pixels depuis le FPGA vers le noyau.
Quant à l'entité de notre programme, elle regroupe principalement les signaux de la caméra (Pclk, Href, Vsync, Y), les signaux relatifs au processeur (data, ctrl, addr, clk), un signal de validation de la RAM (ENA), la ligne d'interruption (TIM1) ainsi que des signaux d'interface. Notre programme est fourni en annexe. La partie VHDL suit le logigramme suivant :



Concernant la RAM, d'un point de vue choix technologique, nous avons pris une RAM à double ports permettant, d'une part, la réception des pixels venant du capteur CMOS(premier port), d'autre part, l'envoi de la donnée vers le processeur (second port). Il nous faut donc une RAM dont le premier port ayant un bus de données de 8 bits, afin que celui-ci coïncide correctement avec le bus Y de la caméra, et il faut, sur le second port, un bus de données de 16 bits, afin que celui-ci coïncide avec le bus data du processeur. Nous avons choisis la `RAMB16_S9_S18` dans les bibliothèques Xilinx. Cette RAM est un composant VHDL générique fourni par Xilinx. Parmi ses entrées du port A, nous avons le signal ENA qui valide l'écriture sur le port A (donnée 8 bits) : si ENA est a 1, on peut écrire. Nous avons directement connecté le signal Href étant donné que nous devons écrire seulement si Href est à 1 (phase de envoi des pixel par le capteur). La liaison entre notre IP et la RAM s'est faite de façon graphique sur Xilinx, comme cela est présenté si dessous :



Remarque : il est important de noter que le FPGA est alimenté en 3.3V, ce qui fait que les niveaux de tensions recevables sur les entrées de ce FPGA ne doivent pas excéder 3.3V. Or, le capteur CMOS que nous devons interfacer a des sorties dont les niveaux de tensions sont de 5V étant donné qu'il s'alimente en 5V. Il est cependant possible que le capteur nous délivre des sorties ayant un niveaux de tensions de 3.3V, pour cela, il est nécessaire de retirer le strap JP2 du support C38A et de fournir 3.3V sur l'entrée DOVDD (Digital Ouput VDD) du support C38A, les 3.3V sont bien entendu disponible sur la carte apf9328. Les alimentation restantes peuvent toujours être alimentée en 5V.


Strap JP2

6 Programmation de la carte

6.1 Principe de lancement

Pour lancer le système, nous devons lancer les programmes un par un. Il faut dans un premier temps, une fois le noyau chargé sur la carte par la bootloader U-boot, charger le module fpgaloader qui permet le chargement des fichiers binaires pour le FPGA :

```
# modprobe fpgaloader
 fpgaloader v0.9
```

Une fois ce module chargé, nous devons chargé notre module noyau que nous avons concu via la commande suivante :

```
# insmod MonModule.ko
```

Ces deux commandes saisies, nous pouvons vérifier le bon chargement de ces deux modules via la commande lsmod.

Nous implémentons ensuite notre fichier binaire représentant l'IP de capture image via la commande :

```
# dd if=capture_image.bin of=/dev/fpgaloader
```

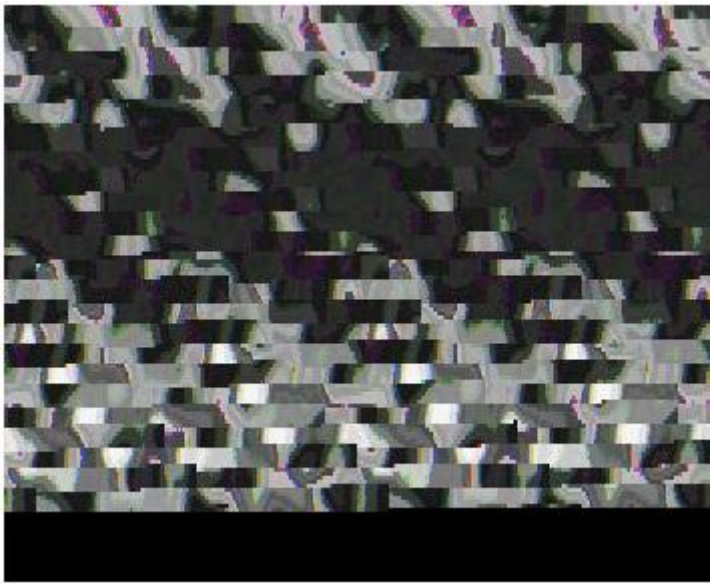
Il est nécessaire de générer le périphérique charnière, appelé aussi nœud système, « mydriver » de façon manuel afin que les deux parties du driver puisse interagir, ce périphérique est généré dans le dossier /dev. Pour cela, il faut utiliser la commande mknod comme ceci :

```
mknod /dev/mydriver c 250 0
```

Il est bien entendu qu'il faut conserver le même numéro de majeur, dans notre cas 250, (variable major dans le module noyau) afin que le tout puisse fonctionner.
En dernier lieu, nous lançons notre programme via le programme tournant dans l'espace user via la commande suivante :

```
./ProgrammeUser
```

6.2 Flashage de la carte au démarrage



Afin de faire un lancement permanent dès le démarrage de la carte, il est possible de rédiger un script en langage schell que nous appelons bootOV7620 dans lequel nous incluons toutes les commandes précédentes :

```
#! /bin/sh
modprobe fpgaloader
insmod MonModule.ko
dd if=capture_image.bin of=/dev/fpgaloader
mknod /dev/mydriver c 250 0
./ProgrammeUser
```

En première ligne, la commande!/bin/sh sur la indique que ce script doit être exécuté par le shell sh. Ainsi, il suffit de saisir la commande `./bootOV7620`.

Afin de lancer le système au démarrage, il faut mettre ce script dans un dossier spécifique au lancement des applications au démarrage de la carte. Ce dossier est le dossier nommé init.d dont le chemin est /etc/init.d.
Une fois lancé, on récupère ainsi le fichier image.ppm dans /tmp. Lorsqu'on l'ouvre, on recupere l'image suivante :

Cette image n'est pas le résultat attendu, il permet néanmoins le transfert des pixels depuis le FPGA vers l'espace utilisateur. Cependant, il doit y avoir un souci de synchronisation d'image.

Ce problème est peut être lié à un problème de déclenchement d'interruption, voir d'assemblage des différents programmes. Le manque de temps nous permet pas d'avoir une l'image souhaitée.

7 Conclusion

Nous venons de présenter le travail effectué durant ces quatre mois et demi de stage. Le résultat fourni de l'application principale, l'interfaçage de la carte avec le capteur CMOS OV7620, n'est pas malheureusement pas celui attendu, faute de temps nous ne pouvons l'améliorer. Cependant, il est concluant sur certains points, notamment la validation et le bon fonctionnement de chacune de nos parties : le driver (espace noyau et usr) écrit pour cet application assure bien le transfert des pixel depuis le FPGA, ce qui nous permet de répondre à certains points fixés en début de stage notamment la maitrise de la carte, la communication des deux plateformes et aussi l'apprentissage du développement d'un driver sous Linux. L'objectif de départ était de créer un système embarqué le plus intégrable possible, celui-ci n'est pas remis en cause étant donné que nous faisons appel à aucun composant externe (exemple RAM externe suffisamment grande pour stocker une image complète).

Le travail a fournir dans l'avenir serait, d'une part, de trouver la faille qui empêche de restituer l'image souhaitée, (je pense que cette faille est peut être due soit à un mauvais déclenchement de l'interruption voir à une mauvaise gestion des timings du capteur CMOS par notre code VHDL). D'autre part, configurer le capteur via son bus I2C nous permettrait de restituer une image en couleur, il serait dommage de brancher le bus UV de la caméra pour la restitution d'une image couleur par souci d'intégration : brancher le second bus rajouterai une série de huit files supplément, d'autant plus que ce bus monopoliserait huit entrées/sorties supplémentaires sur le FPGA Spartan 3, ce qui représenterai environ 10

D'un point de vue personnel, je tire, de ce stage, une expérience positive : ce stage m'a été très lucratif en terme de connaissances, j'ai pu découvrir notamment le développement de drivers sur un système Linux, la notion de système d'exploitation embarqué mais aussi la notion de développement en langage C sur processeur avec « un haut niveau de programmation », avant ce stage, cette notion m'était assez abstraite étant donné que les méthodes vue précédemment consistaient à passer par les registres du processeur (bas niveau). De plus, ce projet est entièrement autonome : n'intégrant aucune entité du laboratoire ce stage m'a permis de développer mon autonomie et d'apprendre à gérer mon temps. En conclusion, le résultat obtenu, bien que l'on peut en tirer profit, n'est malheureusement pas celui fixé (image brouillée) mais, d'un point de vue personnel, l'apport en connaissance acquis durant ces quatre mois et demi m'est très fructueux.