Campus des Cézeaux
BP 125
63173 AUBIERE CEDEX
FRANCE

CERN
Organisation Européenne
pour la recherche Nucléaire
CH-1211 GENEVE 23
SUISSE

LOGICIEL HAUTE PERFORMANCE POUR CARTE TAGNET

RÉDIGÉ PAR
SÉBASTIEN GONZALVE

EXPOSÉ DANS LE CADRE DU STAGE DE TROISIÈME ANNÉE.

RESPONSABLE DE STAGE :
HANS M ÜLLER

AVRIL-SEPTEMBRE 2002

REMERCIEMENTS

Je remercie M. Hans MÜLLER, physicien au CERN et chef de la section ED/DTB, pour le travail passionnant qu'il m'a confié, et pour la confiance qu'il m'a accordée.

Je tiens aussi à exprimer toute ma gratitude envers messieurs D. ALTMANN, A. GUIRAO, F. VINCI DOS SANTOS pour l'aide précieuse qu'ils m'ont apportée, et pour m'avoir accueilli dans leur bureau.

Je souhaite également remercier Mlle Carmen GONZALEZ, pour sa charmante présence et pour sa bonne humeur permanente.

Plus généralement, je remercie toutes les personnes travaillant dans le service pour leur très chaleureux accueil et leur aide.

Je tiens aussi à saluer M. MESNARD pour son aimable visite durant mon stage.

Glossaire et notations

BIOS	Basic Input/Output System : Collection de fonctions de base servant à accéder au matériel de manière standard. C'est aussi la partie de l'ordinateur qui initialise les périphériques au démarrage.
CERN	Centre Européen pour la Recherche Nucléaire.
CMC	Common Mezzanine Card : Standard IEEE P1386
Driver	Programme chargé de superviser le fonctionnement d'un périphérique.
FLIC	Flexible Input/Output Card : Carte développée au CERN dont l'architecture flexible rend son utilisation possible dans de nombreux projets.
LHC	Large Hadron Collider : Le Grand Collisionneur de Hadrons sera le prochain accélérateur de particules du CERN. Il devrait pouvoir permettre d'explorer la matière à des dimensions jamais atteintes.
MCU	Monitoring Control Unit : système embarqué basé sur une architecture Intel, dessiné selon le standard PPMC, pour le contrôle distant des cartes mère des RU
OS	Operating System : Système d'exploitation. Le système d'exploitation est un programme généralement lancé au démarrage de l'ordinateur, et chargé de gérer les ressources de la machine. Sans système d'exploitation, il faudrait travailler en langage machine (binaire) ce qui n'est pas réaliste pour les systèmes complexes que nous utilisons de nos jours.
PCI	Peripheral Component Interconnect : Spécification de bus permettant des transferts rapides entre les différents périphériques et le processeur de l'ordinateur.
PPMC	Processor PCI Mezzanine Card VITA 32 -199X
RU	Readout Unit : unité de lecture utilisée sur le réseau TAGnet pour communiquer avec les processeurs du tore
SCI	standard IEEE 1596 assurant un débit de 10Gbit/s
Scheduler	terme difficilement traduisible servant à désigner un programme de gestion de la charge dans le cas d'une architecture parallèle

RÉSUMÉ

Afin de faire entrer en collision les quarks qui composent les protons, il a été entrepris de construire le plus puissant accélérateur de particules du monde : le LHC (Large Hadron Collider). LHCb est l'une des quatre expériences du LHC, elle est destinée à mettre en évidence une asymétrie dans la production des particules secondaires issues de la désintégration des mésons B qui pourrait expliquer la disparition de l'anti-matière après le Big-Bang.

Le flot de données généré par les détecteurs est gigantesque et est constitué en grande partie d'événements non significatifs. Il faut donc opérer un tri afin de ne conserver que les événements pour lesquels la probabilité de trouver les particules cherchées est suffisamment élevée.

Une série de triggers est donc disposée en sortie des détecteurs, chacun éliminant une partie du « bruit ». Mon travail, au sein de LHCb, consistait à développer une méthode temps-réel pour transmettre la disponibilité des CPUs dans un tore pour le trigger de niveau 1 (le deuxième après le détecteur).

Afin de faire face au très haut débit, une architecture parallèle de processeurs à été mise en place. La répartition des tâches entre les différents esclaves se fait grâce au système TAGnet que notre équipe propose.

Le maître TAGnet est constitué d'une partie matérielle (une carte FLIC), et une partie logicielle (programme C). Le travail qui m'était demandé consistait à réaliser la partie logicielle du maître TAGnet contrôlant la carte FLIC via le bus PCI du PC maître dans l'architecture parallèle .

Mots Clés : TAGnet, FLIC, PCI, langage C, architecture parallèle.

ABSTRACT

In order to collide quarks that compose protons, it was decided to built the most powerful beam accelerator in the world : LHC (Large Hadron Collider). LHCb is one of the four LHC experiences and is devoted to point out an asymmetry in production of secondary particles due to disintegration of B-meson that would explain that anti-matter disappeared after the Big-Bang.

The data-flow generated by detectors is gigantic and consist for a big part of non-significant events. Telling good data from bad one is essential in order to keep events for which probability to find the searched particles is hight enough.

Several triggers are queued to detectors' output, each one removing a part of "noise". My job, in LHCb experiment, consisted in developing a real-time method for free CPU distribution for level-1 trigger (second one after the detector).

In order to face the very hight data flow, a parallel architecture of processors was settled. Task scheduling between slaves is done thanks to the TAGnet system proposed by our team.

TAGnet master is made of both hardware (FLIC) and software (C program) parts. I've been asked to make the TAGnet's software side that drives FLIC via the architecture's master PCI bus.

Key words : TAGnet, FLIC, PCI, C-language, parallel architecture.

TABLE DES MATIÈRES

REMERCIEMENTS

GLOSSAIRE ET NOTATIONS

RÉSUMÉ

ABSTRACT

TABLE DES MATIÈRES

INDEX DES ILLUSTRATIONS

INTRODUCTION..9

I.PRÉSENTATION DES COMPOSANTS DU SYSTÈME 10

1. Présentation du système...10

a)LHCb, trigger niveau 1...10
b)La FLIC...13
c)TAGnet...15
d)Linux..16

2.Notions à propos du bus PCI..17

a)Espace de configuration...18
b)Transferts de données via PCI..19
c)Gestion de la mémoire des périphériques PCI.......................................19
d)Accès à la mémoire PCI avec Linux...20

3.Linux et la gestion du bus PCI..21

a)Linux et droits d'accès 24
b)Version de noyau..25
c)Fichiers spéciaux..26

II.RÉALISATION ET ÉTUDE 27

1.Choix du type de driver...27

2.Positionnement du driver au sein du système...28

3.Module du noyaux..29

4.Déboguage du module..31

5.Génération de C-TAGs...33

III.RÉSULTATS ET VALIDATION...36

1.Test de la FLIC...36

2.Utilisations du module..38

3.Tests de vitesse...40

4.Validation d'une boucle du système TAGnet..41

a)Nouvel algorithme..41

b)Tests avec le nouvel algorithme...42

c)Validation avec transfert SCI...42

CONCLUSION...46

RÉFÉRENCES BIBLIOGRAPHIQUES

ANNEXES

INDEX DES ILLUSTRATIONS

Illustration 1 : Coupe de l'ensemble du détecteur de LHCb.................................10
Illustration 2 : Topologie en tore (les points rouges sont les processeurs)...........11
Illustration 3 : Boucle TAGnet et tore de processeurs...11
Illustration 4 : Placement du PC maître au sein de la topologie...........................12
Illustration 5 : Schéma bloc de la FLIC...13
Illustration 6 : Photographie de la FLIC..14
Illustration 7 : Photographie de la carte mezzanine S-link..................................14
Illustration 8 : Structure des TAGs...16
Illustration 9 : Espace de configuration PCI..18
Illustration 10 : Appel de périphériques sans utiliser de driver............................22
Illustration 11 : Utilisation des périphériques avec un driver...............................23
Illustration 12 : Différents appels lors de l'utilisation d'un driver.........................23
Illustration 13 : Le noyau en détail..24
Illustration 14 : Boucle de fonctionnement pour les C-TAGs..............................33
Illustration 15 : Champ de bit pour les C-TAGs..34
Illustration 16 : Exemple de mauvais résultat lors du premier test de RAM........36
Illustration 17 : Exemple d'utilisation du module avec LabVIEW.......................38
Illustration 18 : Signaux mesurés sur le bus PCI à l'aide d'un traceur numérique.39
Illustration 19 : Écran de sortie après le test SCI-Transfert FLIC........................41
Illustration 20 : Boucle TAGnet complète..42
Illustration 21 : Résultat obtenu avec le traceur PCI...43

INTRODUCTION

La recherche en physique des particules a pour but de trouver les éléments constitutifs de la matière. Elle utilise de grands accélérateurs de particules afin de faire entrer en collision des grains de matière (électrons, protons, etc...) de manière à observer les briques fondamentales de notre univers.

Le CERN est un haut lieu de la recherche scientifique dans ce domaine, il a accueilli le plus grand de ces accélérateurs, le LEP qui faisait entrer en collision des électrons. Pour étudier les interactions entre les particules hadroniques fondamentales, les quarks, il a été entrepris de construire d'ici 2007 un accélérateur encore plus puissant, le LHC (Large Hadron Collider).

Parmi les différents travaux de recherche qui auront lieu, LHCb sera une expérience dédiée à la recherche de la violation CP par mesure des produits de désintégration. Dans le principe il s'agit toujours d'accélérer des particules et de les faire entrer en collision afin de « voir » quelles sont les briques fondamentales constitutrices de la matière. Pour LHCb, un détecteur appelé VELO (VErtex LOcator) est chargé de mesurer les « traces » laissées la désintégration des mésons B. Ces traces (les vertex) sont faciles à trouver et servent donc de trigger.

Ce type de détecteur génère une très grande quantité d'information qu'il est impossible de stocker pour des problèmes évidents de vitesse et de place. C'est pourquoi, il a été mis en place une série de « trigger » permettant de sélectionner seulement les événements pour lesquels la probabilité de trouver quelque chose est assez élevée.

Afin de traiter les données à une vitesse suffisante, le système est constitué d'une architecture parallèle de processeurs basée sur une topologie torique. Comme toute architecture parallèle, ce système nécessite un ordinateur « maître » dont le rôle est de répartir les données entre les différents esclaves. Les communications entre le maître et les buffers externes remplis pas le détecteur VELO, sont réalisées grâce au protocole TAGnet proposé par M. Hans Müller.

Le maître a aussi une fonction de régulateur, et de gérant du système (surveiller les erreurs). Il se divise en deux parties : une partie matérielle et une partie logicielle. Alors que la partie matérielle s'occupe du rythme des transfert ainsi que de la mise en forme des TAGs, la partie logicielle est chargée du rôle de supervision et de répartition de la charge. Le but de mon stage était de réaliser un logiciel performant pour maître TAGnet capable de générer des TAGs avec une fréquence d'au moins 1MHz.

I. PRÉSENTATION DES COMPOSANTS DU SYSTÈME

1. Présentation du système

a) LHCb, trigger niveau 1

LHCb est l'expérience dédiée à la recherche du meson b (beauté). Elle met en oeuvre des détecteurs de très grande taille avec un débit de données extrêmement élevé. Afin de pouvoir faire face à ce flot incessant de mesures, une sélection par niveaux a été mise au point. Les triggers sont disposés en cascade de telle sorte que chacun retire les données qu'il considère comme mauvaises et passe les autres à l'étage de trigger suivant (selon le principe du tamisage successif pour du sable).

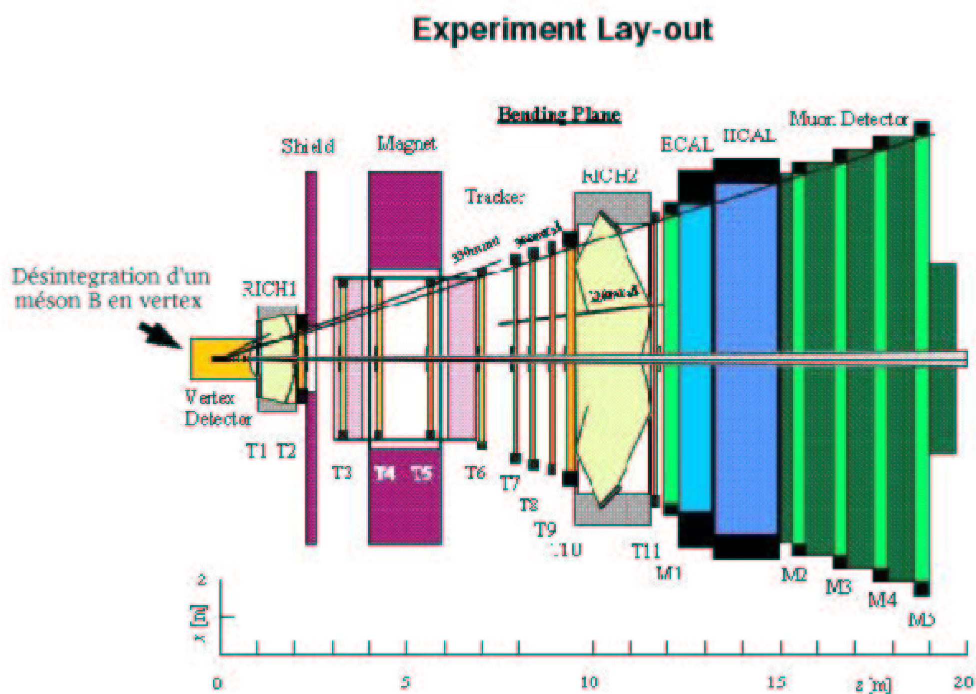

Illustration 1 : Coupe de l'ensemble du détecteur de LHCb

Le trigger niveau 1 est chargé de retirer les échantillons pour lesquels il n'existe pas de vertex[1]. Pour ce faire, il doit analyser les données issues du détecteur (Silicon Strip Sensor) pour ne garder que les meilleurs événements. Ce traitement doit être effectué dans un temps très limité (le traitement doit s'effectuer à une fréquence minimum de 1MHz), c'est pourquoi une architecture parallèle de processeurs est utilisée. La topologie retenue est une surface torique qui correspond, de manière plus intuitive, à une grille dont les extrémités sont bouclées, comme montré sur l'illustration 2.

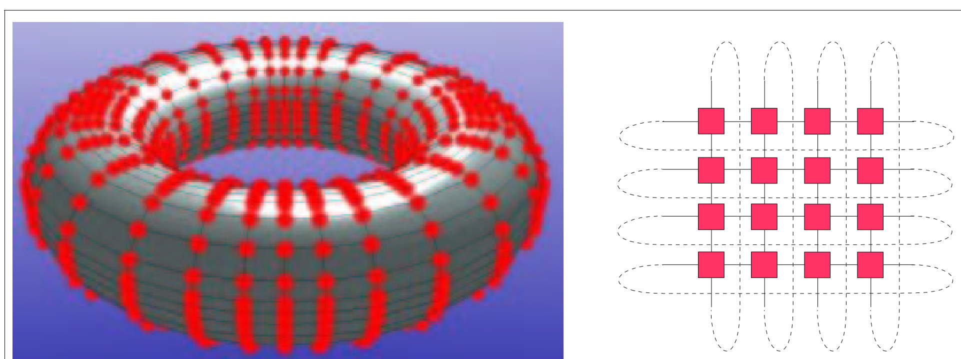

Illustration 2 : Topologie en tore (les points rouges sont les processeurs)

Comme toute architecture parallèle, le système de traitement des données de LHCb possède un maître chargé de diriger ses esclaves. Le PC arbitre est chargé de repartir les données issues des détecteurs, entre les différents PC esclaves afin que ces derniers les traitent. Le protocole retenu pour la distribution du travail est le système TAGnet [1] [2] dont nous allons voir le fonctionnement dans la suite. On notera, pour l'instant, que le circuit des TAGs est indépendant du reste des communications qui ont lieu au sein du tore.

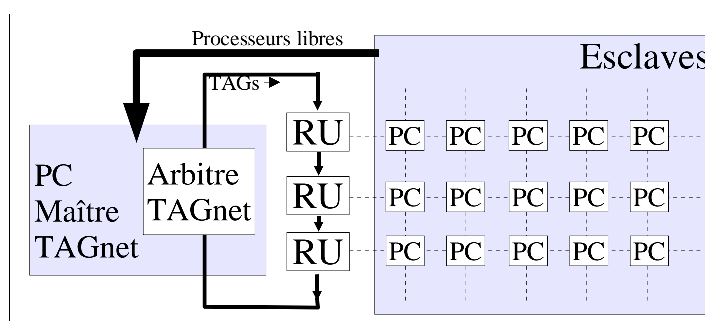

Illustration 3 : Boucle TAGnet et tore de processeurs.

1 Le vertex est le nom donné à la trajectoire de certaines particules issues de la désintégration des mésons B. C'est une condition sinequanone pour la présence de la particule à étudier. Pour plus de détails sur la théorie, il convient de lire les ouvrages de physique des particules.

Le maître TAGnet est constitué de deux parties distinctes : une logicielle, appelée « scheduler », et une partie matérielle basée sur une FLIC dont le fonctionnement est décrit plus loin.

Afin de connaître les adresses des processeurs libres, le PC maître possède une carte réseau SCI par laquelle chaque esclave indique son état (illustration 3). Les coordonnées des processeurs sont alors stockées en mémoire (RAM du PC), pour pouvoir être ensuite triées avant d'être renvoyées aux esclaves TAGnet par le biais de TAGs.

En y regardant de plus près, on s'aperçoit facilement que le PC maître ne diffère des PC esclaves que par la présence d'une FLIC sur son bus PCI [3]. Afin de simplifier l'architecture globale, le PC maître est donc considéré comme un PC standard du tore, auquel on a ajouté une FLIC, et dont la tâche principale n'est pas de traiter les données, mais de les repartir entre ses différents voisins (illustration 4). Il référence ainsi les CPUs libres et leur envoie des données à traiter.

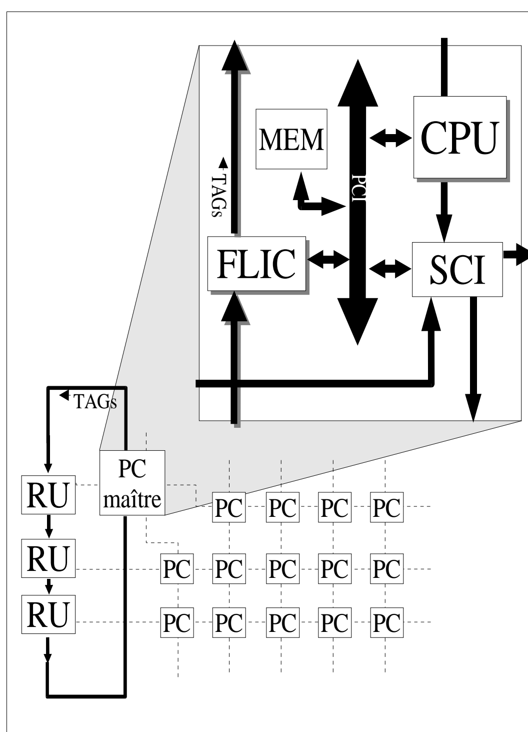

Illustration 4 : Placement du PC maître au sein de la topologie.

Pour ce faire, le maître génère un flot continu de TAGs indiquant dynamiquement quels sont les processeurs libres qui prennent en charge les calculs.

b) La FLIC

La FLIC est une carte PCI développée au CERN pour réaliser différentes tâches. Son nom est l'acronyme de Flexible I/O Card (carte E/S flexible). Elle utilise le standard « PCI universal » 64 bits. Sa partie principale est organisée autour d'un FPGA ORCA (Lucent) (illustration 5) dont le comportement est reprogrammable dynamiquement via le bus PCI. Pour le projet dans lequel je travaillais, la FLIC est la partie matérielle (« Hardware ») du maître TAGnet, travaillant pour émettre les TAGs vers les RU. La FLIC tient ses ordres de fonctionnement du scheduler TAGnet (partie logicielle qui est l'objet de ce stage).

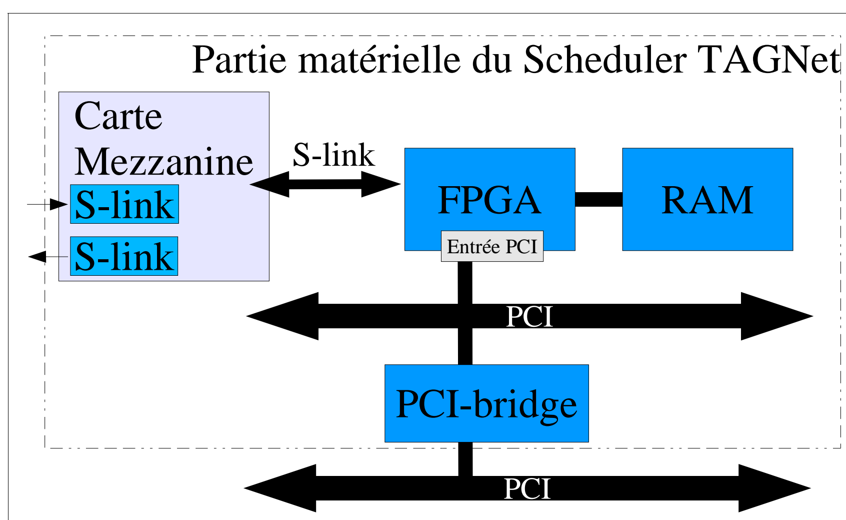

Illustration 5 : Schéma bloc de la FLIC

L'illustration 6 présente une photographies de la FLIC. On remarquera le FPGA ORCA au centre, est les banc de SDRAM à droite.

L'illustration 7 est une photographie de la carte mezzanine S-link pouvant être connectée à la FLIC.

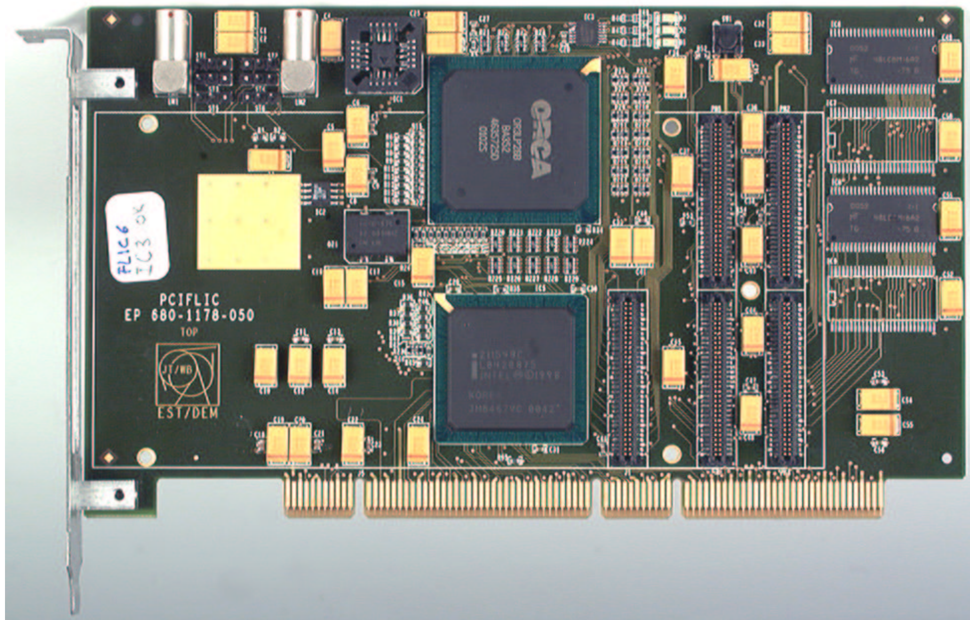

Illustration 6 : Photographie de la FLIC

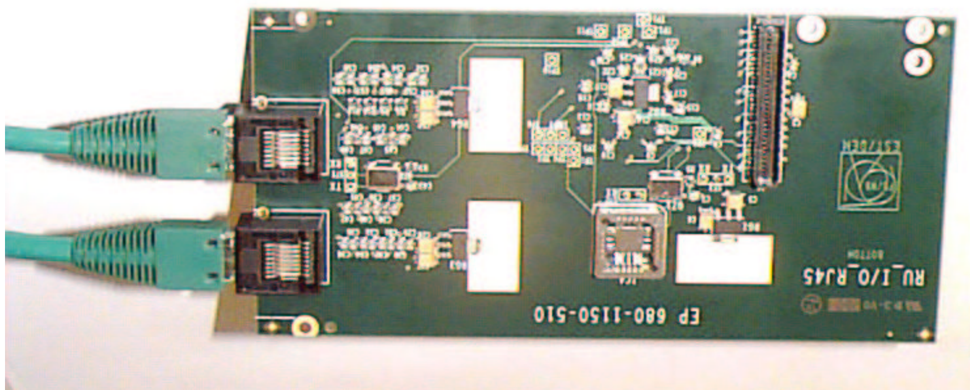

Illustration 7 : Photographie de la carte mezzanine S-link

c) TAGnet

Ce paragraphe décrit de manière simplifiée le fonctionnement du TAGnet, et s'appuie largement sur les documents de M. Müller. Afin de rester suffisamment concis, certain aspects ne sont volontairement pas développés.

TAGnet est un système original de répartition de trafic pour des événements à haut taux de transfert de données. Il a été conçu et développé par M. Hans Müller afin de faire face aux haut débits de données issus du trigger de niveau 1 de LHCb. Ce système est conçu pour assurer la supervision et le bon fonctionnement d'une architecture parallèle de processeurs dont les esclaves sont situés sur un tore. Chaque ligne du tore (cf illustration 4) est reliée à une unités de lecture (Readout Unit ou RU). Ces dernières sont reliées par un bus TAGnet sur lequel transitent les informations provenant du maître TAGnet. Les RU constitues donc les esclaves du maître TAGnet.
Le maître émet des paquets, appelés TAG, regroupant l'information destinée aux RU.
De fait, il existe deux types de TAGs : les TAGs de commande (C-TAG) et les TAGs messagers (M-TAGs).
De façon simplifiée, un C-TAG contient l'adresse de 12 bits d'un processeur libre ainsi que l'adresse de l'évènement qui lui est donné à traiter. Un M-TAG contient, pour sa part, un message vers un ou plusieurs esclaves.

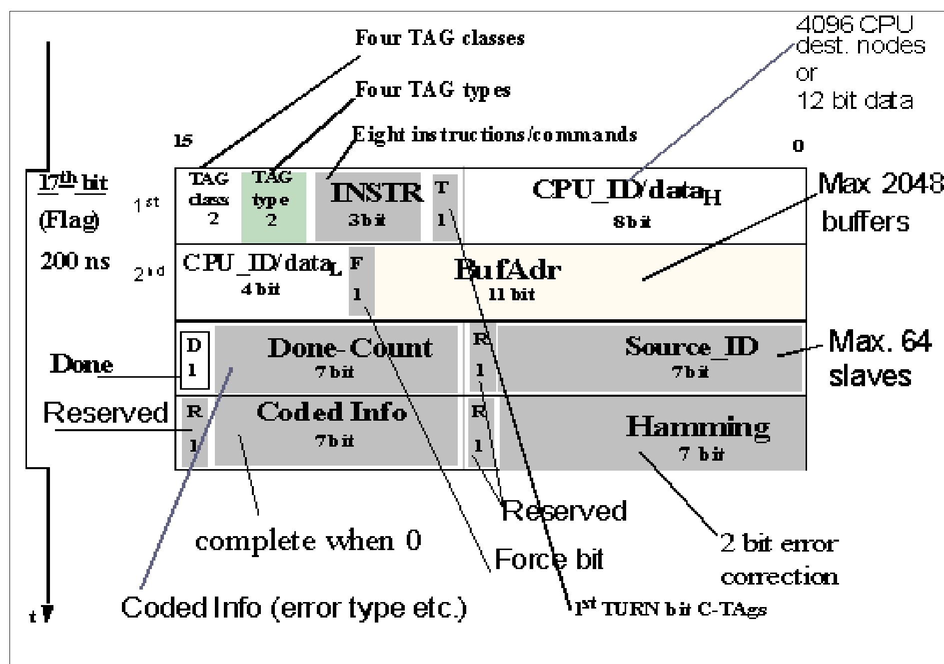

Illustration 8 : Structure des TAGs

Les TAGs sont envoyés de manière continue sur le bus TAGnet qui relie toutes les RU en cercle. Parallèlement au bus véhiculant les données, le bus TAGnet possède une ligne d'horloge servant à la bonne synchronisation des esclaves. Ce « heartbeat » (littéralement, battement de coeur) d'une fréquence de 5MHz permet de plus de vérifier la bonne circulation des TAGs.

d) Linux

Linux est un système d'exploitation très sophistiqué et gratuit dont les sources sont libres d'accès et de modification. Initialement créé par Linus Torvalds sur le modèle des UNIX propriétaires, ce projet proposant de créer un nouvel OS en partant de rien, a tout de suite passionné la communauté des développeurs du monde entier. Rapidement, l'OS est devenu stable et a constitué la base de nombreuses applications.
Depuis ses début assez « confidentiels », Linux a lentement pris de l'ampleur, si bien qu'aujourd'hui, il est reconnu par les passionnés d'informatique aussi bien que par les industriels comme étant l'un des OS les plus performants du moment, offrant, de surcroît, la possibilité d'être « adapté » librement aux besoins de son utilisateur.

Dans le projet présent, Linux a été retenu car, de part son architecture interne, il permet d'accéder relativement simplement aux périphériques (mémoire, bus PCI, etc..), tout en offrant l'ensemble des outils nécessaires à la programmation. De manière pratique, il n'est pas nécessaire de développer un driver pour pouvoir accéder au matériel (alors que c'est indispensable pour MS Windows, par exemple). Cela est particulièrement intéressant pour effectuer des tests simples sur un nouveau périphérique.

2. Notions à propos du bus PCI

Le bus PCI (Peripherial Component Interconnect) [3] est, de nos jours, l'extension de bus la plus commune, si ce n'est la seule dans les PC actuels.
Ce standard est aussi très utilisé dans la plupart des ordinateurs récents tels les stations Alpha, les stations Sun ou PowerPC.
Généralement, le bus PCI est utilisé à une fréquence de 33MHz avec des données de 32 bits. Cela-dit, la norme décrit des utilisations de données de 64bits dont la fréquence peut monter à 66MHz. Les machines exploitant des bus PCI 64bits à 66MHz sont généralement des serveurs nécessitant des taux de transfert de données très élevés.
Cette partie n'a pas pour but d'expliquer la norme PCI, mais plutôt de présenter les différentes notions qui sont indispensables à la compréhension des programmes réalisés.
Il est intéressant de noter que la norme PCI ne définit pas seulement un protocole, mais aussi les connecteurs, et le squelette de fonctionnement des cartes. Cette architecture bien définie est sans doute une des clefs du succès de ce bus, et représente une aide non négligeable dans l'aide à la compréhension du fonctionnement des périphériques PCI.

a) Espace de configuration

Lorsque le PC démarre, le BIOS configure automatiquement chaque carte PCI (indispensable pour le fonctionnement correcte des cartes). Linux se paramètre ensuite en vérifiant les cartes installées (Carte vidéo, carte réseau, carte son etc...) et charge les drivers correspondants. Dans le cas de la carte FLIC que nous utilisons, le chargement du driver peut se faire à ce moment là ou être chargé par la suite sans que cela ne pose de problème.
Chaque carte PCI possède un jeu de registres appelé « espace de configuration » dont le contenu est défini de manière fixe par la norme PCI (illustration 9).

<table>
<tr><th colspan="4">OCTET</th><th></th></tr>
<tr><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td colspan="2">ID DEVICE</td><td colspan="2">ID VENDEUR</td><td>00</td></tr>
<tr><td colspan="2">Registre d'état</td><td colspan="2">Registre de commande</td><td>01</td></tr>
<tr><td colspan="3">Code de classe</td><td>ID révision</td><td>02</td></tr>
<tr><td>BIST</td><td>Type d'entête</td><td>Temps de latence</td><td>Longueur du cache</td><td>03</td></tr>
<tr><td colspan="4">Adresse de base 0</td><td>04</td></tr>
<tr><td colspan="4">Adresse de base 1</td><td>05</td></tr>
<tr><td colspan="4">Adresse de base 2</td><td>06</td></tr>
<tr><td colspan="4">Adresse de base 3</td><td>07</td></tr>
<tr><td colspan="4">Adresse de base 4</td><td>08</td></tr>
<tr><td colspan="4">Adresse de base 5</td><td>09</td></tr>
<tr><td colspan="4">Pointeur de bus CIS</td><td>10</td></tr>
<tr><td colspan="2">ID sous-système</td><td colspan="2">ID vendeur du sous-système</td><td>11</td></tr>
<tr><td colspan="4">Adresse de base pour expansion ROM</td><td>12</td></tr>
<tr><td colspan="4">Réservé</td><td>13</td></tr>
<tr><td colspan="4">Réservé</td><td>14</td></tr>
<tr><td>Max_Lat</td><td>Min_Gnt</td><td>Broche d'interpt</td><td>Ligne d'interpt</td><td>15</td></tr>
</table>

Illustration 9 : Espace de configuration PCI

En lisant cette plage de registres, le système est capable de reconnaître à quel type de périphérique il a à faire. Ainsi, la carte est référencée grâce à sont numéro de vendeur (chaque constructeur a un numéro qui lui est réservé, par exemple pour la FLIC, le constructeur est Lucent dont le numéro est 11C1h) et un numéro désignant le type de carte chez ce fabriquant (5401h pour la FLIC). La deuxième partie importante de ces registres sont les pointeurs des adresses de base. Ces adresses sont celles des espaces mémoire ou bancs de registres disponibles sur la carte. Ce sont des adresses définies par le BIOS lors du démarrage de l'ordinateur. La carte sait ainsi quelles adresses passant sur le bus lui sont destinées. Ces adresses sont aussi très importantes pour un programme voulant utiliser la carte : il ne doit utiliser que l'adresse du registre comme adresse de début de la mémoire (d'où adresse de base).

b) Transferts de données via PCI

Le bus PCI [4] autorise des écritures ou lectures en rafale. Ce mode, appelé « burst », permet d'écrire (ou lire) un grand nombre de données consécutives dans un intervalle de temps très réduit. Généralement, ce type de transaction est utilisé pour accéder à de la mémoire SDRAM présente sur la FLIC. En effet, la SDRAM possède généralement elle-aussi un mode burst, il est ainsi possible d'effectuer des transferts extrêmement rapides entre la mémoire du PC et la mémoire de la carte.
Par ailleurs, le bus PCI autorise bien entendu des écritures "normales", c'est à dire qu'il est possible de transférer les données unes à unes, mais ce type de transfert est extrêmement lent en comparaison avec le mode burst.

c) Gestion de la mémoire des périphériques PCI

Les cartes PCI possèdent généralement des bancs de mémoire pour stocker des données localement avant ou après les traitements. Il est possible de connaître les capacités en mémoire des périphériques en utilisant la commande *lspci.*

```
[root /sbin/] > lspci -vvs 2:f.0
02:0f.0 Non-VGA unclassified device: Lucent
Microelectronics Orca FPGA (rev 02) (prog-if 80)
Subsystem: Lucent Microelectronics Orca FPGA
Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV-
VGASnoop- ParErr- Stepping- SERR- FastB2B-
Status: Cap+ 66Mhz+ UDF- FastB2B+ ParErr- DEVSEL=medium
```

```
>TAbort- <TAbort- <MAbort- >SERR- <PERR-
Latency: 64 (63750ns min, 63750ns max)
Interrupt: pin A routed to IRQ 5
Region 0: Memory at e8000000 (32-bit, prefetchable)
[size=64M]
Region 5: Memory at ed000000 (32-bit, non-prefetchable)
[size=32K]
Capabilities: [50] #06 [0080]
```

L'antépénultième et l'avant dernière lignes sont remarquables car elles indiquent les emplacements ainsi que la taille et le type de mémoire présent sur la carte (ici une FLIC). On remarque dans le cas présenté ci-dessus qu'il existe deux régions de mémoire manipulant des mots de 32 bits. La première, d'une taille de 64Mega-mot32bits, est associée à BAR0 (Registre d'adresse de base 0). La deuxième est associée à BAR5, et possède 32K-mot32bits de capacité.
Les BARs sont particulièrement intéressants car ils permettent une utilisation très flexible des périphériques PCI. En effet, les adresses servant de référence pour les bancs de mémoire étant allouées dynamiquement lors de l'initialisation du système, on s'assure ainsi que les différentes cartes auront bien des plages d'adresses qui ne se recouvrent pas.

d) Accès à la mémoire PCI avec Linux

Pour écrire dans ces plages mémoire depuis un programme C sous Linux, la façon la plus simple consiste à « mapper » la mémoire, c'est à dire la rendre utilisable comme une partie de la mémoire interne. Lors de cette opération, Linux associe une plage d'adresse de sa mémoire au banc de mémoire à mapper. Les PC n'ont généralement qu'un volume de RAM très réduit par rapport à la plage des adresses possibles (2^{32} adresses possibles), il est dès lors envisageable de donner une plage de ces adresses aux périphériques (ces adresses sont, d'ailleurs, très au delà des dernières adresses RAM). De cette manière, les espaces mémoire des périphériques vont apparaître comme des parties de la mémoire centrale, et c'est Linux qui va gérer de manière transparente les transferts PCI.

Cela se révèle très intéressant du point de vue du programmeur, car on peut ainsi accéder à de la mémoire présente sur une carte comme s'il s'agissait de mémoire classique (avec utilisation de pointeurs, par exemple).

Pour ce qui est de la programmation, mapper la mémoire est la façon la plus simple pour accéder à la mémoire réelle du périphérique (d'autres méthodes sont possibles, mais elles sont beaucoup plus compliquées).

De plus cette opération est possible aussi bien depuis un programme utilisateur (espace utilisateur) que depuis l'espace noyau (zone mémoire où s'exécutent exclusivement les tâches du noyau ou des drivers).

3. Linux et la gestion du bus PCI

Comme exposé auparavant, le système détecte les périphériques PCI lors du démarrage, l'OS est ensuite chargé de les gérer. Alors que certaines cartes sont prises en charge par l'OS (par exemple la carte vidéo), d'autres nécessitent l'exécution de programmes dédiés (cas des cartes expérimentales, par exemple), et donc, ont besoin d'être référencés par l'OS pour pouvoir être accessibles. Différentes solutions ont été retenues par les quelques systèmes d'exploitation du marché avec plus ou moins de réussite. Dans le système Linux, les programmeurs ont défini une structure de données destinée à regrouper les différentes informations d'un périphérique PCI. Les structures sont chaînées entre elles, ce qui permet de parcourir très facilement le bus afin de retrouver une carte en particulier. Afin de pouvoir examiner les périphériques PCI, il convient d'installer, au cas où ce ne serait pas déjà fait, le package « pciutils-XXX.rpm ». Par ailleurs, il est important d'avoir installé le package « pciutils-devel-XXX.rpm » contenant les fichiers d'entête des bibliothèques PCI, afin de ne pas avoir de problème lors de l'édition des liens (linkage) par la suite.
Le simple utilitaire *lspci (fourni dans pciutils)* parcourt la liste chaînée des périphériques PCI et affiche des informations sur les cartes.
Exemple :

```
[gonzalve] gonzalve\> /sbin/lspci
00:00.0 Host bridge: Intel Corporation 440BX/ZX - 82443BX/ZX Host bridge (rev 03)
00:01.0 PCI bridge: Intel Corporation 440BX/ZX - 82443BX/ZX AGP bridge (rev 03)
00:07.0 ISA bridge: Intel Corporation 82371AB PIIX4 ISA (rev 02)
00:07.1 IDE interface: Intel Corporation 82371AB PIIX4 IDE (rev 01)
00:07.2 USB Controller: Intel Corporation 82371AB PIIX4 USB (rev 01)
00:07.3 Bridge: Intel Corporation 82371AB PIIX4 ACPI (rev 02)
00:0e.0 Ethernet controller: Intel Corporation 82557 [Ethernet Pro 100] (rev 02)
00:10.0 PCI bridge: Digital Equipment Corporation DECchip 21154 (rev 05)
01:00.0 VGA compatible controller: ATI Technologies Inc 3D Rage Pro AGP 1X/2X(rev 5c)
02:0f.0 Non-VGA unclassified device: Lucent Microelectronics Orca FPGA (rev 02)
```

La première colonne indique la position de la carte dans l'architecture PCI. Les informations sont disposées de la manière suivante :
< numéro de bus > : < numéro de dispositif > . < numéro de fonction >
Ainsi la dernière ligne nous informe que sur le bus n°2, dispositif n°f, fonction 0, se trouve une FLIC (Non-VGA unclassified device: Lucent Microelectronics Orca FPGA).
De manière générale, Linux fournit toute une bibliothèque de fonctions permettant d'accéder aux cartes PCI. La principale difficulté est de se familiariser avec ces fonctions ; la programmation ne présente ensuite pas plus de difficulté que pour des programmes « classiques ». Malheureusement, ces fonctions ne sont pas les mêmes lorsque l'on travail dans l'espace utilisateur ou dans l'espace noyau. Cette différence vient du fait que les besoins ne sont pas les mêmes pour une programme bas-niveau (noyau) et un programme haut-niveau (utilisateur). Il faut donc pouvoir jongler entre les différentes fonctions lors du développement, ce qui représente une difficulté non-négligeable.

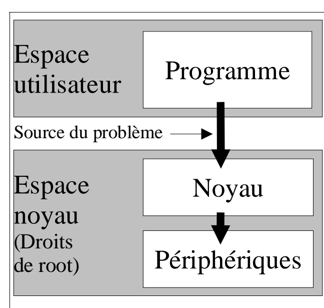

Illustration 10 : Appel de périphériques sans utiliser de driver.

L'illustration 10 montre les différentes couches d'appel d'un programme voulant accéder aux périphériques directement depuis l'espace utilisateur. Il apparaît clairement que le programme, même appelé depuis l'espace utilisateur, accède à des fonctions nécessitant les droits du super utilisateur (désigné par « root » dans les systèmes UNIX). Cela signifie que le programme ne fonctionnera pas s'il est appelé par tout autre personne que root. Par conséquent, le programme devra être exécuté avec les droits de root, avec les nombreux inconvénients que cela entraîne (le développement se fait donc en utilisant le compte root et donc rend les occasions de "casser" le système plus nombreuses ; les erreurs de segmentation peuvent avoir des conséquences graves ; etc...)
Pour résoudre les problèmes d'accès aux périphériques, il convient de développer un driver. Celui-ci fonctionne dans l'espace noyau, mais reste visible et utilisable depuis un programme externe (cf illustration 11). Cette façon de voir le noyau et l'ensemble des périphériques qu'il contrôle comme une entité invisible depuis un programme, et seulement accessible grâce à

une poignée de fonctions, rappelle fortement les principes du modèle objet[2]. En effet, le noyau peut être vu comme un objet dont la partie publique serait appelée le driver (en réalité, *les* drivers).

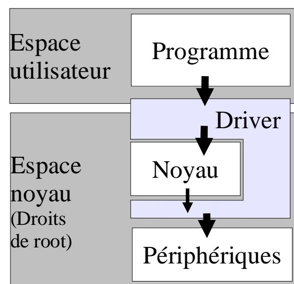

Illustration 11 : Utilisation des périphériques avec un driver.

De manière pratique, le noyau propose un certain nombres de méthodes (on a ici aussi la terminologie du modèle objet) pour appeler des fonctions bas niveau depuis l'espace utilisateur [5].

Seulement, cette transparence n'est en fait qu'apparente, puisqu'il y a en réalité toute une série de couches à traverser avant d'arriver au périphérique voulu. Par exemple, imaginons que l'on veuille lire les données provenant de la souris. A première vue, il est très simple de le faire grâce à la fonction *read* pour laquelle on spécifie le fichier d'entrée de la souris (ie /dev/mouse dans la plupart des Linux). C'est là que le driver intervient : /dev/mouse est un fichier associé au driver de la souris, et le noyau sait alors que lorsque l'utilisateur accède à ce fichier, il doit utiliser les fonctions du driver. Ce dernier fournit donc une version spécialisée de *read* qui renvoie les données en provenance du port souris (illustration 12).

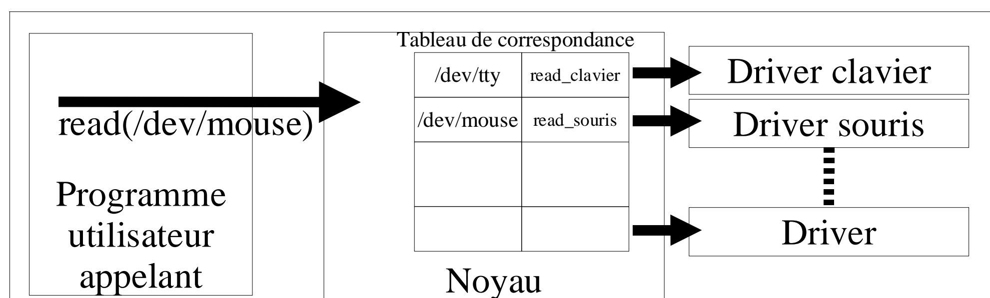

Illustration 12 : Différents appels lors de l'utilisation d'un driver.

Cette technique repose sur les méthodes de gestion des données sous les systèmes UNIX. En effet, l'OS considère l'ensemble des périphériques dont il a la charge comme étant des fichiers (un répertoire est un fichier, la

2 : D'ailleurs, bien que le langage C ne soit pas un langage objet, les techniques de développement des noyaux modernes s'organise de plus en plus vers une vision orientée objet du noyau)

mémoire est un fichier, le port série est un fichier, etc...). Ce choix technologique particulier peut-être déstabilisant pour un néophyte, mais simplifie grandement le développement d'applications (peu importe le type de périphérique avec lequel on veut travailler, on utilise toujours la fonction *open* pour y accéder) Ainsi, la programmation de drivers peut être vue comme une surcharge des fonctions polymorphiques d'accès aux fichiers.

a) Linux et droits d'accès

Tous les systèmes d'exploitation les plus connus (comme MS-Windows ou les systèmes UNIX propriétaires) reposent sur une architecture interne identique ou analogue à celle exposée ici. En effet, tous possèdent un noyau (ou micro-noyau) autour duquel viennent se greffer des couches de contrôle (pilotes de matériel, contrôleurs de périphériques etc... voir illustration 13).

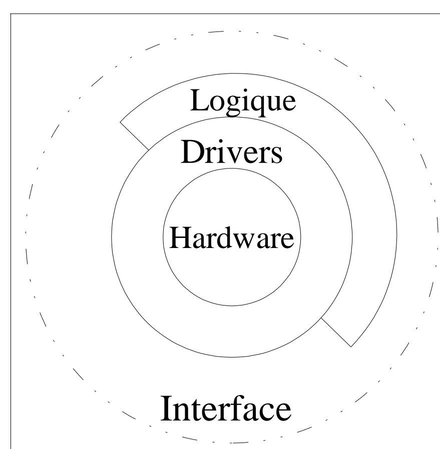

Illustration 13 : Le noyau en détail

Cependant, bien que les schémas de fonctionnement soient similaires, la mise en oeuvre (le code interne) est très différente, c'est pourquoi ce que j'expose ici s'applique plus particulièrement au système d'exploitation Linux.

Linux est un système d'exploitation multi-tâches, multi-utilisateurs. Son noyau est donc chargé de gérer le bon ordonnancement et l'exécution des tâches que l'ordinateur doit exécuter, ainsi que la conservation des droits des différents fichiers auxquels les programmes ont accès. Cette partie du système d'exploitation est donc très importante, mais surtout représente une partie sensible. En effet, ce sont les bugs du noyau qui amènent l'OS à se bloquer (les fameux « écrans bleu de la mort » sous

MS Windows, par exemple). Linux, du fait que les sources sont librement accessibles et modifiables, est devenu un exemple de stabilité du noyau. Pourtant, rien ne sert d'avoir un noyau très stable si un programme quelconque peut venir interférer avec lui. C'est pourquoi Linux (et tous les vrais OS multi-utilisateurs) place une frontière nette entre les processus noyau et les processus utilisateur (avec un statut particulier pour le super utilisateur). Cela permet d'empêcher un utilisateur quelconque d'accéder à des ressources auxquelles il n'est pas sensé accéder (par exemple l'espace mémoire d'un autre utilisateur, ou la mémoire vidéo) et ainsi établit un plus grand contrôle du système (on le rend plus stable).

Comme toute solution, celle-ci ne présente pas que des avantages. Le principal problème dans le cas présent est que les ressources PCI deviennent inaccessibles pour un programme « normal », c'est à dire qu'il est impossible de commander une carte son, par exemple, par un programme utilisateur. Cela peut paraître très paradoxal : à quoi sert une carte que l'on ne peut pas utiliser ? En fait, c'est là le rôle des drivers. Ils sont comme un pont entre l'espace utilisateur et l'espace du noyau. Le driver est généralement un module du noyau qui possède des fonctions spéciales appelables depuis un programme utilisateur et donnant accès aux ressources. Cependant, le super utilisateur (root) garde le droit d'utiliser tous les fichiers de l'ordinateur, et donc, il peut accéder librement aux périphériques. C'est pourquoi, pour des raisons de simplification dans le développement, ou pour effectuer de simples tests, il peut être intéressant d'utiliser des programmes de type utilisateur depuis l'espace du super-utilisateur. Il convient néanmoins dans ce cas de bien savoir ce que l'on fait, car un programme lancé en super utilisateur devenant incontrôlable peut faire de gros dégât sur le système lui même.

b) Version de noyau

Linux est un OS en constante évolution, les développeurs cherchant à coller au mieux à l'évolution des technologies. Pour ce faire, il a parfois été nécessaire de remodeler partiellement ou totalement l'architecture du noyau (récemment, ce sont les périphériques USB qui ont poussé à modifier le noyau). Le noyau 2.4.XX représente une évolution majeure dans le développement des noyaux Linux [6], de nombreux changements ont été apportés. Les drivers actuels font donc face à une difficulté supplémentaire : il est essentiel de garder la compatibilité avec les noyaux 2.2.XX, au moins dans les premiers temps (jusqu'à ce que le 2.4.XX se soit imposé). Cela se répercute sur le code des drivers par une grande quantité de directives de pré-compilation (le pré-processeur du

compilateur ne conserve que les parties de code compatibles avec le noyau pour lequel on compile).

Cela-dit, il est important de noter que la programmation d'applications destinées à être exécutées dans l'espace utilisateur reste quasiment la même, quelle que soit la version du noyau. Cela provient du fait que le noyau n'est que très peu visible depuis un programme utilisateur et du fait que ce sont les drivers qui prennent en charge la plus grande partie des problèmes de compatibilité.

c) Fichiers spéciaux

Pour permettre aux programmes utilisateurs d'accéder aux fonctions du noyau, nous avons vu que l'on faisait appel à un driver. Cela dit, on se rend bien compte qu'un driver seul ne représente pas une solution satisfaisante puisqu'il réintroduit les mêmes difficultés de droits d'accès : le driver ne peut être exécuté dans l'espace utilisateur (dans ce cas il serait tout bonnement un programme simple), et il ne peut être exclusivement en espace noyau (sinon il devient inaccessible aux programmes appelant). Il faut donc qu'il y ait une « porte d'entrée » du driver [7]. Cet accès est fourni par des fichiers spéciaux présents dans /dev/. Ils sont aisément reconnaissables lorsqu'on liste les propriétés des fichiers :

```
[gonzalve] dev$ ls -al hda1 psaux
brw-rw----    1 root     disk       3,   1 Aug 30  2001 hda1
crw-------    1 root     root      10,   1 Aug 13 13:40 psaux
```

La partie intéressante est située en tête des droits d'accès : les lettres c et b sont mises ici pour signifier que ces fichiers sont associés à des drivers. Le c correspond à un driver de caractères (Char driver) et b correspond à un driver de blocs (block driver). Ce sont ces fichiers qui servent de passerelle entre l'espace utilisateur et l'espace noyau. Ainsi pour accéder aux données de la partition hda1, il suffit d'ouvrir en lecture (avec un simple appel à open()) le fichier hda1. Ensuite, toute lecture ou écriture effectuée avec ce fichier conduira à l'appel des fonctions spécifiques du driver de disque dur. Dans la pratique, personne n'accède aux fichiers de cette manière car il y a de nombreuses couches situées au dessus des appels aux disques durs, mais c'est bien ce principe qui est derrière toute lecture.

II.RÉALISATION ET ÉTUDE

1. Choix du type de driver

Il existe deux façons bien distinctes de réaliser un programme accédant aux périphériques. Soit on le destine à être exécuté dans l'espace utilisateur, soit on le destine à être exécuté dans l'espace noyau. Bien évidement, écrire un programme pour l'espace utilisateur est bien plus simple pour quelqu'un n'ayant jamais fait de développement bas niveau pour Linux. On pourrait même lister les différents avantages à utiliser l'espace utilisateur :

- On peut utiliser toute la bibliothèque standard du C.
- On peut utiliser un débogueur classique pour le développement (en évitant les techniques compliquées de déboguage dans l'espace noyau)
- Si le driver se bloque, il est possible de le tuer avec la commande kill.
- Il est possible de « swapper » la mémoire (mettre une partie de la RAM sur le disque dur afin de libérer de la mémoire). Cela peut être utile pour des drivers relativement importants et qui ne sont appelés que relativement rarement.

Mais demeurent les inconvénients :

- Les interruptions ne sont pas gérables depuis l'espace utilisateur.
- L'accès direct à la mémoire n'est possible que par l'utilisation de mmap sur /dev/mem, et cette opération n'est possible que pour le super utilisateur.
- L'accès aux ports E/S n'est possible qu'après l'appel de ioperm ou iopl. Cela n'est possible que pour le super utilisateur.
- Le temps de réponse est plus élevé car un changement de contexte est nécessaire pour effectuer les transfert d'informations entre le client et le matériel.
- Si le driver a été « swappé » sur le disque, le temps de réponse devient extrêmement long.

En conclusion, on se rend aisément compte qu'un driver s'exécutant dans l'espace utilisateur ne peut effectuer qu'un nombre réduit d'opérations. En général, le développement de drivers pour l'exécution dans l'espace utilisateur est utilisé pour « l'exploration » de nouveaux matériels (voir comment une nouvelle carte se comporte), mais un vrai driver se doit d'être exécuté dans l'espace noyau pour être efficace. C'est pourquoi j'ai fait le choix de développer un driver sous forme de module noyau pour la FLIC.

2. POSITIONNEMENT DU DRIVER AU SEIN DU SYSTÈME

Comme présenté auparavant, chaque driver est associé à un fichier spécial présent dans /dev/. Lors de l'installation du driver sous forme de module, la routine d'initialisation indique au noyau à quel fichier il s'associe. Cette opération s'effectue à l'aide de la fonction *register_chrdev(unsigned int major, const string name, file_operation*fop)* pour les char-drivers et par la fonction *register_blkdev(unsigned int major, const string name, file_operation*fop)* pour les block-drivers. Bien que les architectures de ces deux types de drivers soient très différentes, les explications qui suivent sont valables pour les deux.

Reprenons l'exemple à propos des drivers de la souris et du disque dur :

```
[gonzalve] dev$ ls -al hda1 psaux
brw-rw----   1 root     disk   3,   1 Aug 30  2001 hda1
crw-------   1 root     root   10,   1 Aug 13 13:40 psaux
```

Les numéros présents dans le 5ème et 6ème champ sont appelés numéros majeurs et mineurs (Major and minor numbers). Ce sont les vraies références importantes du point de vue du noyau. En effet, lors de l'enregistrement du driver, on indique quel est le numéro majeur que l'on va utiliser (unsigned int major). Le noyau sait alors associer le fichier spécial, et le driver. La liste des drivers références est dans /proc/devices et ressemble typiquement à la liste suivante :

```
Character devices:
 1 mem
 2 pty
 3 ttyp
 4 ttyS
 5 cua
 7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
137 pts
162 raw
180 usb

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md
 22 ide1
227 dump
```

De manière évidente, les valeurs après « Character devices » sont les char-drivers et les « Block devices » sont les block-drivers. Le numéro indiqué est le numéro majeur. Le numéro mineur n'est pas signalé car son utilité est plus réduite. Il sert généralement à différencier différents appels au même driver.

Il est enfin intéressant de signaler que ces fichiers spéciaux ne sont en fait que de simples i-nodes qui ne sont liés à aucun espace sur le disque dur. Il est bien entendu indispensable que le super-utilisateur crée ces fichiers avant l'installation du driver. Généralement cette tâche est réalisée automatiquement par des scriptes shell qui paramètrent le système et installent les drivers sans qu'il y ait besoin de réaliser d'opérations compliquées. Cela dit, il peut être utile de les créer manuellement grâce à la commande mknod. Bien entendu, il est nécessaire d'avoir les droits de root.

```
mknod /dev/monNoeud b 126 0
```

Le premier paramètre est le nom du noeud que l'on souhaite créer, le second est le type de driver associé (c ou b pour (resp) un char-driver ou un block-driver). Les deux derniers sont les numéros majeur et mineur. En général, les valeurs de 60 à 63, de 120 à 127 et de 240 à 254 sont celles réservées pour des applications locales ou expérimentales, c'est pourquoi j'ai choisis la valeur 126 pour les tests du driver de la FLIC. Cela dit, il serait possible de réaliser une allocation dynamique des numéros majeurs, principalement pour permettre l'utilisation des drivers sur le plus grand nombre de PC possible (un driver de carte son doit fonctionner sur des PC qui ont des configurations matérielles, et donc logicielles très diverses). Dans le cas de la FLIC, l'allocation dynamique ne m'a pas parue utile car le module de la FLIC est destiné à être installé une bonne fois pour toute sur une machine (cela n'empêche pas les tests, mais l'allocation statique permet d'alléger le programme et donc de faciliter le déboggage).

Dans tous les cas, il faut penser à changer les droits d'accès sur le fichier (chmod 666 monNoeud), sans quoi le driver ne sera pas accessible par les utilisateurs standards, et donc sera inutile.

3. MODULE DU NOYAUX

Afin de pouvoir accéder à la mémoire de la FLIC par le biais d'un programme s'exécutant dans l'espace utilisateur, il est nécessaire de recourir à un module noyau (un « driver »).

Le module réalisant les transactions d'écriture et de lecture sur la FLIC doit, de manière évidente, fournir des fonctions de lecture/écriture pour des mots de 32 bits, mais aussi pour des blocs de mémoire. De plus, il doit être possible d'accéder aux bancs de registres (adressés en BAR5).

De manière générale, le module se compose de deux fichiers (flic_mod.c et flic_mod.h annexe B) dont il est essentiel de comprendre la fonction. Flic_mod.c est le fichier dans lequel est écrit le code alors que flic_mod.h est un fichier d'entête classique, si ce n'est qu'il sert de référence pour le module ET pour les programmes utilisateurs. En effet, ce fichier indique quels sont les valeurs des numéro de fonction à passer lors de l'appel de la fonction ioctl(). De plus, il implémente des fonctions « écran » afin de rendre les appels noyau les plus transparents possible.

Il est cependant nécessaire de bien séparer les deux parties de ce fichier (la partie destinée au module et la partie destinée au programme utilisateur) lors de la compilation, car les déclarations ne sont bien entendu pas les mêmes du côté utilisateur et du côté noyau. C'est pourquoi on utilise des directives pour le préprocesseur. Ici, il s'agit juste de savoir si le fichier est lu pour compiler le module ou pour compiler un programme. Or une « variable » existe déjà pour les fichiers sources du noyau, il s'agit de MODULE. Le code s'articule donc autour du test de cette variable :

```
#ifdef MODULE
code destiné au module
#else
code destiné aux programmes utilisateurs
#endif
```

Le préprocesseur identifie ainsi les parties du code à garder pour chacun des deux cas. Il est important de noter que ces identificateurs destinés au préprocesseur font partie intégrante de la programmation bas niveau dans les systèmes Linux. Ils sont, en effet, les seuls moyens de pouvoir fournir dans un même fichier, du code supportant différentes architectures (par exemple PCI 32 bit ou 64bits) ou plusieurs versions (Noyau 2.2.XX ou 2.4.XX). Cela rend cependant le code relativement difficile à lire et à comprendre, car il faut pouvoir se « représenter » le code dans les différentes versions.

Une autre particularité à noter est que l'on utilise ici un « Block driver » (autorisant les accès à n'importe quel endroit de la mémoire et autorisant les transferts par blocs). Cette approche impose l'utilisation de la fonction ioctl() dont j'ai déjà parlé précédemment.

Cette fonction est le véritable pont entre l'espace utilisateur et l'espace noyau. Elle est relativement simple à utiliser mais la programmation doit, bien entendu, suivre le format très rigide de cette fonction, ce qui peut parfois amener à complexifier certaines parties du code.

```
int ioctl(int fd, int cmd, ...);
```

où fd est un descripteur de fichier et cmd est le numéro associé à la fonction du noyau que l'on souhaite exécuter.

On remarque que ce prototype possède les « ... » servant généralement à indiquer que la fonction a un nombre de paramètres variable, le compilateur remplaçant ce « faux » prototype par le réel (avec le nombre de paramètres correspondant au cas précis). Or, dans le cas de ioctl(), il est bien entendu impossible que le compilateur change, ou plus simplement ajoute une fonction ioctl() aux fonctions du noyau.

En fait, « ... » n'est pas utilisé ici pour dire qu'il y a un nombre variable de paramètres. C'est une « astuce » de programmation permettant de s'affranchir du contrôle de type réalisé par le compilateur. Il faut cependant prendre garde à ne pas passer plus de 3 paramètres dans ioctl() car le compilateur ne signalera aucune erreur, mais l'exécution du programme conduira certainement à l'erreur de segmentation. Le dernier paramètre est donc bien optionnel, mais il est toujours présent (le compilateur lui assignera une valeur quelconque s'il n'est pas utilisé) et peut être de tout type.

On utilise alors ce dernier paramètre pour envoyer des données au noyau, mais comme il doit rester unique, il faut avoir recours à des structures afin de pouvoir passer plus d'un élément, ce qui peut se révéler plutôt complexe et lourd. C'est pourquoi j'ai utilisé ici des fonctions intermédiaires (définies dans flic_mod.h) afin de rendre ces appels plus simples de la part d'un utilisateur :

Un programme voulant lire dans la mémoire de la FLIC utilisera la fonction *flic_read_u32(int fd, u32 pos)* où fd est le descripteur de fichier du driver et pos est l'adresse mémoire à lire.

Cette fonction implémente une lecture par l'utilisation de ioctl() et d'une structure spéciale appelée *struct flic_data_to_transfert* (définie dans le même fichier), mais cela n'est pas vu par le programmeur du programme appelant.

4. DÉBOGUAGE DU MODULE

Une fois les différents tests assurant la fiabilité de la FLIC effectués, il est enfin possible de déboguer le module (le driver) en détail. Il est important de noter que cette partie est particulièrement délicate, car un module s'exécute dans l'espace noyau, et donc toute erreur (erreur de segmentation par exemple) entraîne des conséquences bien plus importantes que la simple suppression du programme de la table des processus, comme c'est le cas pour un programme exécuté dans l'espace utilisateur. Cependant, le noyau possède un nombre réduit de fonctions utilisables pour le déboguage et, de plus, les erreurs d'accès mémoire peuvent avoir de graves

conséquences (dans les cas extrêmes, on peut même imaginer qu'on détériore le matériel). Généralement, le blocage d'un module durant son exécution entraîne tout d'abord la perte de contrôle sur le module (sans surprise) mais aussi le blocage de certaines fonctions d'accès au noyau. Dans ce cas, il n'y a pas d'autre alternative que de redémarrer la machine (de manière plus ou moins contrôlée selon le type et la gravité du problème).

Dans l'espace noyau encore plus qu'ailleurs, la fonction affichant des valeurs sur la sortie standard est essentielle (sans elle il est quasiment impossible de savoir ce qui se déroule réellement lors de l'exécution). Comme déjà précisé auparavant, les fonctions du noyau ne portent pas les mêmes noms que les fonctions standards. Cela dit, le nom change, mais cela n'est pas systematique (voir rare) pour ce qui est de prototype. Cela permet alors de se familiariser relativement rapidement avec ces fonctions. Par exemple *printf(const char *format, ...)* trouve son équivalent sous le nom de *printk(const char *format, ...)*, la seule différence étant qu'il est possible de donner une priorité au message en mettant au début de la chaîne de format une valeur de priorité (définies dans <linux/kernel.h>). Par exemple lorsqu'on redémarre la machine (avec la commande reboot), le noyau utilise

```
printk("<0> Le système va redémarrer.\n")
```

ce qui a pour effet d'envoyer un message à toutes les personnes connectées afin de les prévenir. Le <0> est la valeur de priorité ; 0 signifie que ce message est de la plus haute priorité.
Ces messages sont générés par le noyau, et sont généralement (et préférablement pour des raisons de sécurité) affichés sur les consoles de root uniquement. Cela est possible grâce à une variable d'environnement des consoles qui indique à partir de quel niveau de priorité le message va être affiché. De manière générale, le super-utilisateur n'est pas toujours connecté ni prêt à lire tous les messages provenant du noyau, c'est pourquoi ces messages sont archivés dans le fichier `/var/log/messages` . Il est alors possible au super-utilisateur de venir consulter les messages lorsqu'il en a le temps et l'envie.

5. GÉNÉRATION DE C-TAGS

La partie principale du maître TAGnet consiste à envoyer des adresses de processeurs libres aux esclaves TAGnet afin que ces derniers équilibrent la charge. Du point de vu logiciel, le scheduler reçoit des signaux via le réseau SCI de la part des processeurs libres du tore, et il doit transmettre leur

adresse sous forme organisée à la FLIC afin qu'elle émette des TAGs illustration 14).

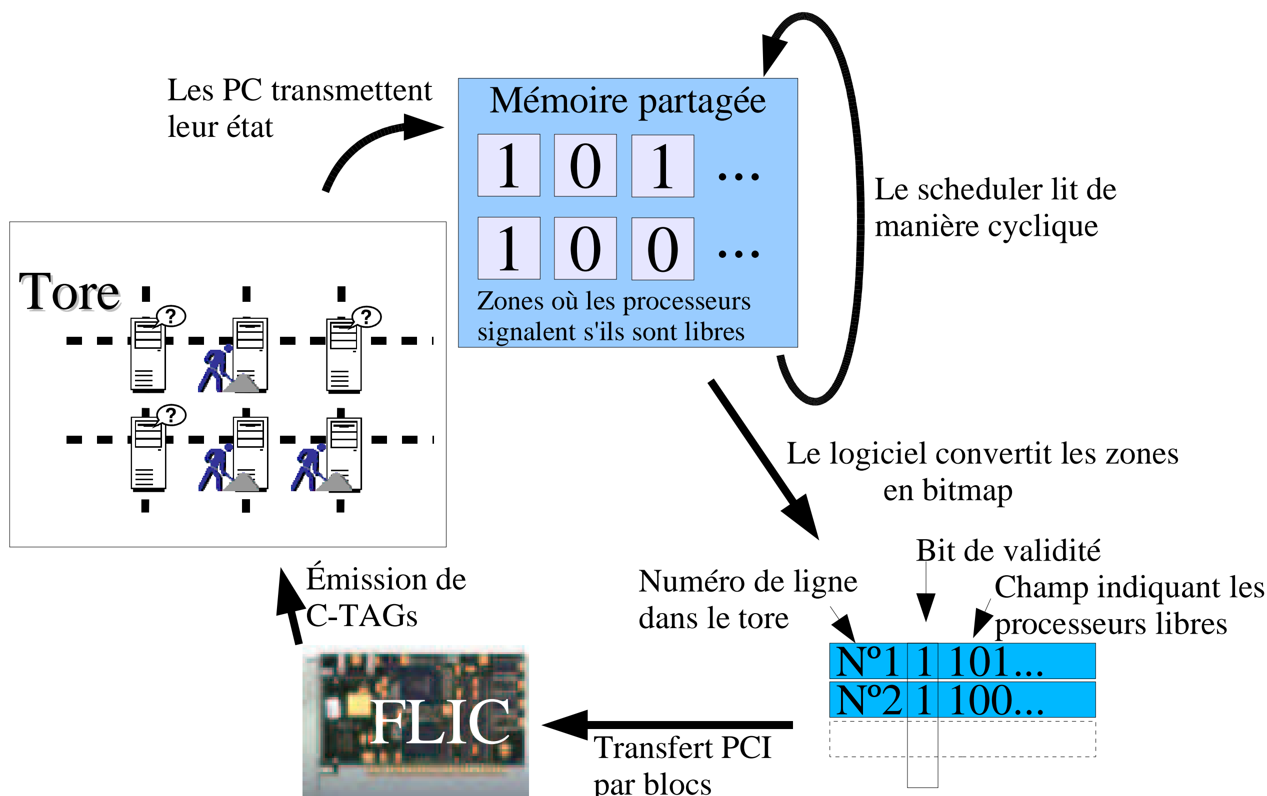

Illustration 14 : Boucle de fonctionnement pour les C-TAGs

Le principe est que l'on ne doit envoyer un ordre de travail qu'aux processeurs libres. Or le temps de traitement n'est pas constant entre les processeurs[1], cela implique que leur libération peut être considérée comme aléatoire. De ce fait, il n'y a pas de moyen pour savoir quel processeur va se libérer à un moment donné. La solution la plus simple et la moins coûteuse en terme de durée, est la lecture cyclique des valeurs décrivant l'état des processeurs. Ceci est très facile sous condition que les processeurs peuvent communiquer leur état à une mémoire partagée entre tous les CPUs du tore. Cette possibilité est pourvu par le réseau SCI (le protocole SCI assure la cohérence de données en mémoire partagée).
Cette zone est en fait un espace de mémoire partagée fragmentée entre les différents PC esclaves. Chacun a son propre espace et doit y placer une valeur non nulle lorsqu'il devient libre. Le scheduler, de son côté, prend en

[1] il ne l'est pas avec un ensemble de processeurs identiques, et dès lors, on peut considérer que l'uniformité du parc n'est pas importante, seule le temps moyen de traitement de l'ensemble représente une valeur critique.

compte cette valeur lorsqu'il lit cette partie de la mémoire, et la réinitialise. Cette étape parait évidente, mais il faut cependant prendre quelques précautions en ce qui concerne la reinitialisation. En effet, il s'agit ici de mémoire partagée sur laquelle un des deux utilisateurs opère des accès « aléatoire ». Il faut donc arriver à garder une cohérence dans l'ordre des écritures. En fait le problème n'apparaît que si le scheduler réinitialise TOUTES les valeurs après lecture. En effet, rien ne dit qu'entre la lecture et l'écriture réalisée par le scheduler, il n'y a pas eu lieu une écriture de la part du PC esclave. Ce cas serait très problématique car un PC serait « perdu ». Une solution simple a été trouvée : il s'agit de ne remettre à 0 QUE les zones dans lesquelles on vient de lire une valeur non nulle. Dans ce cas on peut être sûr que le PC esclave ne va pas faire d'écriture car il s'est déjà signalé comme étant libre.

Une fois les zones mémoires non nulles connues, le scheduler est chargé de transmettre les noms des processeurs libres à la FLIC afin que cette dernière génère les C-TAGs. En réalité, on se contente seulement d'envoyer des mots de 32 bits contenant toutes les informations sur l'état du tore au moment où le scheduler a effectué ses lectures. Chaque mot est constitué comme montré sur l'illustration 15.

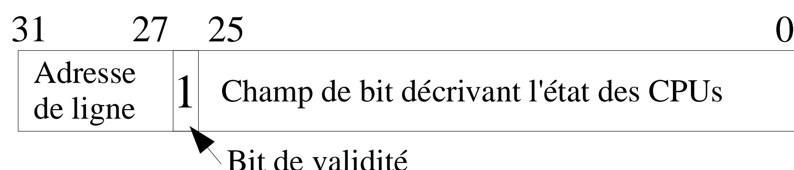

Illustration 15 : Champ de bit pour les C-TAGs

De son côté, la FLIC met à jour une table interne des adresses des CPUs auxquels elle peut envoyer un ordre de travail. Cela est fait de manière transparente pour le scheduler, c'est à dire qu'il ne vérifie pas si les écritures se sont bien déroulées. En fait, cette méthode repose sur le fait que l'électronique de la FLIC peut lire les données bien plus vite que le bus ne peut lui les passer.

Techniquement, la mise à jour se traduit par un simple OU logique entre le tableau de bits déjà stocké dans les registres du FPGA et le champ de bit transféré par la partie logicielle. De cette manière, on est certain de ne jamais « perdre »[1] de processeurs, puisque la seule opération possible est d'ajouter de nouvelles références.

1 Cependant, le cas d'un processeur « perdu » n'est pas si grave car, au dela d'une certaine durée d'attente, les CPUs redemandent un CTAG.

III.RÉSULTATS ET VALIDATION

1. TEST DE LA FLIC

La FLIC est un élément essentiel du scheduler TAGnet, c'est pourquoi nous avons consacré beaucoup de temps à sa mise au point. Le système complet se base sur des transferts de données entre un programme en exécution sur le PC maître et la RAM de la FLIC. Ces transferts mettent donc en jeux des compétences en programmation (partie logicielle), mais aussi des compétences en programmation de FPGA. Je m'occupais, pour ma part, de la programmation du logiciel réalisant les appels PCI, alors que l'ingénieur en électronique était chargé du développement du code VHDL destiné au FPGA ORCA situé sur la FLIC.
Les principales difficultés rencontrées dans le développement « matériel » provenaient (comme habituellement dans ce type de travail) des différences manifestes entre la documentation fournie par le constructeur et le comportement réel du FPGA. À cela sont venus s'ajouter des problèmes avec le simulateur qui ne réussissait pas à reproduire un comportement satisfaisant du FPGA.
De mon côté, je me suis d'abord appuyé sur des programmes simples utilisés jusqu'alors pour effectuer de simples lectures/écritures sur la carte, afin de mieux me familiariser avec les fonctions PCI de Linux.
Ce premier apprentissage m'a permit d'aboutir sur le programme « testram » dont le fonctionnement est décrit en détail dans l'annexe E. Son principe de fonctionnement est relativement simple :

- Détecter la FLIC que l'on souhaite tester (parmi plusieurs FLIC).
- Effacer la plage mémoire que l'on souhaite tester
- Écrire dans la plage mémoire à tester
- Relire les valeurs contenues dans la RAM, et les comparer

Une des premières parties sensibles est l'effacement de la mémoire. En effet, cette opération n'est en fait qu'une écriture. Or on cherche à tester le processus d'écriture sur la FLIC. Il convient donc de ne pas vouer une confiance trop élevée dans cette étape, au moins dans un premier temps. Une étude plus poussée des premiers résultats montre clairement (et sans surprise) que les problèmes d'écriture apparaissent lorsque le taux de transfert PCI devient trop important (Lors des premiers tests, le « burst » ne fonctionnait pas correctement). La solution la plus simple consiste à rajouter des retards dans l'exécution du programme de test de la RAM. Après l'effacement de chaque mot de la RAM, le programme fait une pause (relativement courte : environ 1µsec), puis reprend avec la prochaine

écriture. Cette méthode permet de s'assurer que la mémoire RAM est effectivement effacée, mais elle augmente considérablement la durée du test (jusqu'à 3 minutes pour tester 64 Mega-mot32), ce qui est assez ennuyeux.

Ensuite, il convient de se pencher sur le test des accès RAM. En effet, ce qui nous importe ici c'est de pouvoir s'assurer que les transferts (lectures/écritures) se font suffisamment rapidement. C'est pourquoi les transferts se font ici de la manière la plus rapide possible, afin de voir si la FLIC « supporte » une telle vitesse. Les premiers essais se sont révélées plutôt décevants, de part la présence d'un nombre d'erreurs beaucoup trop important.

```
[root Outils] >  testram S 2:f M 0 100000 T OB
 M function recognised

using /usr/share/pci.ids in order to identify card
Detecting Orca on 2:f....[  OK  ]

 Resetting
reset finished

test begining @ : 0

writing @ 0x40568000
0x000590fa  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x000590fb  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x00059118  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x0005cb22  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x0005cb23  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x0005cb24  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x00065b02  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x00065b03  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x00065b20  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x0006be5e  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x0006be5f  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x0006be60  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x0006e46a  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x00071e9a  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
0x00071e9b  [ 0x12345678 ]  [ 0000000000 ]  [FAILED]
reading.... -
Termiated  @ fffff
```

Illustration 16 : Exemple de mauvais résultat lors du premier test de RAM

L'illustration 16 présente les résultats obtenus lors des premiers tests de la mémoire SDRAM de la FLIC. Les première lignes sont relatives à la détection et à la réinitialisation de la mémoire. Le tableau venant ensuite, présente les erreurs rencontrées. La première colonne désigne l'adresse

mémoire posant problème, la seconde la valeur écrite lors du test et la troisième la valeur relue. La dernière colonne indique seulement si les valeurs correspondent (cela n'est pas très utile dans cet exemple, mais il est possible d'afficher toutes les valeurs, bonnes et mauvaises, et dans ce cas, il est plus facile de repérer les erreurs).

De manière générale, la principale cause d'erreurs était le bridge PCI (composant permettant de créer un « sous-bus » PCI et d'isoler galvaniquement ces deux bus) présent sur la FLIC. En effet, il est apparu que ce dernier avait un comportement assez contradictoire avec celui décrit par ses spécifications. Après de nombreuses modifications et améliorations apportées sur le code VHDL, l'ingénieur en électronique a réussi à assurer un comportement stable et conforme aux exigences du projet.

Un lecteur attentif aura remarqué sur l'illustration 16 que le programme est exécuté avec les droits de root. Cela n'est bien entendu pas une coïncidence : le programme de test de la RAM s'exécute entièrement en mode utilisateur. Cela est principalement dû au fait que ce programme se doit d'être très simple si l'on veut que les utilisateurs pour lesquels il est destiné s'en servent effectivement. Il parait peu réaliste de demander à un utilisateur, désirant simplement tester le bon fonctionnement de sa FLIC, d'installer un module noyau ou de recompiler son noyau Linux avec de nouvelles fonctionnalités. Dans le principe, ce programme doit être rapproché des utilitaires comme *lspci*, ou *lsmod* : des utilitaires de diagnostic simples et efficaces.

2. Utilisations du module

Le principal intérêt de l'utilisation d'un module est qu'il devient très simple de séparer entièrement les fonctions de gestion des transferts vers la FLIC et le code de l'algorithme lui même. Les nombreuses fonctions destinées à mapper la mémoire, à lire les adresses dans les registres d'adresse de base, disparaissent du code des programmes utilisateur (car elles sont dorénavant dans le module). De plus, le déboguage devient plus aisé car les erreurs algorithmiques n'ont de répercutions que dans l'espace utilisateur, alors que le module, lui, reste stable. Il ne faut cependant pas oublier que ce gain se fait grâce à une bonne conception du driver, et qu'il peut arriver de trouver des bogues dans le module lors de cette étape. Dans ce cas, la seule solution reste souvent de redémarrer la machine et de se replonger dans le code du driver.

De manière plus globale, l'utilisation d'un module permet un interfaçage aisé avec d'autres types de programmes. En effet, dans la plupart des cas,

les accès au matériel sont difficiles lorsqu'on utilise des logiciels « haut niveau ». Mais avec l'utilisation de modules noyau, les accès sont rendus transparents, et donc simples. Pour reprendre l'exemple de la carte son, les modules de gestion de la carte sont chargés au lancement de Linux et ensuite sont exploités par des programmes très évolués et de très haut niveau (xmms, CDplayer etc...) dont le code peut être réalisé dans des langages peu destinés à la gestion de périphériques (C++, JAVA, Mathlab?).

J'ai ainsi réalisé un petit programme LabVIEW exploitant le module de la FLIC. Cela permet d'offrir une interface graphique (appréciée des utilisateurs novices) en ne perdant que très peu de temps avec la réalisation de l'interface graphique elle même puisque LabVIEW offre des outils très évolués dans ce sens.

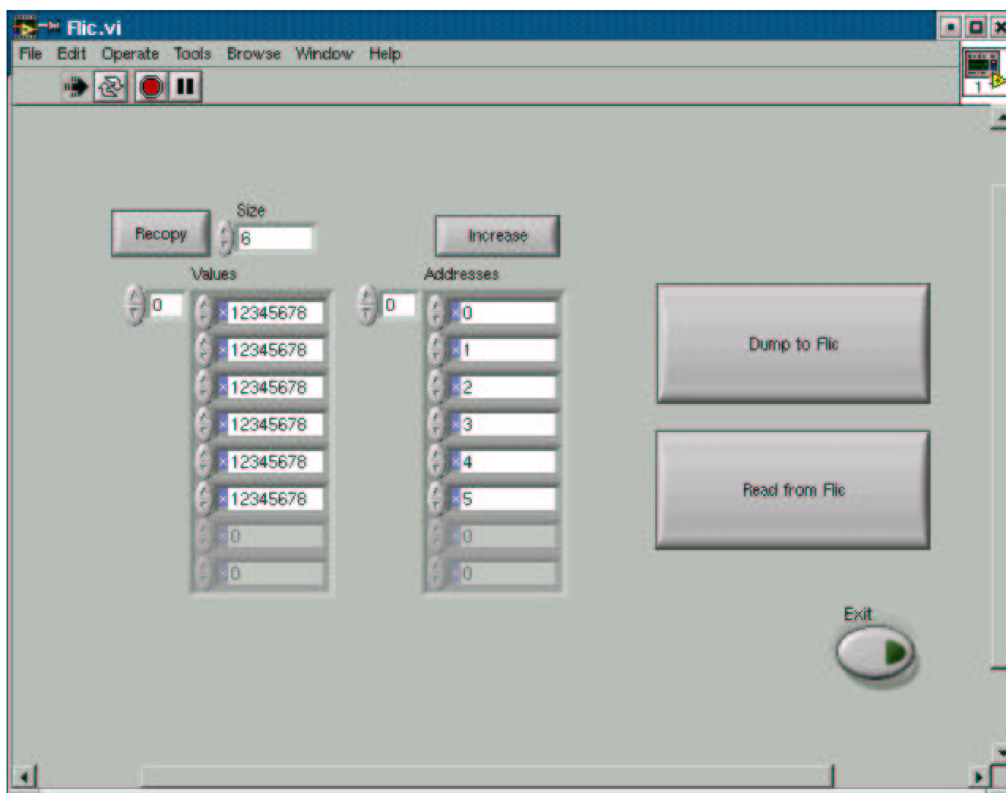

Illustration 17 : Exemple d'utilisation du module avec LabVIEW

L'exemple de l'illustration 17 n'est qu'une simple application d'aide au déboguage lors d'utilisations de la FLIC à des fins de mesures (la FLIC effectue les mesures, elle stocke les résultats, il faut donc un logiciel tiers pour les relire). Cependant la complexité du programme haut niveau ne dépend pas du module, qui lui ne fait qu'assurer les lectures et écritures.

3. TESTS DE VITESSE

Le principal objectif du maître TAGnet est de pouvoir faire face au flot de données provenant des détecteurs avec une fréquence de 1MHz. C'est pourquoi les premiers essais se sont axés sur la faisabilité de tels taux de transfert. L'algorithme de test était relativement simple :

```
début :
   -parcourir la mémoire partagée à la recherche de valeurs
   non nulles
   -mettre en forme les paquets de 32 bits à    envoyer
   -vérifier que les mots précédents ont été lus
   -transférer les données sur la FLIC
   -retourner au début
```

Les premières mesures se sont révélées relativement mauvaises, il fallait plus de 2µsec pour chaque itération. Cette valeur était d'autant plus surprenante que le test du programme seul (sans la partie de transfert PCI), se révélait presque 8 fois plus rapide. En fait, la partie pénalisante en terme de vitesse c'est révélée être la lecture dans la mémoire RAM de la FLIC. Cette lenteur apparente provient du mode de fonctionnement du bridge PCI. En effet, lorsqu'on lit une valeur en RAM, le bridge cherche à optimiser les accès RAM, et lit 16 valeurs consécutives. Or cela pénalise gravement notre programme qui doit donc attendre 16 lectures alors qu'il n'en a demandé qu'une. Cela dit, ces valeurs, bien que décevantes, étaient déjà dans la fourchette acceptable de durée d'exécution. Cependant, il faut s'assurer que la marge libre est assez grande car le PC maître n'a pas pour unique tâche de transmettre des C-TAGs, mais il a aussi une mission de surveillance. Il faut donc prévoir que ces traitements supplémentaires auront un coût en terme de cycles d'horloge, ce qui se traduira par un ralentissement (minime, mais non négligeable) du programme. C'est pourquoi, suite à ces mesures peu conformes aux attentes, il a été décidé de modifier le mode d'accès à la FLIC[1].

Pour effectuer des tests plus poussés, nous avons utilisé un logiciel couplé à une carte PCI servant de traceur numérique (outil permettant de visualiser les signaux passant sur le bus PCI). Ce type d'outil est très cher, mais il permet, outre de visualiser les informations circulant sur le bus PCI, de générer des signaux afin de vérifier le comportement des cartes.

L'illustration 18 montre les cycles d'écriture réalisés par la première version du logiciel du scheduler TAGnet. La mesure du nombre de cycle d'horloge nécessaire pour un transfert complet vers la FLIC donnait une première

[1] C'est pourquoi l'algorithme présenté dans ce paragraphe ne correspond pas à celui décrit dans II.5 Génération des C-TAGs

Illustration 18 : Signaux mesurés sur le bus PCI à l'aide d'un traceur numérique

durée de 2.28µsec pour les transfert d'un mot de 32 bits. C'est en partie suite à ces tests qu'il a été décidé de modifier l'algorithme.

4. Validation d'une boucle du système TAGnet

a) Nouvel algorithme

Afin de résoudre les problèmes de temps d'accès sur la FLIC, il a été décidé de modifier l'algorithme de transfert des mots. En effet, les tests montrent clairement que la partie demandant le plus de temps à se réaliser est le test cherchant à déterminer si l'emplacement où l'on écrit sur la FLIC est libre. Or, ce test n'est pas réellement indispensable puisqu'on peut considérer que la durée de traitement par l'électronique de la carte est bien plus courte que la durée nécessaire pour la mise en forme des données (par le logiciel) et le transfert.

Ainsi, dans la version (celle définitivement retenue et implémentée), on se contentera de transférer les blocs de mots vers la FLIC (en mode rafale), tout en considérant que la carte est toujours prête à les recevoir.

b) Tests avec le nouvel algorithme

Il s'agissait ici de voir à quelle vitesse se faisaient les transferts vers la FLIC. Le principe de ce test était relativement simple, on effectue 1000 transferts, tout en mesurant le temps mis par le programme.

Bien entendu, comme toute mesure, elle implique une modification de ce que l'on observe. Ici, cela se traduit par une perte de temps lors du déclenchement du chronomètre, et lors de son arrêt. Cela dit, nous pouvons considérer que les erreurs induites par ces deux étapes sont négligeables devant la durée totale des transferts. En effet, j'ai estimé la durée de déclenchement du chronomètre logiciel à environ 1µsec, idem pour son arrêt. L'erreur représente donc 2µsec sur la durée totale d'environ 23000µsec, c'est à dire moins de 0.01%. On comprend alors bien ici l'importance de d'effectuer un grand nombre de transferts (pour que les durées de déclenchement soient bien négligeables devant la durée totale).

```
Local segment (id=0x80400, size=1024) is created.
Local segment (id=0x80400, size=1024) is created.
Local segment (id=0x80400) is mapped to user space.
The physical address for local segment is :2f6000
Local segment (id=0x80400) is available for remote
connections.
Waiting for the DMA transfer to be ready ....
Node 8 received interrupt (0x0)

DMA transfer done!

Client data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1024 1024 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
The local segment is unmapped
The local segment is removed
Detecting Orca on 2:f....[  OK  ]
Physical address = 2f6000
duration for the writing of 16*1000 32bits WORDS : 22843 usec
```

Illustration 19 : Écran de sortie après le test SCI-Transfert FLIC

c) Validation avec transfert SCI

Afin de s'assurer du bon fonctionnement de l'ensemble de la boucle TAGnet (illustration 20), il nous faut valider chaque partie une à une. Ici, il s'agit de réaliser et de valider l'interfaçage des fonctions chargées du transfert SCI (signalant les processeurs libres) et des fonctions de traitement et de transfert, à proprement parler.

Les transfert SCI et la mise en place de la mémoire partagée ont été mis au point par un ingénieur en informatique avec lequel je travaillais. Cette partie consiste à créer et superviser un espace de mémoire partagé par deux PC reliés par un réseau SCI. On crée ainsi un segment de mémoire sur un des PC, qui est ensuite référencé par le deuxième grâce au protocole SCI. Les deux machines accèdent donc à la mémoire comme s'il s'agissait d'un espace mémoire classique, toute la supervision et les transferts étant gérés de manière transparente.

La carte SCI que l'on utilise est fournie avec un driver Linux. Il est donc possible là aussi d'accéder aux données depuis l'espace utilisateur. Pour le test du morceau de la boucle TAGnet concernant les transfert SCI et le scheduler TAGnet, il a donc fallut regrouper les fonctions d'accès aux cartes SCI et FLIC, tout en conservant, bien entendu, l'algorithme de transfert en mémoire FLIC.

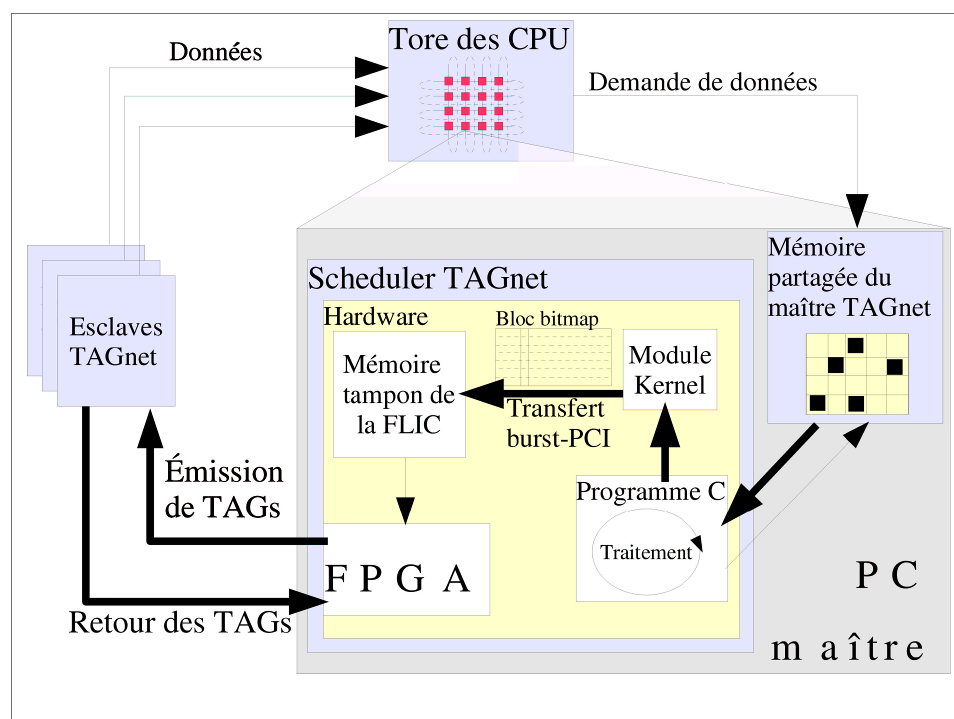

Illustration 20 : Boucle TAGnet complète

Les deux parties (SCI et FLIC) étant suffisamment bien séparées, le déboguage a été relativement aisé, cette étape s'est donc déroulée sans problème majeur.

Pour le test à proprement parler (mesures de vitesse, vérification du bon transfert des données) nous avons utilisé la MCU pour écrire des valeurs dans la mémoire partagée (elle simule ainsi les signaux de libération provenant des processeurs du tore), alors qu'un PC de test équipé d'une carte SCI et d'une FLIC, exécutait le programme du scheduler.

Le principe utilisé était simple : La MCU, après avoir établi le lien SCI assurant l'espace de mémoire partagée, écrit de manière aléatoire à des emplacements de cette mémoire. Cependant, on considère (comme dans le fonctionnement réel du système complet) que seul les cases mémoires à 0 peuvent être mises à 1. Cela est répété un grand nombre de fois (voir un nombre infini de fois) afin de pouvoir effectuer facilement les tests sur le PC maître.

De son côté, le programme du scheduler TAGnet commence par établir les accès à la mémoire partagée, puis exécute l'algorithme présenté plus haut. En affichant les « paquets » transférés, on peut vérifier que les données sont effectivement envoyées à la FLIC (annexe F). Bien entendu, lorsque l'on affiche les données transférées, l'algorithme perd beaucoup en vitesse (chaque itération peut prendre jusqu'à 700µsec), c'est pourquoi pour une utilisation normale, il n'y a pas d'affichage durant les transferts.

La mesure avec le traceur PCI (illustration 21) donne 800 cycles pour une itération de l'algorithme.

Illustration 21 : Résultat obtenu avec le traceur PCI

Avec un bus à 33MHz, cela donne 24µsec par itération, ce qui correspond à la valeur établie avec le chronomètre logiciel. On notera qu'on a ici le temps pris par une itération, c'est à dire le temps qu'il faut pour lire la mémoire partagée, mettre en forme les données (bit-field), et transférer les 16 mots de 32bits. Cela dit le temps pris par le transfert lui même est de 88 cycles, c'est à dire qu'il représente environ 10% du temps, le reste étant dû au traitement sur le PC. On imagine alors bien que les performances seront améliorées avec l'utilisation d'un PC 64bit-66MHz et avec un processeur plus puissant (ici on utilisait un 800MHz).

On remarquera que les transferts PCI ne se font pas d'un seul coup, ce qui pénalise les performances, mais cela est dû au fait que les transferts sur la FLIC ne sont pas prioritaires. Cela trouverait une solution en assurant un transfert en tant maître sur le bus PCI, mais cela nécessite de modifier le mode de fonctionnement de la logique sur la FLIC.

Cependant, ce résultat est déjà largement satisfaisant. Cette modification assurera simplement que les transferts ne sont pas interrompus par d'autres plus prioritaires pour le noyau.

CONCLUSION

Le travail que j'ai effectué au CERN est destiné à servir dans le système TAGnet supervisant le contrôle d'une architecture parallèle de processeurs. Je devais concevoir un programme capable de fournir des adresses de CPU avec une fréquence supérieure ou égale à 1MHz.

J'ai d'abord consacré une période assez longue à l'apprentissage du fonctionnement et des méthodes de programmation du noyau Linux. Parallèlement à cela, j'ai développé des petits outils d'accès et de test de la mémoire de la FLIC, ce qui était une bonne façon de mettre en pratique les notions que je venais d'apprendre.

Par ailleurs, nous avons eu de nombreuses discussions à propos du protocole TAGnet, afin de conjuguer au mieux les besoins, les contraintes techniques et les contraintes liées au protocole lui même. Au fil des semaines, nous avons réussit à nous mettre d'accord sur un mode de fonctionnement, et c'est cette dernière qui a été implémentée. Cela m'a permis d'appréhender les démarches propres à l'ingénieur que sont l'analyse, l'assimilation des travaux existants et la prise en compte de problèmes matériels ou techniques.

La mise au point d'un module noyau a été un choix judicieux car elle permet désormais de modifier le programme principale sans se soucier des appels bas niveau. De plus, il est toujours préférable d'exécuter les programmes en espace utilisateur, et cela n'est possible qu'avec un driver.

Sur un plan plus personnel, il est indéniable que travailler dans un environnement aussi diversifié, autant sur le plan des compétences que sur le plan des nationalités, est sans nul doute une expérience humaine extraordinaire. J'ai ainsi pu développer mes compétences en langues étrangères et en communication.

Enfin, j'ai découvert le fonctionnement du noyau de Linux, ce qui présente une partie très intéressante de mes nouveaux acquis. En effet, cette partie de l'informatique (probablement trop liées à l'OS que l'on utilise) n'est pas abordée en cours sous le biais de la programmation à proprement parler.

Le programme actuel fonctionne de manière satisfaisante, mais il ne constitue qu'une première couche d'un programme de supervision plus global. Cela dit, les applications possibles sont bien plus diversifiées car la FLIC est une carte créée pour être utilisable dans de nombreuses expériences ou applications. Les programmes l'accompagnant ont donc vocation à être utilisés dans d'autres contextes que le TAGnet.

RÉFÉRENCES BIBLIOGRAPHIQUES

[1] TAGnet 1MHz link protocol for event-coherent DMA transferts from source-buffers. LHCb note 2002-XX

[2] TAGnet A hight rate eventbuilder protocol to be presented at 8th workshop on electronics for LHC experiments, Colmar 9-13 Sept 2002

[3] www.pcisig.com ou PCI Local Bus Specification Revision 2.2 December 18, 1998

[4] Tom Shanley/ Don Anderson, PCI system architecture, third edition MindShare, 1995

[5] Remy Card/Eric Dumas/Franck Mevel, The linux kernel book, John Wiley & Sons, 1997

[6] A.Rubini & J. Corbet, LINUX device drivers, Second edition, O'REILLY, 2001.

[7] D. Bovet & M. Cesati, Understanding the LINUX kernel, O'REILLY, 2001

ANNEXES

TABLE DES ANNEXES

Annexe A : Présentation du CERN..III

Annexe B : Code du module..VI

Annexe C : Code du scheduler...IX

Annexe D : Manuel du programmeur...XIII

Annexe E : Code Testram.c...XVII

Annexe F : Contenu des transferts vers la FLIC...........................XXIV

ANNEXE A
PRÉSENTATION DU CERN

1. LE CENTRE DE RECHERCHE

a. Origines et activités

Le CERN (Centre Européenne pour la Recherche Nucléaire), est le plus grand centre mondial de recherche en physique des particules. Situé a la frontière franco-suisse, il a été fondé en 1954. Le laboratoire a été l'une des premières entreprises communes à l'échelle européenne et est devenu un exemple éclatant de collaboration internationale. Le CERN, dont la convention constitutive avait été signée à l'origine par 12 pays, compte aujourd'hui 20 États membres. On y étudie principalement ce qu'est la matière, ce dont elle est faite et les forces qui la maintiennent agglomérée, mais le site regroupe aussi de nombreuses expériences scientifiques de taille plus modeste, mais assurant au site une pluralité de disciplines unique.

Le Laboratoire met à la disposition des chercheurs et des ingénieurs des instruments scientifiques à la pointe de la technologie, développés, dans la plupart des cas, en collaboration avec le plus grandes entreprises mondiales.

Le LEP fut le premier grand accélérateur de particule du monde et a assuré la renommée du CERN auprès des plus grands laboratoires de recherche internationaux. Il a permis de confirmer (ou infirmer) de nombreuses théories concernant les élément constitutifs de la matière et a terminé ses activité en octobre 2000.

Les expériences menées au CERN n'ont pas d'équivalent dans l'histoire des sciences. Conçues et réalisées par des centaines de scientifiques, ces expériences sont souvent gigantesques, autant sur le plan des dimension, que sur les périodes durant lesquelles elles se font (au moins 10 ans pour le LHC). Ce contexte de travail unique a attiré depuis sa création des chercheurs des quatre coin du monde.

b. Une organisation tournée vers l'international

Le programme de recherche du CERN, basé sur son ensemble unique de grandes machines, attire des scientifiques des États Membres de l'union Européenne mais aussi d'un grand nombre d'autre pays parmi lesquels figurent les États Unis d'Amérique, le Japon, le Canada, la Fédération de Russie, la

République Populaire de Chine, Israël, l'Inde, et beaucoup d'autres pour des collaborations plus occasionnelles. (A l'occasion de leur travail au CERN, la plupart des scientifiques en visite sont payés par leur université ou leur institut de recherche). Environ 6500 scientifiques, soit la moitié des physiciens des particules dans le monde, utilisent les installations du CERN. Ils représentent 500 universités et plus de 80 nationalités. La fonction première du CERN est de fournir l'infrastructure de recherche et le support de base à cette vaste communauté d'utilisateurs.

La vie dans un tel creuset de nationalités, de cultures, de formations, d'habitudes de travail, est une expérience pittoresque et enrichissante.

Le CERN joue également un rôle important dans la formation technique de pointe. Une gamme complète de programmes de formation et de bourses attire au laboratoire de nombreux jeunes scientifiques de talent. La plupart font ensuite carrière dans l'industrie où leur expérience du travail dans un environnement multinational de haute technologie est fort appréciée.

c. Des expériences et des hommes

Pour concevoir et construire l'appareillage sophistiqué du CERN et pour assurer son bon fonctionnement, pour aider à préparer, à mettre en oeuvre les expériences scientifiques complexes, à analyser et interpréter leurs résultats et pour mener à bien la multitude de tâches nécessaires au succès d'une organisation aussi grande et spéciale, le CERN emploie un peu moins de 3000 personnes, couvrant un large éventail de compétences et de métiers - ingénieurs, techniciens, ouvriers qualifiés, administrateurs, secrétaires,...

Les activités se repartissent sur deux sites, celui de Meyrin et celui de Prevessin, tous deux situes à la frontière franco-suisse, le premier cote suisse, l'autre cote Français.

En plus des activités scientifiques, sont présents de nombreuses activités de services, telles que les restaurants, maisons de la presse, postes, banques, pompiers, etc... qui donnent au site un aspect de petite ville cosmopolite, comme un modèle réduit du village monde.

Afin d'assurer une gestion cohérente de cette force de travail, le CERN est fondé sur un modèle hiérarchique des compétences, direction, division, section, services. Je travaillais pour ma part dans la division EP

2. La division EP groupe ED section DTb

Le LHC sera le nouveau grand accélérateur du CERN. Il est destiné à faire entrer en collision des hadrons (contrairement au LEP qui accélérait des électrons). Une fois la construction achevée, les chercheurs mettront 4 grandes expériences en place. L'une d'elles est LHCb, destinée à étudier les particules issues de la désintégration du méson B. La division EP (Experimental Physics) est chargée de mettre au point les expériences qui prendront place sur LHC lorsqu'il sera achevé. Le groupe ED (Electronic Design), dans lequel je travaille, s'occupe de la réalisation des systèmes électroniques utilisées pour les expériences. La section DTb s'occupe de certains projets de l'expérience LHCbtels que le trigger 1.
C'est dans cette section que s'est déroulé mon stage, plus particulièrement sur la FLIC et le système TAGnet.

ANNEXE B
Code du module

```c
#ifndef __FLIC_MOD_H__
#define __FLIC_MOD_H__

#include <linux/pci.h>
#include <linux/module.h>
#include <linux/mman.h>
#include <errno.h>
#include <linux/types.h>

#ifndef MODULE
#include <pci/pci.h>
#endif


#define MAJOR_NR 126

struct flic_region_info
{

};

struct flic_data_to_transfert
{
  u32 *data;
  u32 pos;
  u32 nb_of_u32;
};

#define ORCA_VEND 0x11c1
#define ORCA_DEV 0x5401


/*definition of IOCTRL values */
#define FLIC_READ _IOWR(126, 0, unsigned long *)
#define FLIC_WRITE _IOWR(126, 1, unsigned long *)
#define FLIC_GET_MEM_SIZE _IOWR(126, 2, unsigned long *)
#define FLIC_WRITE_BLOCK _IOWR(126, 3, unsigned long *)
#define FLIC_READ_BLOCK _IOWR(126, 4, unsigned long *)
#define FLIC_REG_READ _IOWR(126, 5, unsigned long *)
#define FLIC_REG_WRITE _IOWR(126, 6, unsigned long *)


/* _IOR means that we're creating an ioctl command
 * number for passing information from a user process
 * to the kernel module.
 *
 * The first arguments, MAJOR_NUM, is the major device
 * number we're using.
 *
 * The second argument is the number of the command
 * (there could be several with different meanings).
 *
 * The third argument is the type we want to get from
 * the process to the kernel.
 */



#ifdef MODULE
#include <linux/blkdev.h>
#include <asm/io.h>  /* ioremap */
#include <linux/blk.h>
```



```c

#include <linux/fs.h>
#include <asm/uaccess.h> /* access_ok */
#include <asm/ioctl.h> /* _IO_* */

MODULE_AUTHOR("Sebastien GONZALVE");
MODULE_DESCRIPTION("Flic control module");
//MODULE_LICENSE("to kill whales");
extern int printk(const char* fmt, ...);
//static int Loaded;
static u32* ptr, *ptr_reg;
static struct pci_dev *flic_dev;



#else /* ifndef MODULE */
#include <fcntl.h>
#include <linux/fs.h>


/*****************/
/* read in RAM on flic. fd is the file descriptor of */
/* the registered device. pos is the position in RAM */
/* where to read. */

//int flic_read_dword(int fd, u32 pos);

inline u32 flic_read_u32(int fd, u32 pos)
{
  u32 ret_val;
  //data_to_trans_t *data_to_read;
struct flic_data_to_transfert *data_to_read;

  data_to_read = (struct flic_data_to_transfert *)malloc(sizeof(struct flic_data
_to_transfert));
  data_to_read->data = &ret_val;
  data_to_read->pos = pos;
  data_to_read->nb_of_u32 = 1;
  ioctl(fd, FLIC_READ, data_to_read);

  if(data_to_read->data == NULL)
    {
      errno=ENODATA;
      ret_val = 0xffffffff;
    }
  free(data_to_read);
  return ret_val;

}


/*****************************************************/
/* write in RAM on flic. fd is the file descriptor of */
/* the registered device. pos is the position in RAM  */
/* where to read.                                     */
inline void flic_write_u32(int fd, u32 value, u32 pos)
{

  //data_to_trans_t *data_to_read;
  struct flic_data_to_transfert *data_to_write;
  data_to_write = (struct flic_data_to_transfert *)malloc(sizeof(struct flic_dat
a_to_transfert));
```

```c
  data_to_write->data = &value;
  data_to_write->pos = pos;
  data_to_write->nb_of_u32 = 1;
  ioctl(fd, FLIC_WRITE, data_to_write);

  free(data_to_write);


}
/****************************************************/
/* read blocks of data into memory  on flic.        */
/* fd is the file descriptor of                      */
/* the registered device. pos is the position in RAM */
/* where to read.                                    */
void flic_read_u32block(int fd, u32 *block, u32 nb_of_u32, u32 pos)
{
  //data_to_trans_t *data_to_read;
  struct flic_data_to_transfert *data_to_write;

  data_to_write = (struct flic_data_to_transfert *)malloc(sizeof(struct flic_dat
a_to_transfert));
  data_to_write->data = block;
  data_to_write->pos = pos;
  data_to_write->nb_of_u32 = nb_of_u32;
  ioctl(fd, FLIC_READ_BLOCK, data_to_write);

  free(data_to_write);


}


/****************************************************/
/* write blocks of data into memory  on flic.        */
/* fd is the file descriptor of                      */
/* the registered device. pos is the position in RAM */
/* where to write.                                    */
void flic_write_u32block(int fd, u32 *block, u32 nb_of_u32, int pos)
{
  //data_to_trans_t *data_to_read;
  struct flic_data_to_transfert *data_to_write;

  data_to_write = (struct flic_data_to_transfert *)malloc(sizeof(struct flic_dat
a_to_transfert));
  data_to_write->data = block;
  data_to_write->pos = pos;
  data_to_write->nb_of_u32 = nb_of_u32;
  ioctl(fd, FLIC_WRITE_BLOCK, data_to_write);
  free(data_to_write);


}

u32 flic_mem_size(int fd)
{
  u32 val;
 // printf("pouet");
  ioctl(fd, FLIC_GET_MEM_SIZE, &val);
  return val;
}
```



```c
/* ________________________________ */
/*|                                |*/
/*|   Functions du registers        |*/
/*|________________________________|*/

/*****************/
/* read in Register on flic. fd is the file descriptor of */
/* the registered device. pos is the position in Register */
/* where to read. */


inline u32 flic_reg_read_u32(int fd, u32 pos)
{
  u32 ret_val;
  //data_to_trans_t *data_to_read;
  struct flic_data_to_transfert *data_to_read;

  data_to_read = (struct flic_data_to_transfert *)malloc(sizeof(struct flic_data
_to_transfert));
  data_to_read->data = &ret_val;
  data_to_read->pos = pos;
  data_to_read->nb_of_u32 = 1;
  ioctl(fd, FLIC_REG_READ, data_to_read);

  if(data_to_read->data == NULL)
    {
      errno=ENODATA;
      ret_val = 0xffffffff;
    }
  free(data_to_read);
  return ret_val;

}


/****************************************************/
/* write in RAM on flic. fd is the file descriptor of */
/* the registered device. pos is the position in RAM  */
/* where to read.                                      */
inline void flic_reg_write_u32(int fd, u32 value, u32 pos)
{

  //data_to_trans_t *data_to_read;
  struct flic_data_to_transfert *data_to_write;
  data_to_write = (struct flic_data_to_transfert *)malloc(sizeof(struct flic_dat
a_to_transfert));
  data_to_write->data = &value;
  data_to_write->pos = pos;
  data_to_write->nb_of_u32 = 1;
  ioctl(fd, FLIC_REG_WRITE, data_to_write);

  free(data_to_write);


}

/*****end reg fcts **********/

#endif /* !MODULE */


#endif /* __FLIC_MOD_H__ */
```

ANNEXE C
Code scheduler

```
/*
*  tablCharMod.c  04 September 2002
*  Skeleton of the program needed to transfert data from memoy to flic's ram
*  In transfert_data function, a SCI segment is created, that's where we read
*  cells where CPU are declared as free.
*  Transfert is done in one time, each transfert is 16 u32 words
*  Compiling with WHATTIME will display the duration of transfert.
*  Compiling with DEBUG_FLIC will display data transfered.
*
*/


/*includes*/

#include <stdlib.h>
#include <linux/types.h>
#include <stdio.h>
#include <asm/types.h>
#include <sys/mman.h>
#include <string.h>
#include <sys/io.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/byteorder/swab.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>
#include "flic_mod.h" /* headerfile of module */



/************/
/*   SCI    */


#include "sisci_error.h"
#include "sisci_api.h"
#include "sisci_demolib.h"
#include "testlib.h"

/***SCI*****/



/*****************/
/* globale SCI    */
#define NO_FLAGS          0
#define NO_CALLBACK       NULL
#define DMA_READY         7
#define MAX_SEGMENT_SIZE 16777216
#define INNER_LOOPS_DEFAULT     1000


/*   glob SCI     */
/*****************/
```



```
/******************/
/* globals algo    */

//#define DEBUG
#define NB_COL 16
#define NB_RAW 16
#define ORCA_VEND 0x11c1
#define ORCA_DEV  0x5401


#ifdef DEBUG
#define RAND_MAX 0xffffff
#endif

/* glob alg        */
/******************/





struct memory_use {
  u32 * ptr; /* pointer to mapped memory */
  u32 position; /* where to write next value */
};





u32 * get_data_to_transfert(u8 *tableau)//u8* tableau
{
  u32 i=0, j=0;
  u32 *data;
//unsigned int *logical_address_SCI;
  data = (u32 *)malloc(NB_RAW * sizeof(u32));
 //printf("get_data_to_transfert\n");
  for(i=0;i<NB_RAW; i++)
    {
      *(data + i)=i; /* put raw number in data  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXRAW
   */
      *(data +i) <<= 1;
      *(data +i) |=1;
      //printf("%x\n", *(data+i));
      for(j=0;j<NB_COL;j++)
        {

          *(data+i) <<=1; /*  shift every thing to the left */

          if (tableau[j + i*NB_COL]!=0)
            {

                *(data+i) |= 0x1;
            }
        }
      *(data +i)<<=24-NB_COL; /*  I shift of the number of missing CPU nb    */
      /* this has to be set as an agreement between hard & soft. Here I assume *
/
      /* that the cpu with the smallest addresse has the higher bit.           *
/


#ifdef DEBUG_FLIC
```

```c
      printf("%x\n", *(data +i));
#endif
    }

  return data;
}


/***************************************************************/
/* Reset used CPU from bit field.*/
/* We could consider that we could reset share memory when  */
/* we read values from share memory, but here we ca say that */
/* we can controle if transert was done or not. If there was */
/* a probleme, we can simply cancel the reset, and go on with */
/* the next iteration */
void reset_used_proc(u32 *sent_data, u8 * tableau)
{
  u32 test=1;
  int raw, i;

  for(raw=0;raw<NB_RAW;raw++)
    {
      test=0x800000; //1;
     // test<<= 23;

      /* reset valid bit only for addresses that were valid when I read */
      /* I use a shifting mask (test) to check which bits are set to 1 */

      for(i=0;i<NB_COL;i++)
        {

          if (( *(sent_data + raw) & test) != 0)
            {
              tableau[i +raw*NB_COL] = 0x0; /*reset to 0, but it could be just t
he highest bit */
            }
          test >>=1;

        }
    }
}



inline void  flic_send(int fd, struct memory_use * addr_ptr, u32 *data)
{

  flic_write_u32block(/*int filedesc*/ fd, data, NB_RAW, 0/*position*/);

}


/*
*  main loop of transfert. Perform a cyclic transfert of column to flic.
*
*/
void transfert_loop( int nbloop)
{
  int bou, fd;
  u8/*32*/ *SCI_ptr;

#ifdef WHATTIME
```



```c
  int start, finish;
  struct timeval *tv=(struct timeval *)malloc(sizeof(struct timeval));
  struct timezone *tz=(struct timezone *)malloc(sizeof(struct timezone)) ;
#endif


  u32 *data;


  /*************************************/
  /*                                   */
  /*      SCI Declarations             */
  /*                                   */
  /*************************************/



  unsigned int *logical_address_SCI;

  sci_error_t              error;
  sci_desc_t               sd;
  unsigned int             localAdapterNo   = 0;
  unsigned int             localNodeId      = 0;
  unsigned int             remoteNodeId     = 4;//0;
  unsigned int             localSegmentId   = 0;
  unsigned int             remoteSegmentId  = 0;
  unsigned int             localOffset      = 0;
  unsigned int             remoteOffset     = 0;
  unsigned int             segmentSize      = 1024;//65536;
  unsigned int             offset           = 0;
  unsigned int             loops            = 1;
  unsigned int               innerloops       = 1000;
  int                      counter;
  unsigned int             client           = 0;
  unsigned int             server           = 1;//0;
  unsigned int             MaxDmaSegments   = 1;
  unsigned int             keyOffset        = 0;
  unsigned int             printOut         = 1;
  unsigned int               direction        = 0;
  unsigned int             doAllInOneCall   = 1;

  /*      end of SCI declaration        */
  /*************************************/



  /*************************************/
  /*                                   */
  /*     SCI initialisation            */
  /*                                   */
  /*************************************/

  /* Initialize the SISCI library */
  SCIInitialize(NO_FLAGS, &error);
  if (error != SCI_ERR_OK)
    {
      fprintf(stderr,"SCIInitialize failed – Error code: 0x%x\n",error);
      //return(error);
    }
  /* Open a file descriptor */
  SCIOpen(&sd,NO_FLAGS,&error);

  if (error != SCI_ERR_OK)
    {
```

```
      fprintf(stderr,"SCIOpen failed - Error code 0x%x\n",error);
      //return(error);
    }


  /* Get local node-id */
  error = GetLocalNodeId(localAdapterNo, &localNodeId);
  if (error != SCI_ERR_OK)
    {
      fprintf(stderr,"Could not find the local adapter %d\n", localAdapterNo);
      SCIClose(sd,NO_FLAGS,&error);
      //return(-1);
    }
  /* Create a segmentId */
  localSegmentId  = (((localNodeId << 8) | remoteNodeId) << 8) | keyOffset;
  remoteSegmentId = (((remoteNodeId << 8) | localNodeId) << 8) | keyOffset;


  DmaServerNode(sd,
                localAdapterNo,
                localNodeId,
                localSegmentId,
                localOffset,
                segmentSize,
                keyOffset,
                remoteNodeId,
                remoteSegmentId,
                remoteOffset,
                MaxDmaSegments,
                direction,
                loops,
                innerloops,
                doAllInOneCall,
                printOut);


  if (error != SCI_ERR_OK)
    {
      fprintf(stderr,"SCIClose failed - Error code: 0x%x %s\n",error);
    }

  logical_address_SCI = get_logical_address();

  /*        EO Initialisations          */
  /*************************************/


  /* open special file*/
  fd = open("/dev/flic_node", O_RDWR);
  if(fd==-1) /* unable to open /dev/flic_node */
      {
        perror(" Unable to open driver flie");
        exit(-1);
      }
  printf("flic_node opened\n");

  printf("Physical address = %x\n", get_Physical_address());

#ifdef DEBUG
  printf("memory mapped\n");
#endif
```



```
#ifdef WHATTIME
  gettimeofday(tv,tz);
  start =tv->tv_usec;
#endif

  for(bou=0;bou<nbloop ;bou++)
/*main loop, "for()" should be replaced, when algo checked, by while(1)*/
    {
      data = get_data_to_transfert(logical_address_SCI); /* Get a structure of 1
6 u32 */
      flic_send(fd, 0/*addr_ptr*/, data); /* sent it to flic via module */
      reset_used_proc(data,logical_address_SCI); /* remove 1 in shared memory */
    }

#ifdef WHATTIME

  gettimeofday(tv,tz);
  finish = tv->tv_usec;
//  gettimeofday(tv,tz);
//  bou = tv->tv_usec;

//  printf("Na %x %x\n",tv->tv_sec, finish   );
  printf("duration for the writing of 16*%d 32bits WORDS : %d usec\n",nbloop, finish - start);
// printf("%d\n", bou-finish);
#endif

  /* unmap the local SCI segment and call SCI library (compiled from dma_bench.c
) */
  unmap_local_segment();

 /* Close the file descriptor */
  SCIClose(sd,NO_FLAGS,&error);


  if (error != SCI_ERR_OK)
    {
      fprintf(stderr,"SCIClose failed - Error code: 0x%x %s\n",error);
    }

  /* Free allocated resources */
  SCITerminate();

  close (fd);

}/* end transfert_loop */



int main(int argc, char** argv)
{
  transfert_loop( 1000);
  exit(1);
}


/* EOF */
```

ANNEXE D
MANUEL DU PROGRAMMEUR

1. Module

Cette partie n'a pas pour but d'expliquer en détail la manière de Créer un module noyau, ou un driver PCI. Il s'agit simplement d'expliquer les parties importantes du module créé. Pour plus d'informations sur le fonctionnement des drivers Linux, il convient de se reporter à l'ouvrage de Alessandro RUBINI & Jonathan CORBET: « LINUX device drivers »

Le module noyau a une structure classique pour un noyau de type 2.4
Il y a les deux fonctions indispensables d'initialisation et de fermeture :
`int init_module(void)` et
`void cleanup_module(void)`.
Ces fonctions (qui sont en fait des macros) s'apparentent fortement au constructeur et destructeur d'un objet en programmation orientée objet. En effet, *Init_module(-)* est appelée lors de l'installation du module, et *cleanup_module(-)* est appelée juste avant que le module ne soit supprimé de la mémoire. Ces fonctions permettent divers contrôles comme, par exemple la vérification de la configuration matérielle (on s'assure que le périphérique que l'on est chargé de gérer est bien présent).

L'appel à *register_blkdev(-)* sert à enregistrer le module comme un bloc driver de périphérique. Suite à la bonne exécution de cette fonction, le driver apparaît dans le fichier /proc/devices.

```
register_blkdev(126, "flic", &flic_bdops);
```

Ici « flic » est le nom qui apparaîtra dans le fichier devices et 126 et le major number. Le numéro majeur a été choisi dans une plage réservée aux cartes expérimentales, mais peut être changé. Cependant il faut choisir un numéro une bonne fois pour toute afin d'éviter d'avoir à charger le programme à chaque fois que l'on change de machine.

La stucture :

```
struct block_device_operations flic_bdops = {
  open : flic_open,
  release : flic_close,
  ioctl : flic_ioctl,
  NULL,
  NULL
}
```

définit les pointeurs de fonction pour l'ouverture et la fermeture du périphérique. Elles sont appelées à chaque fois que le fichier (i-node dans /dev/) est ouvert ou fermé. C'est dans cette structure qu'est définie le pointeur pour la fonction *ioctl(-)* qui a une si grande importance.

Pour la définition des numéros de la fonction *ioctl(-)* il est conseillé d'utiliser les macros :
_IOR() pour les ioctl de lecture
_IOW() pour les écritures
_IOWR() pour les lectures/écritures
fournies par Linux de façon à ne pas créer de conflits entre différents drivers. Pour la direction des transferts, on considère le point de vue du programme appelant, c'est à dire que si l'on veut transmettre une donnée au module, il s'agira d'une écriture.

2. QUELQUES NOTIONS PARTICULIÈREMENT INTÉRESSANTES

Les structures liées à la gestion du PCI sous Linux sont définies dans pci.h . Cependant, il faut prendre une double précaution : D'abord, il existe plusieurs pci.h, la plupart du temps liés entre eux par des *#include*; Ensuite, pci.h a de nombreuses directives de pré-compilation, c'est à dire que toutes les déclarations ne sont pas destinées à la même utilisation.
Bien qu'il soit alors un peu fastidieux de chercher à comprendre les structures PCI, cela se révèle fort utile pour une bonne maîtrise des outils.
En effet, les programmeurs des fichiers pci.h se sont fabriqués divers outils, comme l'auto détection des cartes PCI, le référencement des zones mémoire ainsi que leur taille. Cela dit, il y a un certain nombre d'étrangetés dans la définition des structures. Par exemple, la structure *pci_dev,* servant à référencer un périphérique PCI, n'est pas la même selon qu'on est dans l'espace utilisateur ou dans l'espace noyau. Cela est probablement du aux différences de besoin, et de structure, mais elles n'en demeurent pas moins fâcheuses.

Pour les utilisations futures du module gérant la FLIC, il sera nécessaire de pouvoir remonter les données de LOG dans l'espace utilisateur. Pour ce faire, il pourrait être intéressant d'utiliser les fonctions : create_proc_read_entry(). En effet, cette fonction crée un fichier accessible depuis l'espace utilisateur dans lequel le module peut écrire. On peut ainsi relire les statuts de LOG sans rien demander au module (comme un vrai fichier LOG).

3. Compilation

Les différents programmes nécessitent l'adjonction de directives de pré-compilation. Le module de la FLIC doit ainsi être compilé avec la ligne de commande suivante :

```
gcc -DMODULE -D__KERNEL__ -c -O2 -o flic_mod.o flic_mod.c
```

gcc sert (bien évidement) à invoquer le compilateur GNU.
-D est une option de gcc permettent de définir des variables du pré-compilateur. Ici, il convient de définir MODULE et __KERNEL__. MODULE sert à ce que les macros liées aux modules soient effectivement prisent en compte lors de la compilation. En effet, les appels entre le noyau et les modules (partie transparente pour le programmeur du module) suivent une architecture bien définie. Par ailleurs, __KERNEL__ sert à préciser que les fonctions et structures utilisées au sein du module sont celles du noyau et pas celles de l'espace utilisateur (par exemple pci_dev dont nous avons parlé auparavant).
-c sert à préciser qu'il ne faut pas éditer les liens. En effet, un module noyau n'est en fait qu'une bibliothèque de fonctions que l'on ajoute au noyau.
-O2 est placé pour que les routines d'optimisation du gcc soient utilisées. En effet, si cette option n'est pas activée, le compilateur de développe pas les fonctions déclarées *inline* dans le code. Or cela est préférable pour des raisons de performances.
-o <nom> sert comme toujours à donner un nom au fichier de sortie.

Il est à noter que selon les tests ou fonctions que l'on souhaite utiliser, il peut être nécessaire d'ajouter d'autres variables. Par exemple, de nombreux drivers ont des variables servant à préciser quelle version du noyau est présente sur la machine (cela est important car de nombreuses différences sont présentes entre le KERNEL 2.2XX et le 2.4XX). De plus, il peut y avoir des variantes du même programme avec plus ou moins de sorties affichées (pour le déboguage, par exemple).

4. LabVIEW

L'interfaçage avec un programme LabVIEW est relativement simple avec l'utilisation du module noyau. En effet, les accès à la carte sont entièrement pris en charge, reste à appeler une petite bibliothèque depuis labview. Pour le programme décrit dans le rapport, j'ai utilisé interfacelabview.c dans lequel on ne fait que mettre en forme les appels pour le module. Dans ce cas, il n'y a que deux fonctions (lecture d'un mot, écriture d'un mot), mais on peut imaginer bien d'autres fonctions de contrôle ou autre. En fait, toute fonction réalisée ou réalisable en C peut être ainsi appelée depuis LabVIEW, ce qui permet de réaliser

simplement des interfaces graphiques. Cela dit, il ne faut pas perdre de vue que LabVIEW n'est pas un programme dévolu à des tâches performantes en terme de vitesse, il faut donc se garder de vouloir utiliser LabVIEW pour réaliser des nombreuses tâches ou pour superviser des tâches critiques (hautes vitesses).
Pour ce qui est de la compilation, il convient d'utiliser des options particulières pour générer la bibliothèque appelée depuis LabVIEW :
`gcc -fPIC -shared -o <output name> <source file>` (cf : Using External Code in LabVIEW de National Instrument)

ANNEXE E
Code Testram.c

```c
 /******************************************************************\
|        testram2.c perform write across RAM and read back values     |
|       to compare with original and deternine dab sectors in read    |
|                                                                     |
| 06 June 2002                          Sebastien.gonzalve@cern.ch  |
 \******************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <pci/pci.h>
#include <linux/pci.h>
#include <sys/io.h>
#include <errno.h>
#include <asm/pci.h>
#include <fcntl.h>
#include <linux/config.h>
//#include <linux/mm.h>
//#include <asm-alpha/io.h>
//#include <sys/ioctl.h>
#include <sys/stat.h>
//#include <linux/ioport.h>
#define ORCA_VEND 0x11c1
#define ORCA_DEV  0x5401
//u32 begining, lenght;


struct parametre
{
  u32 begining;
  u32 lenght;
  word bus;
  byte disp;
  u32 testval;
  char display_all;
  enum {READ, WRITE, TEST} operation;
  char input_file[256];
};

/* Initilize param structure */
void init_param(struct parametre *param)
{
  /*init*/
  param->display_all = 0; /* default value is display nothing */
  param->bus =0xffff;     /* 0xffff to mean no bus is selected */
  param->disp = 0xff;     /* 0xff to mean no device number is given */
  param->begining = 0;    /* We start writing at 0 if nothing is said */
  param->lenght = 0x1000000; /* finish at the end of the RAM */
  strcpy(param->input_file, "");    /* no input file */
  param->testval = 0x12345678;
  /* end init */
}



/**********************************************************/
/* looking for Orca                                       */
/* we scan pci_dev chain and try to find the good vendo_id */
```



```c
struct pci_dev * find_orca(struct pci_access* pci_acc, struct parametre *param)
{
  struct pci_dev *device;
  struct pci_dev *orca_dev=NULL;



  if(param->bus != 0xffff && param->disp != 0xff)
    printf("Detecting Orca on %x:%x....", param->bus, param->disp);
  else  printf("You didn't specify where is the flic :\n scanning....");
  device = pci_acc->devices;
  while(device)
    {
      if (device->vendor_id == ORCA_VEND)
        {

          if( (device->bus==param->bus && device->dev==param->disp) ||
              (param->bus == 0xffff && param->disp == 0xff) )
            {
              orca_dev=device;
              break;
            }
        }
      device=device->next;
    }

  if(orca_dev == NULL)
    {
      printf("detection failled.\n Verify that Orca is present.\n");
      exit(-1);
    }
//  printf(".....Orca detected\n");
   if(param->bus == 0xffff && param->disp == 0xff)
     printf("Orca found on %x:%x", orca_dev->bus, orca_dev->dev);
   else printf("[ OK ]\n");
  return orca_dev;
}

/*******/
/* err_syntax()*/
/*  display an error message for syntax and exit pgm */
void err_syntax()
{
  errno=EINVAL;
  perror("Syntax error");
  printf("use --help for help\n");
  exit(-1);
}


/*******/
/* err_val(char*) */
/*  display an error message for syntax and exit pgm */
void err_val(char *val)
{

  printf("Bad value : %s\n", val);
  printf("use --help for help\n");
  exit(-1);
}
```

```c
/******/
/* display_help print help message and quit the program */
void display_help()
{
  printf("\n"
           " This program test the flic's RAM and registers.\n\n"
           " Syntaxe is \" testram [option1] ([param1.1] [param1.2])\n"
           "\t\t\t\t [option2] ([param2.1] [param 2.2]) ...\"\n"
           " Options are :\n"
           "\t- M bigining_addr nb_of_dword : specify where to write in the RAM \n"
           "\t\tand how many words should be manipulated\n"
           ""
           "\t- AM : specify that we work with all memory\n"
           "\t- S bus:device : to specify bus number and device number where to find \n"
           "\t\tthe flic. This parameter can be ommited, and then the program\n"
           "\t\twill use the first flic it find during bus scan. If there is \n"
           "\t\tonly one flic on the PC, this parameter is useless.\n"
           "\t- V value : set the value to be written in memory during test\n"
           "\t- OB : display only adresses & value where RW failed\n"
           "\t- R : Option to say that we want to read\n"
           "\t- W : Option to say that we want to write\n"
           "\t- T : Option to say that we want to test RAM. It means that values are\n"
           "\t\tgoing to be written in RAM, and then read back and compared to\n"
           "\t\toriginal tvalues. A status (OK or Failed) is displayed for\n"
           "\t\teach word. (it mean that every word will make an output line,\n"
           "\t\tand so it could be useful to combine this option with OB when\n"
           "\t\ttesting huge range of word)\n"
           "\t- F _file_name_: Set the file to use. If R option, it is an output file.\n"
           "\t\twith W it is an input file (Mean that data that are going to\n"
           "\t\tbe written in RAM are data stored in this file)\n"

           "\n"
           );
  exit(0);
}





/*****/
/* Check format of input line */
int check_format(int argc, char **argv, struct parametre *param)
{
  enum { W, R, T, AM, help, S, V, OB, F, M, bad} Com;
  int i=1;
  char *pos;

  /* init */
  param->bus = 0xffff;
  param->disp = 0xff;
  Com=bad; /* command is bad per default */


  if((argc)<2) /*no param*/
    {
      err_syntax();
    }

  while(i<=argc-1 || argc == 1)
    {
      /*printf("%d,",i);*/
```



```c
      if (!strcmp( argv[i], "S")) Com = S;
      if (!strcmp( argv[i], "R")) Com = R;
      if (!strcmp( argv[i], "M")) Com = M;
      if (!strcmp( argv[i], "W")) Com = W;
      if (!strcmp( argv[i], "T")) Com = T;
      if (!strcmp( argv[i], "--help")) Com = help;
      if (!strcmp( argv[i], "AM")) Com = AM;
      if (!strcmp( argv[i], "V")) Com = V;
      if (!strcmp( argv[i], "OB")) Com = OB;
      if (!strcmp( argv[i], "F")) Com = F;


      switch(Com)
        {
        case S :
          if(argc<i+2)
            {
              printf("S not enought parameters\n");
              err_syntax();
            }
          pos = strchr(argv[i+1], ':');
          if(!pos)
            {
              printf(" S wrong parameters\n");
              err_val(argv[i+1]);
            }
          *pos='\0';
          param->bus =(word)strtoul(pos-1, (char**) NULL, 16); /* char before :
*/
          param->disp = param->disp & strtoul(pos+1, (char**)NULL, 16); /*char a
fter':'*/
//        printf(" \n\n\n\n\n%d\n",param->disp );
          i+=2; /* shift 3 param further */
          break;

        case V :
          if(argc<i+2)
            {
              printf("V not enought parameters\n");
              err_syntax();
            }

          param->testval = (u32)strtoul(argv[i+1], (char**) NULL, 16);
          i+=2; /* shift 3 param further */
          break;


        case M :
          if(argc-1<i+2) /* if there are less than 2 args  */
            {
              printf(" M wrong parameters\n");
              err_syntax();
            }
          printf(" M function recognised\n");
          if((strtoul(argv[i+1], (char**)NULL, 16) > 0x1000000) || /* begining f
urther than memory size*/
             (strtoul(argv[i+1], (char**)NULL, 16) < 0)) /* negative value for b
egining */
            {
              err_val(argv[i+1]);
            }
```

```
          // printf("  %x\n",(strtol(argv[2], (char**)NULL, 16) +strtol(argv[3],
 (char**)NULL, 16)));
          if((strtoul(argv[i+2], (char**)NULL, 16)<0) || /* negative size */
             ( (strtoul(argv[i+1], (char**)NULL, 16) +
                strtoul(argv[i+2], (char**)NULL, 16)  ) > 0x1000000) || /* end a
dresse outside of RAM */
             (errno == ERANGE)) /* value out of range of convert function */
            {
              err_val(argv[i+2]);
            }

          /* when M params are OK */
          else
            {
              param->display_all = 1;
              param->begining = strtoul(argv[i+1], (char**)NULL, 16);
              param->lenght = strtoul(argv[i+2], (char**)NULL, 16);
            }
          i+=3;
          break;

        case AM :
//        param->display_all = 0;
          param->begining =0;
          param->lenght=0x1000000;
          i++;
          break;

        case OB :
          param->display_all = 0;
          i++;
          break;

        case T :
          param->operation = TEST;
          i++;
          break;

        case R :
          param->operation = READ;
          i++;
          break;

        case W :
          param->operation = WRITE;
          i++;
          break;

        case F :
          if (argv[i+1]!=NULL)
            {
              strcpy(param->input_file, argv[i+1]);
            }
          else
            {
              printf("F File name is needed\n");
              exit(-1);
            }
          i+=2;
          break;
```



```
        case help :
          display_help();
          break;

        case bad :
          err_syntax();
          break;

        default :
          printf("Warning, this message signal that there is an internal programme error\n"
                 "please report\n");
          exit(-1);
          break;

        }
    }
  return 0;
}




/*********************************/
/* execute command read write or test */
void Exec_com(u32* ptr, struct parametre *param)
{
  u32 *end, *i, offset, Val, pos, dummy;
  int cmpt;
  char progress[4]= { '-','\\','|','/'};
  FILE *fp;

          /************************************************************/
    /*--------------           DRAM TEST                        --------------- */



  end = ptr + param->lenght + param->begining;
  offset = param->begining;
  fp = fopen(param->input_file, "rw"); /*try to open input file */

  switch(param->operation)
    {

    case READ:
      fp = fopen(param->input_file, "w"); /*try to open input file */
      if (fp != NULL) /* if file given exists */
        {
          for(i=ptr+offset;i<end;i++) /* loop till end of region*/
            {
              fprintf(fp,
                      "%010x\t%010x\n",
                      (u32)(i-ptr),
                      (u32)(*i));
            }
          fclose(fp);
        }

      else /* if file doesn't exist */
        {
          printf(" No file fond for output, display everything on screen\n");

          printf("   addr   [ read-back  ]\n");
```

```
          for(i=ptr+offset;i<=end-1;i++)
            {
              printf("%0#10x  [ %0#10x ]\n",
                     (u32)(i-ptr),
                     (u32)(*i));
            }
        }
      break;

    case WRITE:
      printf("\n Resetting \n");
      for(i=ptr+offset;i<end;i++) /* loop to write data in dram*/
        {
          *i = 0x0; /* set value to 0 */

          for(cmpt=0;cmpt<500;cmpt++); /*delay*/

          if(((u32)i/4)%0x6000==0) printf("erasing @ %0#x\r",(u32)(i-ptr));
          fflush(stdout);
        }
      printf("reset finished     \n");
      /* end reset */

      fp = fopen(param->input_file, "r"); /*try to open input file */
      if (fp != NULL) /* if file given exists */
        {
          for(i=ptr+offset;i<end;i++) /* loop till end of region*/
            {
              if(fscanf(fp, "%x\n", &Val)==-1)
                {
                  rewind(fp);
                  fscanf(fp, "%x\n", &Val);
                }
              if(((u32)i/4)%0x8==0) for(cmpt=0;cmpt<1500;cmpt++); /*delay every
8 32bit-word*/
              *i=Val;
            }
          fclose(fp);
        }

      else /* if file doesn't exist */
        {
          printf(" No file fond in input, Using Default value %x for writing\n", param->testval);
          for(i=ptr+offset;i<=end-1;i++)
            {
              *i=param->testval;
            }
        }
      break;

    case TEST:
      /* Reset the memory content */
      printf("\n Resetting \n");
      for(i=ptr+offset;i<end;i++) /* loop to write data in dram*/
        {
        *i = 0x0; /* set value to 0 */

        for(cmpt=0;cmpt<500;cmpt++); /*delay*/

        if(((u32)i/4)%0x6000==0) printf("erasing @ %0#x\r",(u32)(i-ptr));
        fflush(stdout);
        }
```



```
      printf("reset finished     \n");
      /* end reset */


      /* test begining */
      printf("\ntest begining @ : %x\n\n", (u32)(offset));
      for(i=ptr+offset;i<=end-1;i++) /* loop to write data in dram*/
        {

          //    *i = (u32)i; /* write address at the very address*/

          *i= param->testval;
                if(((u32)i/4)%0x8==0) for(cmpt=0;cmpt<1500;cmpt++); /*delay*/

          if(((u32)i/4)%0x1000==0) printf("writing @ %0#x\r",(u32)i);
          fflush(stdout);
        }
      putc('\n',stdout);

      sleep(1);
      /************       read back from dram to check values   ******/

      if(param->display_all) /* output for each addr */
        {
          printf("  addr  [  written ] [ read-back ]  RESULT\n");
          for(i=ptr+offset;i<=end-1;i++)
            {
              printf("%0#10x  [ %0#10x ] [ %0#10x ] [%s]\n",
                     (u32)(i-ptr),
                     (u32)param->testval ,
                     (u32)(*i),
                     ((u32)param->testval == (u32)(*i) ? " OK ": "FAILED"));
              dummy = *ptr; /* read at 0 position to fflush bridge */
            }
        }
      else /* display only addr with bad values */
        {
          cmpt=0;
          for(i=ptr+offset;i<=end-1;i++)
            {
              if((u32)i%2000==0)
                {
                  printf("reading.... %c\r", progress[(cmpt++%4)]);
                  fflush(stdout);
                }
              if(param->testval != (u32)(*i))
                {
                  printf("%0#10x  [ %0#10x ] [ %0#10x ] [%s]\n",
                         (u32)(i-ptr),
                         (u32)param->testval ,
                         (u32)(*i),
                         "FAILED");
                }
              dummy = *ptr; /* read at 0 position to fflush bridge */
            }
        }
      break; /*end test */

    }/* end switch*/

#ifdef OULALA
  /* Reset the memory content */
```

```
  printf("\n Resetting \n");
  for(i=ptr+offset;i<end;i++) /* loop to write data in dram*/
      {
        *i = 0x0; /* set value to 0 */

        for(cmpt=0;cmpt<500;cmpt++); /*delay*/

        if(((u32)i/4)%0x6000==0) printf("erasing @ %0#x\r",(u32)(i-ptr));
                fflush(stdout);
      }
    printf("reset finished    \n");
    /* end reset */


    /* test begining */
    printf("\nbegining @ : %x\n\n", (u32)(offset));

/*********if fopen ***********/

    if (fp != NULL) /* if file given exists */
      {
        pos=0;
        for(i=ptr+offset;i<=end-1;i++)
          {
            if(fscanf(fp, "%x\n", &Val)==-1)
              {
                rewind(fp);
                fscanf(fp, "%x\n", &Val);
              }
            *i=Val;
          }
        fclose(fp); /*close file*/

      }/*end fopen*/


/***********else*********/
    else /* case param  When no input file  , we use value given by V option */
      {
        for(i=ptr+offset;i<=end-1;i++) /* loop to write data in dram*/
          {

            *i= param->testval;
            if(((u32)i/4)%0x8==0) for(cmpt=0;cmpt<1500;cmpt++); /*delay every 8
32bit-word*/

            if(((u32)i/4)%0x1000==0) printf("writing @ %0#x\r",(u32)i);
            fflush(stdout);
          }
        putc('\n',stdout);

        sleep(2);
        /************       read back from dram to check values  ******/

        if(param->display_all) /* output for each addr */
          {
            printf("  addr  [  written ] [ read-back ]  RESULT\n");
            for(i=ptr+offset;i<=end-1;i++)
              {
                printf("%0#10x  [ %0#10x ] [ %0#10x ] [%s]\n",
                       (u32)(i-ptr),
                       (u32)param->testval ,
```



```
                       (u32)(*i),
                       ((u32)param->testval == (u32)(*i) ? " OK ": "FAILED"));
              }
          }
        else /* display only addr with bad values */
          {
            cmpt=0;
            for(i=ptr+offset;i<=end-1;i++)
              {
                if((u32)i%2000==0)
                  {
                    printf("reading.... %c\r", progress[(cmpt++%4)]);
                    fflush(stdout);
                  }
                if(param->testval != (u32)(*i))
                  {
                    printf("%0#10x  [ %0#10x ] [ %0#10x ] [%s]\n",
                           (u32)(i-ptr),
                           (u32)param->testval ,
                           (u32)(*i),
                           "FAILED");
                  }
              }
          }
      }/*end if input ==NULL*/

#endif
          printf("\nTermiated  @ %x \n", (u32)(i-ptr-1));
}





#ifndef NOMAIN


int main(int argc, char **argv)
{
  u32 id, *ptr, *ptr_reg;
  struct pci_access *pci_acc;
  u32 BAR0, BAR5;
  int fd;
  u32 ADDR_USR;
  struct pci_dev *orca_config_space;
//  u32 *end, *i, offset;
//  u32 cmpt, ok;
 // char progress[4]= { '-','\\','|','/'};
  struct parametre param;


  init_param(&param);

  check_format(argc, argv, &param); // &begining, &lenght, &bus, &disp);
  printf("\n");
  /* Scan the PCI bus, and store the config space in pci_acc */

  pci_acc = pci_alloc();
  pci_init(pci_acc);
  pci_scan_bus(pci_acc);
  printf("using %s in order to identify card\n", pci_acc->id_file_name);
```

```
  orca_config_space = find_orca(pci_acc, &param ); //param->bus, param->disp);



/************************/
 /* under construction */



//pci_setup_cache(orca_config_space,  orca_config_space->cache, 10000);



/**************************/

#ifdef DEBUG
/* lines under are not really useful... */
  id = pci_read_long(orca_config_space,0);
  printf(" les donnees d'identite sont : %x \n", (u32)id);
  pci_write_byte(orca_config_space, 0x0c, 0xf); /* cache line size */
  id = pci_read_long(orca_config_space,0x4);
  printf(" les donnees des registres de statut sont : %x \n", (u32)id);
/*-----------------------------------------*/
#endif

//  printf("\v\v\t Memory test....\n\n");

  BAR0= orca_config_space->base_addr[0] &  PCI_BASE_ADDRESS_MEM_MASK;
  BAR5= orca_config_space->base_addr[5] &  PCI_BASE_ADDRESS_MEM_MASK;


  /*-------------Initialize memory---------------------------------*/
  /* mapping memory in order to use pointers to acces directly the */
  /* memory space                                                  */


  fd = open("/dev/mem", O_RDWR);/*Prefetchable memory*/
  if(fd==-1) /* unable to open /dev/mem */
      {
        perror(" Unable to map memory");
        exit(-1);
      }

  ptr = (u32 *) mmap(
                     NULL,
                     orca_config_space->size[0] * 4, /* !! size of memory in byt
e  !! */
                     PROT_READ | PROT_WRITE,
                     MAP_SHARED,
                     fd,
                     BAR0);
  if(ptr==NULL)
    {
      perror("echec mmap");
      exit(-1);
    }

  ptr_reg = (u32 *) mmap(
                         NULL,
                         orca_config_space->size[5] *4,/* !! size of memory in b
yte  !! */
```



```
                         PROT_READ | PROT_WRITE,
                         MAP_SHARED,
                         fd,
                         BAR5);

    /*********************************************************************
**/

    ADDR_USR = 0x10; /* setting where we want to write in register*/


#ifdef DEBUG
    printf("ptr_reg : %x\n ptr_reg + %#x : %x\n\n", (u32)ptr_reg, (u32)ADDR_USR, (u32)(ptr_
reg+ADDR_USR));
    *(ptr_reg + ADDR_USR) = (u32)ADDR_USR;

       printf(" on y lit : %#x\n", (u32)*(ptr_reg+ ADDR_USR));
    printf(" ptr : %#x\n", (u32)ptr);

#endif

    /* test Ram*/

    Exec_com(ptr, &param);

    /* give back memory space */
    munmap(ptr, orca_config_space->size[0] * 4);/* !! size of memory in byte !!*
/
    munmap(ptr_reg, orca_config_space->size[5] * 4);


    close(fd); /* close memory */

    return 0;



}

#endif

/* EOF */
```

ANNEXE F
Contenu des transferts vers la FLIC

```
Local segment (id=0x80400, size=1024) is created.
Local segment (id=0x80400, size=1024) is created.
Local segment (id=0x80400) is mapped to user space.
The physical address for local segment is :2f4000
Local segment (id=0x80400) is available for remote connections.
Waiting for the DMA transfer to be ready ....
Node 8 received interrupt (0x0)

DMA transfer done!

Client data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1024 1024 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
The local segment is unmapped
The local segment is removed
Detecting Orca on 2:f....[  OK  ]
orca found
Physical address = 2f4000
memory mapped
 1e45700
 3cdfb00
 57c6200
 770be00
 953f400
 bc8ae00
 d5b5100
 f9e5f00
 11214500
 130f7300
 1550b900
 17f93600
 19267400
 1bc89800
 1d57f900
 1f65fb00
 1000000
 3100000
 5000000
 7000000
 9000200
 b000000
 d000000
 f000000
 11000000
 13008000
 15080000
 17040000
 19000000
 1b000000
 1d008000
 1f000000
 1000000
 3000000
 5000000
 7208400
 9000400
 b000000
 d000000
 f004000
 11000000
 13000000
 15000000
 17000800
```



```
 19000000
 1b000000
 1d800000
 1f000000
 1000000
 3000000
 5000000
 7080000
 9000200
 b000100
 d000000
 f010000
 11000000
 13000000
 15000000
 17000000
 19001000
 1b000000
 1d000000
 1f002000
 1040000
 3000000
 5401000
 7000000
 9000000
 b000000
 d000800
 f000000
 11020000
 13000000
 15002800
 17000000
 19000000
 1b000000
 1d000000
 1f000000
 1000000
 3002000
 5000000
 7000000
 9000000
 b040000
 d000000
 f900000
 11000000
 13000000
 15200000
 17040200
 19000000
 1b000000
 1d000000
 1f000000
 1080000
 3000000
 5000000
 7021000
 9000000
 b000800
 d000000
 f000000
 11000000
 13080000
```

```
15000100
17100000
19000000
1b100000
1d200000
1f800000
1000000
3000000
5000000
7000000
9000100
b000000
d000000
f000000
11000000
13000200
15000000
17000000
19000000
1b000000
1d080000
1f000400
1000000
3040000
5000000
7200000
9004200
b000000
d000000
f000000
11200000
13000000
15004000
17000000
19000000
1b080000
1d000000
1f000000
1008000
3000000
5004000
7000000
9000000
b080000
d000000
f800000
11002000
13002000
15000000
17200000
19000000
1b000000
1d000000
1f000000
1002000
3000000
5000000
7000000
9008000
b000000
d000000
f000000
```



```
11000000
13000000
15008000
17000800
19000000
1b040000
1d000000
1f000000
1000200
3008000
5000000
7000000
9080000
b001000
d000000
f000000
11000200
13000000
15000800
17000000
19000000
1b010000
1d000000
1f100000
1004000
3000000
5000400
7000000
9200800
b000000
d000000
f000000
11000000
13001000
15000000
17000000
19080000
1b000000
1d040000
1f000000
1020000
3000000
5000000
7100000
9000000
b000000
d000000
f000100
11080000
13000000
15100100
17000000
19000400
1b080000
1d000000
1f000000
1000000
3000000
5000400
7200000
9010000
b000000
```

```
d000000
f020000
11000000
13000000
15000400
17000000
19080000
1b000200
1d000800
1f000000
1000000
3000800
5900000
7800000
9002000
b000000
d000000
f000000
11040000
13000000
15000000
17000000
19000000
1b000000
1d000000
1f021000
1090000
3000000
5000000
7000000
9000000
b000000
d000800
f000000
11000000
13000000
15000000
17204000
19000000
1b000000
1d000800
1f080000
1001000
3000000
5800000
7000000
9000000
b000000
d000000
f000000
11020000
13000000
15000000
17002000
19100000
1b000000
1d000000
1f400000
1040000
3000000
5000000
7100200
```



```
9000000
b800000
d010000
f000000
11000000
13004000
15100000
17010000
19000000
1b104000
1d100800
1f000000
1000000
3080000
5000000
7080000
9000000
b000200
d000000
f000000
11000000
13000000
15800000
17000000
19000000
1b000000
1d000000
1f210000
```