# Étude et réalisation d'un driver CAMAC pour Amiga.

Milán Zofka

16 juin 1992

*A MM. les Professeurs*
*Harms, Extermann et Bourquin*[a],
*ainsi qu'à Jean-Yves Muespach*[b].
*Tous m'ont permis de venir à bout*
*de ce travail.*

[a] *Université de Genève*
[b] *Support Commodore Suisse*

# Table des matières


**1 Introduction** 1

1.1 Les besoins et l'existant . . . 2

1.2 Les ressources . . . 3

1.3 Pré-développement . . . 4

**2 Etude** 6

2.1 L'électronique . . . 7

2.2 Communication avec le CAMAC . . . 8

2.3 Multi-tâches et temps réel . . . 9

2.4 Messagerie . . . 10

2.5 Driver . . . 11

**3 Driver** 13

3.1 Autoconfiguration . . . 14

3.1.1 Autres méthodes . . . 15

3.1.2 Illustration . . . 15

3.2 Lien dynamique et librairie résidente . . . 17

3.2.1 Illustration . . . 18

3.3 Synchrone et asynchrone . . . 19

3.3.1 Synchronicité du CAMAC . . . 20

3.3.2 Les requêtes d'entrées-sorties . . . . . 21
3.3.3 Illustration . . . . . 21
3.4 Communication . . . . . 22
3.5 Les types d'actions . . . . . 24
3.5.1 L'obligatoire . . . . . 24
3.5.2 Le nécessaire . . . . . 25

**4 Outils de développement** 27
4.1 Librairie . . . . . 28
4.2 Les interruptions . . . . . 29
4.2.1 Pourquoi? . . . . . 29
4.2.2 Comment? . . . . . 30
4.2.3 Illustration . . . . . 32
4.3 Contenu de la librairie . . . . . 33

**5 Prise de données** 35
5.1 Qui fait quoi et quand? . . . . . 35
5.2 Combien de temps cela dure-t-il? . . . . . 39
5.3 Exemple d'utilisation . . . . . 40

**6 Conclusions** 43
6.1 Utilisation future . . . . . 43
6.2 Compléments . . . . . 44
6.3 Epilogue . . . . . 45

**Bibliographie** 46

# Préface

*Il y a dans la communauté scientifique une catégorie bien particulière de chercheurs, les physiciens. Toujours en quête de l'inconnu, le physicien explore les lois naturelles qui l'entourent et cherche à comprendre les mécanismes qui les régissent.*

*Mais la physique est un vaste domaine, et de l'infiniment petit il y a un fossé avec l'infiniment grand. Au fait, ce fossé est-il si grand que ça? Les spécialistes de l'infiniment petit tendent à démontrer que notre infiniment grand est né de notre infiniment petit. Et, pour le prouver, ils construisent dans d'immenses proportions des machines qui domestiquent de minuscules petites choses.*

*Partant de ce petit rien, ils ont œuvré des années durant et ont finalement construit la plus grande machine jamais créée, une gigantesque machine de 27 kilomètres de long représentant l'équivalent de milliers d'années de travail dans des domaines des plus variés.*

*Baptisée le LEP*[1]*, les physiciens se sont partagé leur nouvelle création en quatre parties. Le LEP est un accélérateur de particules de forme pratiquement circulaire qui fait tourner des positrons ($e^+$) dans un sens et des électrons ($e^-$) dans l'autre. A quatre endroits différents, les particules sont dirigées les unes contre les autres pour former une collision de type $e^+e^-$; c'est à ces endroits précis que les physiciens se battent pour obtenir des résultats intéressants.*

*A l'un de ces croisements, se situe l'expérience L3 où près de 600 chercheurs tentent de suivre à la trace*[2] *le cheminement de ces particules après collision. Pour pouvoir mener à bien leur recherche, les physiciens ont fait appel à des mécaniciens, des électroniciens, des informaticiens, etc... Bref, à toute la communauté scientifique susceptible d'apporter sa contribution à la recherche fondamentale dans la physique des particules.*

*Le résultat: 34 universités et laboratoires du monde entier ont en 8 ans, soit 1100 hommes/année de travail, mis au point un détecteur de 16 mètres de diamètre pesant 8000 tonnes.*

*L'informatique est la servante de ces chercheurs, elle est fidèle, docile et prête à rendre tous les services qui lui sont demandés, pour autant qu'ils lui soient clairement présentés... En communiquant efficacement avec l'électronique, les ordinateurs deviennent des interprètes pour les physiciens. Ce rôle d'interprète, les ordinateurs ne l'ont pas acquis tout seuls, les*

[1] Large Electron Positron Collider

[2] Au sens propre.

*informaticiens y sont tout de même pour quelque chose.*

*Si on observe un physicien de près, on se rend compte qu'il est capable de domestiquer toute une panoplie de disciplines. En plus de ses qualités de théoricien, on peut lui mettre un tournevis, un fer à souder ou un clavier entre les mains, il s'en sortira toujours. L'informaticien scientifique ayant aussi cette aptitude, qu'est-ce qui les différencie? Tout simplement une connaissance plus approfondie du domaine dans lequel il est roi. L'informaticien est bon en informatique et le physicien est bon en physique.*

*Une vérité pas si évidente si l'on sait qu'il y a encore 15 ans, le physicien devait tout programmer, tout mettre en œuvre et tout comprendre. L'informaticien était alors un physicien spécialiste en informatique ou un mathématicien appliqué, mais le titre proprement dit ne signifiait rien.*

*Petit à petit, le cercle infiniment petit des informaticiens est venu gonfler le cercle infiniment grand des physiciens et les détails de la programmation sont devenus des mystères réservés aux experts en ce domaine.*

*Si le LEP a été conçu par des physiciens, la réalisation des détails est revenue aux spécialistes. Bien que 'nul' en physique des particules, un informaticien peut très bien comprendre le phénomène qu'il observe et tenter de l'analyser avec soin. En fait, ne cherchant pas à trouver un résultat qui convienne aux hypothèses des physiciens, l'informaticien cherche une solution élégante et efficace au problème qui lui est posé, de façon à ce que le physicien ait tout loisir d'utiliser les programmes sans avoir à se poser de questions sur la réalisation.*

*Ce présent ouvrage est la description d'un maillon infime de l'une des plus grandes expériences qui ait été réalisée en physique des particules. Ce maillon infime est nécessaire à l'étude du phénomène se produisant dans le détecteur L3 et a donné entière satisfaction puisqu'il est utilisé à chaque fois que la machine LEP se met en route.*

*Les physiciens auraient très bien pu aboutir aux mêmes résultats sans l'aide des informaticiens, il faut le reconnaître. Toutefois en demandant à un informaticien de bien vouloir traiter le problème, ils ont non seulement permis à un étudiant de rédiger un travail de diplôme, mais ils ont surtout réalisé qu'ils avaient bien assez de problèmes avec leurs particules et que tout compte fait, l'informatique est une spécialisation qu'ils ne sont plus obligés de maîtriser.*

# Chapitre 1

# Introduction

Au Laboratoire Européen pour la Physique des Particules (CERN), il y a un accélerateur de particules nommé le LEP [1], qui accélère, dans des directions opposées, des électrons ($e^-$) et des positrons ($e^+$). Cet accélerateur est un tunnel souterrain de forme circulaire, et les particules qui y circulent sont dirigées les unes contre les autres en quatre points différents.

A ces quatre intersections se trouvent des expériences qui consistent à suivre le déroulement des désintégrations $e^+e^-$ et à en saisir le plus d'informations possible. Une de ces expériences porte le nom de L3.

L'expérience L3 est constituée de dizaines de parties différentes formant un détecteur de particules et une chaîne d'acquisition de données. Le détecteur a pour but de fournir des signaux électriques reflètant l'évènement physique et la chaîne d'acquisition sert à véhiculer les informations du détecteur à l'unité de stockage, qui est généralement un disque ou une bande magnétique.

La chaîne d'acquisition de données est elle-même divisée en plusieurs parties, selon la teneur et le volume d'informations que chacune d'elles est capable de véhiculer. Pour mener à bien une des parties de la chaîne d'acquisition, un micro-ordinateur a été utilisé. Alors qu'aucun périphérique et qu'aucun logiciel n'était à disposition sur ce micro-système pour mener à bien cette tâche, tout a dû être développé et mis au point.

Ce présent ouvrage traite de la façon dont un micro-ordinateur a été utilisé pour devenir un système d'acquisition de données interactif. Mais le domaine est vaste et l'accent sera mis sur la manière dont a été développé le *driver* CAMAC [2], ce *driver* permettant de gérer au mieux le trafic électronique↔mémoire-centrale, compte tenu des caractéristiques du système d'exploitation de la machine et des spécifications techniques du CAMAC.

## 1.1 Les besoins et l'existant

Situons maintenant le problème dans son environnement. Sans entrer dans les détails, voyons comment le détecteur L3 fonctionne. Le détecteur a été construit autour du point précis où ont lieu les collisions $e^+e^-$. Afin de suivre la trajectoire des particules résultant de cette désintégration, toute une série de sous-détecteurs *enrobent* le point d'impact. Le premier, le plus au centre, est une chambre à fils basée sur le principe d'expansion temporelle et connue sous le nom de TEC[1].

Autour de la TEC, se trouve le calorimètre électromagnétique au BGO[2], puis le calorimètre hadronique. Viennent enfin les chambres à muons. Les mystères de ces sous-détecteurs sont longuement décrits dans toute une série de publications [3] disponibles sur simple demande auprès du service de publications du CERN.

En ce qui concerne ce travail, seul le calorimètre électromagnétique au BGO nous intéresse. Le BGO est un cristal[3] qui a la particularité d'émettre des photons lors du passage d'une particule dans son volume, c'est donc un scintillateur. Afin de quantifier la qualité de ce scintillateur, on doit le calibrer. Calibrer le scintillateur signifie que l'on doit mesurer la quantité de photons émis lorsqu'une particule ayant une énergie connue le traverse. Une fois le *rendement* connu, il est alors possible de déterminer l'énergie d'une particule traversant le cristal en comparant le nombre de photons émis avec ceux émis lors de la calibration.

Le critère qualité-quantité est donc dépendant de cette calibration et c'est justement ce critère qui a déterminé le choix d'un tel sous-détecteur à cet endroit.

Les détails de la calibration sont également longuement traités dans une série de publications [4], mais il y a différents points qu'il faut retenir:

- La calibration a lieu lorqu'il n'y a pas de faisceau de particules provenant du LEP.
- Le temps nécessaire à la calibration n'est pas un facteur important, tant qu'il n'est pas d'une durée jugée inadmissible.
- Bien que le résultat soit d'une importance capitale pour la validité des valeurs détectées lors d'une prise de données, la technologie utilisée ne doit pas forcément être exemplaire et, par extension du point précédent, ne doit pas forcément être de très grande vitesse.

Compte tenu de ces différents points, le choix du matériel électronique pour effectuer une partie de cette calibration a été la norme CAMAC [2]. Le CAMAC est une norme divisée en quatre parties:

[1] Time Expansion Chamber
[2] Bismuth Germanium Oxide
[3] Le calorimètre est constitué de 12000 cristaux au total

1. [EUR 4100] Spécification de la mécanique d'une baie[4], de son bus, de ses alimentations et de ses signaux électriques, ainsi que la manière dont les modules électroniques doivent communiquer dans la baie.

2. [EUR 4600] Spécification d'un bus parallèle, nommé *branche*, qui permet de connecter jusqu'à sept baies à une unité de contrôle, généralement un ordinateur.

3. [EUR 6100] Spécification d'un bus sériel qui permet de connecter jusqu'à huit baies à une unité de contrôle, puis à un réseau (très rarement utilisé).

4. [EUR 6500] Extension de la norme EUR 4100, en ce qui concerne l'arbitration du bus de la baie, dans le cas où plusieurs modules désireraient en être le maître (rarement utilisé).

Bien que pas très performante (10 MBits/sec. dans le cas EUR 4600) cette norme permet d'insérer 25 modules électroniques par baie et 7 baies par *branche*, soit 175 modules qui peuvent être pilotés par un ordinateur équipé d'un seul contrôleur. De plus, toute une panoplie de modules CAMAC capables de générer ou de lire des signaux électriques de toutes catégories ont été construits au cours des dernières années et ne nécessitent plus aucun développement.

## 1.2 Les ressources

Dès qu'une nouvelle expérience est mise à l'étude, on détermine quelle technologie sera utilisée. Lorsqu'une précédente expérience est démontée, on peut récupérer une partie de l'électronique et éventuellement la réutiliser dans une nouvelle expérience, pour autant que la technologie soit encore à même de remplir les tâches qui lui sont assignées. Dans le cas de L3, les prises de données *lentes* ont été laissées au CAMAC. Mais voilà, le CAMAC était utilisé auparavant, soit:

1. par le biais d'ordinateurs de type VAX 11 ou Nord 100; dans ce cas il fallait un contrôleur de *branche* se connectant dans une baie à part, nommée *"system-crate"*, et, du *system-crate* à l'ordinateur, il fallait un câble spécial qui venait se brancher directement sur le bus de la machine;

2. par un micro-ordinateur de type PC; dans ce cas les contraintes devenaient importantes, étant donné qu'il n'existait pas de contrôleurs de branche répondant à la norme EUR 4600, limitant ainsi le contrôle à une seule baie, et à condition que cette dernière soit équipée d'un contrôleur spécifique et non standard.

En prenant en considération le premier cas, on sait dans quelle direction on s'engage: équipements nécessitant des contrats de maintenance dont le coût est proportionnel au prix

[4]Boîte métallique dans laquelle on peut loger un certain nombre de modules électroniques répondant à la même norme

de l'installation, disponibilité du système dépendant du nombre d'utilisateurs en ligne et du *system manager* (à qui il faut verser un salaire), complexité de connexion et parasitage du bus par le *system-crate*. De surcroît, les cycles CAMAC isolés (sans DMA) sont soumis aux règles d'entrées-sorties d'un système multi-utilisateurs, c'est-à-dire qu'ils doivent passer par un noyau superviseur qui ralentit passablement le déroulement.

Dans le second cas, l'achat d'un contrôleur de baie spécifique est une dépense inutile, car on trouve très facilement des contrôleurs standard qui ne sont plus utilisés et qui sont disponibles à moindre frais. De plus, la limitation à une seule baie par PC est un obstacle rapidement insurmontable.

La calibration du BGO nécessite l'utilisation de deux à trois baies CAMAC, ce qui élimine la seconde possibilité. L'utilisation d'un équipement de grande envergure équipé d'un *system-crate* est une solution envisageable, mais la maintenance d'un tel système se chiffre en milliers de francs par an.

Toutefois dans le laboratoire il y a des micro-ordinateurs utilisés par un ingénieur technicien qui ne sont ni des PC, ni des "gros" calculateurs, mais qui offrent les avantages des deux types sans en présenter les inconvénients. Seulement deux points noirs viennent ternir l'avantage de cette petite machine: elle ne possède pas de contrôleur de branche CAMAC et il n'y a aucun logiciel d'acquisition de données.

La décision est finalement prise. Le micro-ordinateur étant performant et d'un coût très faible, il est décidé de développer une interface aux normes EUR 4600, ainsi que les logiciels nécessaires à la calibration du BGO. Les raisons sont très simples: les développements étant en partie donnés à des étudiants diplômants, le coût se révèle être peu élevé comparativement à quelqu'autre solution, et comme dans toute expérience de grande envergure on cherche à minimiser les coûts de chaque phase de la construction...

## 1.3 Pré-développement

La machine choisie pour la calibration du BGO est un micro-ordinateur Amiga[5]. Cette machine, bien que peu répandue sur le marché professionnel de la micro-informatique, dispose d'un système d'exploitation multi-tâches temps-réel, est dotée de divers coprocesseurs et est construite autour d'un micro-processeur de la famille Motorola 68000[6].

Le système de fenêtres et de menus déroulants est intégré au système d'exploitation, ce qui lui donne une cohérence et une rapidité intéressante.

Une particularité de la machine est l'absence quasi-totale d'adresses mémoires réservées. En effet, seules quelques adresses sont fixes, dont: les interruptions, l'emplacement où se trouve l'adresse de base de la librairie permettant d'accéder aux fonctions du système d'exploita-

[5] Amiga est une marque déposée par la société Commodore-Amiga Inc.

[6] Divers modèles de machines existent, ils vont actuellement du 68000 au 68030/68882 à 25 Mhz.

tion, l'adresse d'initialisation des périphériques lors de l'autoconfiguration ainsi que quelques adresses réservées par les coprocesseurs.

Cette particularité apporte quelques complications lors du développement des périphériques[7] et des logiciels puisqu'il faut les définir de façon à ce que **tout** soit relogeable. Cependant, la *relogeabilité* apporte une sécurité sur l'avenir puisque le constructeur s'engage de cette façon à rendre tout développement compatible avec les machines et les systèmes d'exploitation du futur.

Le bus de l'Amiga ressemble passablement à un bus VME [5]. Une documentation technique de la machine ayant été fournie [6], non sans peine, par le constructeur, la réalisation de l'interface CAMAC a pu rapidement démarrer et le premier prototype est sorti des laboratoires en quelques mois. Le travail avait été donné à un diplômant en physique secondé par un ingénieur technicien de recherches[8] [7].

Une fois le travail de diplôme sanctionné, le prototype a été repris et une version *commerciale* répondant totalement aux normes a été produite. Le résultat: une interface CAMAC capable de contrôler une branche CAMAC standard, avec des contrôleurs de baies standards, construite avec une technologie standard, bref, permettant de récupérer tous les modules CAMAC utilisés dans les précédentes expériences.

La première partie du cahier des charges ayant été remplie, il a fallu passer à la seconde, la partie logiciel. A ce moment, seul un tout petit groupe de personnes croyaient à l'utilisation d'un Amiga dans L3. L'orientation vers un ordinateur de type VAX 11 reprenait le dessus, mais à force de persévérance, et surtout performances à l'appui, c'est l'Amiga qui a obtenu le *droit* de calibrer les lampes au Xenon du BGO.

[7] Qui n'ont pas d'espace d'adressage propre, voir paragraphe 2.1

[8] L'auteur de cet ouvrage!

# Chapitre 2

# Etude

Avant de se placer devant un écran et se mettre à programmer, il faut étudier les besoins et les ressources mises à disposition de tous les côtés. Les besoins dans une expérience de la taille de L3 ne sont jamais très clairs au départ. Souvent il arrive que les besoins changent au cours du temps, c'est un des grands intérêts de la recherche. Les ressources aussi changent; le système d'exploitation de n'importe quel ordinateur est constamment en progression et l'électronique est régulièrement adaptée aux besoins des nouveaux critères de l'expérience.

Il faut donc impérativement déterminer ce qui est stable et invariable avant de commencer la recherche. Dans le cadre d'un système d'exploitation, il y a toute une série de mécanismes qui restent en place définitivement. En fait, c'est la présentation des mécanismes à l'utilisateur qui reste identique, la manière de les mettre en place pouvant varier. Mais un système d'exploitation évolue. Si on tente de contourner certains mécanismes qui semblent "lourds" à première vue, on ne peut plus garantir un fonctionnement à long terme.

Le CAMAC est une norme industrielle. De par l'âge de sa création, on peut considérer que les variations sur certains détails de cette norme seront insignifiantes; toutefois la norme CAMAC ne définit pas le fonctionnement des modules, seule est invariable la façon de communiquer. Dans notre cas, il nous faut faire abstraction du fonctionnement des modules et mettre à disposition des mécanismes de dialogue valables pour tous.

Voyons maintenant le problème dans son ensemble. On a choisi le CAMAC, ainsi qu'une brochette de modules capables de transformer l'information provenant d'un phénomène physique en données numériques, puis on a décidé d'écrire des logiciels capables de lire ces données.

Si chaque programme doit, à chaque fois, contrôler tous les détails de la communication entre l'électronique et le micro-ordinateur, on va multiplier d'autant les risques d'erreur. Par contre, si les programmes font appel à un mécanisme de communication qui gère les détails, les risques d'erreur ne sont plus proportionnels au nombre d'utilisateurs mais dépendent de la qualité de ce mécanisme.

Les logiciels de prise de données ne se souciant plus des protocoles de communication accèdent

alors aux données en faisant appel à un mécanisme ayant toutes les compétences pour ce travail. Dès lors, ils deviennent plus clairs et plus fiables. On peut répéter cette approche plusieurs fois (ne l'a-t-on pas déjà fait en introduisant *l'interprète* qu'est l'interface CAMAC?) en augmentant le niveau d'abstraction d'une fois à l'autre.

Dans notre cas, une nouvelle composante entre en ligne de compte: le temps. En effet, le CAMAC et ses modules, bien que dépassés, fonctionnent à des vitesses proches de la durée de quelques opérations élémentaires d'un micro-processeur. En introduisant à chaque étape un mécanisme permettant de faire abstraction des détails, on limite la vitesse de communication entre les programmes et les modules électroniques. Il faut donc trouver un équilibre entre la rapidité et la facilité.

Alors que le développement décrit dans le paragraphe 1.3 s'occupe d'interfacer une branche CAMAC au bus de l'Amiga, il faut maintenant étudier le moyen d'implanter le mécanisme qui s'occupera des détails de communication entre un programme et l'électronique qui se trouve dans les baies CAMAC. Pour cela, une étude rigoureuse des différents protocoles et autres mystères du système d'exploitation doit être entreprise.

## 2.1 L'électronique

Avant d'entreprendre l'étude des différentes possibilités offertes par le système d'exploitation, voyons un peu comment l'interface CAMAC fonctionne et de quelle façon il faut la considérer.

L'interface CAMAC est un périphérique répondant aux spécifications Zorro II [1] et à la norme EUR 4600. En clair, cela signifie qu'elle respecte d'un côté les spécifications du bus de la machine et que, de l'autre côté, elle répond aux normes CAMAC jusqu'au niveau d'une branche.

Trois parties bien distinctes nous concernent directement:

- La procédure d'autoconfiguration.
- Les registres de lecture, d'écriture et de commande.
- La gestion des interruptions.

La procédure d'autoconfiguration, du point de vue de l'électronique, est destinée à éviter toute confusion sur le bus. En effet, le micro-ordinateur a été développé de façon à évoluer dans le temps. L'espace d'adressage de la machine est flexible et ni le code, ni l'électronique ne doivent imposer des adresses mémoire définitives; d'ailleurs, il n'y a que très peu d'adresses qui soient réservées par le système d'exploitation ou par des périphériques [8]. En ce qui concerne les cartes introduites dans les connecteurs du bus, elles ne doivent en aucun cas

[1] Zorro II est le terme donné au bus de l'Amiga qui inclu les spécifications électriques, mécaniques, ainsi que la procédure d'autoconfiguration [6].

avoir un espace d'adressage déterminé. Lors de l'allumage ou de la remise à zéro (reset) du système, la procédure d'autoconfiguration va s'adresser à la première carte qui se trouve sur le bus en activant certains signaux électriques. Cette carte devra donner des informations telles que le numéro de fabriquant, modèle, numéro de série, espace-mémoire utilisé ou fourni, utilisation de quelles interruptions, adresse relative d'un morceau de code, et ainsi-de-suite. La procédure d'autoconfiguration choisira alors une adresse de base qui sera transmise à la carte et qui sera mémorisée dans le système d'exploitation. Une fois la première carte configurée, elle donne le contrôle à la suivante et ainsi-de-suite, jusqu'à la dernière.

On voit immédiatement l'avantage d'une telle procédure: plusieurs exemplaires d'une même carte peuvent être connectés autant de fois que nécessaire sans aucune intervention physique, vu que la procédure d'autoconfiguration se charge de donner des adresses de base qui ne se recouvrent jamais. De ce fait, toutes les cartes, bien qu'identiques, ont un espace d'adressage distinct. Au niveau des logiciels, il faut d'abord consulter le système d'exploitation pour connaître la position du périphérique avant d'adresser ses registres qui se trouvent à une position relative à cette adresse de base.

Les différents registres, qui sont alors des adresses-mémoire, peuvent être lus ou écrits selon les besoins. C'est en écrivant dans un registre bien précis que l'on démarre un cycle CAMAC décrit plus en détail dans le paragraphe suivant.

Les interruptions permettent à la carte de signaler qu'une requête urgente est à traiter. Un registre d'un type un peu particulier permet de spécifier pour quelles raisons on désire que le périphérique interrompe le système d'exploitation.

La réalisation de l'interface CAMAC a donné lieu à quelques publications qui décrivent en détail son fonctionnement. Le chapitre suivant, beaucoup plus technique, traite de la façon d'accéder aux registres. En résumé: l'interface une fois connectée au bus, permet de contrôler une branche CAMAC en lisant et en écrivant des données à certaines adresses-mémoire relatives à une position de base.

## 2.2 Communication avec le CAMAC

L'électronique qui se trouve dans les baies CAMAC répond à un certain protocole de communication. Pour comprendre son fonctionnement, voyons comment se déroule un cycle CAMAC dans une baie, puis étendons le principe dans le cadre d'une branche.

Une baie est divisée en 25 emplacements dans lesquels peuvent se glisser des modules. Chaque emplacement est désigné par un numéro que l'on nomme *"station"*. Les modules qui se trouvent dans ces stations communiquent par l'intermédiaire d'un bus. Sur le bus il y a, entre autres, 24 bits pour le transit des données, un système de sélection divisé en adresses et en fonctions, ainsi que 25 lignes d'arbitration, soit une par station.

Afin de démarrer un cycle CAMAC dans une baie, il faut sélectionner la station (N) et

spécifier quelle adresse (A) et quelle fonction (F) on désire exploiter sur le module situé dans cette station. Après avoir correctement manipulé les fils de contrôle du bus, le module effectue l'action demandée et, au besoin, transfère les informations sur les bits de données. On appelle cela un cycle NAF, il dure exactement 1 $\mu$-seconde.

En fait, ce cycle est géré par un contrôleur de baie occupant les deux dernières stations. Dans le cadre d'un système à une seule baie, le contrôleur doit communiquer directement avec l'unité de commande (ordinateur); dans notre cas, nous avons à faire à la norme étendue. La norme étendue permet de connecter plusieurs baies en série. Pour différencier les baies, sept fils supplémentaires sont rajoutés sur le bus; on peut donc en brancher sept en série. Pour accéder aux baies, il faut rajouter un préfixe (C) à tous les cycles CAMAC qui deviennent des cycles CNAF.

Pour démarrer un cycle CAMAC dans une branche, il faut procéder de la même façon que pour un cycle NAF, mais en spécifiant la ou les baies concernées. Les bits de contrôle sont un petit peu différents pour les cycles CNAF et sont gérés soit par une baie spéciale dans le cas des "gros" ordinateurs, soit par un contrôleur de branche qui, dans notre cas, n'est rien d'autre que l'interface CAMAC. La durée totale d'un cycle CNAF est de 2.2 $\mu$-secondes, le temps utilisé par le contrôleur de baie pour effectuer le cycle NAF étant compris.

Les modules sont donc des "esclaves" puisqu'ils ne peuvent être que commandés à distance. Cependant, ils ont la possibilité d'informer le contrôleur de baie qu'ils désirent être entendus. Si le contrôleur de baie est programmé pour accepter ce genre de requêtes, il signalera sur un bit de contrôle de la branche qu'un module a quelque chose à dire. Cette procédure sommaire s'appelle *"Look At Me"* (LAM), et un cycle un peu particulier (Graded LAM) permet de déterminer quelle station s'est manifestée (mais pas dans quelle baie!). Dans le cas de l'interface CAMAC, les requêtes LAM peuvent être dirigées sur le gestionnaire d'interruptions.

## 2.3 Multi-tâches et temps réel

Maintenant que nous avons une idée générale sur le fonctionnement du CAMAC, voyons quelle est la philosophie du système d'exploitation. Mis à la disposition du grand public à la fin 1985, le système d'exploitation de l'Amiga a introduit différents concepts innovateurs pour la micro-informatique domestique. La tendance générale de voir Unix s'implanter sur une majorité de systèmes a poussé le constructeur à s'orienter vers une solution multi-tâches.

La pratique du temps partagé sur les mini-ordinateurs était devenue usuelle, mais sur les micro-ordinateurs, elle constituait un défi. Le but n'était pas de permettre à plusieurs personnes de travailler simultanément sur un même processeur, mais d'offrir la possibilité d'exécuter simultanément plusieurs programmes. Conçu initialement pour "tourner" sur des processeurs Motorola 68000 fonctionnant à 7 MHz, le système d'exploitation se devait d'être le plus performant possible.

De par leur conception, les micro-ordinateurs, toutes marques confondues, ont en général offert la possibilité de traiter les interruptions en temps-réel. Du moment que l'Amiga offrait en plus la possibilité d'exécuter plusieurs programmes simultanément, il ne devait pas enlever la notion de temps-réel à la machine. C'est pour cette raison que le concept de "noyau superviseur" n'a pas été retenu au profit du concept de "noyau temps-réel".

Le noyau temps-réel de l'Amiga répond au nom de *"Exec Kernel"* [9, 10]. Son fonctionnement est basé sur une série d'anneaux du type *"Round-Robin"*, un anneau par niveau de priorité. En clair, cela signifie que tout processus prêt à être exécuté, et qui possède le niveau de priorité le plus élevé, obtiendra le processeur en égale proportion avec les autres processus prêts de même niveau de priorité. S'il n'existe plus de processus prêts à priorité maximale, ce sont les processus du niveau de priorité précédant qui ont droit au processeur, et ainsi de suite. Ce qui implique que la notion de priorité est traitée de manière absolue et non de manière relative, comme dans le cas de UNIX, ou de manière dynamique dans le cas de VMS.

Il y a deux exceptions à cette règle:

1. Une interruption physique survient, un mécanisme d'identification est alors immédiatement mis en route, c'est la garantie qu'une interruption est reconnue dans un laps de temps fini. D'ailleurs, c'est de cette façon qu'Exec prend la main régulièrement pour décider à qui il doit la céder, une circuiterie spéciale de basse priorité ayant été conçue spécialement à cet effet.

2. Un processus décide explicitement qu'il prend la main jusqu'à nouvel avis. En fait, il masque l'interruption physique permettant à Exec de reprendre le contrôle mais laisse les autres interruptions actives, ce qui garantit toujours le temps-réel au niveau de la reconnaissance de l'interruption (pas forcément dans son traitement).

Les implications du point de vue de la programmation sont importantes. Premièrement, l'attente active doit être absolument bannie de toute programmation. Ensuite, les ressources communes ne peuvent pas être accédées sans précautions préalables. Enfin, une discipline stricte concernant l'utilisation du processeur doit être observée, surtout en ce qui concerne la seconde exception citée précédemment. Bref, ce qui est valable avec un *super-ordinateur* reste valable avec Exec.

## 2.4 Messagerie

Avant d'entrer dans le principe de fonctionnement de la messagerie inter-processus, il faut savoir que, pour des raisons de performances générales[2], vu qu'en principe il n'y a qu'une seule personne qui utilise le système à la fois et que la confidentialité des données n'est pas

[2]On se demande parfois si ces contraintes sont toujours objectives, compte tenu des performances des nouveaux modèles d'Amiga

une nécessité, seul le mode superviseur du processeur est utilisé par toutes les tâches. De plus, la mémoire n'est pas virtualisée.

L'exécution simultanée de plusieurs processus n'est pas d'un grand intérêt s'il n'existe pas un moyen de communiquer entre-eux. Exec ne joue pas seulement le rôle d'arbitre entre les différentes tâches, il est également le gérant d'un système de communication.

Tous les programmes qui désirent communiquer, et c'est le cas ne fût-ce que pour une entrée-sortie, doivent au préalable créer un canal de communication appelé un *"port"*. Ce *port* est créé par l'intermédiaire de fonctions que Exec met à disposition au travers d'une librairie.

Une fois le canal de communication établi, on envoie des messages dans le *port* du processus avec qui l'on désire dialoguer. Ce processus nous répond à l'aide du même mécanisme, mais cette fois dans notre propre *port*.

Ces messages ne sont en fait que des pointeurs sur les informations que l'on désire transmettre. Du moment que la mémoire n'est pas virtuelle, ce mécanisme est extrêmement performant puisqu'on ne "transmet" pas le message mais que l'on met un pointeur en queue. Pour "signaler" le processus qui attend le message, Exec active un bit indiquant que *"si la main doit être donnée au processus avec qui l'on communique, ce bit ne représente pas un obstacle"* (masque de conditions d'exécution d'un processus).

Ce type de communication est valable quel que soit le type de processus étant susceptible d'utiliser le processeur. Par exemple, le gestionnaire de fenêtres et de menus déroulants (*Intuition*) [11] est un processus qui communique avec les programmes à l'aide de ce mécanisme. Le port série est accédé en communiquant avec un processus nommé *serial.device*. Le clavier a un processus apparenté qui s'appelle *input.device* et il communique avec un autre processus qui s'appelle *console.device* (le programmeur peut communiquer avec les deux processus, le choix dépend de l'application).

## 2.5 Driver

Mais il existe des processus ayant un rôle bien précis à jouer, ce sont les drivers. Ces drivers, ou gestionnaires d'entrées-sorties, communiquent également à l'aide du même mécanisme décrit dans le paragraphe précédent.

Dans un environnement multi-tâches, on ne peut pas se permettre d'accéder à une ressource sans précautions préalables. En effet, si un programme désire écrire des informations sur un disque et qu'il prend la liberté d'écrire des données dans un fichier quelconque sans se soucier de son utilisation par un autre programme, on peut gager que des problèmes apparaîtront très rapidement.

D'un autre côté, certaines ressources, telles que notre disque cité plus haut, sont en principe capables d'être partagées par plusieurs programmes. Dans l'exemple précédent, si deux programmes désirent écrire *simultanément* deux fichiers différents, il n'y a a priori pas de

contre-indication.

Pour palier à ce genre de problèmes, on attribue un unique processus à chaque ressource. Ce processus *traite* des *requêtes* d'entrées-sorties. En fait, pour utiliser la ressource, on doit communiquer avec son processus gestionnaire en lui envoyant des messages contenant des indications sur le *travail* que l'on désire lui voir accomplir.

Le driver est ce gestionnaire. En traitant les requêtes les unes après les autres[3], il garantit qu'aucun travail en cours n'est interrompu par un autre programme sans précautions préalables. De plus, il gère la disponibilité de la ressource en refusant, ou en mettant en attente, les requêtes qui ne pourraient pas être prises en considération immédiatement (cas d'une imprimante qui serait utilisée simultanément par plusieurs programmes).

L'intérêt majeur pour le programmeur est de ne pas avoir à se soucier de l'utilisation de la ressource par un autre processus. Si, lors de la première demande d'utilisation de la ressource, son driver répond que la requête ne peut pas être honorée, le programmeur peut alors prendre les mesures qui s'imposent pour continuer la bonne marche du programme. Sans le driver, le programmeur n'a pas la garantie que lorsque son programme devra passer la main à un autre processus, ce nouveau processus ne viendra pas *détruire* le travail qu'aurait dû accomplir la ressource.

Une vision encore plus agréable du driver est son rôle d'interprète. Le terme de *driver* signifie littéralement *conducteur* en anglais. C'est bien un nom qui convient puisqu'il *manœuvre* la ressource alors que le programme lui donne des ordres. En gros, c'est un *chauffeur* qui conduit pour vous sans que vous ayez à savoir conduire ou à connaître les règles de la circulation. Mais, cet avantage n'est pas propre à un driver de système d'exploitation multi-tâches, on trouve également des drivers dans la micro-informatique mono-tâche. La différence fondamentale est que, dans notre cas, le driver est un processus, tandis que dans le cas simple, il est un ensemble de fonctions activées par un programme ou par une interruption.

[3] Principe de la messagerie vue au paragraphe précédent, les messages sont mis en queue et sont traités les uns après les autres.

# Chapitre 3

# Driver

La réalisation d'un driver nécessite une bonne connaissance du système d'exploitation pour lequel il est écrit. Comme nous l'avons vu auparavant, le driver est une *porte de communication* avec la ressource physique ou logique à laquelle il est associé. Cette *porte de communication* doit impérativement respecter certaines règles qui permettront à un programme de l'utiliser de façon *transparente*.

Pour exprimer de façon plus claire le terme de "transparence", prenons l'exemple d'une imprimante. Cette dernière peut être connectée soit à un port sériel, soit à un port parallèle, soit à une quelconque interface-réseau. Si un logiciel est à même de produire un flux à imprimer, ce n'est pas au programme de gérer la connexion physique de l'imprimante, et encore moins le protocole de communication.

En écrivant un driver qui respecte les règles de base définies par le système d'exploitation, on permet aux autres logiciels de ne pas avoir à se soucier des règles de communication propres à la ressource pour laquelle le driver a été écrit. Dans le cas de notre imprimante, on communique avec un port ou un autre strictement de la même façon, alors qu'une ligne sérielle asynchrone n'a rien de commun avec une ligne parallèle synchrone.

Le terme de *certaines règles* est un peu vague. En effet, certaines ressources sont plus compliquées que d'autres, mais toutes doivent avoir des points communs. L'initialisation, la lecture, l'écriture, l'annulation d'un précédent ordre, la remise à zéro ainsi que la signalisation de fin d'utilisation sont des actions que tout driver devrait comprendre. Cependant, certaines actions diffèrent d'un driver à l'autre. L'initialisation d'un driver de ligne sérielle étant assez sommaire, elle peut se révéler plus compliquée si l'on a à faire à une ligne réseau, car à quel moment doit-on préciser le nom du noeud avec lequel on veut communiquer?

Le driver étant la partie *visible* d'une ressource physique ou logique, il faut que ce dernier soit accessible par n'importe quel programme. Si la ressource est inexistante ou hors d'état, le driver doit refléter cette situation. L'installation d'un driver dans un système d'exploitation est donc d'une importance capitale.

Le noyau du système d'exploitation tient compte des drivers de la façon suivante:

1. Tant qu'il n'existe pas de relation entre un périphérique et un driver, ce dernier est ignoré (voir §3.1).
2. Une fois le driver autoconfiguré, il devient un objet passif au même titre qu'une librairie résidente et réentrante, et cela jusqu'à ce qu'un programme y fasse appel (voir §3.2).
3. Du moment qu'un programme utilise le driver:
   (a) Il devient un objet actif s'il doit constamment s'occuper du périphérique. C'est le cas du driver *ethernet* qui détache une tâche indépendante pour surveiller le trafic réseau. Ce cas ne nous concernant pas, il ne sera pas traité dans cet ouvrage.
   (b) Il reste un objet passif, même s'il est capable de traiter les entrées-sorties de façon asynchrone, et pour autant que l'asynchronicité puisse être réalisée à l'aide d'interruptions provenant du périphérique (voir §3.3).

   Dans tous les cas, la communication entre le driver et le programme est gérée par la messagerie mise à disposition par Exec (voir §3.4).

## 3.1 Autoconfiguration

Lors de la mise en route de la machine (ou lors d'un reset), les périphériques connectés au système sont catalogués dans une liste appelée `ConfigDev`. Lors de l'initialisation du système d'exploitation, les périphériques internes de la machine tels que port sériel, parallèle, audio, etc..., et qui ne sont pas dans cette liste `ConfigDev`, se voient automatiquement affublés de leurs drivers correspondants qui se trouvent dans le directoire `devs:`.

Dans le directoire `sys:Expansion` se trouvent les drivers dits *autoconfigurables*. Un driver du directoire `sys:Expansion` est un *bout de code* associé à une ressource physique décelée à l'allumage et présente dans la liste `ConfigDev`. A chaque driver correspond un fichier ayant même nom mais dont l'extension est `.info`. Ce fichier est un descripteur qui contient diverses informations telles que l'imagerie à visualiser (icône) par le gestionnaire de fenêtres, le type de données (`Tool` dans notre cas), ainsi qu'une liste de paramètres qui seront transmis au driver associé. Ces paramètres sont la signature du périphérique, à savoir, le numéro de constructeur et le modèle (dans notre cas 2202/88). On peut également y introduire quelques paramètres propres au périphérique.

Afin de mettre en relation périphériques et drivers, la commande `BindDrivers` du système d'exploitation doit être exécutée. Cette commande examine les descripteurs du directoire `sys:Expansion` et les compare avec les périphériques de la liste `ConfigDev`. Si les signatures correspondent et que le bit `CONFIGME` du périphérique est présent, le driver (dont le descripteur en est la signature) est chargé en mémoire et la fonction "Initialisation-Physique" du driver est exécutée. C'est la fonction "Initialisation-Physique" qui recevra en argument, la liste des paramètres définis dans le descripteur du driver.

Cette commande `BindDrivers` est en principe exécutée à chaque démarrage de la machine. Cependant, il est possible de mettre un driver dans le directoire `sys:Expansion` et de taper la commande `BindDrivers` à n'importe quel moment, seuls les périphériques non encore pourvus d'un driver (dont le bit `CONFIGME` est présent) seront concernés par cette commande. On voit d'emblée qu'une fonction de type "Libère-Périphérique" doit être également existante dans le driver; elle doit remettre le périphérique dans un état initial, remettre le bit `CONFIGME` et éventuellement signaler au système d'exploitation d'enlever le driver de la mémoire si plus aucun lien logique ne subsiste pour ce driver.

Le développement d'un driver autoconfiguré est donc assez agréable vu qu'il est possible d'installer un driver, de le tester, de le corriger ou de l'améliorer, d'enlever la précédente version de la mémoire, puis de le réinstaller en mémoire pour recommencer un cycle de développement. Le tout sans réinitialiser le système d'exploitation.

### 3.1.1 Autres méthodes

Il existe encore deux autres systèmes permettant d'installer un driver en mémoire.

- Le système dit `Mount`. On donne une définition exhaustive des paramètres du périphérique dans le fichier `devs:MountList` puis on tape la commande `Mount xyz`. En fait, et hormis le fait que la commande `Mount` permette de sélectionner le driver que l'on désire utiliser, elle permet également d'installer des drivers pour des ressources logiques (comme les disques virtuels), ainsi que d'autres types de ressources résidentes.
- Le système dit `romboot`. Lors de l'autoconfiguration physique, un périphérique peut signaler qu'il a un programme d'initialisation à exécuter. Ce programme d'initialisation se trouve dans une ROM du périphérique, et par conséquent dans son espace d'adressage; il sera exécuté avant même le démarrage du système d'exploitation. En général, le programme aura pour mission de *monter* le driver se trouvant également sur la ROM, permettant au périphérique de se comporter comme un *Boot-Device*, c'est-à-dire, un périphérique capable de se comporter comme un disque.

N'étant pas nécessaires à l'élaboration du driver CAMAC, ces deux méthodes ne seront pas developpées dans le présent ouvrage.

### 3.1.2 Illustration

Ces quelques lignes de code illustrent la fonction "Initialisation-Physique" décrite auparavant. Vous noterez au passage l'utilisation de la librairie Expansion qui permet de déterminer rapidement l'adresse de base du périphérique.

```
initRoutine:
        ;
        ; Usage des registres
        ; ====================
        ;
        ; In:
        ; d0 -- CamDev Pointer
        ; a0 -- Segment List pointer (our code)
        ; a6 -- Exec base
        ; Out:
        ; d0 -- 0 = Error, else Device pointer
        ; Misc Used:
        ; a1 -- Expansion Lib Name
        ; a4 -- Adresse d'Expansion library base
        ; a5 -- pointeur sur le device
        ;
        movem.l a0/a1/a4/a5,-(sp)       ; Préserve tous les registres modifiés
        bchg    #1,$bfe001              ; Baisse l'intensité de la led...
        move.l  d0,a5                   ; a5 = pointeur sur le device
        move.l  a6,md_SysLib(a5)        ; sauve le pointeur sur exec
        move.l  a0,md_SegList(a5)       ; sauve le pointer sur notre code
        lea.l   ExLibName,A1            ; Prend le nom de expansion.library
        moveq.l #0,D0                   ; version la plus récente
        jsr     _OpenLibrary(a6)        ; Ouvre expansion.library
        tst.l   D0                      ; Ouvert correctement ?
        bne.s   init_OpSuccess
        ;
init_OpFail:
        ; Alert definitions
AG_OpenLib      EQU     $00030000
AO_ExpansionLib EQU     $0000800A
        ;
        movem.l d7,-(sp)                ; Sauve d7
        move.l  #AG_OpenLib!AO_ExpansionLib,d7
        jsr     _Alert(a6)
        movem.l (sp)+,d7                ; et récupère d7
        bra     Init_End                ; et termine
        ;
        ; Autoconfiguration: donne un pointeur sur device structure
        ; et va chercher les bindings (il semble que le pointeur sur
        ; le device contient le numero de fabriquant !! )
        ;
init_OpSuccess:
        move.l  D0,A4           ; Addresse de expansion.library dans A4
        lea     md_Bindings(A5),A0 ; Ou ranger les bindings
        moveq   #4,D0           ; Ne prend que l'adresse (longueur = 4 bytes)
        move.l  a4,a6           ; Adresse de la librairie Expansion
        jsr     _GetCurrentBinding(a6)
        movea.l 4,a6            ; adresse de EXEC dans a6
        move.l  md_Bindings(A5),D0 ; Prend l'adresse des bindings
        tst.l   D0              ; Si notre carte n'est pas trouvée
        beq     Init_False      ; Sort et unloade le driver
        move.l  D0,A0           ; Récupère l'addresse de la structure config
        move.l  cd_BoardAddr(A0),md_Base(A5)    ; Sauve l'adresse de base
        move.l  cd_BoardAddr(A0),D0             ; Adresse de base dans D0
        add.l   #WORDREL,D0                     ; Première adresse Camac
        move.l  D0,md_Where(A5)                 ; calculée à l'avance
        bclr.b  #CDB_CONFIGME,cd_Flags(A0)      ; Le device est configuré
        move.l  a4,a1           ; Fermeture de la librairie expansion
        jsr     _CloseLibrary(a6)
        move.l  a5,d0           ; retourner pointeur sur le device = OK
        bchg    #1,$bfe001      ; La led... Signifie ça fonctionne
Init_End:
        movem.l (sp)+,a0/a1/a4/a5  ; récuperation des registres
        rts
```

```
        ;
        ; On doit mettre d0 a 0 pour signaler que l'autoconfig n'a pas
        ; trouvé la carte.
Init_False:
        move.l  a4,a1           ; Fermeture de la librairie expansion
        jsr     _CloseLibrary(a6)
        moveq   #0,d0           ; Signale l'erreur
        bra.s   Init_End        ; et retour
```

La fonction permettant d'enlever le driver de la mémoire est illustrée à l'aide de ces quelques lignes de code:

```
;
; Expunge enlève la librairie Camac de la mémoire. C'est une opération
; délicate qui ne doit pas être effectuée si LIB_OPENCOUNT n'est pas nul.
; Note: lors du Expunge, le multi-tâches est desactivé... Ça aide!
Expunge:        ; ( device: a6 )
        tst.w   LIB_OPENCNT(a6)         ; regarde si une tâche nous utilise
        beq     1$                      ; non? alors va purger
        bset    #LIBB_DELEXP,md_Flags(a6)  ; on est encore ouvert alors
        moveq.l #0,d0                   ; mettre le expunge retardé et rendre
        bra.s   Expunge_End             ; la valeur 0 dans d0
1$:
        ; passage où l'on se tue la librairie, donc le device.
        movem.l d1/d2/a1/a5,-(sp)       ; sauvetage des registres utilisés
        move.l  a6,a5                   ; a6 va être utilisé pour Exec
        move.l  md_SegList(a5),d2       ; copie notre seglist dans d2
        move.l  a6,a1                   ; copie du device pointeur dans a1
        movea.l 4,a6                    ; exec library pointeur dans a6
        jsr     _Remove(a6)             ; on s'enlève de la liste des devices
        ;
        ; Dans AutoConfig, le device est libre, le signaler
        ;
        movea.l md_Bindings(a5),a1         ; Autoconfig structure of camac
        bset.b  #CDB_CONFIGME,cd_Flags(a1) ; Le device est configuré
        ;
        ; commandes spécifiques au device pour le 'fermer' ici...
        ;
        ; rendre la mémoire a l'allocateur mémoire
        moveq.l #0,d0                   ; clean MSW
        moveq.l #0,d1                   ; of data registers
        move.l  a5,a1                   ; pointeur du device dans a1
        move.w  LIB_NEGSIZE(a5),d1      ; position relative au dev pointer
        sub.w   d1,a1                   ; soustraite au dev point = adresse
        add.w   LIB_POSSIZE(a5),d0      ; de longueur d0
        add.l   d1,d0                   ; calcul de la place utilisée réelle
        jsr     _FreeMem(a6)            ; merci, a1 et d0 vous indiquent tout
        move.l  d2,d0                   ; Notre SegList dans D0 (valeur retour)
        move.l  a5,a6                   ; récupère device pointer
        movem.l (sp)+,d1/d2/a1/a5       ; récupère les registres sauvés
Expunge_End:
        rts                             ; et retour
```

## 3.2 Lien dynamique et librairie résidente

Lors de l'écriture d'un programme, on peut faire appel à des fonctions d'un type un peu particulier, les fonctions des librairies résidentes et réentrantes. A la différence des librairies

constituées d'un ensemble de fonctions en code objet, les librairies résidentes et réentrantes offrent la possibilité d'être utilisées par plusieurs programmes simultanément, un peu comme les *shared object libraries* de UNIX.

Une autre grande particularité de ces librairies résidentes est la possibilité d'effectuer des liens, au même titre que le ferait l'éditeur de lien, mais de façon dynamique, lors de l'exécution d'un programme.

Pour utiliser une fonction d'une librairie résidente et réentrante, il faut tout d'abord faire appel à `OpenLibrary()` de Exec, qui permet d'accéder à la table des fonctions de cette librairie. Une fois la librairie *ouverte*, on peut accéder à n'importe laquelle de ses fonctions, sans oublier qu'il faut effectuer un `CloseLibrary()` après la dernière utilisation de la librairie.

Si l'on fait le rapprochement entre librairie résidente et driver, on remarque que tous deux ont strictement la même structure. En effet, si l'on désire accéder à un driver, il est nécessaire d'appeler `OpenDevice()`, les fonctions sont des fonctions d'entrées-sorties et le `CloseDevice()` libère les ressources utilisées.

Les fonctions du driver sont très limitées et ne sont que la demande d'une entrée-sortie ou l'avortement d'une précédente demande. Mais alors pourquoi différentier le driver de la librairie? En fait, un programme ne fait jamais directement appel aux fonctions du driver mais utilise le mécanisme d'entrées-sorties du système d'exploitation qui lui, communique avec le driver. Ce mécanisme garantit que toute requête au périphérique est traitée selon le principe d'une queue (FIFO).

### 3.2.1 Illustration

Les points d'entrée des fonctions de la pseudo-librairie qu'est un driver, sont définis dans une table.

```
funcTable:
        ;
        ;------ Les routines standard de toute librairie
        dc.l    Open
        dc.l    Close
        dc.l    Expunge
        dc.l    Null
        ;
        ;------ Les entrées spécifiques à ma librairie
        dc.l    BeginIO
        dc.l    AbortIO
        ;
        ;------ Délimiteur de la fin de la table de fonctions
        dc.l    -1
```

Open et Close sont appelés par le système d'exploitation suite à un `OpenDevice` et à un `CloseDevice` respectivement. Dans le cas des librairies résidentes, ce sont les appels à `OpenLibrary` et à `CloseLibrary` qui ont le même effet.

```
;
; Open met IO_ERROR en erreur.  Si tout a fonctionné
; correctement, LIB_OPENCNT aura un point de plus.
; Pour la version multi-cartes, il faudra initialiser un champ avec
; l'adresse de la carte en fonction de l'unité.
;
Open:           ; ( device:a6, iob:a1, unitnum:d0, flags:d1 )
        ;------ Regarde si l'unité existe (actuellement 1)
        subq    #1,d0           ; l'unité ZERO est interdite
        cmp.l   #MD_NUMUNITS,d0 ; compare avec le nombre que nous possedons
        bcc.s   Open_Error      ; Impossible.
        ;------ Signale que nous avons ouvert encore une fois le device
        addq.w  #1,LIB_OPENCNT(a6)
        moveq.l #0,d0           ; pas d'erreur (C'est Lessieur!)
        rts                     ; et rend la main
Open_Error:
        move.b #IOERR_OPENFAIL,IO_ERROR(a1)    ; erreur signalée dans io_error
        move.b #IOERR_OPENFAIL,d0              ; et dans d0 (retour de la fct)
        rts                                     ; et retour
;
; Du fait que nous n'avons que des appels synchrones et qu'une seule
; unité, La seule chose à faire est de nettoyer IO_DEVICE et de
; decrémenter le nombre d'utilisater de la  librairie.
;
Close:          ; ( device:a6, iob:a1 )
        ;------ protéger contre une autre utilisation future
        moveq.l #-1,d0                  ; mettre pour cela -1 dans
        move.l  d0,IO_DEVICE(a1)        ; IORequest->IODevice
        moveq.l #0,d0                   ; Mean Close is OK
        subq.w  #1,LIB_OPENCNT(a6)      ; 1 de moins qui appelle la librairie
        bne.s   Close_End               ; Si pas le dernier, c'est tout.
        ;------ see if we have a delayed expunge pending
        btst    #LIBB_DELEXP,md_Flags(a6) ; Avons nous une purge en cours
        beq.s   Close_End               ; Non? alors termine
        bsr     Expunge                 ; oui? alors purgeons, purgeons...
Close_End:
        rts                             ; et hop, c'est terminé
```

`Expunge` a été décrite dans le paragraphe 3.1 et `Null` est une fonction redondante définie par le constructeur.

`BeginIO` et `AbortIO` sont les deux seules fonctions que doit comporter le driver. C'est particulièrement dans `BeginIO` que les détails de la requête d'entrée-sortie sont traités.

## 3.3 Synchrone et asynchrone

Selon les périphériques utilisés, les requêtes d'entrées-sorties ne peuvent pas toujours être traitées directement. Soit on attend que le périphérique soit prêt, on effectue le travail demandé puis on rend le contrôle au programme, soit on met la requête en attente et on rend le contrôle au programme. Une fois le périphérique prêt et la requête d'entrée-sortie terminée, on le signale au requérant à l'aide d'un message (voir §2.4).

Ces deux méthodes sont dites synchrones et asynchrones. En principe, le programmeur choisit une des deux méthodes mais il existe également une méthode qui consiste à laisser le

choix au driver. Si le périphérique est prêt à consommer des données, la requête est traitée directement, sinon elle est mise en attente.

Imaginons que l'on désire lire un secteur d'un disque. Si la tête de lecture se trouve au bon endroit, la lecture peut s'effectuer en un temps relativement court et la requête d'entrée-sortie peut avoir lieu de façon synchrone. Mais si au contraire la tête de lecture se trouve à l'opposé du disque, le temps de synchronisation du disque prend des proportions gigantesques pour le processeur et une entrée-sortie asynchrone se révèle être un bon choix. Dans ce cas, le driver attend une interruption, lira les informations le moment venu et signalera au programme requérant que sa requête est terminée.

Le choix de la synchronicité par le driver est assez simple dans ce cas-là mais ne l'est pas pour le programmeur. On peut tout de même dire que dans ce cas précis, la solution asynchrone se révèle être un bon choix, car l'expérience veut que les constructeurs de disques donnent toujours des temps-moyens d'accès aux informations qui sont de l'ordre de plusieurs millisecondes, alors qu'un cycle processeur est inférieur à la microseconde.

### 3.3.1 Synchronicité du CAMAC

Lors du développement de la première version du driver CAMAC, les solutions synchrones et asynchrones ont été programmées. Si un programme demandait explicitement une entrée-sortie asynchrone, le driver rendait le contrôle au programme, effectuait le ou les cycles demandés, puis signalait la complétude de son travail à l'aide d'un message.

Mais après avoir mesuré les performances, il s'est avéré que le mécanisme, trop *lourd* pour un périphérique de ce type, n'était pas idéal. En effet, il a suffi de laisser le choix du type de synchronicité au driver pour se rendre compte que toutes les requêtes étaient traitées de façon synchrone.

Le calcul est simple: un cycle CAMAC dans une branche est de 2.2 $\mu$-secondes exactement. Le driver, pour connaître la disponibilité de l'interface, n'a qu'à effectuer le test d'un bit. Compte tenu que le périphérique se trouve dans un espace mémoire qui ne peut être ni virtualisé, ni caché, ni lu à l'aide du *burst-mode*[1], le test du bit nécessite obligatoirement une lecture préalable d'un emplacement mémoire. On peut donc considérer qu'il faut une lecture indirecte avec déplacement en *worst case* suivi du test qui peut être effectué en *best case* étant donné que le code n'a pas de restriction. Le temps minimum avant de prendre la décision étant de 10 cycles horloge, cela signifie que pour un processeur dont l'horloge est de 25 MHz, il faut au minimum 0.4 $\mu$-secondes pour effectuer le test.

Sachant que le driver exécute d'autres instructions avant de passer au choix même du mécanisme d'entrées-sorties, les 2.2 $\mu$-secondes nécessaires à la complétude du précédent cycle CAMAC sont très largement dépassées.

[1]Mode que l'on trouve à partir du 68030 et qui permet un transfert concaténé de 4 mots de 32 bits d'affilée, de la mémoire centrale à la mémoire cache.

La première version du driver a alors été revue et une solution ne violant pas les règles d'accès au driver a été prise en considération. En fait, toutes les requêtes sont traitées de façon synchrone. Cependant, si le programme veut explicitement une entrée-sortie asynchrone, le driver effectue le travail demandé puis rend le contrôle au programme tout en envoyant le message de complétude nécessaire.

A priori, il ne semble pas que cette seconde méthode soit plus performante. En fait, tout le mécanisme d'empilement des demandes asynchrones est supprimé et l'accès au périphérique, qui nécessitait une série de tests préalable, a été réduit à sa plus simple expression. Le programmeur quant à lui ne voit qu'un accroissement de performance.

### 3.3.2 Les requêtes d'entrées-sorties

Les requêtes d'entrée-sortie sont effectuées à l'aide de fonctions de la librairie Exec. Toutes trois nécessitent un seul argument qui est une *enveloppe* contenant un message pour le driver. Pour spécifier le type de synchronicité que l'on désire utiliser, on fait appel à la fonction `DoIO()` pour les requêtes synchrones et `SendIO()` pour les requêtes asynchrones. Pour laisser le choix au driver, on utilise `BeginIO()`.

Le principe consiste à regarder si le bit `IOB_QUICK` est présent dans le champ `IO_FLAGS` du message contenu dans *l'enveloppe*. Côté driver, il signifie que la requête est synchrone (`DoIO()` et `BeginIO()` mettent ce bit avant de passer le message au driver), côté programmeur, il signifie que la requête a été traitée de façon synchrone (`BeginIO()` met ce bit ainsi qu'un second signifiant que le choix est donné au driver).

Si le programmeur a choisi une solution asynchrone, ou si le driver en a ainsi décidé, il doit alors par la suite contrôler la complétude de l'entrée-sortie, ou attendre passivement, avant d'utiliser le résultat de la requête.

### 3.3.3 Illustration

Une fois la requête envoyée, le système d'exploitation va activer la fonction `BeginIO` du driver et lui donner *l'enveloppe*. Notre cas est relativement simple. Une table contient les points d'entrée des différentes actions possibles et un saut indirect à une de ces adresses est effectué en fonction du contenu de la requête (voir §3.5).

```
;
; BeginIO s'occupe de tous les IO. NOTE: ils sont tous synchrones.
;
BeginIO:        ; (in: iob: a1, device:a6; noout; scratch: a0, d0)
        movem.l a0,-(sp)                        ; et des registres sauvés
        move.w  IO_COMMAND(a1),d0               ; la commande existe ?
        cmp.w   #MYDEV_END,d0                   ; va consulter la table
        bcc     BeginIO_NoCmd                   ; si non, pas de commande
        ; On ne traite pas les cas asynchrones mais on les accepte
        clr.b   IO_ERROR(a1)                    ; pas d'erreur pour l'instant
```

```
        lsl     #2,d0                           ; * 4 pour l'offset dans table
        lea     cmdtable(pc),a0                 ; adresse table dans a0
        move.l  0(a0,d0.w),a0                   ; adresse routine dans a0
        jsr     (a0)                            ; on va à la routine

BeginIO_End:
        movem.l (sp)+,a0                ; récupère les registres
        rts                             ; fin de BEGINIO

BeginIO_NoCmd:                          ; Cette commande n'existe pas
        move.b  #IOERR_NOCMD,IO_ERROR(a1)
        bra.s   BeginIO_End             ; termine BeginIO
```

C'est à la fin de l'entrée-sortie que le bit IOB_QUICK est testé. Son absence signifie qu'il faut renvoyer le message au requérant par l'intermédiare d'Exec.

```
;
; TermIO renvoie l'IOReq à l'utilisateur. Le bit IOB_QUICK détermine si
; l'utilisateur a utilisé SendIO ou DoIO. Vu que BeginIO a mis le bit et que
; la requête était de toute façon synchrone, le cas est identique à DoIO.
;
TermIO:         ; ( in: iob:a1, devptr:a6; noout; noscratch )
        btst    #IOB_QUICK,IO_FLAGS(a1) ; si le bit Quick est mis, on rend la
        bne.s   TermIO_End              ; main à l'utilisateur, sinon on rend
        move.l  a6,-(sp)
        movea.l 4,a6                    ;    (adresse de exec dans a6)
        jsr     _ReplyMsg(a6)           ; un message pour dire terminé.
        move.l  (sp)+,a6                ; récupère a6
TermIO_End:
        rts
```

## 3.4 Communication

Pour utiliser le driver depuis un programme, il faut tout d'abord y accéder. S'il est présent et utilisable, il faut alors créer une *enveloppe* qui permettra d'envoyer des requêtes. Dans cette *enveloppe*, on spécifie le type d'accès que l'on désire effectuer (voir §3.5) puis on la "poste".

En C, pour "poster" les requêtes, les cas se présentent de la façon suivante:

Partant du principe qu'une requête d'entrée-sortie est un pointeur sur une structure de type `struct IORequest` et qu'elle est correctement initialisée avec toutes les informations nécessaires à l'entrée-sortie,

- le cas synchrone se traduit par la ligne de code:

  ```
  DoIO(requête);
  ```

- le cas asynchrone se traduit par différentes alternatives telles que:

```
SendIO(requête);  /* dans tous les cas */
```

puis

```
for (..;..;..) { /* solution permettant de continuer autre chose en attendant */
  if (CheckIO(requête)) WaitIO(requête);

  /* faire autre chose entre temps */
       ...
}
```

ou

```
WaitIO(requête); /* attente passive */
```

ou

```
/* cas de l'attente de plusieurs sources d'interruption */
bitrequête = 1 << requête->IOMessage->mp_SigBit;
quiafini = Wait(unbit | autrebit | bitrequête);
if (quiafini & bitrequête) {  /* si c'est notre requête */
  GetMsg(requête->IOMessage); /* enlever le message de la liste */
     ...
}
```

* Pour terminer, laisser le choix au driver se traduit par:

```
BeginIO(requête);
if (requête->IOFlags & IOB_QUICK) {
  /* requête synchrone , l'IO est terminé */
} else {
  /* traiter comme une requête asynchrone */
     ...
}
```

Exec met également à disposition la fonction `AbortIO(requête)` qui permet de retirer une requête envoyée préalablement de façon asynchrone. Au niveau du système d'exploitation, si la requête est encore en queue, elle sera retirée de la liste et le driver ne verra jamais cette requête. Si, par contre, la requête n'est plus dans la queue, la seconde fonction du driver (nommée souvent `AbortIO` également) est appelée par le système d'exploitation. Le driver décide alors s'il peut, et de quelle façon, interrompre ladite entrée-sortie. Dans notre cas, vu que toute requête est traitée immédiatement, l'avortement se révèle inutile et cette fonction ne fait rien. C'est encore un exemple de la simplification du driver par la suppression de l'asynchronicité vu que l'on n'a pas besoin de tenir à jour une liste de requêtes en attente.

## 3.5 Les types d'actions

Nous l'avons vu préalablement, une requête d'entrée-sortie est un message transmis au driver par l'intermédiaire d'Exec. Ce message est une *enveloppe* contenant toutes les informations nécessaires à son acheminement dans les deux sens ainsi que des détails concernant l'accomplissement du travail demandé.

Les chemins d'accès sont créés en même temps que l'établissement du lien dynamique avec le driver, au moment de l'appel à la fonction `OpenDevice()` (voir §3.2). En fait, on prépare un canal de communication appelé *"Message Port"* [9] qui permettra de spécifier une adresse de retour pour le résultat des requêtes. Le lien dynamique quant à lui, permet d'obtenir l'adresse de base du driver.

Comme dans le cas des fonctions de base du driver, il existe une liste de base, prédéfinie par le constructeur, de types d'accès au périphérique; ce sont les actions fondamentales et obligatoires qu'un driver doit être capable de gérer. Cette liste garantit qu'un périphérique de type A peut être accédé de la même façon qu'un périphérique de type B.

Maintenant, chaque périphérique a ses propres particularités et une liste non standard de types d'accès peut être rajoutée à la liste de base. C'est dans la table consultée par la fonction `BeginIO` du driver que l'on donne les points d'entrée des différentes actions que le driver est capable de traiter:

```
;
; cmdtable est utilisée pour savoir quelle routine va être appelée en fonction
; de la commande dans IOReq->Command
;
cmdtable:                       ; Code      Action
        DC.L    Invalid         ; $00000001 Première obligatoire
        DC.L    MyReset         ; $00000002
        DC.L    Read            ; $00000004
        DC.L    Write           ; $00000008
        DC.L    Update          ; $00000010
        DC.L    Clear           ; $00000020
        DC.L    MyStop          ; $00000040
        DC.L    Start           ; $00000080
        DC.L    Flush           ; $00000100 Dernière obligatoire
        DC.L    NoRdWr          ; $00000200 Fonction camac spéciale
        DC.L    ReadGL          ; $00000400 Special read Graded Lam
        DC.L    Crates          ; $00000800 Quels crates sont online ?
        DC.L    Read16          ; $00001000 Read sur 16 bits
        DC.L    Write16         ; $00002000 Write sur 16 bits
        DC.L    CamacZ          ; $00004000 Initialisation de la branche
        DC.L    CaLis16         ; $00008000 Camac List 16 bits (pseudo DMA)
        DC.L    CaLis24         ; $00010000 Camac List 24 bits (pseudo DMA)
cmdtable_end:
```

### 3.5.1 L'obligatoire

Les neuf premiers types d'accès au périphérique sont les actions standard et obligatoires de tout driver.

- `Invalid` est une action de type *"do nothing"* qui permet de tester le bon fonctionnement du driver puisqu'elle doit rendre la requête avec un bit d'erreur déterminé.

- `MyReset` doit réinitialiser physiquement le périphérique. L'interface CAMAC est dotée d'un mécanisme physique de réinitialisation; c'est à l'aide de cette action que ce mécanisme est activé.

- `Read` et `Write` sont les lectures et écritures normales. La requête doit également contenir un pointeur sur des données ainsi que la quantité d'informations à transférer. Pour certains drivers, il faut également spécifier la destination, c'est le cas du CAMAC.

- `Update` et `Flush` forcent toutes les entrées-sorties en attente, à être exécutées avant toute autre requête. `Update` est une action synchrone et ne rend le contrôle qu'une fois le travail terminé. `Flush` doit par contre garantir qu'aucune autre requête ne sera traitée tant que celles qui sont en attente ne soient terminées; c'est donc à la requête suivante que l'on risque d'attendre. Ces actions n'ont aucun sens dans notre cas étant donné que toutes nos requêtes sont traitées de façon synchrone; c'est encore une amélioration des performances du driver vu qu'il n'est pas nécessaire de gérer des tampons de données.

- `Clear` nettoie les tampons internes du driver. Inutile dans notre cas.

- `MyStop` et `Start` doivent permettre de verrouiller et de déverrouiller le périphérique. Une fois le périphérique verrouillé, toute entrée-sortie asynchrone doit être mise localement en queue et toute entrée-sortie synchrone refusée. De plus, l'action de verrouillage ne doit se limiter qu'au processus désireux de stopper le traitement de ses requêtes.[t] Dans notre cas, implanté au départ, ce mécanisme a une fois de plus été supprimé vu que les tampons internes ont également été supprimés. On pourrait refuser toute requête, qu'elle soit synchrone ou non, mais cela ne correspondrait plus à la définition de base du constructeur... La pratique nous prouve que le verrouillage du CAMAC n'est pas d'une grande utilité et qu'il suffit de ne pas soumettre de requête pour obtenir le même résultat!

### 3.5.2 Le nécessaire

Si les actions définies par le constructeur doivent être identiques d'un driver à l'autre, il est possible d'en définir d'autres pour utiliser le périphérique de façon plus complète et plus efficace. Voici une description des actions définies dans le cadre du CAMAC.

- `NoRdWr` est un cycle CAMAC sans transfert de données. Cette action est identique à une lecture ou à une écriture dans son principe, seul le pointeur sur les données est absent. Ce type de cycle est également appelé `Fonction`, puisqu'il est utilisé pour activer des fonctions spéciales de certains modules.

- `Read16` et `Write16` sont des lectures écritures identiques à `Read` et `Write`. La différence réside dans le format des données transférées. Dans ce cas, le pointeur sur les données pointe sur des mots de 16 bits alors que dans l'autre cas, le pointeur pointe sur des données de 32 bits. Le CAMAC étant défini sur 24 bits, les données seront soit "tronquées" dans le cas présent, soit "étendues" dans le cas normal par adjonction d'un octet de valeur nulle. Par expérience, une majorité de données transférées sont des mots de 16 bits. Le gain de performance de ce type d'accès est mineur puisqu'il n'y a qu'une mise à zéro d'un registre, suivi d'une lecture en adressage indirect indexé et d'une écriture en adressage indirect post-incrémenté de différence entre le mode 16 bits et le mode 32 bits, mais cette différence devient très intéressante lors de transferts multiples car ces trois instructions représentent près d'un tiers du temps total d'un cycle.

- `Crates` permet de connaître la liste des baies qui sont en état de fontionnement. Un registre du périphérique contient l'état instantanné de chaque baie, cette action permet d'en obtenir le contenu.

- `CamacZ` est un cycle CAMAC particulier qui a comme résultat, l'initialisation et la remise à zéro de toute la branche.

- `ReadGL` est une lecture CAMAC réservée aux opérations de *"Graded LAM"*. Ces lectures permettent de déterminer quel module a actionné le mécanisme *"Look At Me"* (voir §4.2 et suivants pour plus de détails au sujet du LAM).

- `CaLis16` permet d'effectuer des cycles CAMAC avec ou sans transfert de données et tous différents. En effet, au lieu de définir un type de cycle précis, on en définit autant que nécessaire dans un tableau de structures. Lors de l'exécution de la requête, le driver effectuera les cycles définis dans le tableau au lieu de n'effectuer que le cycle défini usuellement. C'est en somme un genre de tableau de cycles qui est éxécuté.

- `CaLis24` est identique à `Calis16`. Cependant, l'espace nécessaire pour le stockage des données est constitué de mots de 32 bits et le driver lit ou écrit des mots de 24 bits sur le CAMAC. Si aucun transfert de données n'est de plus de 16 bits, il est recommandé d'utiliser `CaLis16` pour les mêmes raisons que celles invoquées dans les cas de `Write16` et `Read16`.

# Chapitre 4

# Outils de développement

Bien qu'un driver joue le rôle d'interprète entre l'électronique et le programmeur, sa programmation n'est pas des plus aisées. Le programmeur occasionnel est toujours surpris par la complexité des entrées-sorties. Pourquoi faut-il se plonger dans une théorie sur les *ports* de communication et sur la messagerie entre processus, alors que l'on ne désire envoyer que quelques mots à un périphérique?

Nous l'avons vu dans le chapitre précédent, la programmation du driver CAMAC n'est pas très aisée. Il faut initialiser toute une série de structures avant le premier accès puis, avant chaque cycle, le préparer minutieusement avant de le *poster* correctement. Ce n'est pas seulement l'accès au driver qui représente un travail considérable, le respect absolu des règles d'utilisation des structures imposées par le système d'expoitation est également un élément capital pour le bon fonctionnement du driver et nécessite une discipline stricte de programmation.

Pour écrire des petits programmes utilisant le CAMAC, il serait agréable de pouvoir y accéder de façon simple, sans devoir se poser de question sur les mécanismes *intimes* des entrées-sorties.

En créant des outils d'aide au développement, on se facilite la tâche pour l'avenir et on diminue les risques d'erreurs. En construisant ces outils et en mettant son code source et objet à disposition des programmeurs, on apporte un outil didactique puisqu'il tient en plus le rôle d'exemple et de mode d'emploi. Et bien sûr, tout cela doit être intégré en respectant les contraintes de "visibilité" imposées par l'encapsulation du driver.

Ce chapitre se contentera de présenter une librairie permettant de programmer des cycles CAMAC le plus simplement possible. On pourrait imaginer un système encore plus simple basé sur des idéogrammes que l'on déplacerait à l'aide de la souris ou que l'on sélectionnerait d'une façon ou d'une autre, supprimant ainsi toute programmation. Mais nous n'en sommes pas là; un tel outil demanderait un tel travail de conception qu'il représenterait à lui seul un travail de recherche...

## 4.1 Librairie

Pour pouvoir accéder au driver, nous avons vu au chapitre précédent qu'il fallait au préalable préparer un canal de communication ainsi qu'une enveloppe.

Partant du principe que ces deux entités seront utilisées dans chaque requête d'entrée-sortie, un unique appel à une fonction initialisant le tout ne serait pas un luxe.

A la fin de tout programme utilisant le CAMAC, il faut bien évidemment rendre au système toutes les ressources utilisées par le programme. Comme dans le cas de l'initialisation, on pourrait laisser ce travail à une seule fonction.

Deux fonctions, c'est déjà le début d'une librairie... Voyons maintenant de quelle façon on pourrait encore faciliter la tâche du programmeur.

La majeure partie des cycles CAMAC sont des lectures, des écritures et l'exécution de fonctions de contrôle. Ces cycles ont la particularité d'être quasiment identiques à quelques détails d'initialisation près. De plus, le choix d'une lecture ou d'une écriture ne dépend que de la valeur de l'argument `Fonction`.

Le choix de la baie avec laquelle on désire communiquer n'est qu'une écriture dans une structure. Une macro-instruction pour le préprocesseur du compilateur C pourrait très bien faire l'affaire. En étendant ce principe, toute une série de macro-instructions *booléennes* pourraient très bien convenir au test de certains bits résultant de cycles préalablement exécutés.

Les contrôleurs de baie donnent la possibilité de modifier les cycles CAMAC de différentes manières. Entre autres, il y a un bit nommé `Inhibit` qui joue un rôle de "modificateur de signification". De plus, ces contrôleurs peuvent gérer le mécanisme LAM en préparant un vecteur désignant le ou les modules désirant communiquer avec l'ordinateur. En principe, ces particularités sont programmables en un cycle.

On peut dès lors imaginer quelques fonctions permettant de cacher les côtés *pénibles* de l'accès au driver. En leur donnant un nom explicite, on améliorera considérablement la lisibilité des logiciels.

Le paragraphe 4.3 détaille les différentes fonctions contenues dans la librairie. Il faut cependant prendre en considération les éléments suivants:

- Avant d'utiliser les fonctions de la librairie, il faut au préalable appeler la fonction d'initialisation qui alloue les ressources nécessaires à tout cycle CAMAC.
- L'utilisation de la librairie introduit deux variables globales. Ces deux variables sont des pointeurs sur les ressources réservées lors de l'initialisation. Il est possible de *profiter* de leur existence pour communiquer directement avec le driver, mais dans ce cas il faut prendre soin de toujours les remettre dans leur état initial.
- Le driver est un élément critique dont l'écriture a fait l'objet d'une attention particuliè-

re, surtout en ce qui concerne la vitesse d'exécution. L'utilisation d'une librairie est une facilité qui peut nuire dans le cas d'applications qui dépendent du facteur temps. En effet, l'appel d'une fonction d'entrée-sortie de la librairie nécessite un empilement des arguments, un saut à ladite fonction, un test de validité des arguments et un ajustement de la pile au retour de l'appel, le tout en sus d'une entrée-sortie classique en utilisant directement le driver. Le prix à payer pour un gain de temps à la programmation peut donc se révéler être important, si le volume d'entrées-sorties est élevé.

- Une fois que l'on a terminé d'utiliser la librairie, il faut rendre au système les ressources réservées au départ. Les variables globales sont dès lors inutilisables.

Si l'on tient compte des points qui précèdent, il est alors possible de trouver un compromis raisonnable entre vitesse d'exécution et clarté de programmation. En effectuant les initialisations et les cycles CAMAC isolés à l'aide de la librairie, on optera pour la facilité et la clarté de programmation, et en programmant le "driver nu" lors des moments critiques en temps, on privilégiera la vitesse d'exécution.

## 4.2 Les interruptions

Nous l'avons vu au paragraphe 2.3, Exec est un système d'exploitation temps-réel. Dans les discussions précédentes, il est fait mention de la capacité que l'interface CAMAC a de générer des interruptions. Voyons maintenant de quelle façon et pour quelles raisons il faut implanter un mécanisme capable de modifier le cours séquentiel d'un programme utilisant le CAMAC.

### 4.2.1 Pourquoi?

Partant du principe qu'un programme d'acquisition de données dialogue avec de l'électronique qui est en état de fonctionner, et que le temps disponible à la saisie de l'information est toujours insuffisant[1], il faut généralement développer des algorithmes permettant de gagner du temps lors des moments les plus critiques. Le premier réflexe est alors de supprimer ce qui semble inutile.

Prenons une expérience qui nécessite l'acquisition d'un grand volume de données provenant d'une multitude de modules électroniques. Si à chaque cycle on se met à contrôler qu'aucune erreur ne s'est glissée, on dira que le programme est fiable mais peu performant. Si au contraire, on ne contrôle la validité des cycles qu'épisodiquement, voire pas du tout, personne ne dira rien jusqu'au jour où l'on se rendra compte que les données acquises ne représentent rien!

[1] Cela peut paraître ironique mais l'expérimentation nous le prouve!

Hormis le fait qu'un cycle ne puisse s'exécuter à cause de l'indisponibilité des mécanismes d'entrées-sorties[2], c'est l'interface CAMAC qui détecte les anomalies physiques et les signale dans son registre d'état. Le driver copiant ce registre dans l'espace fourni lors de la requête, il est possible de le contrôler à tout moment. Mais cette solution, bien qu'élégante, n'est pas toujours satisfaisante.

Imaginons maintenant que l'on ne vérifie jamais la validité des entrées-sorties, mais qu'on désire tout de même être informé d'une défaillance du CAMAC. Pour cela, il faudrait qu'un mécanisme prioritaire soit activé par l'interface dès l'apparition d'une certaine condition.

Dans le même ordre d'idées, et compte tenu du fonctionnement du système d'exploitation, ce mécanisme n'est-il pas aussi valable pour que l'électronique qui se trouve dans les baies CAMAC signale qu'elle veut communiquer? En fait, l'interface CAMAC gère la ligne BD[3] de la même manière que les anomalies de cycles. Seul diffère le bit activé dans le registre d'état. Etant donné qu'il est possible de tester après chaque requête si la ligne BD est active, on pourrait imaginer un algorithme qui permettrait de tenir compte des demandes des modules. Cependant, il ne serait pas envisageable d'attendre passivement que l'électronique se manifeste.

L'approche du problème des erreurs ou du traitement de la ligne BD est différente du point de vue de la programmation, mais toutes deux nécessitent le même mécanisme de débranchement asynchrone. L'interruption étant par définition ce-dit mécanisme, il ne nous reste plus qu'à l'implanter.

### 4.2.2 Comment?

L'interface CAMAC est capable de générer des interruptions *"hardware"*. Cela signifie qu'elle est capable d'intervenir physiquement auprès du processeur pour lui faire changer le déroulement de ses opérations en activant certains signaux électriques.

Les processeurs Motorola de la série 68000 ont 7 niveaux d'interruptions dont le moins prioritaire est le niveau 1. Ce niveau de priorité inférieur est principalement utilisé par Exec[4].

Le niveau le plus prioritaire actuellement utilisé est le niveau 6, ce sont les contrôleurs pressés tels que le DMA et le SCSI qui l'utilisent. A ce niveau, aucune interruption n'est masquable car personne n'utilise le niveau 7 qui est réservé pour utilisation future par le constructeur.

Le niveau qui nous intéresse est le niveau 2. Le CAMAC n'étant pas une ressource "vitale" pour le fonctionnement de la machine, il peut être pris en compte (ou être interrompu) après l'asservissement des ressources plus critiques.

[2]Cas où la requête devrait être refusée par le système d'exploitation, donc à cause d'une faute grave.

[3]Branch Demand.

[4]Une circuiterie interne active régulièrement une interruption de niveau 1 permettant à Exec de changer de contexte.

Pour que les interruptions soient prises en compte en *temps-réel*, il faut qu'elles soient prioritaires sur Exec. Du moment que toute interruption *physique* reçoit un accusé de réception avant que le système d'exploitation n'en soit informé, la condition de base est remplie.

Lorsqu'une interruption est générée, elle possède physiquement un niveau de priorité (vecteur)[5]. Le processeur masque les interruptions d'un niveau inférieur ou égal, sauve quelques registres sur la pile et charge son compteur ordinal avec l'adresse trouvée dans l'emplacement mémoire correspondant au niveau d'interruption.

A partir de ce moment, la suite des évènements dépend du code se trouvant à l'adresse pointée par le compteur ordinal, sauf si une interruption d'un niveau supérieur devait se produire; dans tel cas, cette même procédure serait à nouveau exécutée.

Pour correctement gérer le code des différents niveaux d'interruption, il existe deux mécanismes:

- "Le **handler**". Si, pour un niveau d'interruption précis, on désire prendre le contrôle de la situation, il faut demander à Exec de remplacer le code exécuté par défaut par une procédure fournie à ce propos. Pour répondre aux règles imposées par un *interrupt handler*, le code doit sauver certains registres, ne pas durer plus de quelques millisecondes et, surtout, remettre le processeur dans l'état dans lequel il se trouvait avant l'interruption (*return from interrupt*). Il existe trois sortes de *handlers* par défaut: ceux qui ne font rien comme dans le cas du niveau 7, ceux qui sont privés comme dans le cas du niveau 1 et ceux qui gèrent le mécanisme décrit ci-dessous.
- "L'**interrupt server chain**". Le troisième cas de *handler* permet de hiérarchiser un niveau d'interruption en chaînant plusieurs serveurs. Un serveur d'interruption est une fonction qui peut utiliser librement les registres *scratch*[6], qui doit durer le moins de temps possible et qui doit retourner dans un registre, une valeur indiquant si l'interruption concerne le serveur dont la fonction en est le représentant. En gros, le serveur détermine en quelques instructions si l'interruption le concerne, et retourne une valeur nulle si ce n'est pas le cas. Dans l'affirmative, il quittance l'interruption et prend les mesures qui s'imposent avant de retourner une valeur non-nulle. Si une valeur nulle a été retournée au *handler*, c'est le serveur suivant dans la chaîne qui sera exécuté et ainsi de suite.

Comme on peut le constater, il n'y a pas de limites quant au nombre d'interfaces susceptibles de générer des interruptions. Non seulement on peut physiquement déterminer le niveau d'interruption en spécifiant un vecteur mais, dans un même niveau, on peut déterminer l'ordre dans lequel on désire les traiter.

Exec met à disposition quelques fonctions permettant d'introduire sans risque votre propre *handler* ou *interrupt server*. Il faut cependant faire très attention au code de ces derniers,

[5] A chaque vecteur correspond un emplacement mémoire (déterminé par Motorola) dans lequel se trouve l'adresse d'une fonction à exécuter.

[6] Les deux premiers registres d'adresses et de données.

compte tenu de leur importance dans le bon fonctionnement du système. Il est difficilement pensable que quelqu'un remplace le *handler* de niveau 6 sans pour autant détruire les requêtes du DMA ou du SCSI.

### 4.2.3 Illustration

Le mécanisme mis en œuvre dans notre cas est l'attente d'une interruption par le programme d'acquisition, signalant que le moment de saisir de l'information est arrivé ou qu'une erreur s'est produite.

Cette interruption est une interruption logique *(soft interrupt)* générée par le serveur d'interruption. Pour déterminer les critères qui autoriseront l'interface à interrompre le système, il faut charger un registre avec une certaine valeur lors de l'installation du serveur.

Dans le serveur, pour déterminer la provenance d'une interruption par l'interface CAMAC, une simple lecture à une adresse suffit. De plus, une écriture à une adresse enlève la cause de l'interruption puisqu'elle remet une bascule[7] dans son état initial.

```
;
        SECTION intserver.a
;
IAMPULLING      EQU     7       ; "I am pulling the interrupt" bit de INTCRL1
INTCTRL1        EQU     $40     ; Offset par rapport à l'adresse de base
INTACK          EQU     $40     ; Un write à cette adresse pour enlever INT
SIGNAL          EQU     -6*54   ; Définition de Signal dans Exec
;
        XDEF    _CamServer
;
_CamServer:
        move.l  (a1),a0         ; L'adresse de base de la carte dans a0
        btst    #IAMPULLING,INTCTRL1(a0)  ; est-ce moi ?
        beq.s   pasmoi          ; non ? alors -> pas moi !
        move.b  #0,INTACK(a0)   ; on enlève l'interruption
        movea.l 4,a6            ; Adresse de EXEC dans a6
        move.l  4(a1),d0        ; Le masque de bits à signaler
        move.l  8(a1),a1        ; pointeur sur la tâche en a1
        jsr     SIGNAL(a6)      ; Signale la tâche que interrupt...
        moveq   #1,d0           ; On stoppe la recherche d'interruption
        rts                     ; et retour
pasmoi: moveq.l #0,d0           ; Continue la recherche du serveur
        rts                     ; et retour
;
        END
```

Il a fallu trois instructions pour déterminer si l'interface CAMAC était à l'origine de l'interruption, deux pour quitter le serveur dans la négative, et quelques-unes pour signaler à la tâche en attente que l'interruption s'était produite. Ce sera lors de la reprise du processeur par la tâche attendante que l'interruption sera signalée de façon logique; ce serveur n'est, pour cette raison, pas un serveur d'interruption *temps-réel*.

[7] Bascule de type D se trouvant sur l'interface.

Compte tenu de l'algorithme utilisé pour implanter le serveur, il n'y a qu'une seule tâche qui peut être servie par ce mécanisme. Pour pouvoir servir plusieurs tâches par ce serveur d'interruptions, il faut écrire un noyau qui recoit le signal du serveur et qui le redistribue vers la "bonne" tâche selon un critère prédéterminé (erreurs, LAM, etc...).

## 4.3 Contenu de la librairie

Pour construire la librairie, le compilateur C met à disposition un utilitaire qui archive différents modules de code objet. Pour générer ces modules, il suffit de compiler les différentes fonctions qui constitueront la librairie.

La librairie est un ensemble de fonctions répertoriées qui seront sélectionnées et ajoutées à votre programme lors de l'édition de liens. Cette facilité n'a rien de commun avec les librairies résidentes et réentrantes du paragraphe 3.2 puisque chaque programme a sa propre copie du code, ce qui signifie qu'à chaque modification de la librairie, il faut à nouveau effectuer une édition de liens de tous les programmes.

Voyons sommairement quelles sont les fonctions qui devraient se trouver dans cette librairie. Une description plus détaillée se trouve dans une publication disponible auprès du secrétariat du département de physique nucléaire de l'Université de Genève [12].

- **InitCamac** et **EndCamac** permettent d'accéder et de disposer du CAMAC respectivement.
- **Camac, Camac2, CamacM** et **Camac2M** permettent d'effectuer des cycles CAMAC sur 16 ou 24 bits, une seule fois ou de façon multiple.
- **Crate, AddCrate** et **SubCrate** sont des macro-instructions[8] permettant de spécifier les baies concernées par les futurs cycles CAMAC.
- **StatCamac, DoneCamac, QCamac, XCamac** et **TestBD** sont des macro-instructions permettant de connaître le contenu du registre d'état. Ces macro-instructions donnent des informations concernant le dernier cycle qui se soit déroulé.
- **Inhibit, SetInhibit** et **ClearInhibit** sont des fonctions qui permettent de manipuler le mode Inhibit d'un contrôleur de baie standard.
- **Online** et **TestOnline** sont des fonctions qui permettent de connaître l'état de fonctionnement des baies.
- **EnaCamInt** et **DisCamInt** sont des fonctions permettant d'ajouter ou d'enlever le serveur d'interruption du *handler* de niveau 2.

[8] Définies dans le *header* `CamacLib.h` de la librairie.

- **CCamac** est une fonction qui exécute un cycle qui initialise tous les modules d'une baie. En fait, c'est un ordre qui est donné au contrôleur de baie lui demandant d'effectuer le travail.
- **ZCamac** est une fonction qui exécute un cycle qui initialise toutes les baies de la branche. C'est un cycle complètement indépendant et qui est une fonction spéciale du driver.
- **Lam, RLam** et **LamOnStation** sont des fonctions qui permettent de savoir si un module d'une baie désire communiquer, de connaître la liste des modules d'une baie désireux de se faire entendre, ou de demander à un module précis s'il désire communiquer. Ces fonctions n'attendent pas qu'un évènement se produise, pour cela il faut utiliser l'attente passive d'une interruption logique.
- **CamacGL** est une fonction qui permet d'exécuter un cycle CAMAC de type *graded lam*. Ce cycle est différent des autres et est géré par le driver. Cette fonction permet d'obtenir un vecteur contenant un "`ou logique`" de tous les modules ayant activé la ligne **BD**.

Bien sûr, cette liste de fonctions de base d'une librairie d'aide au développement n'est pas exhaustive, mais elle est suffisamment complète pour permettre à un programmeur de développer rapidement une application utilisant le CAMAC.

# Chapitre 5

# Prise de données

Dans les précédents chapitres, nous avons vu de quelle façon il faut écrire un driver et quels sont les mécanismes qui permettent d'entrer en communication avec lui. Maintenant que nous avons une interface *software* complète, voyons comment écrire une application.

Tant qu'un programme ne nécessite pas de ressources autres que le CAMAC, on ne rencontre en général aucun problème. On peut soit programmer directement le driver, soit utiliser la librairie si le temps n'est pas un critère déterminant.

Toutefois si un programme ne se contente pas seulement de saisir de l'information, mais qu'en plus il la manipule, la situation est bien différente et ce n'est plus forcément une démarche séquentielle synchrone qui vient à l'esprit.

Voyons maintenant comment écrire un programme utilisant plusieurs ressources simultanément, y compris le CAMAC.

## 5.1 Qui fait quoi et quand?

Avant toute chose, il faut déterminer quels seront les besoins de l'application. Avons-nous besoin d'un réseau de télécommunications, utiliserons-nous des fichiers, la présentation des informations se fait-elle en même temps que la saisie, avons-nous plusieurs processus avec lesquels nous désirons communiquer?

En principe, toutes ces ressources seront gérées par un driver ou, dans le cas du gestionnaire de fenêtres, par un processus indépendant.

1. Le gestionnaire de fenêtres et de menus déroulants est un processus qui se nomme *"Intuition"* et avec lequel on communique à l'aide de messages ou de fonctions de sa librairie résidente et réentrante.

2. L'horloge interne *(timer)* est gérée par un driver qui se programme comme le CAMAC.

3. L'accès au réseau est régi par une hiérarchie contituée de processus *("Internet")*, de driver *(ethernet.device)* et de librairie *(sockets)*.

La différence entre un processus résidant et un driver est mince du point de vue de la programmation. Le driver gère des entrées-sorties d'un type bien précis, et le processus est un programme en exécution qui a son champ d'action déterminé. Un point commun unit ces deux types de ressources: la messagerie interne décrite dans le paragraphe 2.4.

Prenons un cas qui nécessite l'utilisation du CAMAC, de l'horloge interne, et du gestionnaire de fenêtres et de menus déroulants.

Les trois ressources influencent l'application de la façon suivante:

1. Il faut saisir une certaine quantité d'informations lorsque le CAMAC se manifeste.

2. Au bout d'un intervalle de temps déterminé, il faut rafraîchir un graphique représentant l'état du système d'acquisition.

3. L'utilisateur de l'application intervient pour modifier le cours des opérations en utilisant la souris ou le clavier.

C'est lors de l'initialisation de chacune de ces ressources que nous allons déterminer leur niveau d'interaction avec l'application, soit:

1. Lors de l'activation de la ligne BD.

2. Lorsqu'un certain laps de temps s'est écoulé.

3. Si l'utilisateur a choisi un élément d'un menu déroulant, s'il a modifié une fenêtre ou s'il a sélectionné un bouton.

Passons maintenant à l'implantation de ces tâches. En ce qui concerne le CAMAC, il faudra installer le serveur d'interruptions et préparer les différentes structures qui permettront d'exécuter des cycles. De plus, si l'application doit effectuer de nombreux cycles rapidement, il faudra préparer les `CamacList` (ensembles de cycles qui doivent être exécutés séquentiellement en une seule requête).

Le *timer* signale qu'un laps de temps déterminé s'est écoulé au travers d'un *port* de communication. L'initialisation consiste alors à créer ledit *port* et à préparer la structure qui spécifiera l'intervalle de temps qui doit s'écouler entre la requête et une interruption logique générée par le *timer* en cas de dépassement.

Pour communiquer et se synchroniser avec d'autres processus, *Intuition* envoie des messages sur un *port* qu'il faut créer. Les fenêtres, les menus, les boutons, les ascenseurs et une

liste de "causes d'interruptions" doivent être préparés en initialisant les structures qui les représentent.

Une fois les structures initialisées, il faut demander les différentes ressources au système d'exploitation.

1. La fonction `InitCamac` de la librairie du paragraphe 4.3 permet d'accéder au driver CAMAC très simplement. Pour obtenir un bit sur lequel on pourra attendre les interruptions, il faut utiliser la fonction `EnaCamInt` de la même librairie. De cette façon on est sûr que le serveur d'interruption est installé de façon réglementaire et qu'aucune autre tâche n'a déjà installé le serveur (voir §4.2.3).

2. L'accès au driver du *timer* est obtenu à l'aide de la fonction `OpenDevice` de Exec. Lors de l'appel à cette fonction, on spécifie l'adresse du *port* préparé au préalable et au travers duquel le driver pourra se manifester. C'est sur un bit contenu dans la structure du *port* qu'il faudra attendre l'interruption.

3. Pour pouvoir entrer en communication avec *Intuition*, il faut accéder à sa librairie résidente et réentrante (`OpenLibrary` de Exec). Etant donné qu'il n'est pas toujours nécessaire d'avoir un canal de communication avec *Intuition*, ce sera lors de la spécification de certains objets qu'il faudra mentionner le *port* préalablement initialisé

Une fois les ressources correctement allouées, il faut effectuer les quelques initialisations de celles-ci.

1. Quelques cycles CAMAC permettent de remettre dans un état connu, les contrôleurs de baies et les modules concernés par les interruptions.

2. L'envoi d'une requête permet de spécifier que l'on désire être averti s'il est telle heure, ou si un certain laps de temps s'est écoulé.

3. La spécification des objets dont *Intuition* devra tenir compte est effectuée à l'aide des fonctions de la librairie résidente auxquelles on a accédé préalablement [11]. Selon les objets, on spécifie sur quel *port* de communication *Intuition* pourra envoyer ses messages (le bit d'interruption y étant contenu). Il n'est pas nécessaire de fournir tous les éléments que devra traiter *Intuition*, mais seulement ceux qui sont utiles pour le moment.

Nos trois bits d'interruption en main et notre système d'acquisition prêt à effectuer son travail, il faut se mettre en attente passive de l'un des trois évènements possibles. Pour cela, il suffit d'utiliser la fonction `Wait` d'Exec et de donner comme argument un "`ou logique`" des bits sur lesquels on désire attendre.

La fonction `Wait` attendra passivement[1] que le CAMAC, le *timer* ou *Intuition* désirent communiquer. Lors du "déblocage" de l'application, un masque sera retourné. Ce masque

[1] Exec ne donnera pas la main à l'application tant qu'un des bits donné en argument ne sera pas activé par une des ressources (sélection non déterministe).

spécifiera quelles ressources ont réactivé l'application. En comparant l'argument et le masque, il est alors possible de déterminer quelles sont les actions à entreprendre. Par exemple:

1. Le signal signifie qu'il faut lire une partie des modules, car de l'information importante s'y trouve. On pourra utiliser des requêtes de type `CamacList` si le temps à disposition est critique ou si les cycles ne dépendent pas d'une situation précise.

   On pourra éventuellement annuler la requête du *timer* puis la soumettre à nouveau si ce dernier était utilisé comme un *WatchDog*.

   Enfin, on pourra signaler à *Intuition* qu'il peut mettre à jour un compteur d'évènements, ou tout autre élément visualisant le contenu des données acquises à l'instant.

2. Un certain laps de temps s'est écoulé sans qu'il ne se soit rien passé sur le CAMAC, notre *timer* s'est alors manifesté[2] et il est temps de prendre les mesures qui s'imposent.

   On peut alors vérifier le fonctionnement de certains modules, activer certains signaux électriques (à l'aide du CAMAC) et visualiser cet état en demandant à *Intuition* d'activer certains boutons ou de mettre à jour un "compteur d'erreurs".

3. *Intuition* veut communiquer avec nous. L'analyse du contenu du message nous permettra de connaître les raisons pour lesquelles nous avons été interrompu. En effet, la multitude de "causes possibles" est divisée en classes et types, selon que l'utilisateur de l'application a utilisé un menu déroulant, qu'il a sélectionné un bouton, modifié la taille d'une fenêtre[3] ou utilisé le clavier.

   *Intuition* étant le propriétaire de la structure dans laquelle se trouve le message, l'application devra envoyer un accusé de réception à l'adresse de retour[4] spécifiée dans chaque message. Ce "quittançage" est absolument nécessaire car c'est la seule manière qu'a *Intuition* de savoir quels messages ont été lus par l'application.

Le serveur d'interruption du CAMAC n'émet qu'un signal sans message particulier, comme nous l'avons vu au paragraphe 4.2.3, mais le *timer* et *Intuition* ne fonctionnent pas sur ce principe. Etant donné qu'il est possible de soumettre plusieurs différentes requêtes simultanément au *timer* et que les "causes d'interruptions" sont multiples pour *Intuition*, il faut aller lire dans le *port* de la ressource qui interromp (`GetMsg` de Exec), le ou les messages mis en queue. C'est en analysant ces messages que l'on trouvera toute l'information dont on aura besoin.

Après avoir pris les mesures qui s'imposent en fonction du masque retourné par la fonction `Wait`, il ne faudra pas oublier de réarmer les interruptions dans le cas du CAMAC, de resoumettre une requête dans le cas du *timer*, avant de se remettre à attendre passivement.

Tôt ou tard, l'application arrivera à la fin; soit parce que le nombre d'évènements à lire sur le CAMAC sera atteint, soit parce qu'il ne se sera plus rien passé depuis un certain laps

[2]Cas où le *timer* serait utilisé en *WatchDog*.

[3]Pour laquelle on désire dessiner un graphique proportionnel à la dimension de la fenêtre.

[4]Qui est celle du *port* de communication d'*Intuition*.

de temps ou qu'il sera l'heure de tout arrêter, soit parce que l'utilisateur en aura décidé ainsi. Dans tous les cas, il faudra rendre toutes les ressources à Exec, étant donné que la mémoire n'est pas virtualisée et que le noyau ne garde pas la trace des tâches ayant alloué une ressource[5].

1. Il faudra alors désarmer les contrôleurs de baie et ôter le serveur d'interruption, puis signaler au driver qu'il ne sera plus utilisé par notre application.

2. Si une requête a été laissée en suspens, il faudra l'annuler. De plus, on signalera au driver du *timer* que notre application ne le sollicitera plus.

3. Toutes les fenêtres devront être refermées, les menus déroulants et les boutons seront ôtés, bref tout ce qui est encore à la charge d'*Intuition* devra être supprimé. Finalement, il faudra lui indiquer que nous n'aurons plus besoin de lui en "fermant" le point d'accès que nous avions auprès de sa librairie.

Une fois toutes ces ressources rendues au système, il ne restera plus qu'à terminer le programme sans oublier de libérer les éventuels espaces mémoire qui auraient été réservés par un mécanisme autre que celui fourni par le compilateur et en fermant les fichiers laissés ouverts.

Bien que rien ne s'oppose à ce que cette application soit exécutée plusieurs fois simultanément, le système d'exploitation n'y voyant aucun inconvénient, il faut tenir compte du fait que le serveur d'interruptions du CAMAC a été conçu pour ne servir qu'une seule application! Laissons notre imagination se représenter les conséquences d'une exécution parallèle du programme décrit dans ces précédentes pages...

## 5.2 Combien de temps cela dure-t-il?

Le driver CAMAC a été écrit en assembleur, la raison évidente qui vient à l'esprit est la vitesse d'exécution.

On a vu dans le chapitre 3 de quelle façon les appels au driver sont gérés et on imagine combien d'instructions assembleur sont nécessaires pour qu'une requête soit traitée. Voyons maintenant à quelle vitesse réelle notre système fonctionne.

Les durées des cycles mentionnées ci-dessous ont été mesurées, à l'aide d'un oscilloscope, directement dans une baie CAMAC et correspondent au temps écoulé entre le début de deux cycles consécutifs. Les tests ont été effectués à l'aide de simples programmes écrits en C, sans optimisation particulière. Les conditions de test correspondent à un environnement de travail normal (toutes les tâches ont la même priorité).

[5]Cette contrainte a comme conséquence que l'on ne peut pas "tuer" une tâche sans risque car, s'il est possible d'enlever une tâche au niveau du noyau, on ne sait pas quel comportement auront les ressources qui continueront à vouloir communiquer avec des "fantômes". De plus, la mémoire aura tendance à se fragmenter.

| Mode | Bits | Durée ($\mu$sec.) |
|---|---|---|
| Normal | 0 | 39 |
| Normal | 16 | 41 |
| Normal | 24 | 47 |
| Multiple | 0 | 11 |
| Multiple | 16 | 13 |
| Multiple | 24 | 19 |
| CamacList | 16 | 28 |
| CamacList | 24 | 32 |

Les trois types de cycles fondamentaux suivants ont fait l'objet de tests:

**Normal:** exécution de n entrées-sorties (`DoIO`) d'un seul cycle.

**Multiple:** exécution d'une entrée-sortie pour n cycles identiques.

**CamacList:** exécution d'une entrée-sortie pour n cycles différents, la taille des données transmises correspondant au cas le plus défavorable lors du transfert des données.

Ces valeurs ont été mesurées sur le modèle Amiga 2000 ayant un processeur Motorola 68000 travaillant à 7.16 MHz, soit la machine la plus lente de la série Amiga. Cependant, l'interface a été testée sur le modèle Amiga 3000 ayant un processeur Motorola 68030 travaillant à 25 Mhz. Le résultat est un accroissement de la vitesse d'un facteur de l'ordre de 6 en moyenne.[6] Ce qui signifie qu'on est arrivé pratiquement aux limites physiques du CAMAC puisque le temps minimum entre 2 cycles doit être de 2.2 $\mu$sec.

On est bien loin des presque 100 $\mu$sec. d'un Olivetti M24 qui ne pouvait gérer qu'une seule baie ou d'une VAX11/750 qui nécessitait près de 10 mS entre deux appels au driver[6].

## 5.3 Exemple d'utilisation

Dans le cadre de l'expérience L3 du LEP, il a fallu mettre au point rapidement un système permettant d'acquérir des données nécessaires à la calibration du BGO.

Sans entrer dans les détails de cette calibration, il faut savoir qu'une source de lumière provenant d'éclairs de lampes au Xénon[7] est utilisée pour quantifier le "rendement" du sous-détecteur qui nous intéresse. Bien que l'énergie émise par les éclairs soit relativement stable, il faut la quantifier quotidiennement.

[6] Limitation due à la "lourdeur" de VMS mais, si l'on possédait un contrôleur de branche avec séquenceur et DMA, le mode CamacList était exécuté par le séquenceur et les cycles se suivaient alors en 2.2 $\mu$sec. (les autres modes n'étant pas améliorés).

[7] Lampe que l'on retrouve dans les machines à Xérographie.

Pour cela, on "prélève" une partie de cette lumière à l'aide de fibres optiques et on la mesure à l'aide de PMs[8] et de photodiodes. Si les photodiodes sont des récepteurs stables, il n'en va pas de même pour les PMs et ces derniers doivent également être calibrés. A cet effet, on utilise des sources d'Americium qui, associées à un scintillateur, émettent des photons ayant une énergie connue.

La calibration de ces lampes au Xénon est donc effectuée en deux temps. Premièrement, on étudie le comportement des PMs lorsque l'énergie provient des sources d'Américium, puis on "mesure" la lumière émise par les éclairs des lampes.

L'acquisition des données est donc séparée en deux parties. En un premier temps on va lire le plus rapidement possible les valeurs que produisent les PMs. Etant donné que les sources d'Américium produisent des désintégrations à une fréquence de l'ordre du KHz, on peut lire l'électronique de façon discontinue.

En un second temps, on va lire les valeurs que produisent les PMs et les photodiodes lorsqu'une lampe au Xénon a produit un éclair.

Pour permettre aux lampes de produire ces éclairs à tour de rôle, il faut programmer des modules CAMAC selon qu'on désire que les éclairs arrivent à intervalles réguliers (on utilise pour cela un pulseur), ou qu'ils soient synchronisés avec le reste de l'expérience.

Comme ces éclairs ne se produisent qu'à une fréquence relativement basse, les lectures n'ont lieu que lorsque le CAMAC génère des interruptions.

A chaque évènement lu, qu'il provienne de la source d'Americium ou d'un éclair de la lampe au Xénon, il est tenu une statistique basée sur des histogrammes. En fait, les convertisseurs analogiques-digitaux fournissent des valeurs comprises dans certaines limites et les histogrammes consistent à enregistrer dans quelle proportion telle valeur a été acquise. Ces histogrammes permettent de déterminer une foule de critères: spectre de la source d'Americium, stabilité des PMs et des photodiodes, dérives, etc...

Lors de la phase de la prise de données, l'utilisateur est informé visuellement de la progression du travail en cours. A la fin de cette phase, quelques valeurs statistiques sont calculées (pour chaque histogramme) et présentées à l'écran.

Puis une série de choix sont offerts à l'utilisateur, lui permettant de visualiser graphiquement un ou plusieurs histogrammes au choix, de sauver les données pour utilisation future, de calculer une distribution plus précise[9], de changer la présentation à l'écran ou de quitter le programme.

Cette convivialité permet de rapidement déceler des défaillances de l'électronique et de mettre au point le système d'acquisition indépendamment du reste de l'expérience. Mais dans

[8] Photo Multiplicateurs.

[9] Basée sur le principe de l'ajustement d'une Gaussienne sur l'histogramme, on rejette alors tous les évènements sortant inconsidérément des valeurs de la Gaussienne ainsi déterminée et l'on recalcule les valeurs statistiques avec les *bons* évènements.

l'immensité de toute la chaîne d'acquisiton L3, on ne peut pas, lors de *"Runs Globaux"*, être constamment devant l'écran pour interagir avec le logiciel. Pour cela, un canal de communication à travers le réseau *Ethernet* a été rajouté.

L'ordinateur principal supervisant toute la chaîne d'acquisition communique alors avec le petit système, lui demande de saisir telle quantité de tel type de données (Americium ou Xénon) avant de lui préciser l'endroit où il peut les envoyer. En fait, il a le même pouvoir que l'utilisateur qui se trouve devant l'écran puisqu'il peut donner les mêmes ordres au programme d'acquisition[10].

Dans la pratique, deux versions de ce programe ont été développées. La première permet de travailler selon la méthode interactive décrite plus haut, et la manipulation des histogrammes y est autorisée. Les interruptions CAMAC sont générées par un pulseur qui est initialisé lors de la mise en route d'une prise de données de type Xénon. La seconde version ne contient pas la partie manipulation des histogrammes mais est capable de dialoguer au travers du canal de communication. De plus, lors d'une prise de données de type Xénon, les interruptions ne viennent plus du pulseur mais sont générées par la partie déclenchement du détecteur L3.

Pour faciliter le développement de ces programmes, toute une série de librairies a été créée selon le même principe que celui utilisé pour la librairie CAMAC. Ces librairies ont facilité la mise au point de plusieurs petits logiciels permettant, entre autre, de régler les alimentations haute-tension des lampes au Xénon ou de tester différentes composantes du système d'acquisition.

Pour terminer, et grâce à la modularité de la conception du système d'acquisition, un programme ressemblant beaucoup à la première des deux versions a été réalisé, la différence principale étant l'origine des données acquises. Dans le cas "normal" on se trouve dans un local climatisé à quinze mètres sous terre et les données proviennent de l'électronique (que l'on sauvera sur un fichier au plus vite), et dans le cas de la version "batarde", on est dans un bureau ensoleillé et les données proviennent du fichier sauvé dans le puits!

[10]On utilise le protocole TCP/IP pour le transport des données à travers *Ethernet* et par-dessus, un petit protocole privé pour le dialogue entre processus.

# Chapitre 6

# Conclusions

L'écriture d'un driver est un travail sans fin puisqu'il y a toujours matière à critiquer. Soit il est trop simple et ne remplit pas toutes les tâches que l'on attend de lui, soit il est très complet mais ses performances laissent à désirer, soit le système d'exploitation change et le driver ne fonctionne plus.

Or un driver est toujours l'élément le plus important dans la vie d'un périphérique. Combien y a-t-il de programmeurs qui ont tenté d'utiliser un périphérique en s'adressant directement à lui? Certes quelques fois cela peut paraître tentant, mais très vite il faut se rendre à l'évidence: sans driver, un périphérique est inutilisable.

Munie de son "conducteur", l'interface CAMAC peut aujourd'hui dignement affronter le "trafic électronique" et, par conséquent, survivre dans le monde de la physique. Tant que le driver existera et sera en état de contrôler les données qu'il manipule, l'interface sera sauve et les Amigas continueront à "hanter" les laboratoires...

## 6.1 Utilisation future

Le CAMAC est aujourd'hui considéré comme une "vieille" technologie. La bande passante d'une branche n'est plus aussi attractive qu'elle le fut au début des années 80. Les machines qui pilotaient le CAMAC il y a dix ans sont presque devenues des pièces de musée et les expériences en physique des particules sont devenues de plus en plus complexes, nécessitant une électronique beaucoup plus performante.

Cependant, une variété presque infinie de modules sont toujours disponibles dans les armoires des laboratoires et toutes les prises de données dites "lentes" sont toujours réalisables avec le CAMAC. De plus, le coût d'un Amiga haut de gamme étant peu élevé, rien ne s'oppose à ce que les anciens ordinateurs soient remplacés par leurs cadettes.

En agissant ainsi, on fait d'une pierre deux coups: on réduit à néant les frais de maintenance

occasionnés par les ordinateurs d'antan et on se sert d'un matériel tout neuf, capable d'utiliser le CAMAC à son plein régime.

Partant de ce point de vue, quelques laboratoires de recherche se sont intéressés à ce développement et ont tenté d'obtenir des exemplaires de l'interface et du driver. Citons entre autres le SLAC (Stanford Linear ACcelerator en Californie) qui vient d'acquérir 10 exemplaires de ce développement.

Quand à L3, on est encore loin d'en voir la fin. LEP phase deux est en cours de réalisation et le programme décrit dans le paragraphe 5.3 en fera partie intégrante, ce qui lui laisse encore quelques bonnes années de fonctionnement.

En fait, la mort du CAMAC est aussi improbable que la disparition du language Fortran. Dans le berceau de l'innovation technologique, on n'abandonne pas facilement les techniques qui ont rendu de bons et loyaux services pendant de nombreuses années!

## 6.2 Compléments

Bien que suffisamment performant quant aux besoins pour lesquels le driver du CAMAC a été conçu, certaines fonctionnalités manquent encore aujourd'hui. Si la nécessité venait à se faire sentir, il serait possible d'améliorer le driver des diverses façons suivantes:

- Seul un contrôleur de branche peut être géré par le driver. En fait, lors de la procédure d'autoconfiguration, le driver ne va pas vérifier si plusieurs contrôleurs sont présents dans l'espace d'adressage de la machine. Les fonctions d'Exec permettent cependant d'accéder à plusieurs unités d'un même périphérique, mais le driver refuse actuellement les appels spécifiant une branche autre que la première.

- Les interruptions ne sont traitées que d'une façon sommaire. En fait, il serait agréable de pouvoir donner un pointeur sur une fonction à exécuter par le serveur d'interruptions, profitant ainsi des avantages du temps-réel.

- Plusieurs processus peuvent accéder à tous les modules d'une branche sans aucune protection. Si le driver tenait une liste des "propriétaires" d'un module, cela permettrait d'éviter qu'un processus "étranger" vienne modifier l'état d'un module en utilisation.

- Les `CamacList` permettent d'exécuter des cycles, tous différents, préalablement définis. Or les données sont rangées de façon contiguës dans un tableau, alors qu'une indirection permettrait de les rediriger directement dans des variables.

- La librairie décrite dans le chapitre 4 devrait être écrite en assembleur et conçue de façon résidente et réentrante. Les avantages seraient multiples: tout d'abord, le gain de temps en exécution permettrait d'utiliser la librairie même dans les phases critiques, étant donné que les arguments seraient, dans ce cas, passés par registre et qu'un choix judicieux de ces derniers permettrait d'accéder au driver en quelques instructions.

Ensuite, il n'y aurait qu'une seule copie en mémoire et enfin, il ne serait pas nécessaire d'effectuer une édition de liens à chaque modification de la librairie.

## 6.3 Epilogue

Maintenant que le système d'acquisition fonctionne et que le CAMAC est devenu un périphérique correctement intégré dans le système d'exploitation de l'Amiga, que peut-on tirer comme conclusions?

Tout d'abord un extrême soulagement! La digestion d'un système d'exploitation multitâches et temps-réel n'est pas une mince affaire. En écrivant un driver, on est confronté à tous les problèmes qui peuvent surgir dans le pire des cas. On passera sous silence les nombreuses réinitialisation, les impressions de "quelque chose de bizarre", les interruptions ininterruptibles et autres coups du sort...

Comment tester un "semblant" de driver? C'est une question qu'aujourd'hui encore se posent ceux qui ont tenté de mettre au point ce type de code.

Mais dans ce capharnaüm, on arrive à s'y retrouver. Mieux encore, on y prend plaisir. Noyau, interruption, messagerie interne, *port* de communication, lien dynamique, point d'entrée, driver, tâche... Tous ces *mots* prennent un *sens*, s'emboitent, correspondent, sont logiquement dépendants et deviennent clairs.

Dans cet immense labyrinthe, on élabore l'esprit de synthèse nécessaire pour maîtriser le domaine dans lequel on évolue, la programmation système.

Bref, l'écriture d'un driver n'est pas une sinécure. L'apprentissage est long, puisqu'il faut s'immiscer dans l'intimité de la machine, mais gratifiant lorsqu'on se rend compte que l'on a appris beaucoup de choses et que ce n'est qu'en se confrontant aux difficultés que l'on découvre "comment ça fonctionne".

Une dernière petite remarque: si l'Amiga ne tient qu'une toute petite place dans le marché de la micro-informatique, ce n'est certes pas parce que son système d'exploitation est considéré comme sommaire, bien au contraire. En écrivant un driver pour Exec, on n'a effleuré qu'une infime partie de ses capacités. Si les professionnels de l'informatique "grand public" se donnaient la peine de consacrer du temps pour la maîtrise de l'Amiga, on aurait probablement des logiciels d'une surprenante qualité à disposition. A suivre...

# Bibliographie


[1] *LEP*, CERN, Service de relations publiques.

[2] *CAMAC Updated specifications*, Commission of the European Communities, Report EUR 8500 en, ISBN 92-825-3597-5

[3] *The construction of the L3 experiment*, North-Holland Physics Publishing, A289 (1990) 35-102.

[4] *The Xenon Monitor of the L3 Electromagnetic Calorimeter*, Université de Genève, UGVA-DPNC 1992/01-150.

[5] *The VME Workshop*, VME bus International Trade Association (VITA), Workshop basé sur IEEE 1014 (sans réf.).

[6] *Commodore Amiga A500/A2000 Technical Reference Manual*, Commodore Business Machines, Inc.,

[7] *Interface entre l'ordinateur Amiga 1000 et une branche Camac avec application au système de calibration du détecteur au BGO de l'expérience L3*, Luc Faravel, Université de Genève, DPNC.

[8] *Amiga Hardware reference manual*, Commodore-Amiga Inc., Addison-Wesley Publishing Inc., ISBN 0-201-18157-6

[9] *Amiga ROM Kernel reference manual: Libraries and devices*, Commodore-Amiga Inc., Addison-Wesley Publishing Inc., ISBN 0-201-18187-8

[10] *Amiga ROM Kernel reference manual: Includes and autodocs*, Commodore-Amiga Inc., Addison-Wesley Publishing Inc., ISBN 0-201-18177-0

[11] *Intuition, the Amiga User Interface*, Commodore-Amiga Inc., CBM Product Number 327267-01

[12] *Programmation du driver CAMAC, version 1.50*, Milan ZOFKA, Université de Genève, DPNC.