Interface Logicielle MPC/1
Manuel du Programmeur

version 3.0

Université Paris 6 - MASI
Alexandre Fenyö, Frédéric Potter, Pierre David

18 décembre 1996

Table des matières

Table des figures

Chapitre 1

Introduction

Ce manuel décrit les spécifications logicielles des interfaces noyau et mode utilisateur des couches de communication de la machine MPC.

1.1 Architecture matérielle

La machine MPC est constituée de trois composantes matérielles, présentées sur la figure 1.1 :

- Un réseau haut-débit composé d'un maillage de routeurs RCube. Chaque routeur possède huit liens HSL à 1 Gigabits/s ;
- Un réseau de contrôle de type Ethernet qui est utilisé pour configurer le système ;
- Un ensemble de micro-ordinateurs PC équipés du système d'exploitation Unix FreeBSD, ainsi que d'un bus PCI sur lequel sont reliées une carte Ethernet et une carte d'interface Fast-HSL vers le réseau HSL. Chaque carte contient un routeur RCube.

Un émulateur de la machine MPC est disponible : il propose toutes les couches spécifiées dans ce document, en émulant celles de plus bas niveau, les couches supérieures étant natives.

1.2 Architecture logicielle

Le mécanisme de communication supporté par la machine MPC est le passage de messages : chaque noeud de la machine possède son propre système

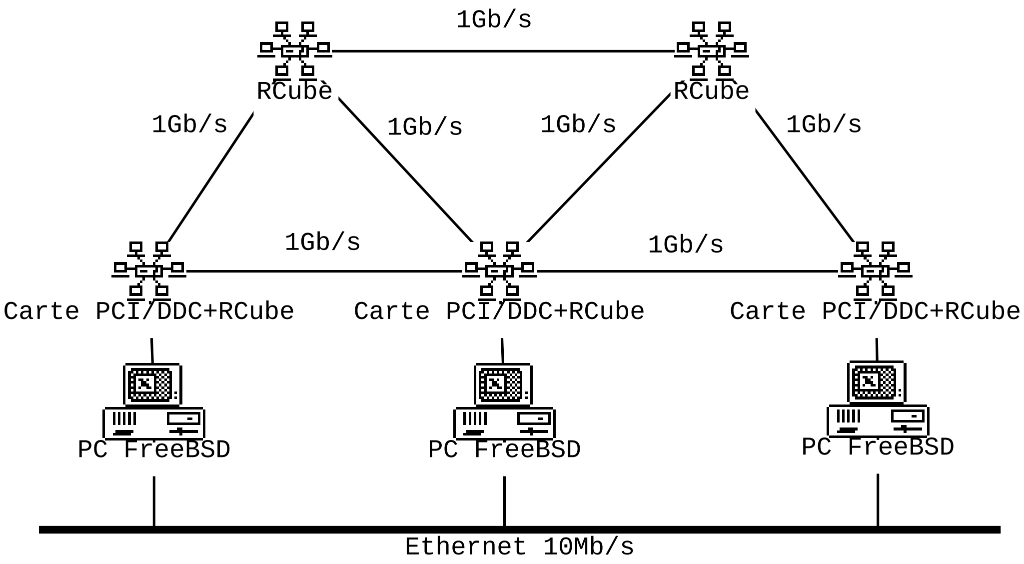

FIG. 1.1 - *Composantes matérielles de la machine MPC*

d'exploitation Unix FreeBSD. Le noyau de l'OS a été modifié pour mettre en oeuvre le protocole de communication DDSLR.

L'architecture logicielle des services de communication de la machine MPC est composée de modules noyau chargeables dynamiquement[1], de démons Unix, ainsi que d'une bibliothèque de programmation.

La figure 1.2 présente les différentes couches logicielles fournies :

- Le pilote (driver) HSL constitue la couche de plus bas niveau, il gère le pseudo-périphérique (pseudo-device) `/dev/hsl` et pilote le matériel ;

- Le pilote CMEM, accessible par le pseudo-périphérique `/dev/cmem`, est un gestionnaire de mémoire physique ; il est chargé au début de la phase d'amorçage de FreeBSD et se réserve une plage de mémoire physique contigüe, qu'il mappe dans l'espace d'adressage virtuel du noyau ; il est capable, par la suite, de fournir des morceaux de cette plage physique contigüe aux modules qui en feront la demande ;

- La couche PUT constitue le service de communication de plus bas niveau, et fournit la fonction *remote write*, de l'espace d'adressage physique local vers l'espace physique d'un noeud distant ;

1. loadable kernel module

- La couche SLR/P fournit un service d'échange de zones de mémoire physique, entre noeuds, sur des canaux virtuels ;
- La couche SLR/V fournit un service équivalent à la couche SLR/P, mais en manipulant des zones localisées dans des espaces de mémoire virtuelle ;
- La bibliothèque USERACCESS permet d'accéder à la fonctionnalité *remote write* de la couche PUT, depuis un processus en mode utilisateur ;
- Sur chaque PC, deux démons Unix sont actifs. Ils dialoguent par le réseau de contrôle et se chargent de la configuration des couches logicielles du noyau des différents noeuds.

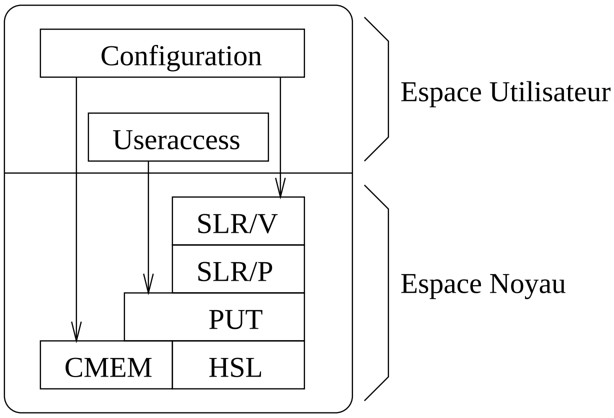

FIG. 1.2 - *Architecture logicielle*

Chapitre 2

Services logiciels

Ce chapitre décrit les interfaces d'accès aux différents services de communication de la machine MPC.

2.1 Les différents niveaux d'accès au logiciel MPC

On distingue quatre niveaux d'accès aux couches logicielles MPC, détaillées sur la figure 2.1 :

- Accès depuis le noyau à SLR/V ;
- Accès depuis le noyau à SLR/P ;
- Accès depuis le noyau à PUT ;
- Accès depuis un processus à PUT.

Avec les différents services logiciels décrits par la suite, la boucle locale n'est pas permise : un noeud ne doit pas s'adresser à lui-même par les services MPC. D'autres services Unix existent déjà pour ce type de communication, comme les tubes nommés, les sockets, ou les IPC System V (Inter Process Communication).

2.2 Les modules présents sur un noeud

La figure 2.2 présente les relations entre les différents modules et démons.

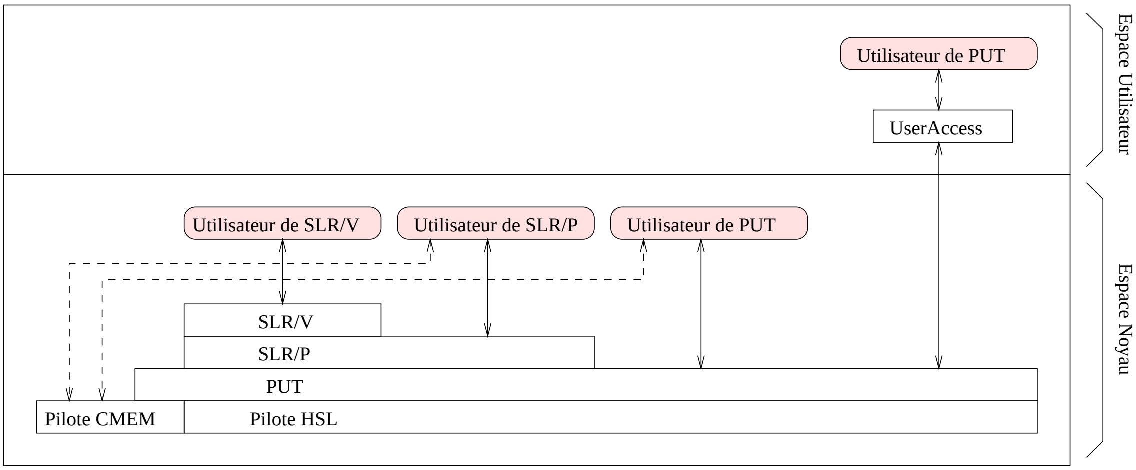

FIG. 2.1 - *Les différents niveaux d'utilisation des couches logicielles.*

2.3 PUT

PUT constitue l'interface la plus proche du matériel.

2.3.1 La liste des pages à émettre et les MI

L'interface PUT permet d'accéder aux fonctionnalités bas-niveau du composant PCI-DDC[1] qui équipe la carte réseau HSL. Ce dernier est capable de transporter des messages entre noeuds. Chaque message est composé d'un ensemble de pages. Un page correspond à une zone contigüe dans la mémoire physique du système. Pour ne pas les confondre avec la notion de page telle qu'elle est définie dans Unix, nous les appellerons pages réseau. On associe à chaque message un numéro de message (Message Identifier), codé sur 24 bits, que nous désignerons par la suite par MI. Cela permet au composant d'identifier les pages qui composent un même message.

Le protocole de communication impose l'unicité du MI : un seul message d'un MI donné, à destination d'un noeud donné, parcourt le réseau à un même instant.

Le composant PCI-DDC accède en DMA à la mémoire : il peut donc

1. Une description complète des fonctionnalités de ce composant et des concepts qu'il met en jeu est disponible dans le document *PCI-DDC Specifications*, accessible à l'URL `http://humanus.ibp.fr/doc/DDC_spec.ps`.

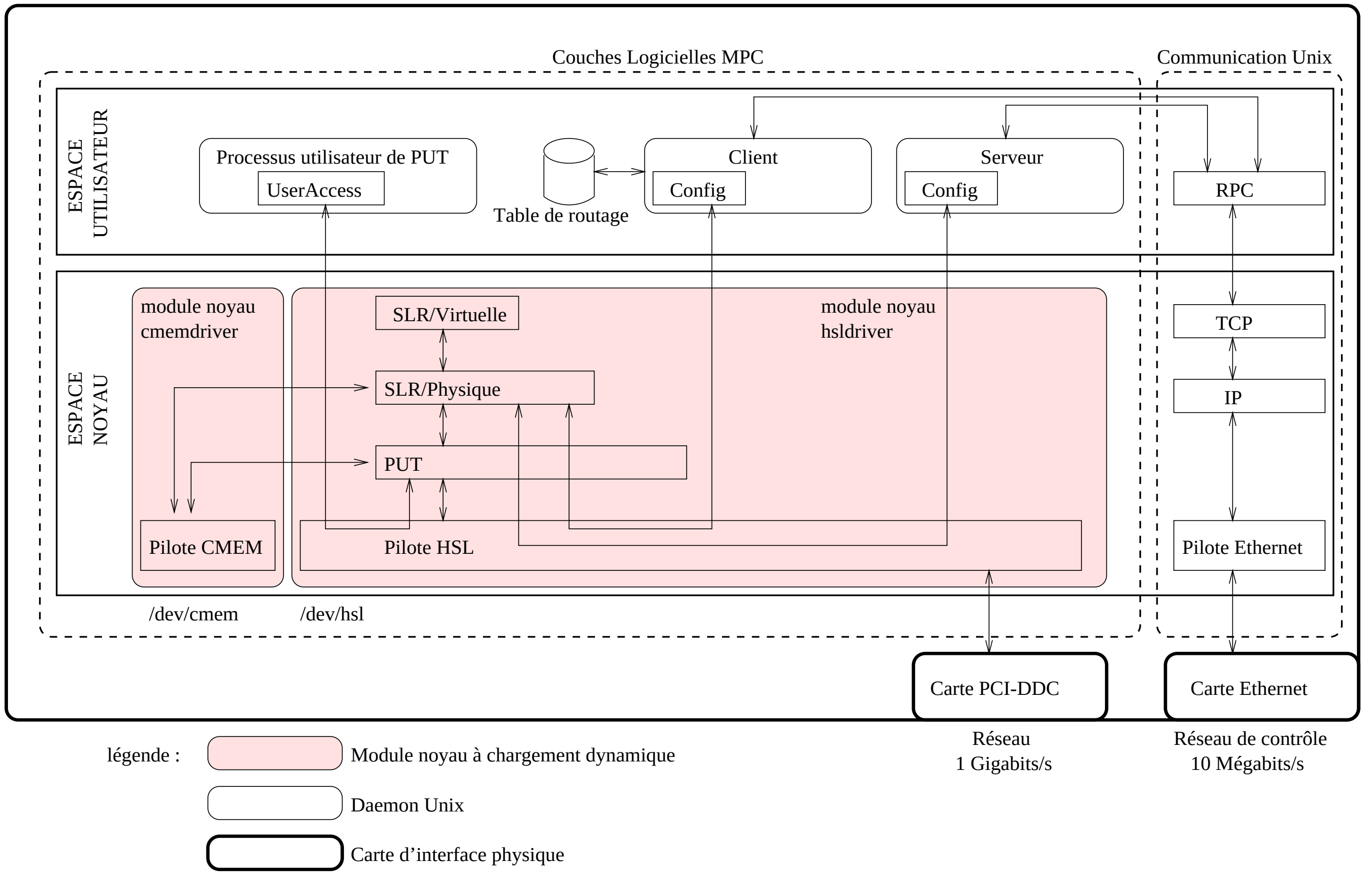

Fig. 2.2 - *Interactions entre modules sur la machine MPC*

acceder à la LPE (Listes de Pages à Emettre). Chaque entrée de la LPE définit une page réseau. Les pages d'un même message doivent être décrites par des entrées de LPE contigües.

Une entrée de LPE est constituée de cinq champs :

- Le champ `routing_part` : il indique le noeud destinataire ;
- Le champ `page_length` : il indique le nombre d'octets contenues dans la page ; remarquons qu'aucun alignement particulier n'est imposé ;
- Le champ `PLSA` (Page Local Start Address) : il indique l'adresse physique où commencent les données dans la mémoire locale ;
- Le champ `PRSA` (Page Remote Start Address) : il indique l'adresse physique dans le noeud distant où les données de la mémoire locale doivent être déposées ;
- Le champ `control` : il est composé de deux parties, un ensemble de drapeaux et le MI associé au message dont cette page fait partie.

Le type `lpe_entry_t`, défini dans `put.h`, décrit les entrées de LPE :

```
typedef struct _lpe_entry {
  u_short routing_part;
  u_short page_length;
  u_long control;
  caddr_t PRSA, PLSA;
} lpe_entry_t;
```

Les bits de poids fort du champ `control` permettent à l'utilisateur de la couche PUT de demander au composant un certain nombre d'actions :

- NOR (Notify Once Received) : le composant PCI-DDC du noeud distant lève une interruption en réception dès que le message contenant cette page est totalement arrivé (s'il est utilisé, ce bit doit être positionné sur l'ensemble des pages d'un même message) ;
- NOS (Notify Once Sent) : le composant PCI-DDC local lève une interruption une fois la page émise sur le réseau ;
- CM (Composed Message) : message composé ;
- LMP (Last Message Page) : indique que la page réseau décrite dans cette entrée de LPE est la dernière du message ;

– LRM (LRM Enable) : informe le composant PCI-DDC qu'il doit utiliser sa table LRM (Liste des Messages Reçus) pour gérer les réceptions (cas d'un réseau adaptatif) ;

– LMI (LMI Enable) : informe le composant PCI-DDC qu'il doit utiliser sa table LMI (Liste des MI) pour gérer les réceptions de messages. Sans table LMI, le composant se bloque après chaque réception d'un message, en attendant un acquittement de prise en compte de réception par le logiciel. Avec une LMI, le composant peut recevoir plusieurs messages sans attendre un acquittement du logiciel.

– SM (Short Message) : message court ;

La description des principes sur lesquels s'appuient les messages courts et les messages composés sont décrits dans la spécification matérielle du composant PCI-DDC (PCI-DDC Specifications, version 1.3a).

Ces drapeaux, ainsi que la partie MI du champ `control`, sont accessibles par les opérateurs suivants, définis dans `data.h` :

```
#define NOR_ISSET(control)  ((control) & NOR_MASK  ? 1 : 0)
#define SM_ISSET(control)   ((control) & SM_MASK   ? 1 : 0)
#define LMP_ISSET(control)  ((control) & LMP_MASK  ? 1 : 0)
#define CM_ISSET(control)   ((control) & CM_MASK   ? 1 : 0)
#define LRM_ISSET(control)  ((control) & LRM_MASK  ? 1 : 0)
#define LMI_ISSET(control)  ((control) & LMI_MASK  ? 1 : 0)
#define NOS_ISSET(control)  ((control) & NOS_MASK  ? 1 : 0)
#define MI_PART(control)    ((mi_t) ((control) & MI_MASK))
```

Le champ `control`, défini sur 32 bits, contient le numéro de MI dans ses 24 bits de poids faible. Ils ne sont pas tous obligatoirement utilisés, le matériel autant que le logiciel pouvant être amenés à reserver moins de mémoire en diminuant le nombre de bits significatifs dans la partie MI. La constante `MI_RANGE`, définie dans `data.h`, indique le nombre de bits significatifs de la partie MI.

2.3.2 Attribution d'une plage de mémoire contigüe

L'action du composant PCI-DDC est commandé par le contenu des entrées de LPE. Ces entrées décrivent des plages de mémoire physique, l'utilisateur des couches basses de communication est donc souvent amené à vouloir se faire attribuer de la mémoire contigüe, afin de faciliter les échanges.

Le gestionnaire de mémoire CMEMDRIVER gère par défaut 1 Mo de mémoire contigüe, lockée. Cette valeur peut être modifiée : elle est stockée dans la constante CMEM_SIZE dans cmem.h. Cette plage est découpée en slots attribués à la demande. Dans l'implémentation actuelle, il peut y avoir jusqu'à 4 slots. Notons que 2 slots sont réservés par PUT et SLR/P. Les différentes utilisations éventuelles de slots de CMEMDRIVER sont décrites sur la figure 2.3.

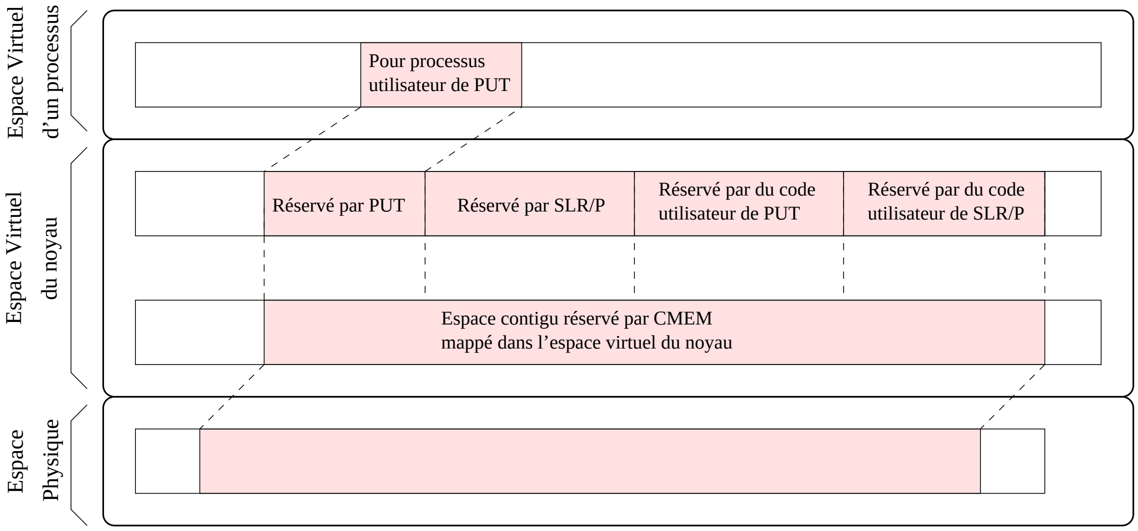

FIG. 2.3 - *Les slots de* CMEMDRIVER

La fonction cmem_getmem() permet une demande d'attribution d'un slot.

```
/*
 ************************************************************
 * cmem_getmem(size, name) : get a slot of contig. memory.
 * size : the size of the slot.
 * name : a string to identify the user of the slot.
 *
 * return values : -1 on error.
 *                 the slot number on success.
 ************************************************************
 */

int
cmem_getmem(size, name)
            vm_size_t size;
            char *name;
```

Pour connaître les adresses physique et virtuelle de début de slot, il faut utiliser cmem_virtaddr() et cmem_physaddr(). CMEMDRIVER garantit que le slot commence sur une frontière de pages Unix.

```
vm_offset_t
cmem_virtaddr(int slot)

caddr_t
cmem_physaddr(int slot)
```

La libération d'un slot se fait à l'aide de `cmem_releasemem()` :

```
/*
 **********************************************************
 * cmem_releasemem(slot) : release a slot of contig. memory.
 * id : the slot number.
 *
 * return value : -1 on error.
 *                 0 on success.
 **********************************************************
 */

int
cmem_releasemem(slot)
                int slot;
```

2.3.3 PUT mode noyau

Les fonctions décrites ici sont déclarées dans `put.h`. L'utilisation en mode noyau de la couche PUT consiste à procéder en suivant 4 étapes : accès au numéro de noeud local, demande d'attribution d'une plage de mémoire contigüe auprès de CMEMDRIVER, enregistrement en tant qu'utilisateur de la couche PUT, demande d'attribution d'une plage de MI, ajout d'entrées à la LPE.

Accès au numéro de noeud local

La fonction `put_get_node()` permet à un utilisateur d'une quelconque couche (PUT, SLR/P ou SLR/V) de connaître le numéro de noeud affecté à une carte Fast-HSL. Le paramètre `minor`, indique le numéro de la carte Fast-HSL concernée (il peut y en avoir plusieurs pour une même carte processeur).

```
/*
 ***********************************************************
 * put_get_node(minor) : get the node affected to a board.
 * minor : board number.
 *
 * return values : node number on success.
 *                 -1 on error (board not initialized).
 ***********************************************************
 */
int
put_get_node(minor)
            int minor;
```

Enregistrement en tant qu'utilisateur de la couche PUT

Pour pouvoir utiliser la couche PUT, il faut s'enregistrer. En effet, plusieurs applications utilisant cette couche peuvent fonctionner simultanément sur la machine MPC, il faut donc pouvoir les différencier, afin de leur distribuer des plages de MI disjointes et de déterminer laquelle des applications prévenir à chaque interruption.

Lors de son enregistrement, l'application fournit donc deux fonctions qui seront appelées par les routines de traitement d'interruption de PUT : l'une en cas de données venant de quitter le noeud, l'autre en cas de données venant d'entrer dans le noeud. Elles prennent en premier paramètre le numéro de la carte réseau HSL qui a généré l'interruption ; le numéro de MI correspondant leur est aussi fourni. La fonction d'interruption pour des données ayant quitté le système reçoit, en plus, l'entrée de LPE correspondante.

La fonction `put_register_SAP()` [2] permet de s'enregistrer :

```
/*
 ***********************************************************
 * put_register_SAP(minor, send, received) : register a user of
 * the PUT interface for one board.
 * minor    : the board number.
 * sent     : the interrupt procedure for data sent.
 * received : the interrupt procedure for data received.
 *
 * return values : -1 on error.
 *                 the registration number for this SAP on success.
 ***********************************************************
 */

int
put_register_SAP(minor, sent, received)
                 int minor;
                 void (*sent) __P((int, mi_t, lpe_entry_t));
                 void (*received) __P((int, mi_t));
```

2. SAP: Service Access Point

Les deux paramètres `sent` et `received` sont des fonctions fournies par l'utilisateur, pour qu'il soit prévenu en cas d'interruption. La première reçoit en paramètres le numéro de la carte qui a généré l'interruption, le MI correspondant et dans le cas de `sent` l'entrée de LPE correspondante.

La fonction `put_unregister_SAP()` permet de terminer un enregistrement :

```
/*
 ***********************************************************
 * put_unregister_SAP(minor, id) : unregister a user of the PUT
 * interface.
 * minor : the board number.
 * id    : the registration number for the SAP.
 *
 * return values : ENXIO  : not enough structures allocated at
 *                          compilation time to handle this board.
 *                 ENOENT : this board is not initialized.
 *                 EBADF  : id out of range.
 ***********************************************************
 */

int
put_unregister_SAP(minor, id)
                int minor, id;
```

demande d'attribution de MI

Plusieurs utilisateurs de la couche PUT peuvent travailler simultanément. Il faut donc un mécanisme pour éviter les conflits lors de l'attribution de plages de MI. La fonction `put_attach_mi_range()` joue ce rôle. Le paramètre `range` doit être une puissance de 2. La plage allouée commencera à un multiple entier de `range`, afin de faciliter la gestion des MI de l'utilisateur.

```
/*
 ***********************************************************
 * put_attach_mi_range(minor, sap, range) : ask for the attribution
 * of a range of MI.
 * minor : the board number.
 * sap   : the registration number for this SAP.
 * range : the size of the range.
 *
 * Note that the size MUST be a power of 2. A test of this requirement
 * should one day be included in this function.
 * The first MI of the allocated range will be a multiple of the range
 * to simplify the work of the user of this interfacae.
 *
 * return values : 0      : success.
 *                 ENXIO  : not enough structures allocated at
 *                          compilation time to handle this board.
 *                 ENOENT : this board is not initialized.
 *                 EBUSY  : this SAP already has an allocated set
 *                          of MI.
 *                 E2BIG  : range is too big to be allocated, it should
 *                          be reduced or a new compilation of the
 *                          PUT interface should be required after
 *                          having increased the constant MI_SIZE.
 *                 ENOMEM : not enough MI left.
 ***********************************************************
 */

int
put_attach_mi_range(minor, sap, range)
                 int minor;
                 int sap;
                 u_long range;
```

Pour connaître le début de la plage de MI attribuée lors de l'opération précédente, on utilise `put_get_mi_start()` :

```
/*
 ***********************************************************
 * put_get_mi_start(minor, sap) : get the first mi allocated
 * for this board.
 * minor : the board.
 * sap   : the registration number affected to the user of this
 *         interface.
 *
 * return values : the required MI.
 ***********************************************************
 */

mi_t
put_get_mi_start(minor, sap)
                 int minor;
                 int sap;
```

Ajout d'entrées à la LPE

Pour ajouter une description de page réseau à la LPE, il faut utiliser la fonction `put_add_entry()`. Contrairement aux autres fonctions des différentes couches, celle-ci peut être appelée même lors d'un traitement d'interruption.

```
/*
 ************************************************************
 * put_add_entry(minor, entry) : add an entry to the LPE.
 * minor : board number.
 * entry : entry to add.
 *
 * return values : 0      : success.
 *                 ENXIO  : not enough structures allocated at
 *                          compilation time to handle this board.
 *                 ENOENT : this board is not initialized.
 *                 EAGAIN : can't add entry because the LPE is full.
 ************************************************************
 */

int
put_add_entry(minor, entry)
              int minor;
              lpe_entry_t *entry;
```

Etat de la LPE

On peut consulter l'état de la LPE avec `put_get_lpe_free()`.

Lorsqu'on envoie un message composé de plusieurs pages, ces dernières doivent être décrites par des entrées successives de la LPE.

Pour cela, il faut masquer les interruptions avec `splhigh()` (pour empêcher l'utilisation de `put_add_entry()` qui est succeptible d'être appelée en interruption), vérifier que la LPE contient suffisamment d'entrées libres (avec `put_get_lpe_free()`), et si ce n'est pas le cas, s'endormir (avec `tsleep()`) sur l'évènement `hslfreelpe_ident`. Cet évènement est activé par l'interface PUT quand une entrée se libère dans la LPE.

```
/*
 ***********************************************************
 * put_get_lpe_free(minor) : the number of free entries in the LPE.
 * minor : the board number.
 *
 * return values : the number required.
 ***********************************************************
 */

long
put_get_lpe_free(minor)
                 int minor;
```

2.3.4 PUT mode utilisateur : USRPUT

La bibliothèque USERACCESS permet l'accès à l'interface PUT en mode utilisateur. Elle est destinée aux applications qui prennent le contrôle complet de la machine, et qui désirent effectuer des écritures directes en mémoire physique d'un noeud distant. L'objectif est de s'affranchir du surcoût inhérent aux passages en mode noyau et aux déroutements en cas d'interruption.

Cette bibliothèque s'adresse donc à une classe d'applications bien particulière : il n'y a pas de gestion d'interruptions levées par la carte réseau HSL, seules des techniques de *polling* pouvant permettre à l'application de réagir à la réception de données. L'application doit donc être seule sur la machine afin que le *polling* ne pénalise pas d'autres processus.

L'ensemble des fonctions qui suivent est disponible dans la bibliothèque `libuseraccess.a` ; elles sont déclarées dans `useraccess.h`.

Avant d'utiliser la couche USERACCESS, il faut appeler `usrput_init()`, et en fin d'utilisation `usrput_end()`. Ces fonctions retournent -1 en cas d'erreur, 0 en cas de succès.

```
int
usrput_init()

int
usrput_end()
```

Une plage de mémoire physique contigüe est réservée auprès du module CMEM, à destination de la bibliothèque USERACCESS, lors du chargement du module HSLDRIVER. Cette plage est initialement « mappée » dans l'espace virtuel du noyau. La fonction `usrput_get_contig_mem()` permet de « remapper » cette zone dans l'espage virtuel du processus utilisateur. Elle retourne NULL en cas d'erreur.

```
opt_contig_mem_t *
usrput_get_contig_mem(size)
                        size_t size;
```

La structure `opt_contig_mem_t`, retournée par `usrput_get_contig_mem()` en cas de succès, est définie dans `driver.h` :

```
typedef struct _opt_contig_mem {
  caddr_t ptr;
  size_t  size;
  caddr_t phys;
} opt_contig_mem_t;
```

Elle offre les informations suivantes : les adresses physique et virtuelle de la mémoire contigüe qui vient d'être mappée dans l'espace du processus, ainsi que la taille réellement allouée (cette taille peut être différente de celle demandée). Il est possible de modifier la taille de la zone mémoire réservée à une application utilisatrice de la bibliothèque USRPUT, en modifiant la variable `OPTIONAL_CONTIG_RAM_SIZE`, dans `put.h`.

La fonction `usrput_get_node()` permet de connaître le numéro de noeud local. Elle retourne -1 en cas d'erreur, le numéro de noeud en cas de succès :

```
int
usrput_get_node()
```

Enfin, la fonction `usrput_add_entry()` permet d'ajouter des entrées dans la LPE. Elle retourne 0 en cas de succès. En cas d'erreur, elle retourne -1, et positionne la variable externe `errno`. Les codes d'erreur possibles sont : ENOENT (la carte n'est pas initialisée) et EAGAIN (la LPE est pleine, il faut réessayer plus tard).

```
int
usrput_add_entry(entry)
lpe_entry_t *entry;
```

Rappelons que lorsque cette fonction retourne -1, et que `errno` vaut EAGAIN, il faut reprendre l'opération après un temps d'attente.

2.4 SLR/P

La couche SLR/P fournit un service de transfert de données préalablement « lockées » en mémoire, et adressées en adresses physiques. C'est à l'utilisateur de cette couche de garantir que les données sont « lockées » en mémoire. La communication se fait à travers des canaux virtuels. Un noeud peut utiliser un même numéro de canal quand il s'adresse à deux autres noeuds distincts. Le numéro de canal utilisé par deux noeuds qui communiquent via SLR/P doit être le même.

L'utilisateur de SLR/P peut aussi profiter des services de CMEMDRIVER, afin d'allouer des zones physiques contigües (cf. section 2.3.2).

Deux opérations sont définies : SEND sur le noeud émetteur et RECEIVE sur le noeud récepteur. SEND est accessible par la fonction `slrpp_send()`, et RECEIVE l'est par la fonction `slrpp_recv()`. L'opération RECEIVE consiste à envoyer au noeud émetteur la description physique des pages locales « lockées », prêtes à accueillir des données. L'opération SEND consiste à émettre par l'intermédiaire de PUT les pages locales, à l'aide de la description des pages distantes. Ainsi, l'opération SEND ne génère aucun trafic réseau tant qu'un RECEIVE n'a pas été fait sur le canal correspondant.

Ces opérations sont non-bloquantes et ne doivent pas être utilisées en interruption, car elles font usage de la primitive noyau `tsleep()`.

La primitive `tsleep()` est utilisée lorsque des ressources vitales sont indisponibles, par exemple quand des tables sont pleines.

Les fonctions étant non bloquantes, l'utilisateur a besoin de savoir quand les données ont été réellement envoyées, afin, par exemple, de pouvoir délocker les zones en question. Pour cela, chacune de ces fonctions demande un pointeur vers un fonction, dite fonction de callback, ainsi qu'un entier appelé paramètre de callback. Suite à l'opération SEND, lorsque les données ont quitté le noeud et sont entrées dans le réseau, la fonction de callback est appelée avec le paramètre de callback et le nombre d'octets émis. De même, suite à l'opération RECEIVE, lorsque les données ont été reçues, une interruption est générée par la carte réseau HSL, et la fonction de callback est appelée avec le paramètre de callback.

Les fonctions `slrpp_send()` et `slrpp_recv()` ont besoin d'une description physique de la zone de mémoire locale allouée par le processus utilisateur pour la réception ou l'émission. Pour cela, le processus fournit un tableau de structures de type `pagedescr_t`. Le champ `addr` désigne donc une adresse physique de début de zone mémoire :

```
typedef struct _pagedescr {
  caddr_t addr;
  size_t  size;
} pagedescr_t;
```

Les fonctions `slrpp_send()` et `slrpp_receive()` prennent en argument la longueur des données à émettre ou recevoir. Si les longueurs ne sont pas identiques, c'est la plus petite de ces deux valeurs qui conditionne la taille des données réellement transmises. Dans ce cas, les données non transmises doivent faire l'objet d'un nouveau couple d'opérations SEND et RECEIVE pour compléter le transfert : en effet, sur un canal donné, à un RECEIVE correspond un et un seul SEND, même si les données n'ont pas pu être entièrement envoyées.

La fonction de callback de l'émetteur reçoit en paramètre le nombre d'octets réellement émis. La fonction de callback du récepteur ne reçoit pas cette information car elle n'est pas fournie par le composant PCI-DDC.

Le processus utilisateur qui désire fournir cette information au processus récepteur peut utiliser le mécanisme suivant :

- Le processus émetteur envoie M octets. Les quatre premiers octets forment un entier qui contient le nombre M.
- Le processus récepteur fournit un tampon de N octets ($N > 4$).
- Lorsque les données sont envoyées, le processus émetteur reçoit en paramètre de sa fonction de callback le nombre d'octets effectivement envoyés : $min(N, M)$.
- Lorsque les données ont été reçues, la fonction de callback du processus récepteur est appellé. Elle lit les 4 premiers octets dans le tampon de réception et en déduit l'entier M. Elle calcule alors $min(N, M)$, c'est-à-dire le nombre d'entier effectivement reçus.

Notons que les quatre premiers octets des tampons d'émissions et de réception ne sont pas forcément contigüs avec les données qui suivent car les fonctions d'émission et de réception définissent un message par un ensemble de zones mémoires éventuellement discontinues.

Le prototype de la fonction `slrpp_send()` est le suivant :

```
/*
 ************************************************************
 * slrpp_send(dest, channel, pages, size, fct, param, proc) :
 * try to send data to a remote node on a specific channel.
 * dest    : remote node.
 * channel : channel identifier.
 * pages   : description of pages to send.
 * size    : number of pages to send.
 * fct     : callback function called when data have been correctly
 *           sent.
 * param   : parameter provided to the callback function when called.
 * proc    : proc structure of the calling process.
 *
 * If the data cannot be sent, a timeout() is generated and
 * the kernel will call ddslrp_timeout() until this procedure
 * can effectively send the data.
 *
 * return values : 0          : success.
 *                 ENXIO      : SLR/P not initialized or not enough
 *                              structures allocated at compilation
 *                              time to handle this board.
 *                 EINVAL     : size is NULL.
 *                 EWOULDBLOCK : tsleep() returned EWOULDBLOCK.
 *                 EINTR      : tsleep() returned EINTR.
 *                 ERESTART   : tsleep() returned ERESTART.
 *                 EFAULT     : should not occur, unrecoverable error,
 *                              contact the author if it happens.
 *                 ENOENT     : this board is not initialized.
 ************************************************************
 */

int
slrpp_send(dest, channel, pages, size, fct, param, proc)
           node_t dest;
           channel_t channel;
           pagedescr_t *pages;
           size_t size;
           void (*fct) __P((caddr_t, int));
           caddr_t param;
           struct proc *proc;
```

Le prototype de la fonction `slrpp_recv()` est le suivant :

```
/*
 ***********************************************************
 * slrpp_recv(dest, channel, pages, size, fct, param, proc) :
 * ask to receive data from a remote node on a specific channel.
 * dest    : remote node.
 * channel : channel identifier.
 * pages   : description of pages ready to receive data.
 * size    : number of pages ready to receive data.
 * fct     : callback function called when data have been correctly
 *           received.
 * param   : parameter provided to the callback function when called.
 * proc    : proc structure of the calling process.
 *
 * return values : 0           : success.
 *                 ENXIO       : SLR/P not initialized.
 *                 EINVAL      : size is NULL or we're the remote node
 *                               (local loop not allowed).
 *                 EWOULDBLOCK : tsleep() returned EWOULDBLOCK.
 *                 EINTR       : tsleep() returned EINTR.
 *                 ERESTART    : tsleep() returned ERESTART.
 *                 ENXIO       : not enough structures allocated at
 *                             : compilation time to handle this board.
 *                 ENOENT      : this board is not initialized.
 ***********************************************************
 */

int
slrpp_recv(dest, channel, pages, size, fct, param, proc)
          node_t dest;
          channel_t channel;
          pagedescr_t *pages;
          size_t size;
          void (*fct) __P((caddr_t));
          caddr_t param;
          struct proc *proc;
```

Lorsque la communication sur un canal est terminée, il faut le réinitialiser avant de l'utiliser à nouveau. La fonction `ddslrpp_reset_channel()`, qui accomplit ce travail, prend en argument le nom d'un noeud distant, ainsi qu'un numéro de canal. Elle va donc réinitialiser la communication sur ce canal, uniquement avec le noeud distant indiqué.

```
/*
 ***********************************************************
 * ddslrpp_reset_channel(node, channel) : reset the sequence numbers
 * on a channel with a node.
 * node    : remote node.
 * channel : channel identifier.
 ***********************************************************
 */

void
ddslrpp_reset_channel(node, channel)
                      channel_t channel;
                      node_t    node;
```

2.5 SLR/V

La couche SLR/V permet de fournir un niveau d'abstraction supplémentaire. Elle fournit les mêmes services que la couche SLR/P, à la différence près que les zones mémoire sont adressées en adresses virtuelles, et non plus en adresses physiques.

Avant d'effectuer des opérations de communication avec cette couche, il faut s'assurer que les zones de mémoire passées en arguments des fonctions de SLR/V sont bien « lockées ».

Pour ce faire, la fonction `slrpv_mlock()` est disponible :

```
/*
 ***********************************************************
 * slrpv_mlock(lock_addr, lock_len, proc) : locks into memory
 * the physical pages associated with the virtual address range
 * starting at lock_addr for lock_len bytes.
 * lock_addr : beginning of the range.
 * lock_len  : length of the range.
 * proc      : proc structure of the calling process.
 *
 * Note that the lock_addr parameter should be aligned to a
 * multiple of the page size.
 *
 * return values : 0      : success.
 *                 EINVAL : The address given is not page aligned
 *                          or the length is negative.
 *                 EAGAIN : Locking the indicated range would
 *                          exceed either the system or
 *                          per-process limit for locked memory.
 *                 ENOMEM : Some portion of the indicated address
 *                          range is not allocated.
 *                          There was an error faulting/mapping
 *                          a page.
 ***********************************************************
 */

int
slrpv_mlock(lock_addr, lock_len, proc)
            vm_offset_t lock_addr;
            vm_size_t   lock_len;
            struct proc *proc;
```

Pour délocker des données, il faut utiliser `slrpv_munlock()` :

```
/*
 ***********************************************************
 * slrpv_munlock(lock_addr, lock_len, proc) : locks into memory
 * the physical pages associated with the virtual address range
 * starting at lock_addr for lock_len bytes.
 * lock_addr : beginning of the range.
 * lock_len  : length of the range.
 * proc      : proc structure of the calling process.
 *
 * Note that the lock_addr parameter should be aligned to a
 * multiple of the page size.
 *
 * return values : 0      : success.
 *                 EINVAL : The address given is not page aligned
 *                          or the length is negative.
 *                 ENOMEM : Some portion of the indicated address
 *                          range is not allocated.
 *                          Some portion of the indicated address
 *                          range is not locked.
 ***********************************************************
 */

int
slrpv_munlock(lock_addr, lock_len, proc)
              vm_offset_t lock_addr;
              vm_size_t   lock_len;
              struct proc *proc;
```

La fonction d'émission, `slrpv_send()`, est très proche de son homologue de la couche SLR/P :

```
/*
 ************************************************************
 * slrpv_send(dest, channel, pages, size, fct, param, proc) :
 * try to send data to a remote node on a specific channel.
 * dest    : remote node.
 * channel : channel identifier.
 * pages   : virtual address of the beginning of the data.
 * size    : size of data.
 * fct     : callback function called when data have been correctly
 *           sent.
 * param   : parameter provided to the callback function when called.
 * proc    : proc structure of the calling process.
 *
 * If the data cannot be sent, a timeout() is generated and
 * the kernel will call ddslrp_timeout() until this procedure
 * can effectively send the data.
 *
 * return values : 0           : success.
 *                 E2BIG       : an internal array used to handle
 *                               a description of the physical pages
 *                               composing the area the user wants to send
 *                               is too small (the area is made of
 *                               too many uncontig. groups of pages).
 *                 ENXIO       : SLR/P not initialized or not enough
 *                               structures allocated at compilation
 *                               time to handle this board.
 *                 EINVAL      : size is NULL.
 *                 EWOULDBLOCK : tsleep() returned EWOULDBLOCK.
 *                 EINTR       : tsleep() returned EINTR.
 *                 ERESTART    : tsleep() returned ERESTART.
 *                 EFAULT      : should not occur, unrecoverable error,
 *                               contact the author if it happens.
 *                 ENOENT      : this board is not initialized.
 ************************************************************
 */

int
slrpv_send(dest, channel, pages, size, fct, param, proc)
           node_t dest;
           channel_t channel;
           caddr_t pages;
           size_t size;
           void (*fct) __P((caddr_t, int));
           caddr_t param;
           struct proc *proc;
```

Le prototype de la fonction de réception est le suivant :

```
/*
 ***********************************************************
 * slrpv_recv(dest, channel, pages, size, fct, param, proc) :
 * ask to receive data from a remote node on a specific channel.
 * dest    : remote node.
 * channel : channel identifier.
 * pages   : virtual address of the beginning of the area ready to receive
 *           data.
 * size    : size of the area.
 * fct     : callback function called when data have been correctly
 *           received.
 * param   : parameter provided to the callback function when called.
 * proc    : proc structure of the calling process.
 *
 * return values : 0           : success.
 *                 E2BIG       : an internal array used to handle
 *                               a description of the physical pages
 *                               composing the area the user wants to send
 *                               is too small (the area is made of
 *                               too many uncontig. groups of pages).
 *                 ENXIO       : SLR/P not initialized.
 *                 EINVAL      : size is NULL or we're the remote node
 *                               (local loop not allowed).
 *                 EWOULDBLOCK : tsleep() returned EWOULDBLOCK.
 *                 EINTR       : tsleep() returned EINTR.
 *                 ERESTART    : tsleep() returned ERESTART.
 *                 ENXIO       : not enough structures allocated at
 *                             :  compilation time to handle this board.
 *                 ENOENT      : this board is not initialized.
 ***********************************************************
 */

int
slrpv_recv(dest, channel, pages, size, fct, param, proc)
           node_t dest;
           channel_t channel;
           caddr_t pages;
           size_t size;
           void (*fct) __P((caddr_t));
           caddr_t param;
           struct proc *proc;
```

Pour permettre la transmission d'une information supplémentaire, par exemple la taille des données envoyées, deux fonctions d'émission et de réception sont mises à disposition. Il s'agit de `slrpv_send_piggy_back()` et `slrpv_recv_piggy_back()`. Elle prennent non plus une, mais deux zones mémoire en argument, qu'elles émettent en une seule et même opération. Nous nommerons piggy-back cette zone supplémentaire. On peut ainsi résoudre le problème des envois et réceptions de taille différente. Nous avons vu, dans la section concernant SLR/P, que l'émetteur peut connaître la taille des données réellement envoyées, à l'aide d'un paramètre de la fonction de callback. Par contre, c'est à l'utilisateur d'implémenter un mécanisme pour que le récepteur puisse connaître aussi cette valeur. L'émetteur peut, par exemple, utiliser le piggy-back pour envoyer la taille des données qu'il a demandées à émettre. Le récepteur n'a plus qu'à calculer le minimum entre la taille indiquée dans le piggy-back et la taille des données qu'il a lui-même demandées à recevoir, pour obtenir la taille des données qui viennent d'être transmises.

Voici le prototype de `slrpv_send_piggy_back()` :

```
/*
 ************************************************************
 * slrpv_send_piggy_back(dest, channel, pages, size, pages2, size2,
 * fct, param, proc) :
 * try to send data to a remote node on a specific channel.
 * dest    : remote node.
 * channel : channel identifier.
 * pages   : virtual address of the beginning of the first set of data.
 * size    : size of data.
 * pages2  : virtual address of the beginning of the second set of data.
 * size2   : size of data.
 * fct     : callback function called when data have been correctly
 *           sent.
 * param   : parameter provided to the callback function when called.
 * proc    : proc structure of the calling process.
 *
 * If the data cannot be sent, a timeout() is generated and
 * the kernel will call ddslrp_timeout() until this procedure
 * can effectively send the data.
 *
 * return values : 0           : success.
 *                 E2BIG       : an internal array used to handle
 *                               a description of the physical pages
 *                               composing the area the user wants to send
 *                               is too small (the area is made of
 *                               too many uncontig. groups of pages).
 *                 ENXIO       : SLR/P not initialized or not enough
 *                               structures allocated at compilation
 *                               time to handle this board.
 *                 EINVAL      : size is NULL.
 *                 EWOULDBLOCK : tsleep() returned EWOULDBLOCK.
 *                 EINTR       : tsleep() returned EINTR.
 *                 ERESTART    : tsleep() returned ERESTART.
 *                 EFAULT      : should not occur, unrecoverable error,
 *                               contact the author if it happens.
 *                 ENOENT      : this board is not initialized.
 ************************************************************
 */

int
slrpv_send_piggy_back(dest, channel, pages, size, pages2, size2,
      fct, param, proc)
                      node_t dest;
                      channel_t channel;
                      caddr_t pages, pages2;
                      size_t size, size2;
                      void (*fct) __P((caddr_t, int));
                      caddr_t param;
                      struct proc *proc;
```

Voici le prototype de `slrpv_recv_piggy_back()` :

```
/*
 ************************************************************
 * slrpv_recv_piggy_back(dest, channel, pages, size, pages2, size2,
 * fct, param, proc) :
 * ask to receive data from a remote node on a specific channel.
 * dest    : remote node.
 * channel : channel identifier.
 * pages   : virtual address of the beginning of the first area ready
 *           to receive data.
 * size    : size of the area.
 * pages   : virtual address of the beginning of the second area ready
 *           to receive data.
 * size    : size of the area.
 * fct     : callback function called when data have been correctly
 *           received.
 * param   : parameter provided to the callback function when called.
 * proc    : proc structure of the calling process.
 *
 * return values : 0           : success.
 *                 E2BIG       : an internal array used to handle
 *                               a description of the physical pages
 *                               composing the area the user wants to send
 *                               is too small (the area is made of
 *                               too many uncontig. groups of pages).
 *                 ENXIO       : SLR/P not initialized.
 *                 EINVAL      : size is NULL or we're the remote node
 *                               (local loop not allowed).
 *                 EWOULDBLOCK : tsleep() returned EWOULDBLOCK.
 *                 EINTR       : tsleep() returned EINTR.
 *                 ERESTART    : tsleep() returned ERESTART.
 *                 ENXIO       : not enough structures allocated at
 *                             :  compilation time to handle this board.
 *                 ENOENT      : this board is not initialized.
 ************************************************************
 */

int
slrpv_recv_piggy_back(dest, channel, pages, size, pages2, size2,
      fct, param, proc)
                      node_t dest;
                      channel_t channel;
                      caddr_t pages, pages2;
                      size_t size, size2;
                      void (*fct) __P((caddr_t));
                      caddr_t param;
                      struct proc *proc;
```

Chapitre 3

Modèles de programmation

Ce chapitre présente divers modèles de programmation adaptés aux couches PUT et SLR/V. Les exemples de codes sont fournis sans traitement d'erreur, pour plus de clarté.

3.1 Programmation de la couche PUT

La couche PUT autorise des échanges DMA de noeud à noeud. On présente ici un exemple d'émission d'un message constitué de deux pages à destination du noeud 1. La première page contient 3 octets et commence en A0000h en mémoire physique, la deuxième page contient 2 octets et commence en B0021h. La première page doit être placée en C0000h dans la mémoire du noeud destinataire et la deuxième page en D0000h. La machine est munie d'une unique carte Fast-HSL, le paramètre minor utilisé est donc nul.

La figure 3.1 page suivante présente l'organigramme du protocole utilisé.

La figure 3.2 page 32 présente le code correspondant.

3.2 Programmation de la couche SLR/V

3.2.1 Programmation par appels bloquants

Les points d'entrée des différentes API de la machine MPC sont non bloquants. Le mécanisme de callback permet de les transformer en appels bloquants.

L'intérêt d'un appel bloquant est de simplifier l'algorithme que le programmeur doit implémenter, l'appel ne se finissant que lorsque les données ont été effectivement envoyées. Le programmeur peut donc réutiliser le tam-

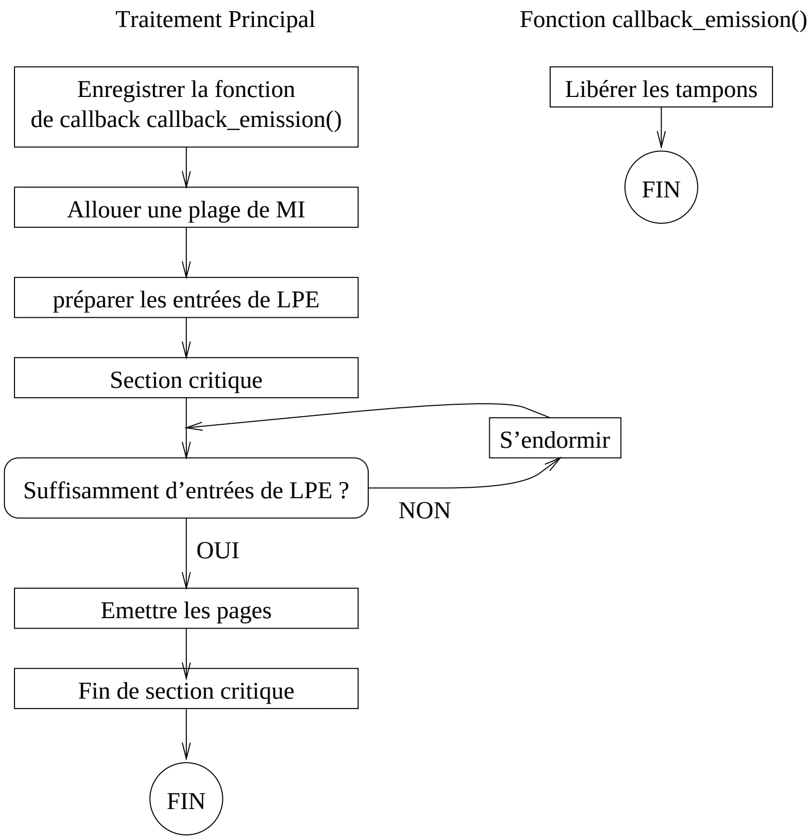

FIG. 3.1 - *Modèle d'émission avec PUT*

```
int node, sap_id, s;
mi_t mi_start;
lpe_entry_t lpe_entry[2];

/* indiquer la fonction de callback après émission */
sap_id = put_register_SAP(0, callback_emission, NULL);

/* allouer une plage de 1 MI */
put_attach_mi_range(0, sap_id, 1);

/* demander le numéro du 1er MI de la plage (donc l'unique MI de la plage) */
mi_start = put_get_mi_start(minor, sap_id);

/* préparer les deux entrées de LPE qui composent le message */
lpe_entry[0].routing_part = 1;
lpe_entry[0].page_length  = 3;
lpe_entry[0].PLSA         = 0xa0000;
lpe_entry[0].PRSA         = 0xc0000;
lpe_entry[0].control      = NOR_MASK | MI_PART(mi_start);

lpe_entry[1].routing_part = 1;
lpe_entry[1].page_length  = 2;
lpe_entry[1].PLSA         = 0xb0021;
lpe_entry[1].PRSA         = 0xd0000;
lpe_entry[1].control      = LMP_MASK | NOS_MASK | NOR_MASK | MI_PART(mi_start);

/* passer en section critique */
s = splhigh();

/* attendre la libération de deux entrées de LPE consécutives */
while (put_get_lpe_free(0) < 2)
  tsleep(&hslfreelpe_indent, HSLPRI, "FREELPE", 0);

/* demander l'émission */
put_add_entry(0, lpe_entry);
put_add_entry(0, lpe_entry + 1);

/* fin de section critique */
splx(s);

/* fin d'utilisation de la couche PUT */
put_unregister_SAP(0, sap_id);
```

FIG. 3.2 - *Exemple de code utilisant PUT*

pon d'émission juste après l'envoi. Avec un appel non bloquant, il doit s'assurer que l'envoi est effectivement réalisé avant de réutiliser le tampon.

Le mécanisme utilisé ici s'appuie sur un entier défini globalement dans le noyau, et qui sert à la fois d'indicateur d'occupation du tampon et d'évènement de reveil pour le couple tsleep()/wakeup().

On présente ici l'envoi de 1000 octets commençant à l'adresse A0000h de la mémoire virtuelle du processus émetteur, vers le noeud 1, sur le canal 3.

La figure 3.3 présente l'organigramme du protocole utilisé.

La figure 3.4 page suivante présente le code correspondant.

La figure 3.5 page suivante présente la fonction de callback associée à ce mécanisme.

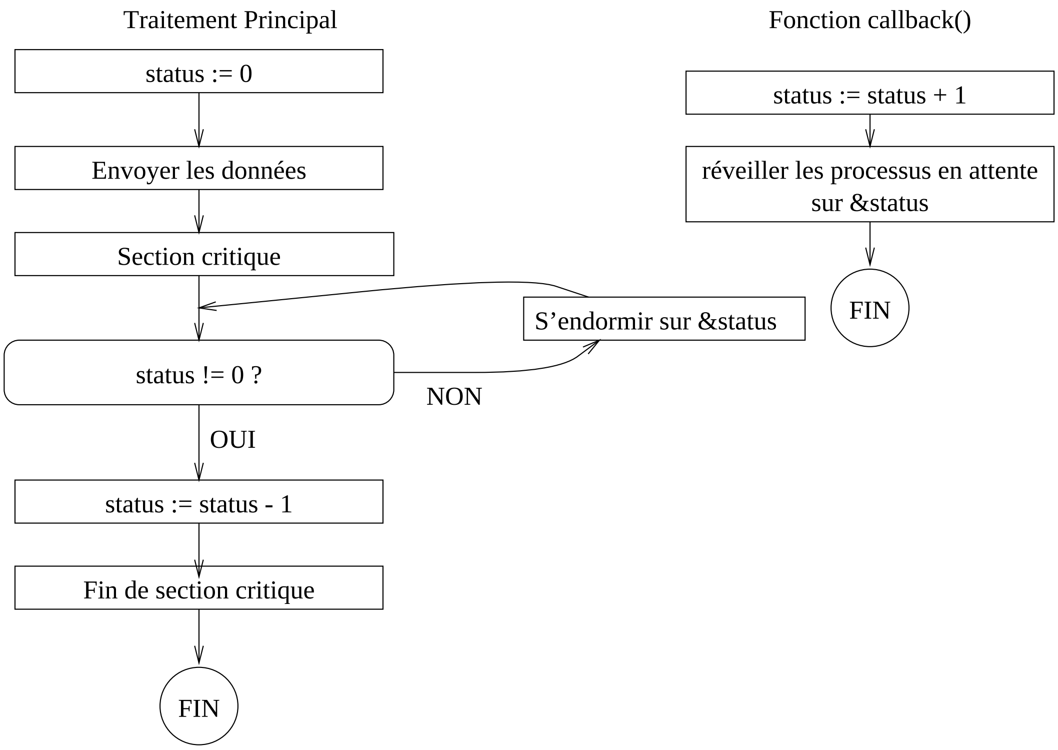

Fig. 3.3 - *Modèle d'émission bloquante avec SLR/V*

3.2.2 Programmation optimale par appels non bloquants

La programmation par appels non bloquants permet de continuer à profiter des services du processeur lorsque les données d'un tampon sont en cours

```
int status, s;

/* mettre à zéro l'entier status */
status = 0;

/* envoyer les données */
slrpv_send(1, 3, 0xa0000, 1000, callback, &status, p);

/* passer en section critique */
s = splhigh();

/* tester si les données ont été envoyées */
while (!status)
  tsleep(&status, HSLPRI, "DATASENT", 0);

/* mise-à-jour de status */
status--;

/* fin de section critique */
splx(s);

/* ici, le tampon d'émission peut être réutilisé */
```

FIG. 3.4 - *Exemple de code bloquant avec SLR/V*

```
void
callback(param, size)
         caddr_t param;
         int size;
{
/* signaler qu'une lecture est possible */
  (*param)++;

/* réveiller les processus en attente */
  wakeup(param);
}
```

FIG. 3.5 - *La fonction de callback*

d'envoi. Le programmeur doit alors faire un effort particulier pour effectuer des opérations qui ne font pas usage du tampon en cours d'émission, juste après l'opération `slrpv_send()`.

Pour réaliser ce modèle de communication, on part de l'algorithme d'envoi bloquant et on rajoute une section juste après l'envoi des données.

La figure 3.6 présente l'organigramme du protocole utilisé.

La figure 3.7 page suivante présente le code correspondant.

La fonction de callback utilisée reste la même que dans le cas bloquant.

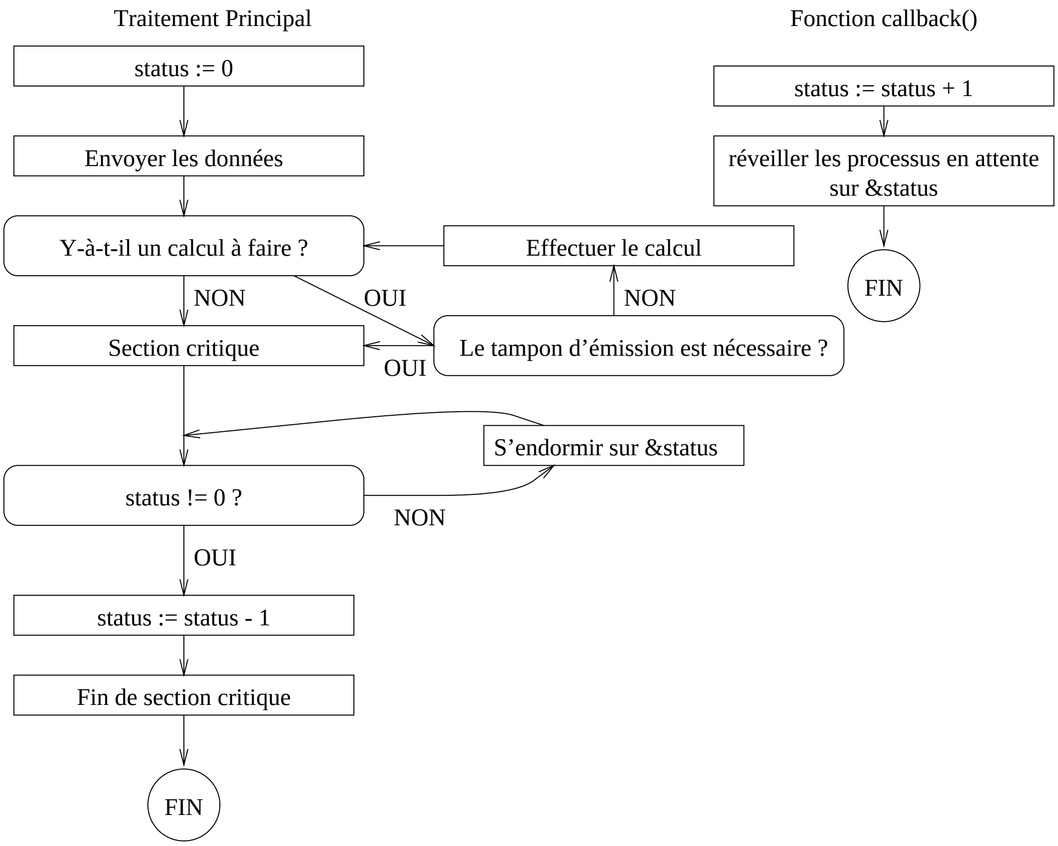

FIG. 3.6 - *Modèle d'émission non bloquante avec SLR/V*

```
int status, s;

/* mettre à zéro l'entier status */
status = 0;

/* envoyer les données */
slrpv_send(1, 3, 0xa0000, 1000, callback, &status, p);

/* EFFECTUER ICI DES OPERATIONS NE FAISANT PAS USAGE DES 1000 OCTETS COMMENCANT EN 0xA0000 */

/* passer en section critique */
s = splhigh();

/* tester si les données ont été envoyées */
while (!status)
  tsleep(&status, HSLPRI, "DATASENT", 0);

/* mise-à-jour de status */
status--;

/* fin de section critique */
splx(s);

/* ici, le tampon d'émission peut être réutilisé */
```

FIG. 3.7 - *Exemple de code non bloquant avec SLR/V*

Chapitre 4

Conclusion

Les couches de communication de haut niveau (couches SLR/P et SLR/V) décrites ici sont accessibles uniquement dans le noyau. Pour en profiter hors du noyau, une implémentation de l'interface socket, sans copie de tampons et s'appuyant sur la couche SLR/V, est en cours de développement par Nicolas Souchu et Pierre David du laboratoire PRiSM.

La couche de communication de bas niveau est accessible depuis le noyau (couche PUT), et hors noyau avec la bibliothèque USERACCESS.

Ainsi, avec l'arrivée de l'implémentation des sockets sur SLR/V, l'utilisateur pourra choisir de travailler au niveau d'abstraction qu'il souhaite, bas niveau ou haut niveau, autant dans le noyau que dans un processus utilisateur.