MASTER EN SCIENCES INFORMATIQUES

Document d'aide à l'écriture d'un driver pour le système UNIX

Barras, Guy

Award date:
1978

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

DOCUMENT D'AIDE A L'ECRITURE D'UN "DRIVER" POUR LE SYSTEME UNIX

Promoteur : DEMARTEAU J.

Guy BARRAS

Mémoire présenté
en vue de l'obtention du grade de
Licencié et Maître en Informatique

ANNEE ACADEMIQUE 1977 – 1978

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
NAMUR
INSTITUT D'INFORMATIQUE

DOCUMENT D'AIDE A L'ECRITURE D'UN "DRIVER" POUR LE SYSTEME UNIX

Promoteur : DEMARTEAU J.

Guy BARRAS

Mémoire présenté
en vue de l'obtention du grade de
Licencié et Maître en Informatique

ANNEE ACADEMIQUE 1977 – 1978

REMERCIEMENTS.

Je tiens à remercier J. Demarteau pour l'intérêt qu'il a porté à ce mémoire et pour l'aide qu'il m'a constamment prodiguée, aussi bien lors de l'étude, que lors de la rédaction de ce travail.

Qu'il me soit permis de remercier également Monsieur Milgrom, professeur à l'Université Catholique de Louvain et ses assistants P. Courtois et R.M. Simpson, pour l'acceuil qu'ils m'ont réservé au sein de leur groupe et les nombreux conseils qu'ils m'ont donnés.

TABLE DES MATIERES .

Première partie :

INTRODUCTION .

PREMIERE PARTIE: INTRODUCTION

Une multitude de systèmes d'exploitation sont actuellement disponibles sur le marché informatique. Ces systèmes sont généralement très vastes (parfois plusieurs centaines de milliers de bytes) et souvent très complexes, ce qui rend leur compréhension difficile pour une seule personne.

Cependant, il existe un système qui possède la majorité des caractéristiques d'un grand système d'exploitation, tout en restant abordable par une personne, il s'agit du système UNIX.

C'est cette facilité d'accès et le fait que nous disposons d'une machine sur laquelle UNIX peut être implémenté, en l'occurence le PDP11/45 de la Faculté de Chimie, qui nous ont poussé à implémenter UNIX sur le PDP.

Malheureusement, la version que l'on possède de UNIX n'inclut pas tous les drivers nécessaires à la conduite des périphériques se trouvant à la Faculté de Chimie (table traçante, écran graphique, lecteur de cartes) ou de certains périphériques que l'on voudrait connecter plus tard.

Le problème de raccordement software du périphérique se pose donc pour UNIX. Sa résolution implique une connaissance intime du système et du contrôleur du périphérique.

Pour notre part, nous avons effectué un apprentissage du système global, en tant qu'utilisateur, pour ensuite étudier plus en détails la partie du système UNIX concernant les entrées-sorties. Toute cette étude nous a permis d'acquérir les connaissances nécessaires à l'écriture du driver du lecteur de cartes.

Par après, nous avons remarqué la difficulté d'utiliser tel quel le lecteur et nous avons du écrire un programme utilitaire résolvant ce problème.

Face à tout ce travail, nous n'avons peut-être pas élaboré beaucoup de considérations théoriques sur UNIX, mais nous sommes malgré tout parvenus à écrire un système UNIX correct, incluant le driver du lecteur de cartes.

En rédigeant ce travail, nous souhaitons qu'il puisse servir de syllabus de référence pour quelqu'un qui serait désireux d'écrire un driver supplémentaire pour le système UNIX et qui pourrait se retrouver confronté aux mêmes problèmes que ceux que nous avons rencontrés nous-mêmes. Nous espérons également fournir une synthèse des entrées-sorties, la plus claire et compréhensible qui soit.

Deuxième partie :

LE SYSTEME D'EXPLOITATION UNIX

PRESENTATION GENERALE

DEUXIEME PARTIE : LE SYSTEME D EXPLOITATION UNIX

PRESENTATION GENERALE

2.1. QU'EST-CE QUE UNIX ?

UNIX est le nom donné à un système d'exploitation mis au point par les laboratoires de la " Bell company " pour une partie de la gamme d'ordinateurs PDP 11 de DEC (Digital Equipment Corporation).

La version qui nous intéresse a été écrite pour les machines numérotées 40,45 ou 70 de PDP 11 .

Il s'agit d'un système multi-utilisateur et interactif d'où sa dénomination de " Time-Sharing " .

Les concepteurs de UNIX poursuivent un but principal : créer un système puissant mais surtout très simple,élégant et facile à utiliser .

Par après, ils purent montrer qu'un système d'exploitation puissant et interactif ne demande pas nécessairement un investissement important, ni en équipement, ni en personnel pour sa conception .

La version de UNIX considérée a été écrite dans un langage de haut niveau : le " C " . Ce langage, bien que très évolué, permet d'effectuer de nombreuses opérations pour lesquelles on doit généralement employer l'assembleur . Cela est rendu possible par le fait que le " C " supporte les notions de types et de structures, permettant de traiter des adresses réelles .

Unix fournit également des programmes supplémentaires au noyau du système proprement dit : des compilateurs (C, Fortran, ...) interpréteurs, "debuggers", éditeur ... afin d'aider l'utilisateur dans son travail .

2.2. OPTIONS FONDAMENTALES

La conception d'un système d'exploitation suit souvent un ou plusieurs guides sur lesquelles viennent se greffer les autres concepts d'un O.S. .

Un exemple est un système basé sur une gestion de mémoire découpée en partitions . Dans ce cas,c'est la gestion de la mémoire qui provoque l'acceptation ou la mise en attente d'un processus, et qui donne une priorité différente à un processus plutôt qu'à un autre .

On voit ici que tous les choix faits dans ce système vont dès lors dans le sens d'une optimisation de la place mémoire .

Bien entendu,plusieurs critères peuvent être choisis, sans pour autant être en contradiction les uns avec les autres .
Dans l'exemple précédent, la maintenance d'un système de fichiers évolué ne porte en aucun cas préjudice à l'optimisation de la place mémoire .

En ce qui concerne le système UNIX, plusieurs options fondamentales ont vraisemblablement dû être prises .

- La gestion des ressources a été basée sur la notion de processus . Tout processus peut être reconnu par le système et entrer en compétition à tout moment avec les autres pour les différentes ressources.

- UNIX maintient un système de gestion de fichiers par niveaux (une arborescence) et protégés, incluant également la notion de fichiers sur volume montable .

- Unix banalise les fichiers et les périphériques ainsi que les différentes interactions entre processus .

-UNIX est un système bien structuré, donc très facilement reconfigurable et modifiable .

Ces options prises, les créateurs de UNIX écrivirent ce système dans le but évident de créer un système simple et surtout très facile à employer . Il n'est donc pas étonnant que l'on ait à sa disposition un JCL bien conçu et complet .

Voyons plus en détails les différents points précisés plus haut .

2.2.1. LA GESTION DES PROCESSUS .

§ -1. Définition d'un processus

Pour UNIX, un processus est l'exécution en ordinateur d'une image.

Une image est tout l'environnement de l'exécution d' un programme en machine, c'est-à-dire :

- une photo de la mémoire du programme qu'il exécute ,
- la valeur des registres généraux,
- l'état des fichiers ouverts,
- le directoire de travail .

C'est donc en quelque sorte, l'état courant d'un " pseudo-ordinateur " monoprogrammé occupé à exécuter un programme .

§ -2. Allocation de la mémoire pour un processus .

Comme UNIX est un système multi-utilisateurs et multi-tâches (un utilisateur peut créer plusieurs processus en même temps), plusieurs processus peuvent réclamer en même temps les différentes ressources de l'ordinateur .

En ce qui concerne la mémoire, si un processus en exécution doit avoir son image en mémoire centrale, par contre, un processus en attente d'exécution n'aura son image en mémoire que si aucun processus plus prioritaire ne désire entrer dans le système .

Dans ce cas, l'image est placée sur mémoire auxiliaire (Swap device) .

La mémoire allouée à une image est logiquement divisée en trois segments .

La première partie contient le texte du programme lui-même . Ce segment commence à l'adresse virtuelle Ø dans la zone de mémoire virtuelle allouée à cette image .
Pendant son exécution,cette zone est déclarée " Write-protect " et peut donc être partagée par plusieurs processus exécutant le même programme .

La deuxième partie est le segment de données . Cette zone est non-partageable, inscriptible, initialisée et sa dimension peut être étendue par un appel-système .

La dernière partie est le "Stack " segment . Elle est placée en fin de zone virtuelle et sa dimension fluctue de la même façon que le pointeur du "Stack hardware " de la PDP 11 .

§ -3. Les primitives de UNIX pour la gestion des processus .

UNIX contient un certain nombre de routines qui servent à gérer les processus . Elle permettent de créer,de synchroniser,de terminer ... des processus .

Ces routines de base, appelées primitives du système, peuvent être utilisées dans d'autres programmes pouvant effectuer des opérations complexes en employant ces quelques routines .

-La création d'un processus

Un processus peut en créer un nouveau par la simple exécution d'un appel système : le FORK .

Lors de l'exécution d'un"fork" dont la forme est :

```
processid =  fork(label)
```

,

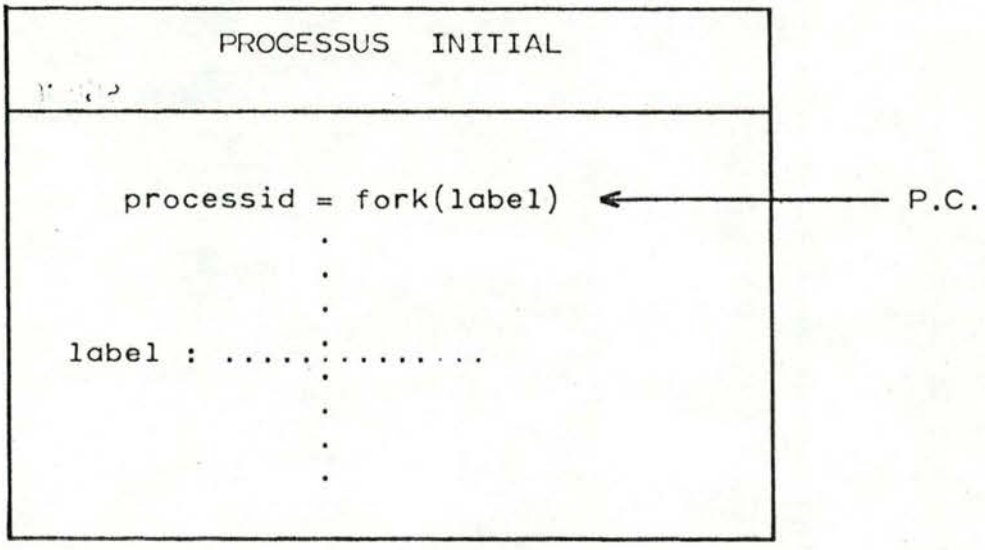
le processus se dédouble et ils reprennent chacun leur exécution de façon indépendante .

Chacun possède logiquement une copie de l'image de la mémoire . Le premier processus est appelé "parent" et son exécution continue en séquence après le "fork" .

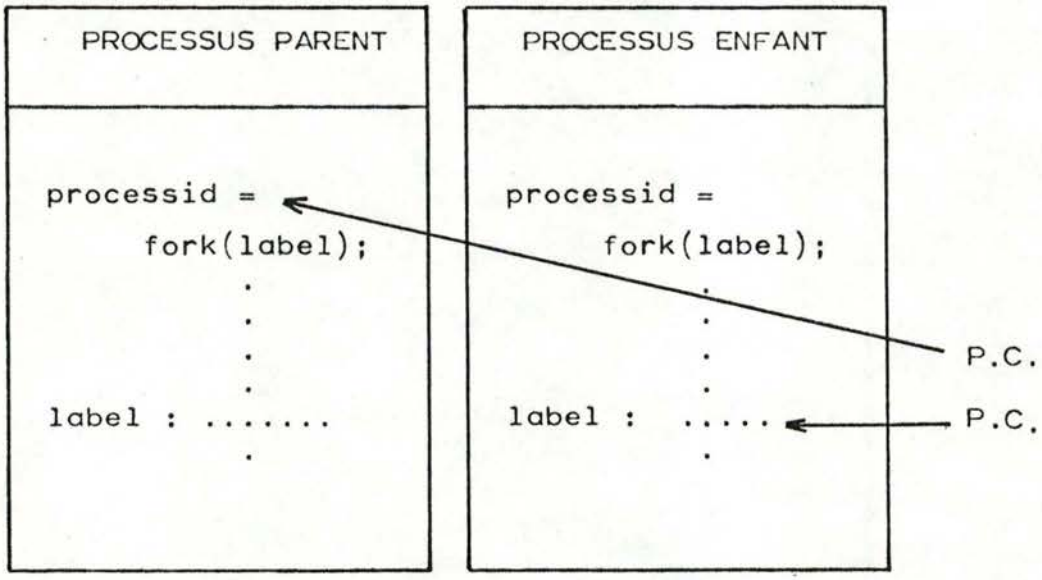
L'autre processus, l'"enfant" commence son exécution à l'adresse spécifiée par label .

Le numéro du processus "enfant" est retourné dans l'entier processid et peut être utilisé par le "parent".

Exemple :



Après l'exécution du " fork ", la situation suivante se présente :



Remarque : Les processus "parent " et " enfant " ont exactement la même image (mêmes instructions, mêmes fichiers ouverts etc ...) mais le programme counter est positionné à des valeurs différentes dans les deux cas .

- La communication inter-processus

L'appel système PIPE dont la forme est :

```
filep = pipe()
```

a pour effet de créer un canal inter-processus, appelé un PIPE .
Ce canal est reconnu, aussi bien par le parent que par l'enfant qui a été créé par un appel fork (de même que tout autre fichier d'ailleurs) .

Dès lors, chaque processus peut exécuter des ordres READ ou WRITE, en utilisant comme descripteur de fichier, celui retourné par l'appel pipe, en l'occurence filep .

C'est ainsi que des informations peuvent être passées d'un processus à un autre et inversément.

Le système UNIX gère ce canal en endormant et en réveillant aux moments opportuns les processus qui fournissent ou recoivent des informations du canal de données .

- Exécution d'un programme à partir d'un processus

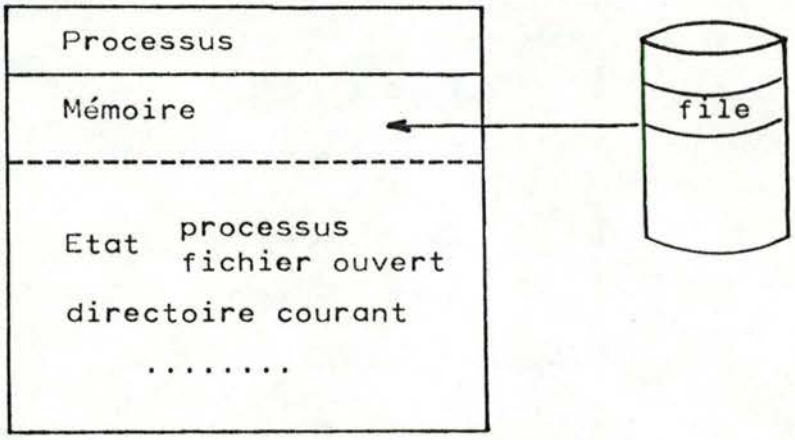
L'appel système EXECUTE, dont la forme est:

```
execute (file, arg1 , arg2 ,...argn)
```

permet à un processus d'exécuter un programme stocké dans le fichier nommé file en lui passant une série de paramètres nécessaires à son exécution : arg1,arg2,...argn, au lieu de continuer l'exécution de son propre programme .

Il est à noter que la fin de l'exécution du programme file provoque la terminaison du processus .

Il n'y a pas de retour en séquence après l'exécute dans le processus qui a lancé cet ordre .



L'exécution d'un execute provoque le transfert du programme file en lieu et place du code et des données du processus qui donne l'ordre execute . Ce code est donc détruit .

Cependant, l'état du processus et des fichiers ouverts et les relations entre processus restent inchangés .

- La synchronisation entre processus

L'appel système WAIT dont la forme est :

```
processid = WAIT
```

provoque la mise en attente d'un processus " parent ", jusqu'à ce qu'un de ses enfants ait terminé son exécution . Le numéro de ce processus est alors retourné dans l'entier processid .

- La terminaison d'un processus

L'appel

```
Exit(status)
```

provoque la terminaison d'un processus, détruit son image et ferme les fichiers ouverts.

Il est à noter que le status est disponible au processus parent (s'il existe) et qui effectue un WAIT .

§ -4. Interaction entre les utilisateurs et le système

Afin de permettre aux utilisateurs d'exécuter certaines commandes à partir de leur terminal, UNIX associe à chacun d'eux qui se fait connaître au système, un processus : le SHELL .

Le Shell lit les lignes envoyées par l'utilisateur, les interprète et provoque l'exécution d'autres programmes, suivant l'interprétation faite de la commande reçue . Pour ce faire,le Shell utilise les primitives vues précédemment (fork, execute, wait) .

Grâce à la souplesse du système de gestion des processus, et notamment par la possibilité de synchroniser différents processus et de créer des canaux de communication entre eux (les pipes), le Shell peut permettre l'utilisation des "FILTRES " et du principe du multi-tâches .

Le système des filtres permet à l'utilisateur d'envoyer une série de commandes séparées par des " | ", ce qui provoque l'exécution simultanée des commandes et la création de "pipes",de façon que les informations en sortie de la $i^{ème}$ commande servent d'information d'entrée de la (i + 1)ème .

EX: Si on désire lire un fichier sur bande magnétique,lui faire subir deux transformations, qui peuvent être faites par les programmes transf1 et transf2 et ensuite diriger ce fichier vers le spool de l'imprimante; on entrera, par exemple, la commande

```
cat /dev/mtØ | transf1 | transf2 | lpr
```

au lieu de la suite de commandes :

cat /dev/mtØ > work1 lire la bande magnétique qui se trouve sur le dérouleur n°Ø et placer le fichier lu dans le fichier de travail work1 .

transf1 <work1 >work2	effectuer la première transformation en prenant comme input le fichier work1 et comme output un autre fichier de travail : work2
transf2 <work2 >work1	opération similaire à la précédente . work1 contient alors le fichier transformé . Il reste à l'imprimer .
lpr work1	provoque l'impression du fichier work1 en le plaçant dans le fichier spool dont le nom sera placé dans une file d'attente des fichiers à imprimer .
rm work1 work2	provoque la supression des fichiers work1 et work2 .

Cet exemple montre non seulement que le système des filtres facilite le travail de l'utilisateur, mais aussi qu'il optimise grandement le fonctionnement de la machine, dans le sens que toutes les commandes se déroulent "simultanément" .

On profite donc de tous les avantages de la multiprogrammation, même si le travail du système pour gérer ce type de commande est assez conséquent . En effet, il doit synchroniser l'exécution en endormant et réveillant certaines commandes, selon l'état des canaux de communication entre processus .

Le " multi-tâche ", c'est donner la possibilité à un utilisateur de créer plusieurs processus, qui s'exécuteront simultanément, mais sans interférence les uns avec les autres .

Pour cela, il suffit de faire suivre la commande du signe & . A ce moment, UNIX renvoit un numéro de processus et est prêt à accepter une autre commande .

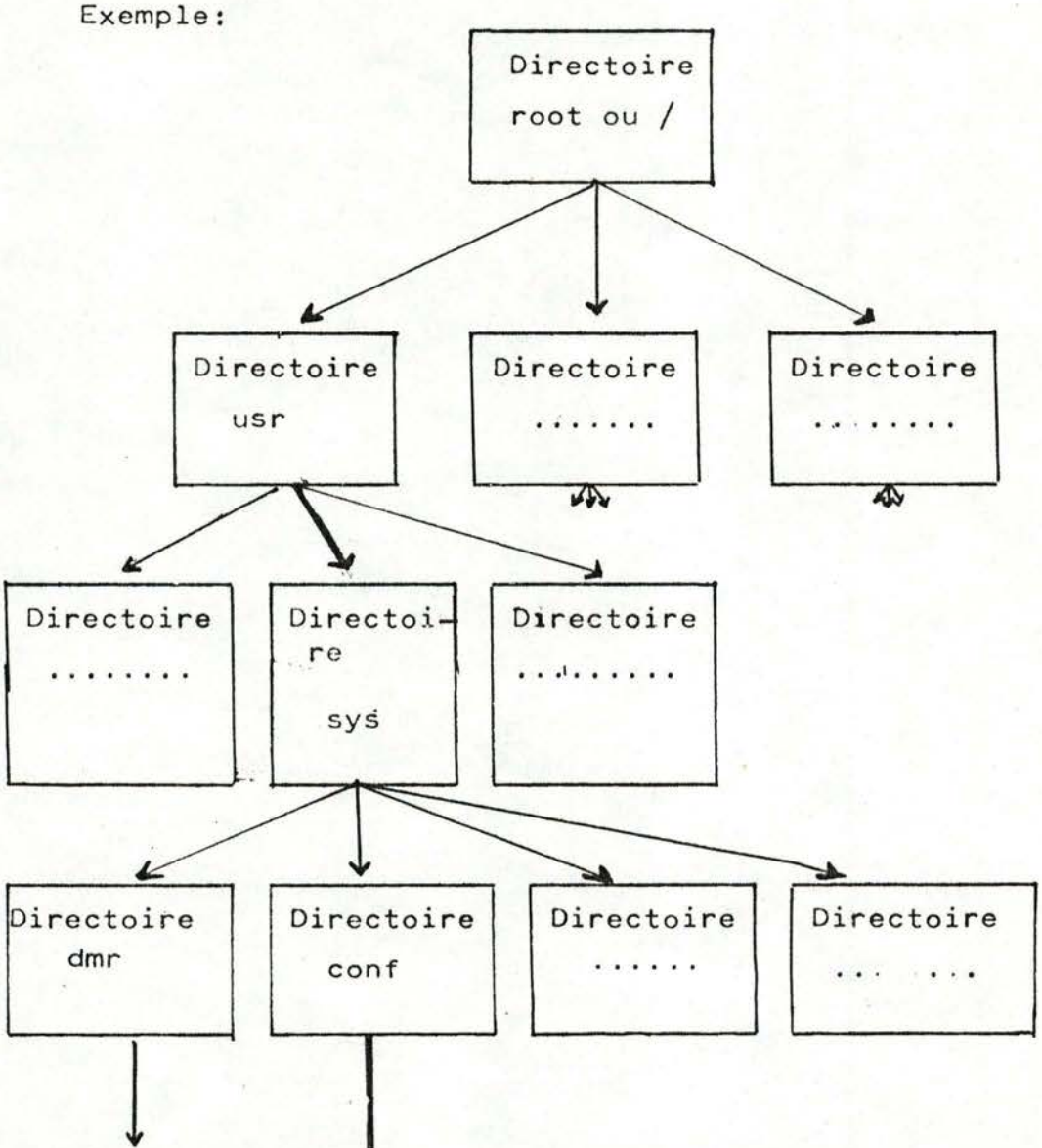
2.2.2. LA GESTION DES FICHIERS .

§ -1. La structure logique des fichiers .

UNIX maintient une structure arborescente de fichiers, de façon que le système puisse retrouver aisément un fichier lorsqu'on lui fournit le nom de celui-ci .

Le nom d'un fichier ou " pathname " est une suite de noms séparés par le caractère " / " (ex: /usr/games/wump) . Le dernier nom est le nom du fichier lui-même, c'est-à-dire le nom d'une feuille de l'arbre (wump) . Les autres sont des noms de directoires qui correspondent chacun à un noeud de l'arborescence .

Exemple:



Contient les sources des programmes dits "drivers" et notamment cr.c, le driver du lecteur de cartes que nous étudierons plus tard .

contient les fichiers servant à définir la configuration du système et notamment :
l.s (/usr/sys/conf/l.s) et
c.c (/usr/sys/conf/c.c)

Les fichiers qui sont des programmes-sources pour aider le programmeur à reconfigurer facilement le système c-à-d mkconf.c et sysfix.c

-Sortes de fichiers :

UNIX distingue trois sortes de fichiers : les fichiers ordinaires, les directoires et les fichiers spéciaux .

Les fichiers ordinaires sont composés d'une suite de caractères formant des lignes séparées par le caractère " newline " (code octal Ø 15) .
UNIX ne considère aucune structure particulière pour un fichier . Malgré tout, certains d'entre eux possèdent une structure bien définie . Il s'agit des fichiers de sortie de l'assembleur, du chargeur ou du compilateur (a.out) . Cependant, cette structure est contrôlée par les programmes qui les utilisent, non par le système UNIX .

Un directoire est donc un fichier qui sert de noeud dans l'arborescence .
Il contient une série de noms de fichiers lui appartenant, auxquels sont associés des pointeurs vers ces fichiers .
Evidemment, les fichiers appartenant à un directoire peuvent être également des directoires .

Remarque : L'utilisateur peut se positionner à n'importe quel noeud dans l'arborescence. Dès lors, pour nommer un fichier, il lui suffit de donner les noms à partir de cet endroit dans l'arborescence.
De plus, au moment où l'utilisateur entre en relation avec UNIX, (commande LOGIN), il se voit positionné à un endroit de l'arborescence qu'il a choisi une fois pour toutes. Cet endroit est spécifié dans la table des utilisateurs (fichier / ETC/ PASSWD).

Exemple: soit le fichier /usr/sys/conf/l.s.
Si le directoire courant de l'utilisateur (le noeud de l'arborescence où il est positionné à ce moment) est /usr/sys, le fichier pourra s'appeler conf/l.s. Ce nom ne commençant pas par /, on lui adjoindra automatiquement le directoire courant: /usr/sys.

La troisième sorte de fichier est celle composée des fichiers spéciaux.

A chaque périphérique est associé un fichier spécial. Celui-ci possède un nom de la même structure qu'un fichier ordinaire ou qu'un directoire.

De plus, on peut exécuter un ordre de lecture ou d'écriture sur un fichier spécial, exactement comme sur un autre, mais cela provoquera l'activation du périphérique associé au fichier.

Les avantages des fichiers spéciaux sont que les périphériques ou les fichiers normaux ont la même syntaxe de noms, sont lus et écrits de la même façon et que ce système permet d'employer la même procédure de protection, quel que soit le fichier à protéger.

§ -2. L'Implémentation du système de fichiers.

Le but de ce travail n'est pas de décrire la façon dont UNIX a implémenté son système de fichiers. Neanmoins, afin de comprendre le fonctionnement des entrées-sorties qui seront étudiées dans la partie suivante,

il est bon de synthétiser cette implémentation, ainsi que le fonctionnement des tables réservées par le système des entrées-sorties, afin de pouvoir reconnaître un fichier.

Sur chaque unité comportant un système de fichiers, on trouve, en plus des fichiers (ordinaires, directoires ou spéciaux), une table décrivant ces fichiers: la table i-list. Chaque ligne de cette table est un "file's inode" et contient la description d'un fichier.

Contenu de l'inode d'un fichier:

- propriétaire du fichier
- bits protection
- adresses sur disque ou bande où se trouve le fichier. C'est un ensemble de 8 adresses, soit directes, indirectes ou doublement indirectes, selon la taille du fichier (8 blocs, 8 blocs et 8 x 256 blocs, 8 x 256 x 256 blocs) .
- dimension (en nombre de caractères).
- date de la dernière modification.
- nombre de liens du fichier, c'est-à-dire le nombre de fois qu'il apparaît dans un directoire. En effet, le même fichier peut se trouver dans des directoires différents (commande ln).
- bits indiquant s'il s'agit d'un directoire, d'un fichier spécial.
- bits indiquant s'il s'agit d'un grand ou d'un petit fichier.

Une entrée de directoire contient un nom de fichier et un pointeur vers le fichier lui-même. Ce pointeur est un entier, appelé i-number et est utilisé lors de l'accès au fichier comme index dans la table i-list.

De plus, UNIX maintient une série de tables définissant l'ensemble des fichiers ouverts dans le système global, mais également par processus.

Lors de l'ordre "open", une recherche du fichier est effectuée sur le disque, par l'intermédiaire des directoires et de la table i-list, jusqu'à ce que l'inode correspondant au fichier soit trouvé.

Dès lors, trois tables du système sont mises à jour:

- la table inode qui contient une copie des lignes de la table i-list correspondante aux fichiers ouverts.
- la table file qui est une table unique pour tout le système et qui contient des "flags" indiquant:
    * le mode ouverture du fichier
    * un compteur qui s'incrémente quand un processus ouvre ce fichier et qui se décrémente si on le ferme. Ce compteur permet de décider quand l'entrée de la table peut être supprimée (count = $\emptyset$).
    * l'"offset" qui indique où le prochain "read" ou "write" doit être effectué sur le fichier.
    * un pointeur vers l'entrée correspondante de la table i-node.

- la table u-ofile, qui est une table réservée pour chaque processus. Le seul contenu de cette table est une série de pointeurs vers les entrées de la table file correspondante aux fichiers ouverts pour ce processus. C'est le numéro de ligne de cette table u-ofile qui est retourné par l'ordre "open" ou "create".

Le schéma suivant peut donc être établi.

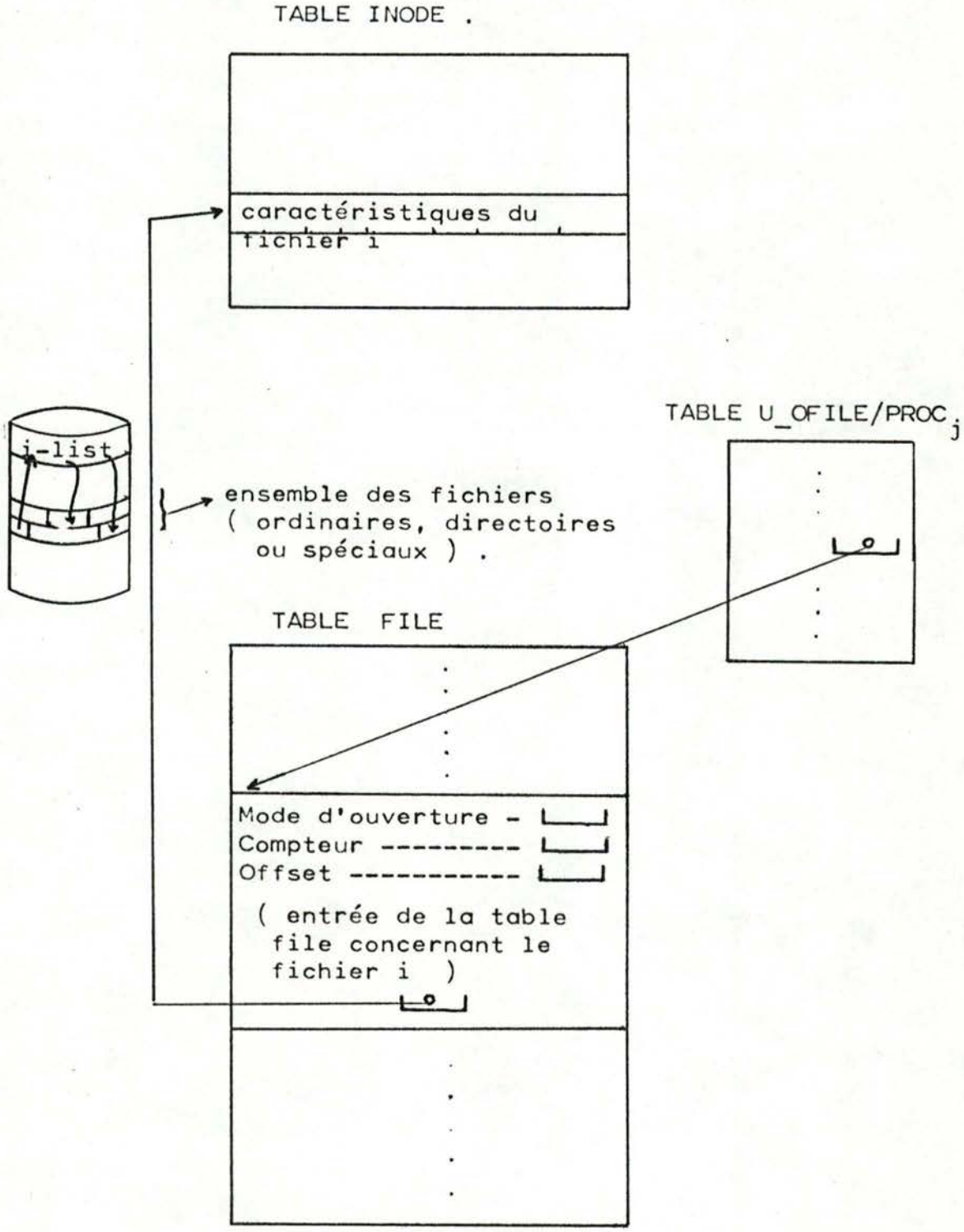


TABLE INODE .
caractéristiques du fichier i
i-list
ensemble des fichiers (ordinaires, directoires ou spéciaux) .
TABLE U_OFILE/PROC j
TABLE FILE
Mode d'ouverture -
Compteur ---------
Offset ----------
(entrée de la table file concernant le fichier i)

On remarquera donc que la recherche des caractéristiques d'un fichier ne se fait qu'une seule fois par l'intermédiaire du nom du fichier (à l'open). Par après, le descripteur retourné par l'open suffit à identifier ce fichier.

§ -3. La protection des fichiers.

En ce qui concerne la protection, UNIX associe à chacun des fichiers une série de 9 bits servant à définir les permissions associées à ce fichier.
Ces 9 bits forment trois ensembles de trois bits, qui sont relatifs à un type d'utilisateur.
Il s'agit -du propriétaire
-du groupe auquel le propriétaire appartient
-d'un utilisateur quelconque.
A chacun de ces ensembles correspond trois bits, spécifiant la permission de lire, d'écrire ou d'exécuter le fichier.

Ainsi, le propriétaire d'un fichier peut spécifier (par la commande "chmod") les permissions d'accès au fichier pour lui-même, pour le groupe ou pour les autres. Cependant, il ne lui est pas possible de le faire directement pour une personne particulière et pas pour d'autres.

Il est à noter que la modification de ces bits n'est admise que par le propriétaire d'un fichier ou par le "super-utilisateur", qui possède toutes les permissions sur tous les fichiers déclarés dans le système.

2.2.3. UNIX: UN SYSTEME FACILEMENT MODIFIABLE ET RECONFIGURABLE.

Il est nécessaire de bien distinguer dans UNIX le noyau du système et ses "utilitaires". En effet, UNIX est fourni sous la forme d'un système de fichiers composé de programmes sources, de programmes objets, de simples tables, de fichiers de données,... Il doit exister au moins un fichier qui est le noyau d'un système d'exploitation UNIX (exemple: /rkunix). Il suffit, dès lors, de charger, puis d'exécuter le bloc Ø du disque système en donnant le nom du fichier contenant le système d'exploitation.

En utilisant un système donné, on peut en créer un autre. Pour cela, il suffit de suivre les conseils stipulés dans le fichier /usr/sys/run, afin de compiler les programmes nécessaires et, ensuite, de relier et assembler les différents modules, afin de créer un nouveau noyau, qui pourra également être "bootstrapper".

Cependant, tous les fichiers n'ont pas servi à la création du système et notamment les programmes correspondant aux commandes et les utilitaires (exemple: le compilateur, l'éditeur, les programmes de maintenance du système,...).

Dès lors, si un arrêt du système et une procédure de création sont nécessaires pour créer un nouveau noyau, l'ensemble des commandes et des utilitaires peuvent être modifiés sans provoquer d'arrêt dans le fonctionnement du système. On peut, par exemple, ajouter une commande qui sera comprise par le "shell", modifier un programme de test de cohérence d'un système de fichier ou modifier la

table des utilisateurs sans recompiler le système.

La reconfiguration:

Pour créer un système reconnaissant un certain nombre de périphériques, deux opérations doivent être effectuées. Tout d'abord, il faut recréer un système après avoir modifié les fichiers qui décrivent la configuration.
Ces fichiers sont: /usr/sys/conf/l.s.
et /usr/sys/conf/c.c.
Cependant, le programme /usr/sys/conf/mkconf.c crée ces fichiers; il suffit de lui donner la sorte et le nombre de périphériques désirés.
Ensuite, il faut créer une série de fichiers spéciaux correspondant chacun aux différents périphériques de la configuration. Pour cela, le programme /etc/mkmod doit être utilisé.

Troisième partie:

LA GESTION DES ENTREES-SORTIES DANS UNIX.

LE PROBLEME DE L'ECRITURE D'UN DRIVER.

TROISIEME PARTIE: LES ENTREES-SORTIES DANS UNIX.

LE PROBLEME DE L'ECRITURE D'UN DRIVER.

Les entrées-sorties sont une partie de UNIX très intéressante, car c'est dans ce chapitre que le système est le plus dépendant de la machine et de son environnement.

Il est donc intéressant de voir comment UNIX essaie de ne pas laisser apparaître les limitations dues à cette dépendance et comment un ordre logique d'un utilisateur peut provoquer l'exécution d'une suite de routines, qui aboutissent aux ordres de fonctionnement d'un périphérique.

Pour cela, nous exposerons le système des entrées-sorties employé par UNIX, en essayant d'en dégager une structure de fonctionnement générale, en partant des ordres d'entrées-sorties du niveau programme-utilisateur et en analysant le chemin parcouru depuis là, jusqu'à l'activation physique d'un périphérique.

Enfin, nous essayerons de donner quelques directives qui, nous l'espérons, seront utiles pour ceux qui désireraient écrire un "driver" supplémentaire pour le système UNIX.

3.1. LES ENTREES-SORTIES DANS UNIX

3.1.1. STRUCTURE GENERALE DES ENTREES-SORTIES

On entend par système d'entrées-sorties, l'ensemble de toutes les routines permettant aux programmeurs d'employer dans leurs programmes des ordres " évolués " qui;grâce à l'exécution d'un certain nombre de ces routines,provoqueront le bon déroulement de l'ordre demandé .

Ces ordres sont du type :

Type d'opérations , fichier concerné , paramètres

Exemple :

```
int filedesc ;
char buffer  96 ;
        :
filedesc = open ("/dev/lp",1) ;
        :
write (filedesc,buffer,96 ) ;
        :
```

L'ordre open est relatif au fichier spécial /dev/lp qui

est associé à l'imprimante ; le paramètre 1 stipule que ce fichier est ouvert en écriture . Dès lors, tout ordre de lecture sera refusé .

L'appel-système " open " retourne une valeur qu'il place dans l'entier filedesc . Cette valeur est en fait le numéro du fichier ouvert pour ce processus . Dès lors, l'unique valeur contenue dans filedesc suffira à désigné le fichier /dev/lp . En effet, l'ordre write ne rappelle plus le nom du fichier mais bien son numéro contenu dans filedesc .

Les paramètres de l'ordre write spécifient l'adresse début de la zone à imprimer (buffer) et le nombre de caractères à écrire .

On remarquera que le programmeur ne doit se préoccuper ni du problème de la concurrence pour la ressource imprimante ni des opérations techniques à effectuer pour l'impression d'un fichier (provoquer un saut de page, envoyer des caractères à l'imprimante au rythme où celle-ci les demande, provoquer le passage à la ligne etc ...) .

Unix considère deux classes de périphériques appelés " block et character device " .

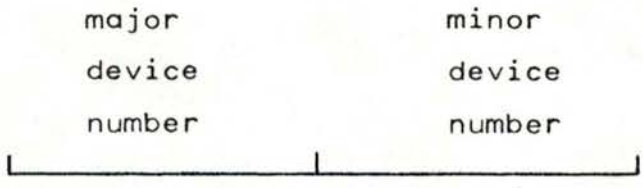
Les périphériques déclarés " block devices " sont les bandes et les disques magnétiques dont l'unité de transfert est un bloc de 512 bytes . De plus, les " block device " peuvent contenir un système de fichiers montables .

Comme l'interface de ces périphériques de ces périphériques est très structuré, les " drivers " correspondants

deviennent donc plus simples de par le partage de nombreuses routines et d'un ensemble de zones tampons .

Les " character devices ", par contre, ont un interface beaucoup moins évolué et surtout très différent pour chaque device . Beaucoup plus de choses doivent donc être faites par les " drivers " eux-mêmes, les rendant ainsi beaucoup plus complexes .

Les périphériques sont distingués par UNIX grâce à un mot contenant son numéro. Ce numéro est composé de deux parties:



Le "major number" sélectionne quel driver traite ce périphérique; le "minor number" , par contre, n'est pas utilisé par le système, mais est passé comme paramètre au driver, qui l'utilise pour orienter son action vers l'un ou l'autre des périphériques qu'il "conduit".

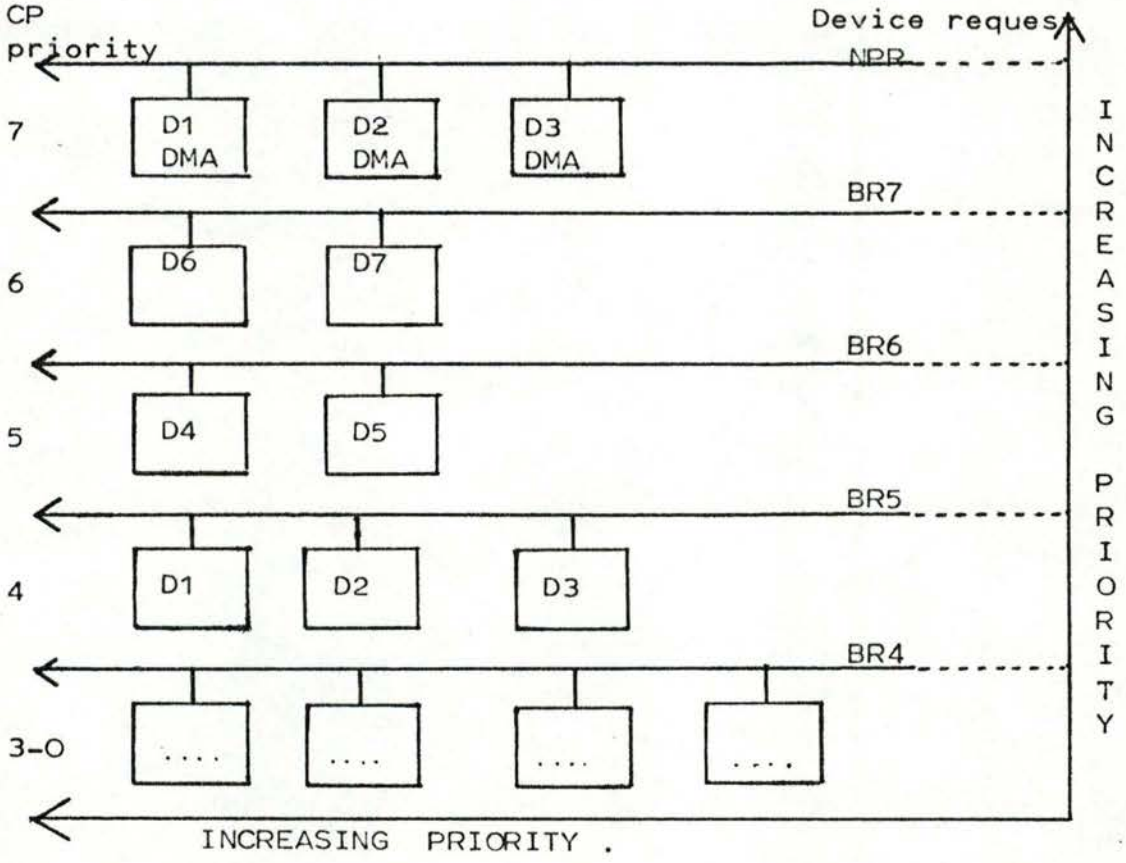
Avant de voir l'enchaînement des routines du système d'entrées-sorties, voyons un peu les interactions du système avec le hardware du PDP 11.

3.1.2. LES INTERRUPTIONS ET LES APPELS SYSTEMES DU PDP 11/40-45.

§ -1. Le système d'interruption.

Un périphérique demande le contrôle de l'"unibus", c'est, soit pour effectuer un transfert d'informations avec la mémoire, sans l'intervention du processeur, soit pour interrompre l'exécution d'un programme et forcer le processeur à exécuter une instruction se trouvant à une adresse spécifiée.

Le PDP 11 a une structure d'interruptions à plusieurs niveaux.



Une demande de contrôle de l'"unibus" par un périphérique peut être relative à une des cinq lignes de la figure précédente. La plus haute priorité est assignée au "non-processor-request" (npr), c'est-à-dire, aux transferts avec les mémoires à accès direct qui sont honorées par le processeur entre des cycles d'exécution d'une instruction.

Les demandes correspondant aux lignes BR7 jusqu'à BR4 sont prises en charge par le processeur entre les instructions. La priorité est fixée de façon hardware pour tous les périphériques, excepté pour le processeur où elle est programmable. La priorité du processeur peut être modifiée en positionnant les bits 7, 6 et 5 du ps (processor status register). Ces bits désignent un niveau de priorité qui empêche le traitement d'une demande de contrôle de l'"unibus" par un périphérique dont la priorité est inférieure ou égale à la valeur contenue dans ces bits.

De plus, sur la même ligne de priorité peut être connecté un nombre quelconque de périphériques, dont la priorité augmente d'autant plus qu'ils sont reliés plus près du processeur.

La procédure d'interruption comporte les opérations suivantes:

- Le périphérique envoie une commande d'interruption et une adresse dans l'espace virtuel du mode "kernel". A cette adresse, on trouve deux mots formant le vecteur d'interruption du périphérique. Ces deux mots contiennent respectivement l'adresse de la routine de service du périphérique et un nouveau ps pour le processeur.

- Le processeur sauve son program counter register (pc) et son program status register (ps) sur la pile "kernel", charge le premier mot du vecteur d'interruption dans son pc et le second dans son ps.

- Le pc du processeur étant chargé avec l'adresse début de la routine de service du périphérique, le processeur exécute donc cette routine.

- A la fin de la routine, on trouvera généralement une instruction rti ou rtt (return from interrupt), qui provoquera le chargement des deux mots du sommet de la pile, respectivement dans le pc et dans le ps du processeur.

- La routine de gestion de l'interruption peut être interrompue par une autre plus prioritaire, dès que les nouveaux pc et ps sont chargés.

§ -2. Les appels systèmes: l'instruction "trap".

Toute une série d'événements peuvent causer le saut du processeur vers une adresse, fonction de cet événement. Cette série comprend la panne de secteur, l'erreur d'adressage, l'erreur de parité,... et l'exécution d'une instruction "trap".

Dès qu'un tel événement se produit, le processeur effectue les mêmes opérations que pour les interruptions. Dans le cas qui nous intéresse d'une instruction "trap",

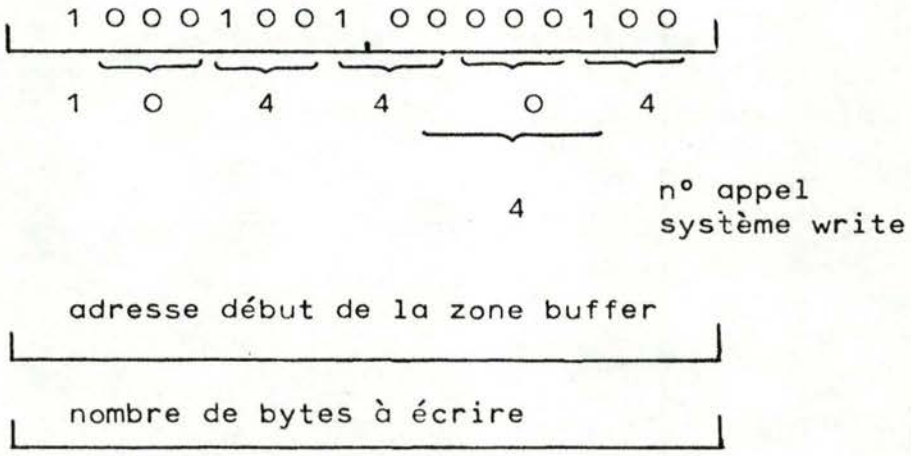
le processeur trouve son pc et son ps dans un vecteur situé à partir de l'adresse 34 de la mémoire centrale.

Remarque: on peut trouver en annexe, une copie de l'appendice B du "Processor Handbook" du PDP 11/40.

L'instruction "trap", en elle-même, possède un paramètre allant de Ø à 377 (en octal), permettant de spécifier de quel appel système il s'agit. (exemple pour UNIX: read = 3, write = 4). Les autre paramètres nécessaires au fonctionnement de l'appel sont placés juste après l'instruction.

exemple: sys write; buffer; nbytes

- l'instruction sys est synonyme de l'instruction "trap".
- cette instruction sera traduite en mémoire de la façon suivante:



3.1.3. LES INTERRUPTIONS DU POINT DE VUE SOFTWARE.

Nous avons vu, que pour exécuter la routine de service de l'interruption, le processeur chargeait son pc et son ps grâce aux informations trouvées dans un vecteur d'interruption dont l'adresse lui est fournie lors de l'interruption.

Analysons le fichier /usr/sys/conf/1.s où les différents vecteurs d'interruption sont définis. (Ce fichier se trouve en annexe du travail à la page 122) .

Ce fichier est, en fait, un programme écrit en assembleur, qui sera assemblé et lié avec les autres programmes de UNIX.

Le but du programme est de décrire les vecteurs d'interruption et de "trap" dans le bas de la mémoire et de faire démarrer l'exécution des différentes routines de traitement de l'interruption.

Afin d'y voir plus clair, analysons le cas d'une interruption lancée par le lecteur de ruban perforé.
L'interruption est accompagnée de l'adresse du vecteur d'interruption, en l'occurence 70.
A l'adresse 70, on trouve

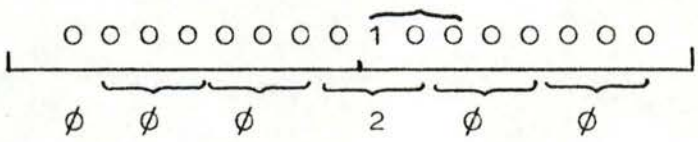
pcin; br4

pcin est une adresse définie dans le programme. Cette adresse est alors chargée dans le pc du processeur.

De plus, le ps est chargé avec la valeur br4, qui est définie par l'instruction (équivalente à un "equate")

br4 = 2ØØ

Le ps devient donc



On remarquera que les bits 7, 6 et 5 de ce ps sont positionnés à (100)2, c'est-à-dire 4, qui représente donc la priorité affectée au processus.

Dès lors, le processeur se branche à l'adresse "pcin" avec la priorité 4.

A cette adresse, on trouve

pcin; jsr rØ , call; -lpint.

Cette instruction "jump to subroutine" provoque le sauvetage du registre Ø sur la pile, le chargement dans rØ de l'adresse de l'instruction suivante et le chargement dans le pc (program counter) de l'adresse de la routine à exécuter (call).
Dans ce cas, cette instruction provoque le branchement vers la routine call, le registre Ø pointant vers le mot contenant -lpint, qui est l'adresse de la routine de traitement de l'interruption.

La routine call:

Après avoir sauvé les informations nécessaires pour le retour et le passage des paramètres à une routine écrite en C (ps, reg1,...), la routine call provoque l'appel de la routine dont l'adresse est contenue dans le registre Ø, c'est-à-dire la routine lpint.

La routine lpint:

lpint est une des routines du "driver" de l'imprimante. Elle a pour but de reconnaître de quelle interruption il s'agit en analysant le mot d'état de l'imprimante situé à l'adresse Ø777514 et, dès lors, d'effectuer une série d'actions en relation avec d'autres routines du driver. (voir fichier /usr/sys/dmr/lp.c).

Remarque: lorsque les interruptions de plusieurs périphériques sont gérées par la même routine, le numéro du périphérique doit être passé comme paramètre à cette routine. Pour ce faire, le ps est chargé, non seulement avec la priorité du processeur, mais aussi avec le numéro du périphérique qui s'inscrit dans les bits réservés au code condition du ps (bits 3 à Ø).

La routine call décode ce ps et passe le numéro du périphérique comme paramètre à la routine du driver.

3.1.4. LES APPELS SYSTEMES DU POINT DE VUE SOFTWARE.

Une instruction "sys" contient logiquement trois informations:

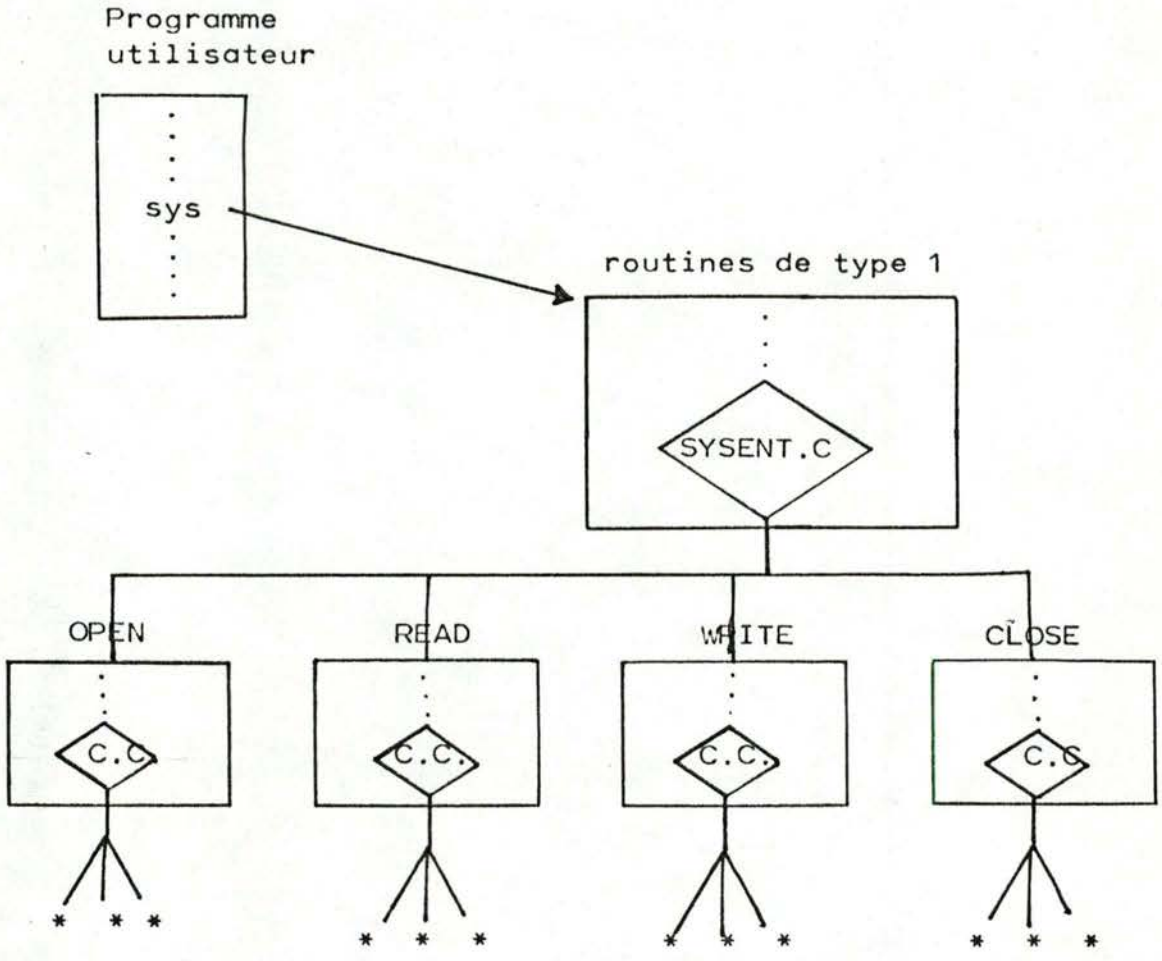
- il s'agit d'un appel système
- cet appel système est d'un type bien particulier
- il concerne un périphérique donné

De même, les routines qui sont exécutées lors d'un appel système peuvent être rassemblées en trois groupes distincts:

- celles communes à tous les appels systèmes
- celles spécifiques à un type d'appels systèmes, mais exécutées quel que soit le périphérique concerné
- celles spécifiques à un type d'appels systèmes et à un périphérique particulier.

L'ensemble des routines pour un périphérique donné forme le "driver" de ce périphérique.

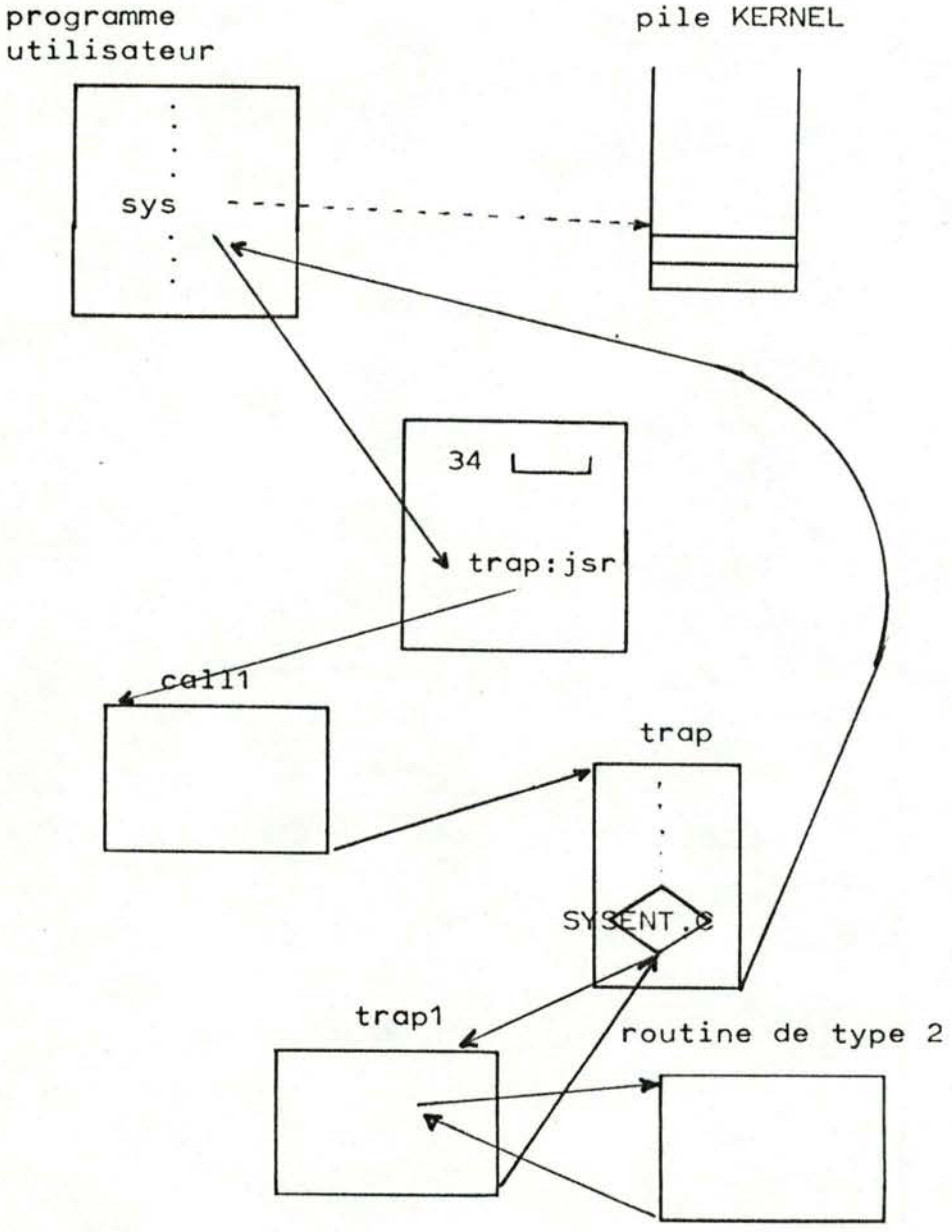
Le schéma suivant peut donc être établi:



(*): routines de type 3: routines du driver correspondant à l'appel système et au périphérique spécifiés dans l'instruction "sys".

§ -1. Partie commune à tous les appels systèmes.

Les routines appartenant à cette partie se présentent comme suit:



Nous avons déjà vu que, lors d'un appel système, le processeur charge le mot d'adresse 34 dans son pc et le mot d'adresse 36 dans son ps.

A l'adresse 34 définie dans le fichier /usr/sys/conf/1.s, on trouve

```
trap ; br7 + 6
```

Le processeur se branche donc à l'adresse trap avec une priorité égale à 7 et les bits du code condition positionné à 6, afin de pouvoir reconnaître s'il s'agit d'un "trap" du type "system entry".

L'adresse trap se trouve dans le fichier m45.s. Après avoir exécuté certaines opérations concernant la gestion de la mémoire, le processeur exécute l'instruction

```
jsr    rØ, call1  ; -trap
```

L'exécution de cette instruction provoquera

- le sauvetage du registre Ø sur la pile
- le transfert dans rØ de l'adresse du mot suivant le jsr (le rØ pointe donc vers le mot contenant l'adresse de la routine -trap).
- le branchement à la routine spécifiée par le deuxième opérande: soit call1..

La routine call1

La sous-routine call1 sauve sur la pile les différentes informations nécessaires à l'appel de la routine trap

et au passage des paramètres et appelle cette routine grâce à l'instruction

```
jsr        pc    ,*(rØ)+
```

La procédure trap (dev,sp,r1,nps,ro,pc,ps)

La procédure trap est écrite en C et peut être trouvée dans le fichier /usr/sys/ken/trap.c.

Les arguments de cette procédure sont les mots sauvés sur la pile "kernel" par le hardware et le software, pendant la prise en charge du trap. Leur séquence est définie par le hardware et les détails de la séquence d'appel des procédures en C.

Une instruction "switch" guide le traitement selon la valeur du paramètre "dev" (nom donné au paramètre correspondant aux bits du code condition du ps).

Ici, nous nous intéressons au cas où "dev" =6, c'est-à-dire lorsqu'il s'agit d'un trap de type "system entry".

L'instruction callp = & sysent fuiword (pc-2) & 077 place dans la zone callp l'adresse d'une ligne du tableau sysent, qui contient l'adresse de la routine de gestion de l'appel système (read, write, open, close, seek,...).

Le tableau sysent est défini dans le fichier sysent.c, comme étant un tableau à une dimension et est redéfini dans trap.c comme un tableau de structure composée de deux entiers.

Voyons donc l'effet de cette instruction qui montre bien comment, en C, on peut écrire de façon très compacte

pc-2 pointe vers l'instruction sys elle-même, puisque pc est le paramètre qui correspond au PC sauvé par le hardware lors de l'instruction sys. Il pointait alors vers l'instruction suivant le sys.

fuiword (pc-2) Comme la routine fuiword() permet de prendre un mot dans un espace mémoire réservé à un utilisateur à l'adresse donnée par l'argument, fuiword (pc-2) fournit, dès lors, le code de l'instruction, qui après une opération and (&) avec la valeur Ø77 fournit le numéro de l'appel système (bits Ø à 5 de l'instruction sys).

L'instruction se résume dès lors à:

callp = &sysent (numéro de l'appel système)

callp contiendra donc bien l'adresse de la ligne du tableau sysent spécifiée par le numéro de l'appel système contenu dans l'instruction sys.

Après avoir exécuté cette instruction, le processeur
-initialise l'indicateur d'erreur u.u-error à Ø
-garnit le tableau u.u_arg[ ] avec les paramètres se trouvant juste après l'appel système.
-incrémente le pc sur la pile de façon qu'au retour, on reprenne l'exécution juste après la liste des

paramètres de l'appel système.

-appelle la routine trap 1 avec, comme argument, l'adresse de la routine système qu'il faut appeler. (cette adresse est fournie par le deuxième entier de la ligne du tableau sysent pointée pour callp.)

Au retour, on teste l'indicateur d'erreur u.u error. S'il est > Ø et < 1ØØ, on positionne le bit Ø du ps se trouvant sur la pile et on retourne u.u error via le registre Ø.

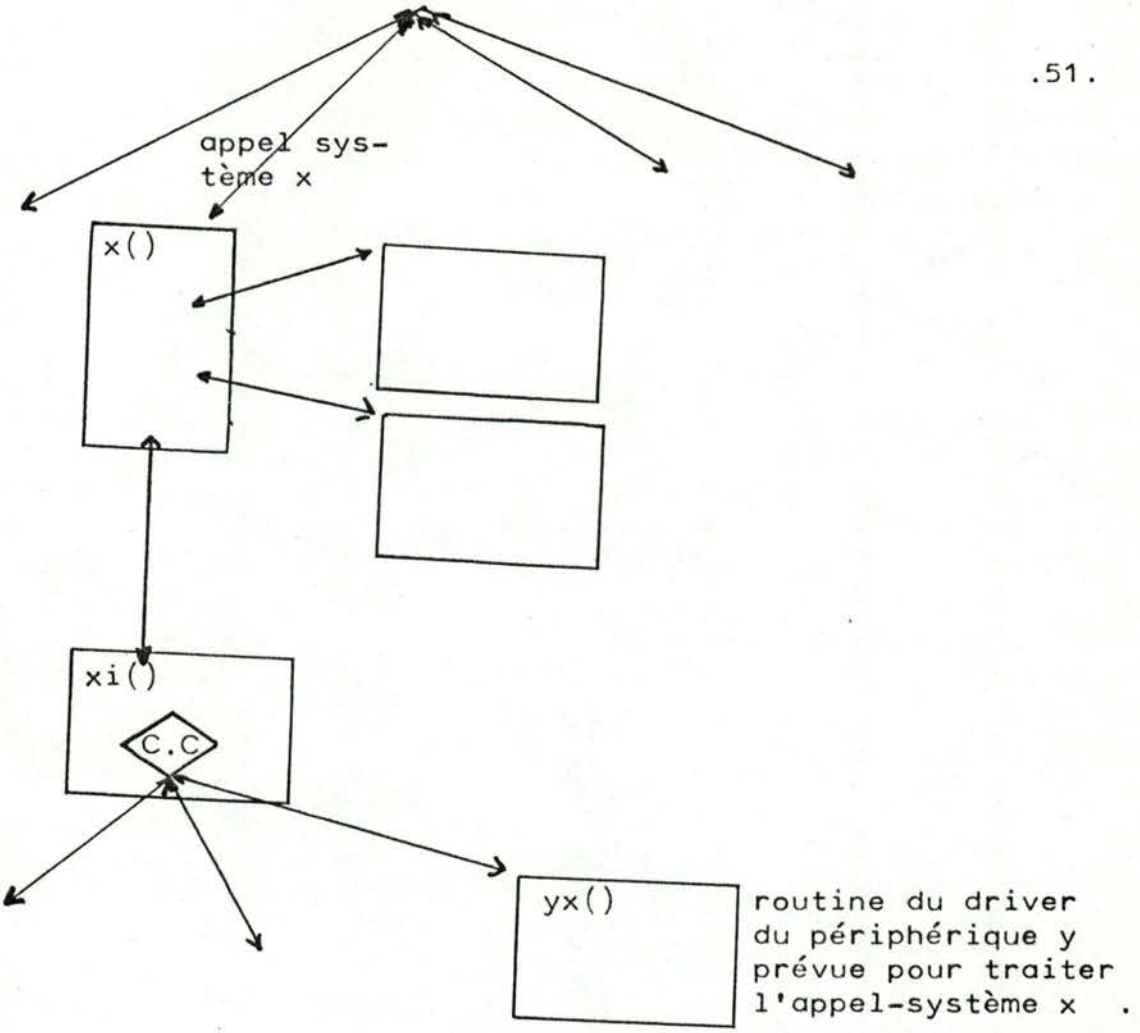
La routine trap 1 (adresse)

La routine trap 1 a pour but principal, d'appeler la routine dont elle a reçu l'adresse comme paramètre.

§ -2. Partie propre à un type d'appels systèmes.

Toutes les routines que l'on a vues précédemment, étaient communes à tous les appels systèmes. Elles servaient, en quelque sorte, à prendre en charge l'appel système, le décoder, préparer ses paramètres et sauver les informations nécessaires au retour, avant d'appeler une routine spécialisée pour le traitement d'un appel système donné.

Généralement, les routines composant cette deuxième partie correspondent au schéma suivant



Le grand nombre d'appels systèmes nous empêche de passer en revue l'ensemble des routines composant cette deuxième partie, bien qu'elles soient généralement très courtes.

Voyons quelles sont les opérations effectuées à partir de l'appel des routines read() et write() par call1 . Les routines read() et write() ont à effectuer des opérations similaires et peuvent, dès lors, partager une grande partie du code.

C'est pourquoi, les appels systèmes provoquent tous deux l'appel de la routine rdwr.

La routine rdwr() a pour but de

- convertir l'identification du fichier dans le programme utilisateur en un pointeur dans la table "file", dont l'entrée correspondant au fichier a été créée par l'appel système open.

- positionner des valeurs standards dans la structure u (voir fichier /usr/sys/user.h en annexe). Les variables u.u base, u.u count et u.u_segflg sont positionnées avec les paramètres respectifs, soit u.u arg[0], u.u_arg[1] et Ø.

- appeler readi ou writei, selon le cas.

- au retour, mettre à jour l'"offset" du fichier et positionner la valeur retournée au programme utilisateur avec le nombre de caractères qui ont été transférés.

Les routines readi() et writei() ont comme fonction d'orienter le traitement vers la routine du driver correspondante. Trois informations sont nécessaires à ces routines pour trouver l'adresse de la routine du driver à appeler, grâce à la consultation du fichier /usr/sys/conf/c.c (voir en annexe).

Ces informations sont:

- il s'agit d'un périphérique "block" ou "character".
- le "major number device".
- le type d'appel système (read, write, seek,...).

Le fait qu'il s'agisse d'un périphérique "block" ou "character" permet de sélectionner l'une des deux tables définies dans /usr/sys/conf/c.c.

Le "major number device" sélectionne la ligne correspondante de la table.

Le type d'appel système spécifie la colonne à considérer. On peut, dès lors, appeler la routine du driver dont l'adresse a été trouvée de cette manière.

§ -3. Les routines du driver.

Un "driver" est donc un ensemble de routines, dont le but est d'exécuter les opérations relatives à un appel système et qui sont propres à un périphérique donné.

exemple: placer un caractère dans le buffer de l'imprimante est une opération qui sera effectuée par la routine lpoutput du driver de l'imprimante.

Les différents "driver" se trouvent tous dans le directoire /usr/sys/dmr.

Les informations qu'une routine d'un driver possède sont les arguments de l'instruction "sys", qui ont été placés dans des entiers de la structure u (u.u base, u.u count,...).

Grâce à ces informations, les routines du driver doivent provoquer le fonctionnement du périphérique qu'elles "conduisent". Pour ce faire, elles doivent gérer des variables internes (buffer, switch,...), mais elles ont surtout comme but de positionner correctement les registres de commandes du périphérique.

Ces registres de commandes sont situés dans le haut de la mémoire (voir liste en annexe). Ils sont généralement composés d'un mot d'état du périphérique et d'un

ou plusieurs buffers, où les caractères sont, soit placés pour envoyer au périphérique, soit reçus du périphérique selon qu'il s'agit d'une opération de lecture ou d'écriture.

Les registres de commandes sont modifiés, non seulement par les routines du driver, mais aussi par le périphérique lui-même.

3.2. LES PROBLEMES RELATIFS A L'ECRITURE D'UN DRIVER.

3.2.1. LES DIFFERENTS PROBLEMES QUE L'ON PEUT RENCONTRER.

On a vu que les différents paramètres des appels systèmes sont stockés dans des zones connues en mémoire (par exemple: u.u base, u.u count,...).

Les problèmes rencontrés lors de l'écriture d'une routine de driver sont généralement de quatre ordres:

1. l'activation des différents organes du périphérique.

2. la bufférisation des informations, en vue de faire le lien entre le buffer de l'utilisateur et celui du périphérique, en tenant compte des valeurs des paramètres de l'appel système.

3. la synchronisation des différentes routines du driver.

4. la détection de "zones critiques" dans le déroulement d'une routine.

§ -1. L'activation des différents organes du périphérique.

Ici, le problème est de dégager la structure de fonctionnement du périphérique concerné. Pour cela, on n'a à sa disposition que la description des registres de commandes du périphérique qui sont adressables comme s'il s'agissait de mots situés dans le haut de la mémoire.

Généralement, un de ces registres est le "status" du périphérique. Il est composé de 16 bits, qui ont chacun leur signification propre. Il est nécessaire, non seulement de comprendre la signification de chacun de ces bits, mais aussi de voir leur influence possible sur la valeur d'autres bits du status et sur le comportement du périphérique, selon qu'un autre bit est positionné ou non.

Il s'agit certainement ici du problème le plus difficile à surmonter lorsqu'on écrit un driver, et cela pour plusieurs raisons:

- on n'a à sa disposition qu'une description du status et rarement la logique de fonctionnement du périphérique.
- la documentation relative aux périphériques n'est pas toujours suffisante pour pouvoir dégager une structure de fonctionnement global du périphérique, complète et sans

ambiguîté.

- il est très malaisé de tester correctement le comportement du périphérique en fonction d'une valeur donnée du "status", car le moment où les différents bits sont positionnés dépend de l'occurence d'un événement. Cette occurence étant très aléatoire (fin de la lecture d'une carte, moment où l'on appuie sur la touche "return" d'une télétype).

§ -2. La bufférisation des informations.

Considérons un périphérique "character". L'ordre de lecture ou d'écriture dans un programme utilisateur comporte deux paramètres, qui stipulent l'adresse début d'une zone et le nombre de caractères de cette zone. Celle-ci est destinée à recevoir les informations du driver dans le cas d'un ordre "read", ou de les fournir à celui-ci lors d'un ordre "write".

Cependant, le périphérique ne peut traiter qu'un seul caractère à la fois. Dès lors, le driver devra effectuer la gestion des différents caractères en interaction avec le buffer de l'utilisateur et faire en sorte de retourner au programme le nombre de caractères réellement lus ou écrits .

§ -3. La synchronisation des différentes routines du driver.

Un driver est composé de plusieurs routines. Celles-ci doivent pouvoir se synchroniser, c'est-à-dire s'endormir ou s'éveiller, selon qu'un événement s'est produit ou

qu'une autre routine arrive à la fin de son travail ou non.

De nombreuses routines de gestion d'un périphérique sont en relation avec d'autres, et, notamment, avec celle qui gère les interruptions de ce périphérique.

Par exemple, on rencontre souvent le cas d'une routine de lecture qui transfère des caractères à partir d'une file d'attente vers le buffer utilisateur et qui, lorsque cette file devient trop petite, lance la lecture physique d'un caractère et s'endort.

D'autre part, la routine d'interruption est activée lorsque le caractère a été lu. Elle place ce caractère dans la file d'attente et relance une lecture physique et cela, jusqu'à ce que la file atteigne une longueur suffisante.

A ce moment, la routine d'interruption ne lance plus de nouvelles lectures, mais réveille la routine de lecture qui recommence à transférer les caractères lus vers le buffer utilisateur.

§ -4. La détection de "zones critiques" dans le déroulement d'une routine.

Lors de l'exécution d'une routine, il est parfois indispansable que l'on ait la certitude que l'exécution ne sera pas interrompue par une autre routine de priorité donnée.

Dès lors, il est nécessaire de modifier la priorité du processeur pendant l'exécution de ces zones, dites "critiques", et, par la suite, de ramener cette priorité à sa valeur initiale afin de na pas gêner l'exécution d'autres routines, comme celles des interruptions, par exemple, qui doivent parfois recevoir le processeur dans un laps de temps assez limité.

3.2.2. LES OUTILS MIS A NOTRE DISPOSITION POUR ECRIRE UN DRIVER.

Un grand nombre des problèmes explicités plus haut peuvent être résolus en employant une série de routines de UNIX, qui ont pour but d'aider le programmeur dans son travail. Ces routines sont: passc, cpass, wakeup, sleep, spli, getc, putc,...

§ -1. Les routines passc(c) et cpass().

Les routines passc et cpass travaillent sur le buffer utilisateur, dont l'adresse est donnée par u.u_base.

Elles ont pour but, soit de prendre le caractère suivant dans le buffer utilisateur (cpass), soit de placer un caractère dans ce buffer (passc(c)), de mettre à jour u.u_count et de renvoyer la valeur -1, lorsque toute la zone utilisateur a été traitée. Cpass retourne également le caractère lu dans le buffer utilisateur.

Ces deux routines peuvent être trouvées dans le fichier /usr/sys/ken/subr.c (voir annexe).

Grâce à passc(c) et cpass(), le programmeur ne doit ni mettre à jour u.u_count, ni accéder lui-même à une zone réservée par un utilisateur.

§ -2. Les routines getc() et putc().

S'il le désire, le programmeur peut utiliser une file d'attente gérée de façon FIFO par les routines getc et putc. Pour ce faire, le programmeur doit réserver dans son programme une structure qui servira d'entête de file d'attente. Cette structure est composée de trois entiers: un compteur de caractères, l'adresse du premier caractère et l'adresse du dernier caractère.
Les files d'attente sont composées d'une série de blocs chaînés entre eux; chaque bloc pouvant contenir 6 caractères. Il existe une chaîne de blocs libres (cfreelist) pour les différentes files d'attente et, quand c'est nécessaire, un bloc est prélevé dans la liste et chaîné aux autres blocs de la file demandeuse d'emplacements.

La routine getc prend donc un caractère dans la file indiquée par le paramètre, qui est l'adresse de l'entête de la file d'attente. La routine renvoie le caractère prélevé dans la file, excepté si le compteur de caractères est nul. Dans ce cas, elle renvoie la valeur -1. De plus, si possible, elle libère un bloc en le rattachant à la chaîne des blocs libres (cfreelist).

Putc(c,& ...) a pour but de placer le caractère c dans la file spécifiée par le second paramètre, d'allouer un nouveau bloc pour la file, si le dernier s'avère rempli,

d'initialiser une file d'attente dans le cas où une file est vide (lors de l'exécution du premier putc relatif à une entête).

Ces deux routines peuvent être trouvées dans le fichier /usr/sys/conf/m45.s.

L'utilité de ces routines est évidente. Elles permettent au programmeur de synchroniser ses routines d'une façon plus optimale; une routine place des caractères dans la file d'attente, une autre puise dans cette file. On remarque donc le parallélisme dans l'exécution des différentes routines.

Néanmoins, le programme utilisant ces routines devra tester l'état de la file (nombre de caractères de cette file) et, dans certains cas, prendre des décisions, comme celle d'endormir ou de réveiller une autre routine.

§ -3. Les routines sleep() et wakeup().

Ces deux routines permettent de synchroniser l'exécution de processus. Un processus appelle la routine sleep (id, pri). Cet appel endort le processus, jusqu'à ce qu'une autre routine appelle le wakeup avec un paramètre dont la valeur est la même que celle du paramètre id, spécifié lors de l'appel sleep.

Le paramètre pri donne la priorité avec laquelle le processus sera placé dans la file d'attente lors du prochain wakeup.

§ -4. Les routines spl$_i$ ().

Les routines spl$\emptyset$, spl1, spl4, spl5, spl6 et spl7 ont pour but de positionner la priorité du processeur à la valeur $\emptyset$, 1, 4, 5, 6 ou 7, selon la routine qui est appelée. Ces routines sont d'ailleurs très courtes: deux ou trois instructions assembleur. On peut consulter ces routines en lisant le fichier m45.s (voir annexe).

Quatrième partie:

UN EXEMPLE: L'ECRITURE D'UN DRIVER POUR UN LECTEUR DE CARTES.

QUATRIEME PARTIE: UN EXEMPLE: L'ECRITURE D'UN DRIVER POUR UN LECTEUR DE CARTES.

Un driver est donc un fichier composé de différentes routines et de déclarations de variables globales à ces routines.

Le travail d'écriture d'un driver comprend généralement quatre parties :

- écrire le driver lui-même
- recompiler le système en y incluant le driver
- tester le système incluant le driver
- écrire un ou plusieurs programmes utilitaires qui aideront à se servir plus aisément du périphérique.

Voyons donc les différents problèmes qui se posent lors de l'écriture d'un driver pour le lecteur de cartes CR11 de DEC (Digital Equipment Corporation).

4.1. LE LECTEUR DE CARTES CR11.

Une copie des pages relatives au lecteur de cartes CR11 dans le "Peripheral Handbook" de DEC peut être trouvée en annexe.

Cependant, quelques commentaires sur les registres de communications du lecteur de cartes sont nécessaires pour la compréhension de la suite de l'exposé. Ces registres sont au nombre de trois:

- le registre "status"
- le registre "buffer" non compressé
- le registre "buffer" compressé.

Les registres "buffer" sont destinés à recevoir les caractères en provenance du lecteur de cartes.
Chaque caractère lu peut être utilisé à partir d'un des deux buffers au choix. Seul le mode de représentation du caractère change.

Le registre "non compressé" représente un caractère par ses douze bits de poids faible. Chaque bit indique si un trou a été perforé dans la ligne correspondante de la carte.

Le registre "compressé", représente un caractère par ses huit bits de poids faible. Ces bits sont positionnés selon le tableau suivant:

BIT	ZONE (ligne) de la carte
7	12
6	11
5	Ø
4	9
3	8
2-1-Ø	ØØØ si zone 1-7 ØØ1 si zone 1 Ø1Ø si zone 2 Ø11 si zone 3 1ØØ si zone 4 1Ø1 si zone 5 11Ø si zone 6 111 si zone 7

Le registre d'état (status) du lecteur de cartes est composé de seize bits, ayant chacun leur signification propre. Quatre de ces bits provoquent le lancement d'une demande d'interruption. Ces bits sont ceux numérotés 15, 14, 7 et 10.

- le bit 15: error: ce bit est positionné lorsqu'une erreur se produit. Plusieurs types d'erreurs peuvent se produire et ce bit est positionné en même temps qu'un autre qui spécifie de quel type d'erreur il s'agit . Ces bits sont au nombre de quatre .

  Le bit 13: (Hopper check) est " ON " lorsqu'il n'y a plus de cartes dans le lecteur ou s'il y en a trop à la sortie .

  Le bit 12: (Motion check) nous averti qu' une erreur s'est déroulée lors de la lecture d'une carte .

  Le bit 11: (Timing Error) sera positionné chaque fois que l'on n'accèdera pas à un caractère lu avant l'arrivée du caractère suivant.

  Le bit 8: (Reader ready status) qui est positionné lorsque le lecteur est en position off-line. Pousser le bouton " stop " provoquera donc la mise " off-line " du lecteur et le positionnement des bits 8 et 15 .

- le bit 14 (Card done): indique la fin de la lecture d'une carte.
- le bit 7 (Column done): indique qu'un caractère est prêt dans les registres buffer.
- le bit 10 (Reader to on line): est positionné quand le lecteur est on-line.

Les bits 6 et Ø sont également importants. On positionne le bit 6 afin de permettre aux bits 15, 14, 7 et 10 de provoquer une demande d'interruption. Le bit Ø sera mis à 1 lorsque l'on désirera initialiser la lecture d'une carte.

4.2. LES DIFFERENTS APPELS SYSTEMES.

Lorsque l'on veut écrire un driver, il faut décider quels appels systèmes seront traités par ce driver et quels sont ceux qui ne les eront pas.

Ici, il paraît assez évident que, seuls les appels "open", "read" et "close" sont intéressants; on ne voit pas très bien quelles seraient les opérations à effectuer pour gérer un autre appel système comme "write", par exemple.

Notez que certains appels systèmes pourraient être implémentés. Par exemple, l'appel "seek" pourrait provoquer la lecture d'un certain nombre de cartes sur lesquelles aucun traitement ne serait effectué, cependant le temps mécanique de lecture ne serait pas gagné. De plus, rares sont les programmes qui utiliseraient cet appel système.

4.3. LE DRIVER DU LECTEUR DE CARTES.

Le driver du lecteur de cartes est composé:
- de lignes interprétées par un précompilateur de C

- de la déclaration de différentes zones de mémoires globales à toutes les routines du driver.
- des routines proprement dites: cropen()
  crint()
  crread()
  crclose()
  crinput()

4.3.1. LES "INCLUDE" ET LES "DEFINE"

Le premier caractère du fichier, seul sur la première ligne, définit le caractère de contrôle pour un précompilateur du langage C. Ici, ce caractère est # .

Dès lors, toutes les lignes du driver commençant par # sont interprétées par le précompilateur.

Ce précompilateur admet les instructions "define" et "include". Par exemple, l'instruction "define ZERO Ø" provoquera le remplacement de chaque "ZERO" rencontré dans le programme par le littéral Ø. De cette façon, des valeurs constantes employées dans certaines routines peuvent être appelées par un nom plus expressif, et la modification d'une seule instruction "define" suffira pour corriger un ensemble d'instructions.

L'instruction "include" suivie d'un nom de fichier, provoque l'insertion du fichier nommé dans le programme. De ce fait, une zone déclarée dans un fichier pourra être employée dans le driver, en y incluant ce fichier par une

instruction "include".

Dans notre cas, 21 lignes seront interprétées par le précompilateur de "c". Ces 21 lignes sont:

```
#INCLUDE "../PARAM.H"
#INCLUDE "../CONF.H"
#INCLUDE "../USER.H"

#DEFINE         CRADDR          0777160

#DEFINE         READ            01
#DEFINE         EJECT           02
#DEFINE         IENABLE         0100
#DEFINE         DONE            0200
#DEFINE         READY           0400
#DEFINE         BUSY            01000
#DEFINE         ONLINE          02000
#DEFINE         TIMING          04000
#DEFINE         MOTION          010000
#DEFINE         HOPPER          020000
#DEFINE         CDONE           040000
#DEFINE         ERROR           0100000

#DEFINE         CLOSED          0
#DEFINE         WAITING         1
#DEFINE         READING         2
#DEFINE         EOF             3

#DEFINE         CRPRI           30
```

On remarquera ici que l'on inclue les fichiers ../param.H, ../conf.H, ../user.H, c'est-à-dire: /usr/sys/param.H
/usr/sys/conf.H
/usr/sys/user.H
puisque le driver aura le nom /usr/sys/dmr/cr.c et que le directoire courant du programmeur lors de la compilation sera /usr/sys/dmr (si ce n'est pas le cas, une erreur surgira lors de la précompilation), et que .. spécifie le directoire parent du directoire courant, soit /usr/sys .

define CRADDR Ø777160.

Ø777160 est l'adresse début du vecteur de communication du lecteur de cartes.

Vient ensuite, une série de 12 instructions "define" qui définissent des constantes, qui sont telles que seul un bit est positionné à 1. Ces constantes serviront de masque lors d'opérations logiques avec le status. Selon le numéro d'ordre du bit positionné, la constante a un nom identique à celui donné par DEC au bit correspondant du status.

exemple: define TIMING Ø4ØØØ

Ø Ø Ø Ø 1 Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø = Ø4ØØØ

bit n° 15 11 Ø

Le bit 11 est positionné. Le bit 11 du status du lecteur de cartes est "on" lorsqu'il y a un "TIMING ERROR".

On définit, par la suite, 4 valeurs que prendra un indicateur (crstate), qui définira la situation du lecteur de cartes. Ces situations sont les suivantes:

- le lecteur de cartes peut être fermé (CLOSED), c'est-à-dire qu'aucun programme ne l'utilise et qu'il est libre.
- il peut être en attente (WAITING), c'est-à-dire qu'il est ouvert, mais que la première carte n'a pas encore été lue. Nous verrons plus tard l'utilité de cette situation.
- il peut être en lecture (READING), c'est-à-dire qu'il est ouvert et qu'au moins une carte a été lue.
- s'il est dans l'état fin de fichier (EOF), cela signifie qu'une erreur a été détectée et donc que le

cycle de lecture se termine. En effet, seule une erreur peut forcer le driver lui-même à arrêter son exécution. La fin de fichier devra être détectée par le programme utilisateur.

define CRPRI 3Ø définit la priorité avec laquelle un processus endormi sera placé dans la file des processus en attente d'exécution.

4.3.2. LA DECLARATION DES VARIABLES, DES TABLEAUX ET DES STRUCTURES.

Char array [256] déclare un tableau array de 256 caractères dont les valeurs sont données en octal .
Ce tableau servira à traduire le code compressé reçu du lecteur de cartes en code ASCII. Tous les caractères inconnus dans le code ASCII recevront une valeur de Ø377, valeur non reprise dans la liste des caractères ASCII.

Char Buffer [81] déclare un tableau buffer destiné à recevoir les informations provenant de la lecture d'une carte.

L'entier I sert d'indice dans le vecteur buffer.

On définit ensuite une structure composée de trois entiers:

- CRSR (card reader status register)
- CRNCBR (card reader non compressed buffer register)
- CRCBR (card reader compressed buffer register)

Cette structure sera appliquée aux registres de communications du lecteur de cartes, qui se trouvent à l'adresse CRADDR. cr11.crsr→CRADDR pointe donc vers le registre d'état du lecteur de cartes.

L'entier crstate recevra les différentes valeurs définissant la situation du lecteur de cartes (CLOSED, WAITING, READING, EOF).

4.3.3. LES ROUTINES DU DRIVER.

Comme on l'a déjà dit, le driver est composé de 5 routines:

- cropen()
- crint()
- crread()
- crclose()
- crinput()

Afin de comprendre le fonctionnement de ces routines, la démarche suivante sera employée. Voyons donc d'abord les opérations que doivent effectuer chacune de ces routines. Ensuite, nous pourrons les analyser plus en détails, afin de voir de quelle façon ces opérations peuvent être programmées. Cependant, comme l'interaction entre ces différentes routines est importante, il sera important d'expliciter le fonctionnement global du driver.

§ -1. Les routines en général.

- cropen()

Cropen() est appelée lors d'un appel système open où

le fichier /dev/cr a été spécifié. Cette routine a pour but de tester si le lecteur de cartes est déjà occupé par un programme utilisateur ou non.

Dans le cas où il est déjà occupé, il doit prévenir le programme sinon, la lecture d'une carte sera initialisée.

- crint():

La routine crint() est appelée lorsqu'une interruption en provenance du lecteur de cartes est prise en charge par le processeur.
Crint() doit, par analyse de certains bits du "status" reconnaître de quel type d'interruption il s'agit afin de prendre les mesures nécessaires au traitement de ce type d'interruption.

Quatre types d'interruption différents peuvent survenir à partir d'un lecteur de cartes cr11:

1. une colonne de la carte a été lue et le caractère est disponible dans le buffer.
2. une carte vient d'être lue complètement.
3. une erreur survient lors de la lecture d'une carte.
4. le lecteur de cartes passe de l'état "off-line" à l'état "on-line", il s'agit de l'interruption "TRANSITION-TO-ON-LINE".

Dans le premier cas, le caractère doit être sauvé afin que le buffer soit libre pour la réception du caractère suivant.

Si une carte a été complètement lue, il faut réveiller la routine crread() qui a lancé cette lecture et qui s'est endormie en attendant qu'elle l'accomplisse. S'il s'agit de la lecture de la première carte, c'est la routine cropen()

qu'il faudra réveiller afin qu'elle puisse terminer son exécution.

Lorsqu'une erreur survient, un message doit être envoyé et l'état du lecteur de cartes doit être positionné à EOF.

S'il s'agit d'une interruption "TRANSITION-TO-ON-LINE", on devra réveiller la routine crinput qui s'était peut-être endormie parce que le lecteur de cartes ne s'était pas avéré prêt lorsqu'une lecture a voulu être lancée.

- crread():

La routine crread(), appelée lors d'un appel système READ relatif au lecteur de cartes, a pour but de fournir des caractères au programme utilisateur lorsque celui-ci en demande. Lorsqu'elle n'a plus de caractères à sa disposition, elle doit lancer une nouvelle lecture, afin de recevoir de nouveaux caractères. Cette lecture devra se faire en appelant la routine crinput().
N.B.: après avoir lancé la lecture, elle devra s'endormir an attendant la fin de la lecture; elle sera réveillée par crint().

- crclose():

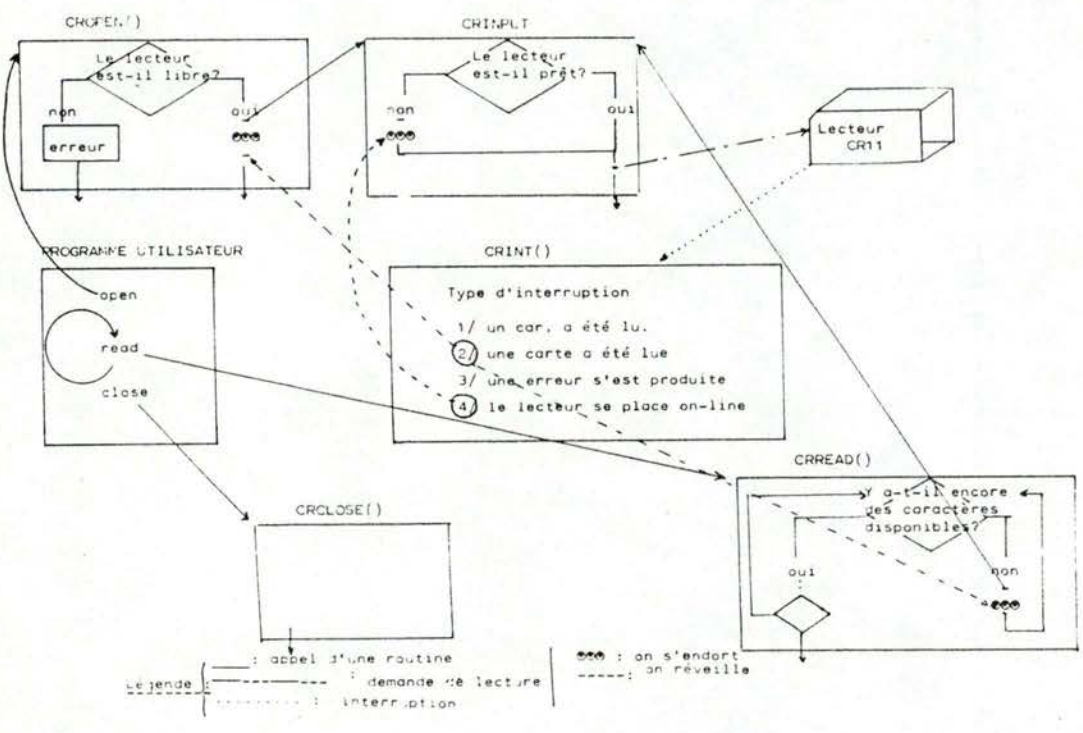
Crclose(), appelée lors d'un appel système CLOSE, placera le lecteur de cartes dans la situation libre (CLOSED) afin de permettre à un autre programme de se servir du lecteur de cartes.

- crinput():

Crinput() est une routine qui doit provoquer la lecture d'une carte si le lecteur est prêt. Si il ne l'est pas, cette routine devra s'endormir et sera réveillée par crinput() lors d'une interruption "TRANSITION-TO-ON-LINE".

On remarquera que, lorsque le driver décide de lire une carte, il essaiera de lire cette carte jusqu'à ce que la lecture soit terminée. Si la lecture est rendue impossible du fait que le lecteur n'est pas prêt (plus de carte, "off-line", en panne,...), le driver patientera jusqu'à ce que cette lecture devienne possible.

§ -2. Schéma général du fonctionnement des routines du driver.



En fait, la routine crread() fournit des caractères à la demande au programme utilisateur et il le fait à partir d'un ensemble de caractères qu'il a reçu à partir de la lecture d'une carte. Quand cet ensemble est vide et que des caractères sont encore demandés, une nouvelle lecture est initialisée.

Ce système se synchronise grâce à crint(), qui réveille les autres routines qui se sont endormies en attente d'un événement.

§ -3. Description des routines du programme cr.c.

- cropen():

```
CROPEN()
{
IF(CRSTATE != CLOSED)
        {
        U.U_ERROR = ENXIO ;
        RETURN;
        }
I = 0 ;
CRSTATE = WAITING ;
CRADDR->CRSR = IENABLE ;
CRINPUT() ;
SLEEP(&CRSTATE,CRPRI) ;
}
```

Dans le cas où le lecteur de cartes n'est pas libre, (CLOSED), on positionne l'indicateur u.u_error à la valeur ENXIO, définie dans le fichier user.h. De cette façon, la valeur -1 sera retournée par l'appel open au programme utilisateur et un message "CANNOT CREATE /dev/cr" s'imprimera à la console qui est à la sortie standard du programme qui a lancé l'open.

Il faut remarquer que CLOSED signifie ∅. De ce fait, lors du premier open du lecteur de cartes après le lancement du système, l'indicateur cr11.crstate sera positionné automatiquement à CLOSED puisque le compilateur C initialise à ∅ toutes les zones déclarées. Par après, cet indicateur sera maintenu par le driver lui-même.

I = ∅: cette assignation initialise l'indice I du vecteur buffer. C'est dans ce vecteur que l'on placera les caractères provenant de la lecture d'une carte.

cr11.crstate = WAITING : ainsi, le lecteur de cartes se trouve dans la situation WAITING: le fichier est ouvert, mais la première carte n'a pas encore été lue.

CRADDR → crsr = IENABLE: grâce à cette instruction, on initialise les bits du "status" à ∅, excepté le bit 6, c'est-à-dire, que l'on se déclare prêt à accepter les interruptions en provenance du lecteur de cartes.

Crinput(): on verra plus tard, que, de toute façon, au retour de cette routine, la lecture d'une carte sera initialisée. C'est donc la routine open() qui provoque la lecture de la première carte.

Sleep (&cr11.crstate, CRPRI): on s'endord après avoir lancé la lecture d'une carte, de façon que l'on ne rende la main au programme utilisateur, que lorsque la lecture de la carte est terminée. En effet, c'est l'interruption de fin de carte qui réveillera la routine open().

- crread():

```
CRREAD()
{

INT Y ;
DO
{
WHILE() > 80 )
        {
        IF(CRSTATE == EOF)
                RETURN ;
       I= 0 ;
        CRINPUT() ;

        SLEEP(&BUFFER,CRPRI) ;
        }
Y = BUFFER[I++] ;
Y =& 0377 ;
}
WHILE(PASSC(ARRAY[Y]) >= 0) ;
}
```

La routine crread() est composée d'une instruction

```
do
.
.
.
```

while (passc(array [y]>=∅); ce qui signifie exécuter les instructions comprises entre le do et le while, exécuter l'appel de la routine passc() avec array[y] comme paramètre et, si cet appel retourne une valeur négative, terminer l'exécution de la routine, sinon recommencer le traitement.

L'entier y, qui est déclaré comme variable locale à la routine, contiendra le caractère suivant. Il sera envoyé dans le buffer utilisateur, grâce à la routine

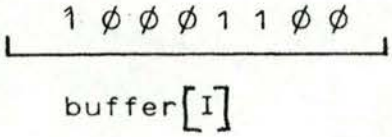
passc(), après avoir subi une traduction en donnant comme paramètre à passc(), non pas l'entier y, mais le caractère situé à l'adresse array[y].

La table array est une table de 256 caractères, dont les valeurs sont telles que le ième caractère de la table contient une valeur dont la signification dans le code ASCII est la même que celle de i dans le code carte compressé.

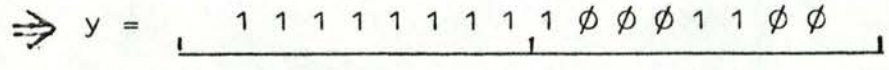
Les instructions comprises entre do et while doivent donc positionner y avec la valeur du caractère suivant. Les instructions sont:

```
        while (I > 80)
                {
                if (cr11.crstate=EOF)
                    return;
        I=Ø;
        crinput( );
        sleep(&cr11.crpri);
                }
y = buffer[y++];
y =& Ø377;
```

Tant que l'indice I ne devient pas supérieur à 80, y est positionné avec le caractère suivant du buffer: y = buffer [y++] et l'indice I est incrémenté.
N.B.: lors de l'assignation d'un caractère (buffer[y]) dans un entier (y), lorsque le bit de poids fort du caractère (dont la longueur est de 1 byte) est positionné, les 8 bits de poids fort de l'entier seront également positionnés.



buffer[I]



Ainsi, seuls les caractères dont la valeur était inférieure à 128 étaient pris en charge correctement. L'instruction.y =& Ø377 trouve donc ainsi sa justification.

Reste donc à analyser le cas où I est supérieur à 80.

```
while(I > 8Ø)
        {
        if(crstate == EOF)
                return;
        I = Ø;
        crinput();
        sleep(&buffer,CRPRI);
        }
```

Il s'agit donc du cas où des caractères doivent encore être envoyés au programme utilisateur, mais où le vecteur a été exploité complètement.

Si le vecteur est dans la situation EOF (si une erreur est survenue lors d'une lecture), on termine l'exécution de crread(), alors que tous les caractères transmis au programme utilisateur. Le nombre de caractères transmis sera retourné de toute façon à ce programme, qui pourra se rendre compte de cette situation.

Comme on a besoin de nouveaux caractères, on va relancer une lecture. Pour cela, on réinitialise l'indice I et on appelle crinput(). On peut, dès lors, s'endormir en attendant que la lecture se termine. Lorsque l'on sera réveillé, on reprendra l'exécution de l'instruction while; le test I >80 sera de nouveau effectué. Si la lecture s'est bien déroulée, le test

sera faux et on pourra positionner y; sinon, dans le cas d'une erreur de lecture, le test sera vrai, ainsi que le test crstate = EOF et on terminera l'exécution du crread.

-crint():

```
CRINT()
{

IF(CRADDR->CRSR&DONE)
        {
        BUFFER[I] = CRADDR->CRCBR ;
        I++ ;
        RETURN ;
        }

IF(C ADDR->CRSR&CDONE)
        {
        IF(CRSTATE == WAITING)
                {
                CRSTATE = READING ;
                WAKEUP(&CRSTATE) ;
                }
        BUFFER[80] = 065 ;
        I=0 ;
        WAKEUP(&BUFFER);
        RETURN ;
        }



IF(CRADDR->CRSR&ERROR)
        {
        IF(CRADDR->CRSR&(MOTION + TIMING))
                {
                U.U_ERROR = EIO ;
                CRSTATE = EOF ;
                }

        CRADDR->CRSR =& 011111 ;
        RETURN ;
        }
WAKEUP(&I) ;
}
```

On distingue les 4 types d'interruptions possibles.

Si le bit DONE est positionné, c'est qu'un caractère a été lu et que l'on doit transférer ce caractère contenu dans le buffer du lecteur de cartes dans le vecteur buffer du driver. Cela est fait par les instructions:

```
buffer[I] = CRADDR → crcbr;
I++;
```

Si le bit CDONE est positionné, c'est qu'une carte a été lue, et que le vecteur buffer contient tous les caractères de cette carte.

Dans le cas où il s'agit de la fin de la lecture de la première carte, c'est-à-dire, si l'indicateur crstate est positionné à WAITING, on réveille la routine cropen et on positionne l'indicateur à READING.

Dans tous les cas,

- on ajoute un caractère à ceux lus sur la carte. La valeur de ce caractère est Ø65. De cette façon, lorsqu'il sera traduit grâce à la table array, il deviendra le caractère NEWLINE du code ASCII. Ainsi, lorsque l'on analysera le fichier résultant de la lecture d'une carte, on remarquera l'image de cartes séparées par des caractères NEWLINE (Ø12).
- on réinitialise l'indice i et on réveille la routine crread. Remarquons que, dans le cas du traitement de la fin de la lecture de la première carte, l'appel de wakeup(&cr11) sera sans effet, puisque crread n'aura pas encore été exécutée et n'a donc pas appelé la routine sleep.

Si le bit ERROR est positionné, c'est donc qu'une erreur s'est produite lors de la lecture d'une carte.

Ces erreurs peuvent avoir des causes très différentes (TIMING ERROR, MOTION ERROR, HOPPER CHECK,...), cependant, seuls les cas irrémédiables seront traités par crint(). Il s'agit des TIMING et des MOTION erreurs. Dans ces cas, on positionne l'indicateur u.u_error à EIO, ce qui provoquera l'impression d'un message et on place le lecteur dans la situation EOF. Dans les autres cas, ces erreurs seront détectées lors du lancement d'une lecture par le fait que le lecteur s'avère off-line. Dans tous les cas, on repositionnera le bit 15 du status (bit ERROR) à Ø, de façon à pouvoir continuer à travailler.

Si les bits DONE, CDONE et ERROR sont simultanément nuls, c'est donc qu'il s'agit d'une interruption "TRANSITION-TO-ON-LINE". Dans ce cas, on réveille crinput().

- crinput().

```
CRINPUT()
{
WHILE(1)
        {

        IF((CRADDR->CRSR & READY) == Ø )
                {
                CRADDR->CRSR++;

                RETURN ;
                }

        CRADDR->CRSR =& Ø175777 ;
        SLEEP(&I,CRPRI);
        }
}
```

- crclose():

```
CRCLOSE()
{
CRSTATE = CLOSED ;
}

CRINPUT()
{
WHILE(1)
        {

        IF((CRADDR->CRSR & READY) == 0 )
                {
                CRADDR->CRSR++;

                RETURN ;
                }

        CRADDR->CRSR =& 0175777 ;
        SLEEP(&I,CRPRI);
        }
}
```

Il suffit de positionner l'indicateur cr11-crstate à la valeur CLOSED, de façon à rendre le lecteur libre pour d'autres utilisateurs.

4.3.4. REMARQUES CONCERNANT L'ECRITURE DU DRIVER CR.C.

§ -1. L'emploi des files d'attente.

Afin de gérer les caractères, le driver utilise un vecteur de 81 caractères (buffer) et un indice (i), alors que la notion de file d'attente peut être utilisée dans UNIX par l'emploi de routines getc() et putc().

Dès lors, pourquoi ne pas utiliser cette facilité? Si on analyse le texte des routines putc() et getc(), on remarque des instructions spl5(), avec comme commentaire: on appelle la routine spl5(), de façon à élever la priorité du processus à 5, ce qui est supérieur aux priorités d'interruption de tous les périphériques "character".

En effet, tous les périphériques "character" reconnus par UNIX, avaient une priorité d'interruption égale à 4.

Malheureusement, le lecteur de cartes est de priorité 6, ce qui interdit l'emploi des routines getc() et putc(). La bufférisation des caractères provenant de la lecture d'une carte doit donc être effectuée par le driver lui-même.

Cependant, un système de bufférisation plus complexe aurait pu être implémenté, de façon à permettre la lecture

d'une carte, en même temps que l'on utilise les caractères de cartes lues précédemment.

Pour cela, une zone mémoire plus grande aurait été nécessaire, les routines auraient dû être plus complexes et plus vastes; le problème des zones critiques se serait compliqué .

Pour toutes ces raisons, nous avons pensé que ce n'était pas nécessaire, d'autant plus que la performance du lecteur de cartes dans le contexte d'un système time-sharing n'est pas primordiale et que cette bufférisation plus complexe n'aurait pas provoqué un comportement très différent du lecteur, celui-ci étant surtout freiné par le temps mécanique de lecture.

§ -2. La longueur d'un enregistrement.

On remarquera également que, malgré la notion de groupe de 80 caractères dans une carte, l'utilisateur peut demander la lecture d'un nombre quelconque de caractères à partir du lecteur de cartes.
exemple: read (filedescp, zone, 17)

Logiquement, nous aurions dû écrire le driver, de manière que, dans ce cas, après 17 caractères vienne s'insérer un caractère NEWLINE.
Cependant, si cette optique avait été choisie, les commandes cat, cp, mv,... de UNIX n'auraient pu être employées, puisqu'elles considèrent un bloc de 512 caractères comme unité de transfert.

Dès lors, le caractère NEWLINE ne serait apparu qu'après 512 caractères et, lors de l'impression sur un terminal ou sur une imprimante, de nombreux caractères auraient été perdus.

Pour ces raisons, l'utilisateur peut écrire un ordre read avec le nombre de caractères qu'il désire, mais le fichier résultant d'une série de lectures sera toujours composé d'ensembles de 80 caractères, séparés par des NEWLINE; seul le nombre de caractères envoyés à la fois au programme dépendra du nombre spécifié dans l'ordre read.

§ -3. Problèmes pratiques.

Le problème le plus important réside dans l'utilisation pratique du lecteur de cartes. Dans le système actuel, un utilisateur possédant une console dans son bureau et qui désire lire un fichier se trouvant sur carte, doit effectuer plusieurs opérations. Il doit se rendre en salle machine et placer son paquet de cartes dans le lecteur. Il doit aussi se faire connaître à une console et demander l'exécution d'un programme de lecture de ses cartes.

Plusieurs problèmes se posent:

1. entre l'instant où le paquet de cartes est déposé dans le lecteur et le moment où le travail est terminé, le lecteur de cartes est rendu indisponible pour d'autres utilisateurs. Il faut remarquer que, généralement, ce temps est allongé, non pas par le temps de lecture des cartes, mais bien par le déplacement de l'utilisateur et son travail à la console.

2. le programme de lecture doit gérer lui-même le problème de la fin du fichier. En effet, aucune notion de fin de fichier n'est associée aux cartes perforées. Nous avons pensé assimiler la fin du fichier au fait qu'il n'y avait plus de cartes dans le lecteur, mais que se passera-t-il pour un fichier qui ne peut être placé entièrement dans le lecteur de cartes?

3. de même que pour le lecteur de cartes, la console employée par l'utilisateur reste également inutilisée pendant le temps de déplacement de celui-ci.

Afin de résoudre tous ces problèmes, la solution suivante a été adoptée.

Nous avons écrit un programme en C: spool.c. Ce programme s'exécute lors du chargement du système et n'a normalement aucune fin. Il a pour but de lire les cartes placées dans le lecteur et de créer des fichiers en fonction de cartes spéciales de commande, et cela, sans intervention d'un utilisateur.

Ainsi, un utilisateur désireux de lire un fichier-carte, entoure ce fichier de deux cartes commandes, place ce paquet dans le lecteur et presse le bouton RESET. Les cartes seront lues et un fichier-disque sera créé. Le nom de ce fichier sera fonction des paramètres donnés par les cartes commandes. Il est donc connu par l'utilisateur.

4.4. LE PROGRAMME SPOOL.C.

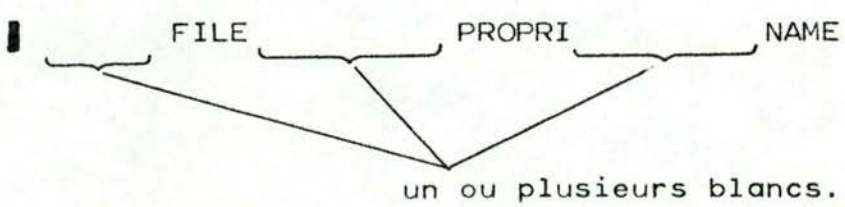
4.4.1. LE TRAVAIL EFFECTUE PAR SPOOL.C.

Spool.c lit les cartes. Lorsqu'il rencontre une carte FILE correcte, il ouvre un fichier-disque dont le nom est /usr/cr/propri.name, propri et name étant les paramètres de la carte FILE. Il continue alors à lire, en écrivant les informations lues dans le fichier ouvert. La rencontre d'une carte END provoque la fermeture du fichier, et, dès lors, on ignorera toutes les cartes jusqu'à la carte FILE suivante.

Notez qu'il n'est pas grave d'oublier sa carte END, à condition que l'utilisateur suivant n'ait pas oublié sa carte FILE.

4.4.2. LES CARTES COMMANDES: FILE ET END.

Chaque fichier-carte doit être précédé d'une carte FILE, dont la description est:



La carte FILE est composée de quatre zones séparées par plusieurs blancs.

1. ▌ est un caractère perforé en colonne 1, inconnu du code ASCII. Pour le créer, on peut employer la touche "multi-punch" du perforateur de cartes et frapper plusieurs caractères dans la colonne 1.

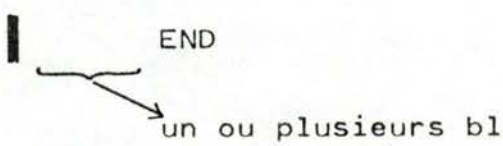
2. FILE: une carte commençant par un caractère inconnu et comportant ensuite le mot FILE, est considérée comme carte début d'un fichier et elle possède deux paramètres, les zones 3 et 4.

3. PROPRI: cette zone contient le nom du propriétaire du fichier. Cependant, aucun contrôle ne sera fait entre ce nom et la liste des utilisateurs. Il s'agit donc, en fait, d'un nom qui empêchera de confondre deux fichiers d'utilisateurs différents, qui ont reçu le même nom. On conseille donc à l'utilisateur de frapper son nom dans cette zone, ainsi, aucune ambiguïté n'est possible.

4. NAME: est le nom du fichier-carte de l'utilisateur nommé par PROPRI.

N.B.: les zones 3 et 4 peuvent avoir plus de huit caractères chacune.

Après les cartes composant le fichier, on trouvera une carte



Cette carte sert à empêcher que les cartes d'un autre utilisateur, qui a oublié sa carte FILE et qui place ses

cartes après les nôtres, ne soient considérées comme faisant partie de notre fichier.

4.4.3. DESCRIPTION DU PROGRAMME SPOOL.C.

Le texte de spool.c peut être consulté en annexe.

§ -1. La définition des variables globales et des constantes.

MP est une constante dont les bits sont tous positionnés à 1. C'est avec cette valeur que sera effectué le test qui décidera si un caractère est inconnu ou non.

BEG, END et ERROR sont trois valeurs que prendra l'entier CARTE lors de l'analyse d'une carte spéciale (commençant par un caractère inconnu), selon que cette analyse décidera qu'il s'agit d'une carte FILE, d'une carte END ou d'une carte erronée.

OPENED et CLOSED sont les valeurs que prendra l'indicateur FILE.

On déclare ensuite une série d'entiers .

File est un entier qui vaut 1 (OPENED), si on a ouvert un fichier-disque et ∅ (CLOSED) dans le cas contraire.

fdocr, fdrcr, fdwsf, fdosf sont des entiers qui recevront les valeurs retournées par les différents appels systèmes employés dans le programme.

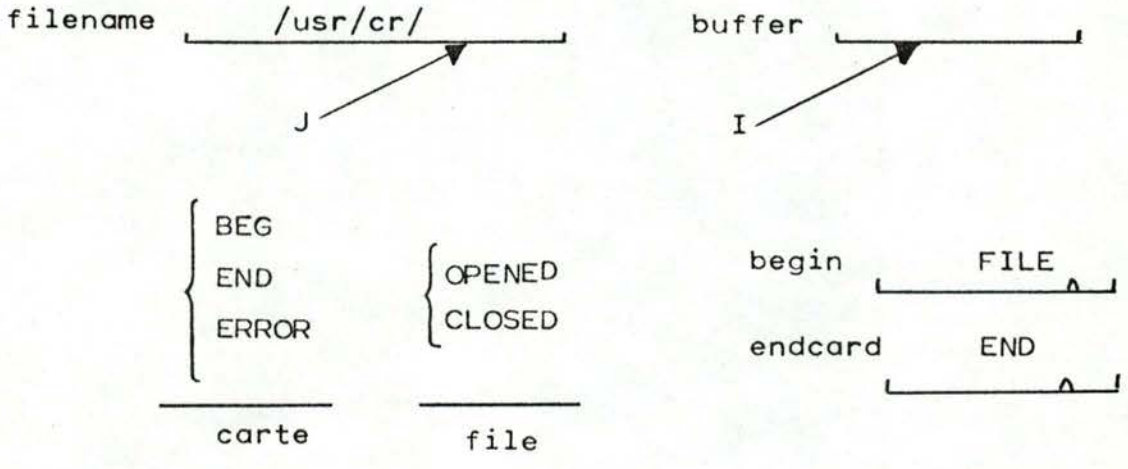
- fdocr: file descriptor open card reader
- fdrcr: file descriptor read card reader
- fdwsf: file descriptor write spool file
- fdosf: file descriptor open spool file

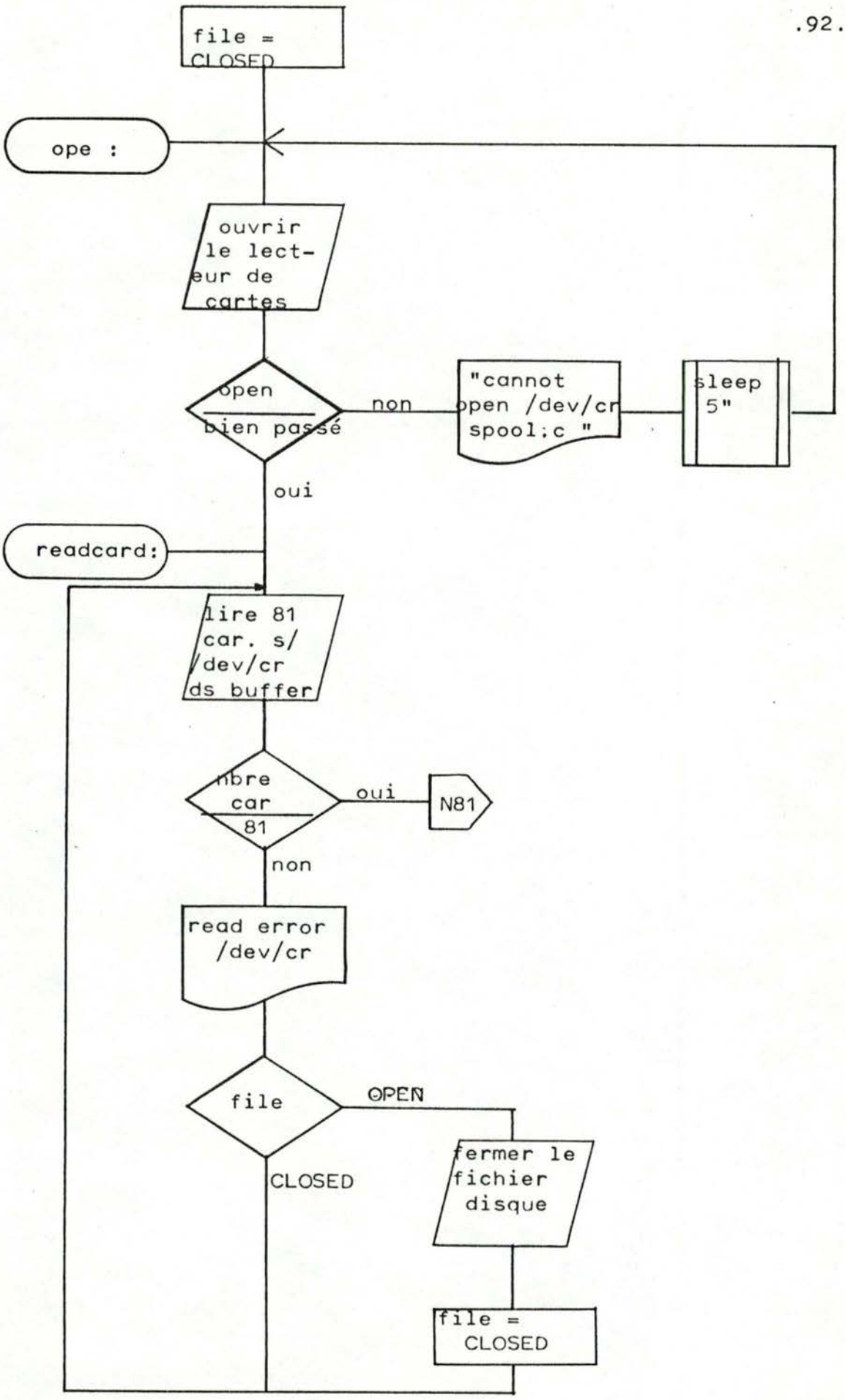
I servira d'indice dans le vecteur buffer et J dans le vecteur filename .

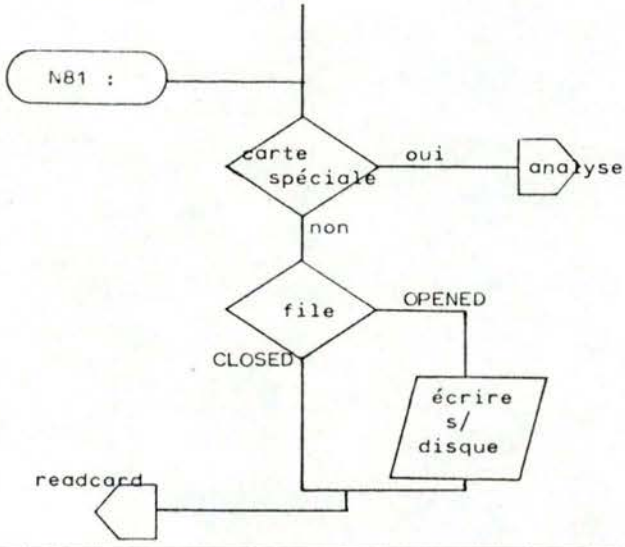
Les tableaux begin [5] et endcard [4] serviront lors de tests pour décider si la deuxième zone d'une carte spéciale est FILE ou END.

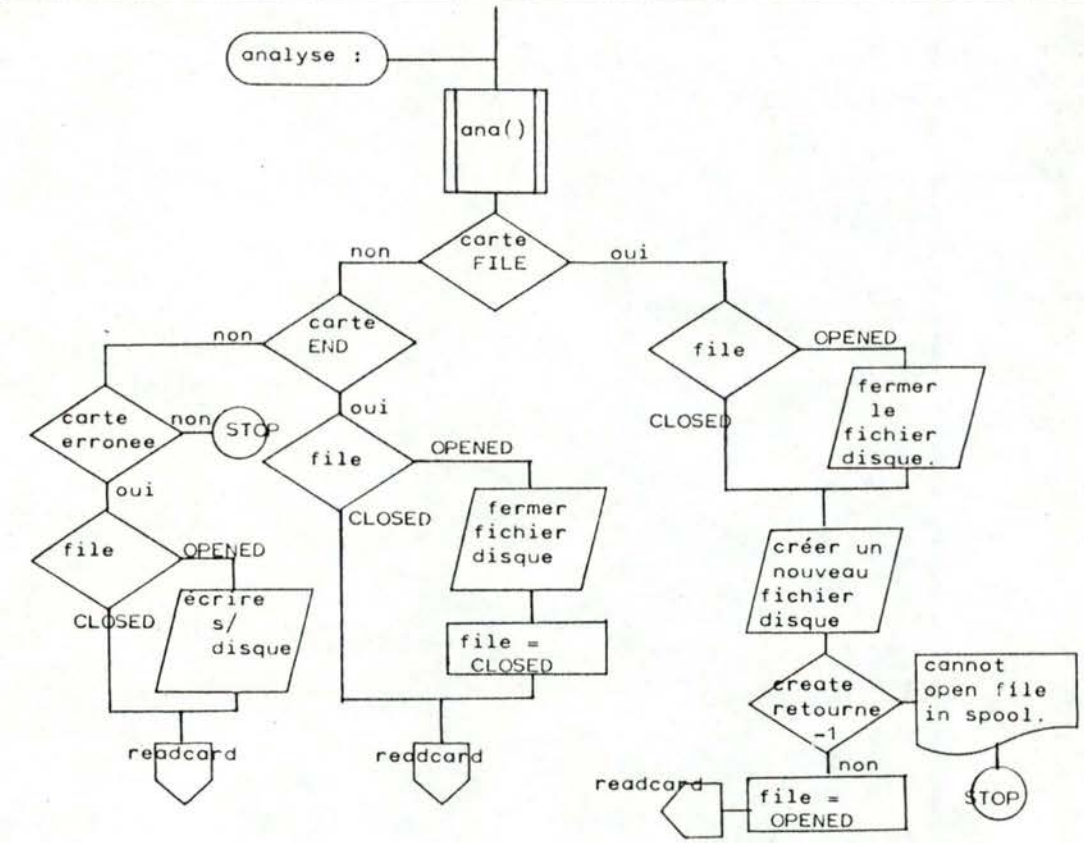
buffer [81] est un vecteur destiné à recevoir les caractères provenant de la lecture d'une carte.

filename [26] contiendra le nom du fichier provenant de la carte FILE. Le début du nom est déjà initialisé: /usr/cr/

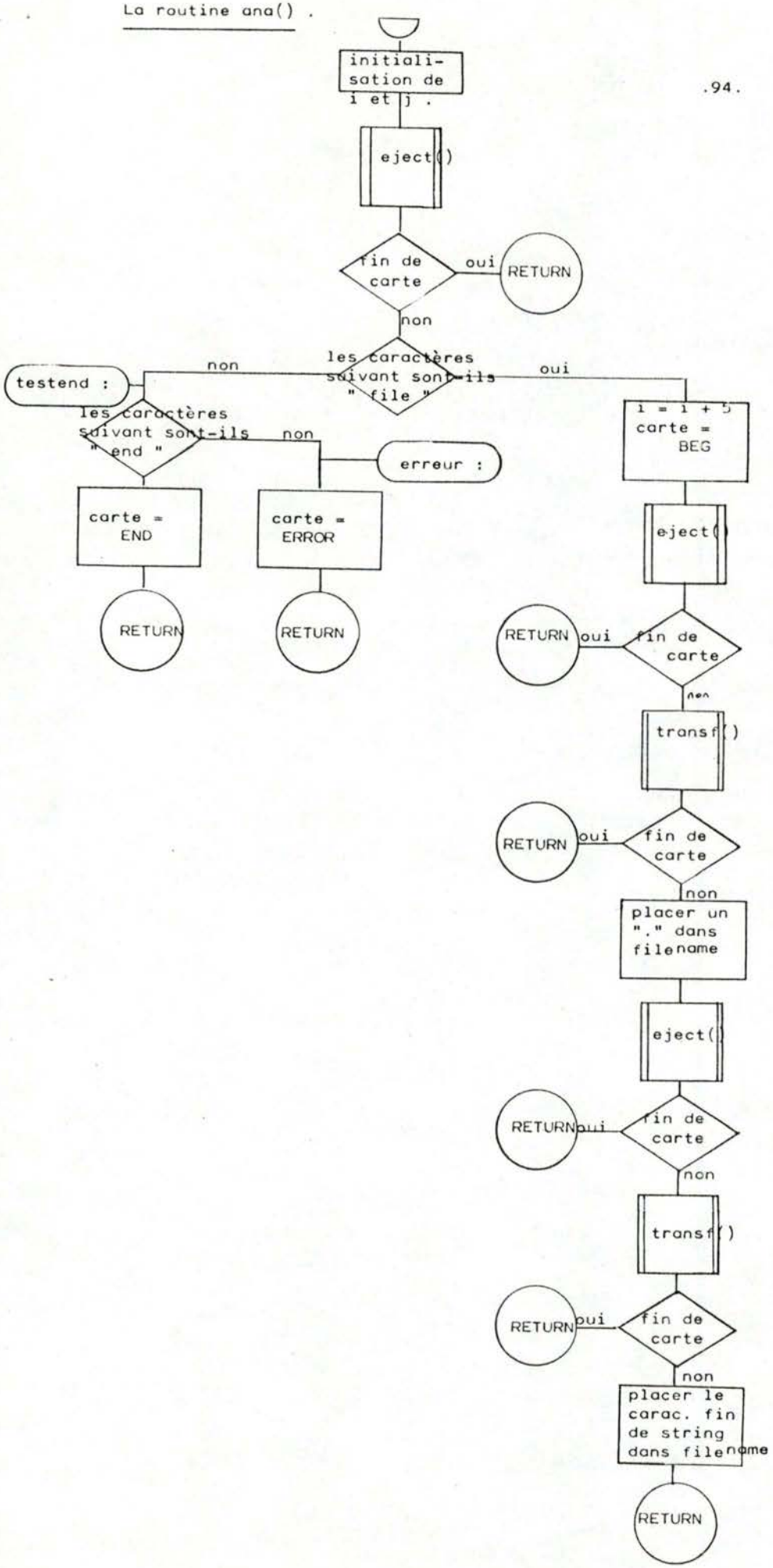



file = CLOSED
ope :
ouvrir le lecteur de cartes
open bien passé
non
"cannot open /dev/cr spool:c "
sleep 5"
oui
readcard:
lire 81 car. s/ /dev/cr ds buffer
nbre car 81
oui
N81
non
read error /dev/cr
file
OPEN
fermer le fichier disque
CLOSED
file = CLOSED

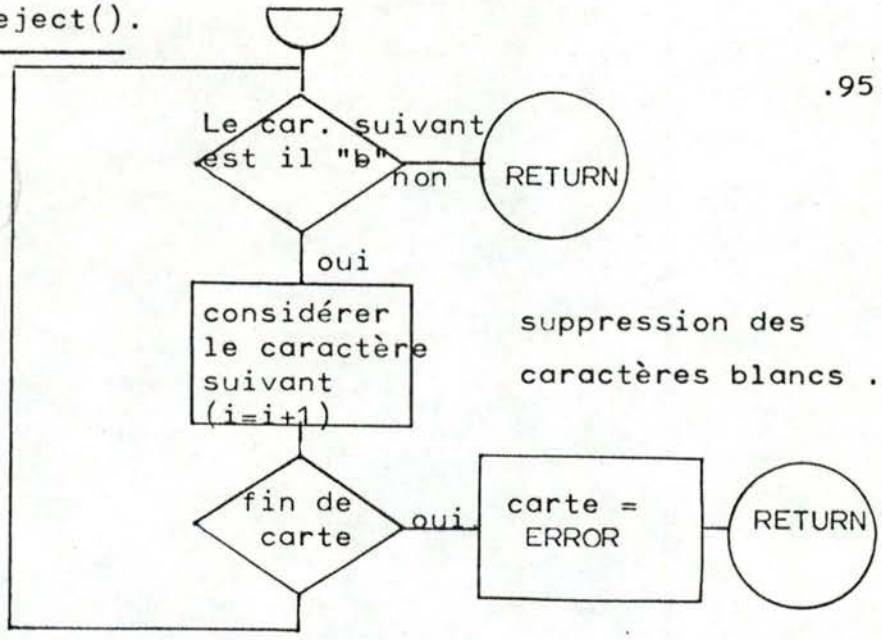

N81 :
carte spéciale
oui
analyse
non
file
OPENED
CLOSED
écrire s/ disque
readcard


analyse :
ana()
carte FILE
non
oui
carte END
non
oui
carte erronee
non
STOP
oui
file
OPENED
CLOSED
écrire s/ disque
fermer fichier disque
file = CLOSED
readcard
readcard
file
OPENED
CLOSED
fermer le fichier disque.
créer un nouveau fichier disque
create retourne -1
cannot open file in spool.
non
file = OPENED
readcard
STOP

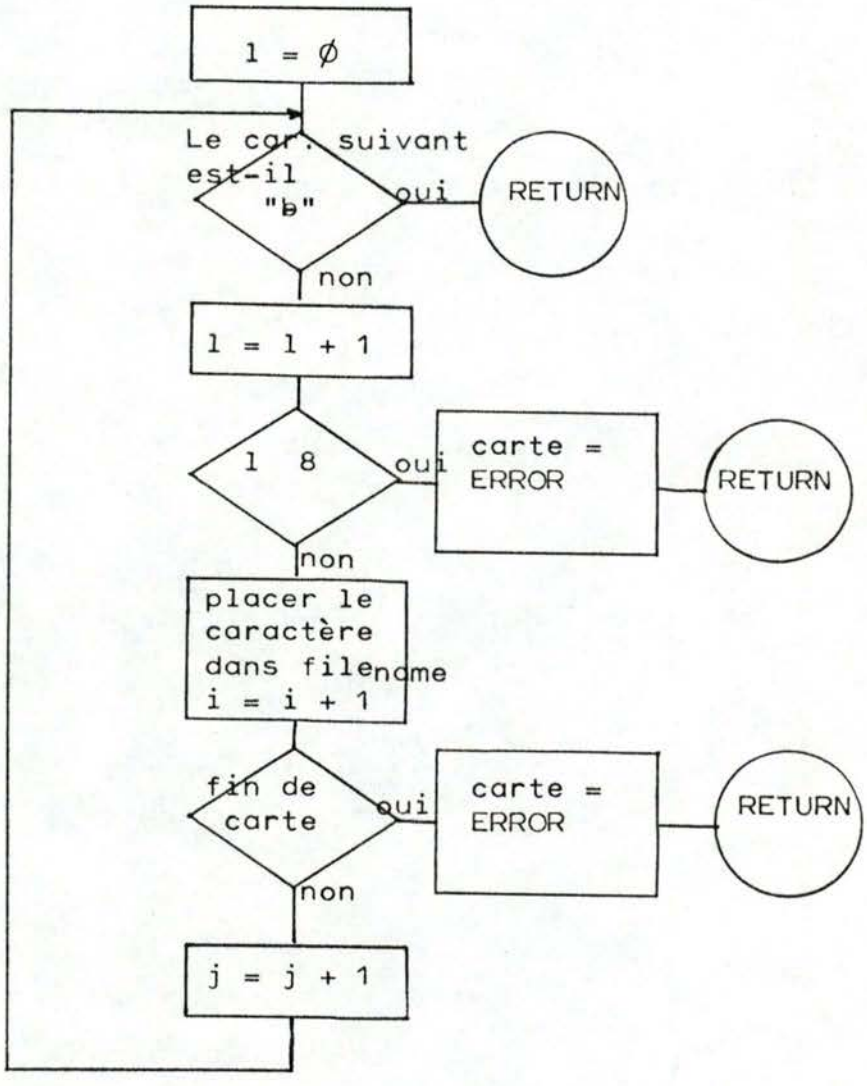
La routine ana() .



La routine eject().



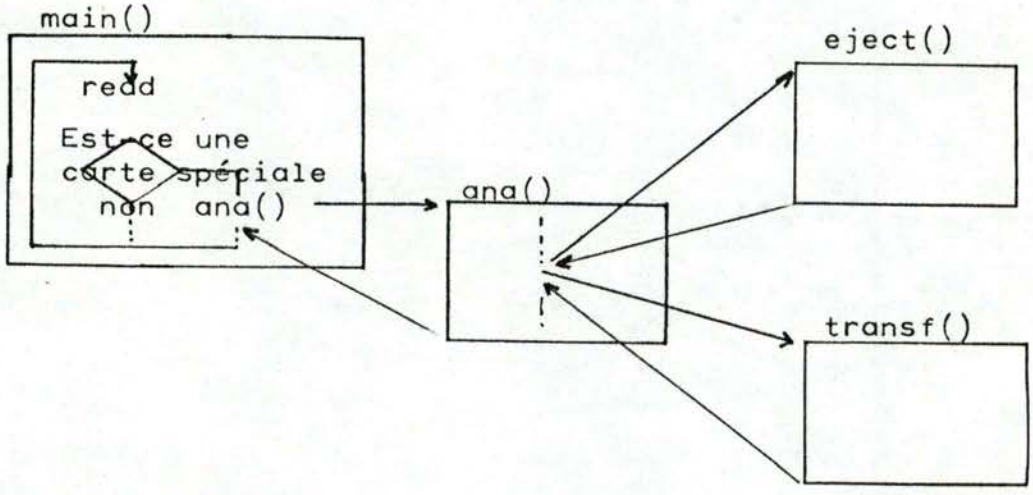
La routine transf().



§ -3. Une description globale des routines de spool.c.

La programmation de spool.c n'est pas intéressante en elle-même. Nous nous bornerons donc à montrer le fonctionnement général de chacune des routines. De plus, les organigrammes précédents et le texte du programme pourront également aider le lecteur désireux d'en savoir plus sur la façon dont ce programme est écrit.

Schéma d'enchaînement des différentes routines:

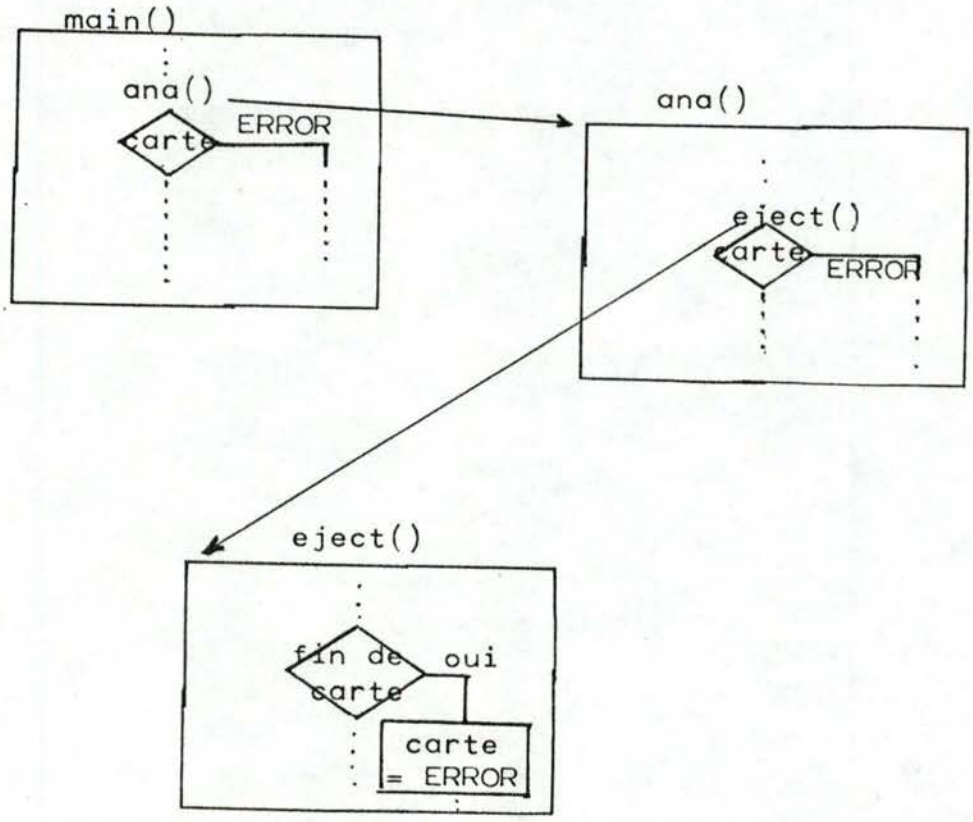


Toutes les routines travaillent sur une série de zones communes. Ainsi, aucun paramètre n'est passé aux routines.

- la routine eject():

Eject() positionne l'indice I, afin qu'il pointe vers le premier caractère différent de b dans le vecteur buffer (à partir de l'endroit où on était déjà arrivé).

Dans le cas où on arrive à la fin de la carte, on positionne l'indicateur carte à la valeur ERROR. Cet indicateur pourra être testé successivement par les routines ana() et par le programme principal.
La situation est la suivante:



Remarque: la technique sera identique pour la routine transf() ou ana().

- la routine transf():

Transf() est appelé lorsque l'indice I a été positionné par eject(), de façon qu'il pointe vers le début d'un paramètre de la carte FILE.

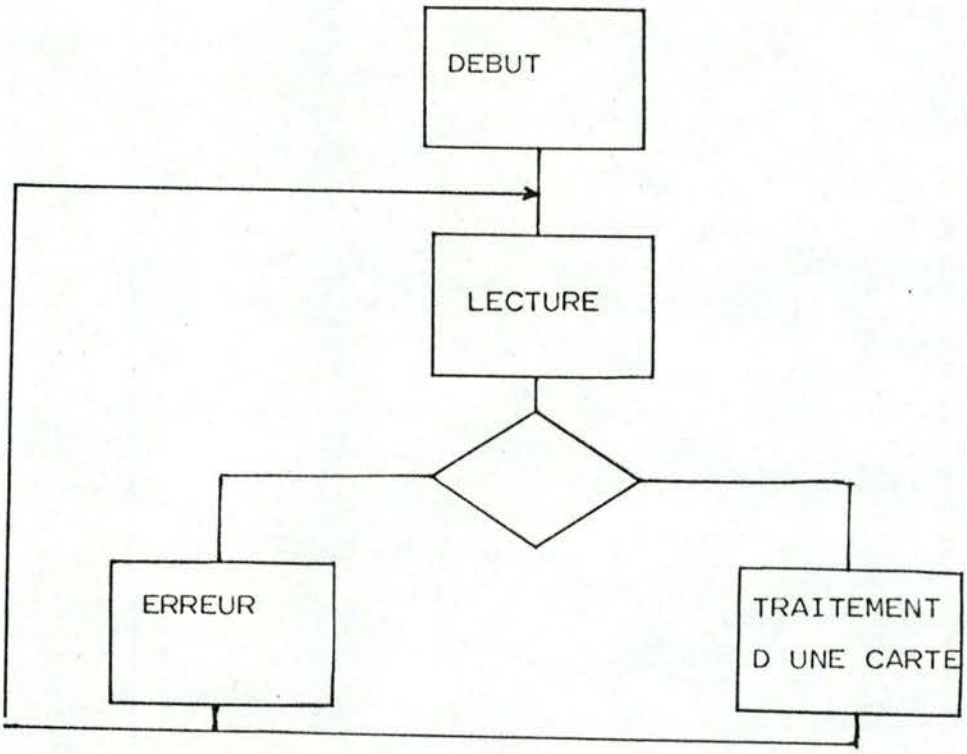
Le but de transf() est de transférer ce paramètre dans le vecteur FILENAME à l'endroit spécifié par l'indice I. Si l'on arrive en bout de carte, ou si le paramètre a une longueur supérieure à 8, on positionnera l'indicateur carte à la valeur ERROR.

- la routine ana():

Maintenant que l'on a vu quels rôles jouent les routines eject() et transf(), nous pouvons parler de la routine ana() qui les utilise.

Rappelons que ana() est appelé lorsqu'une carte spéciale a été découverte (une carte dont le premier caractère est inconnu). ana() a pour but d'analyser cette carte, afin de décider s'il s'agit d'une carte début, d'une carte fin ou d'une carte erronée.
Pour cela, on emploie les routines eject() et transf().
Selon la décision prise par ana(), l'indicateur carte sera positionné à BEG, END ou ERROR.

- le programme principal main():



Cet ordinogramme général montre bien que spool.c est un programme de lecture qui n'a pas de fin.

Au début, on initialise l'indicateur file à CLOSED, afin de spécifier qu'aucun fichier-disque n'a été ouvert. On essaie d'ouvrir le lecteur de cartes. Si l'OPEN renvoie une valeur égale à -1, on patiente 5 secondes avant d'essayer à nouveau.

Dans la boucle de lecture, on teste la valeur retournée par l'appel système read. Si cette valeur est 81, c'est qu'une carte a été lue correctement et on peut exécuter le traitement prévu pour cette carte, sinon c'est qu'une erreur de lecture s'est produite. Dans ce cas, on retourne à la lecture, après avoir imprimé un message.

Cependant, dans le cas où un fichier-disque était ouvert, il sera fermé et l'indicateur file reinitialisé à CLOSED.

Il nous reste donc à voir quel traitement sera effectué par carte lue correctement.

Tout d'abord, on analyse le premier caractère de la carte, afin de voir s'il s'agit d'une carte spéciale ou non. Le cas le plus simple est celui d'une carte normale. Alors, si un fichier-disque est ouvert, les informations contenues sur la carte sont écrites sur le disque, sinon aucun traitement n'est effectué pour cette carte, elle est donc ignorée.

S'il s'agit d'une carte spéciale, on appelle la routine ana(), qui donnera une valeur à l'indicateur carte. Dès lors, des traitements séparés sont programmés selon la valeur de carte.

S'il s'agit d'une carte erronée, on l'écrit sur le disque si un fichier était ouvert.

S'il s'agit d'une carte END, on ferme le fichier-disque, si un fichier était ouvert et on retourne à la lecture.

S'il s'agit d'une carte FILE, on ferme le fichier précédent si un fichier était encore ouvert (c'est le cas où un utilisateur a oublié sa carte END). On crée un nouveau fichier, dont le nom est contenu dans le vecteur filename (il y a été placé par la routine ana().Si cette création s'avère impossible, on arrête l'exécution du programme, sinon on retourne à la lecture.

4.4.4 REMARQUES CONCERNANT L'ECRITURE DE SPOOL.C

Nous avons déjà signalé que le programme spool.c était un programme de lecture qui n'avait pas de fin. En fait, le programme peut terminer son exécution dans le cas suivant.

Si l'ordre d'ouverture d'un fichier-disque renvoie une valeur -1, on arrête le traitement car, si on ne peut ouvrir un fichier sur le disque, on ne pourra en ouvrir d'autres. Il est donc inutile d'essayer de réinitialiser le processus et de continuer.

Cela se produira notamment quand le directoire /usr/cr n'aura pas été crée ou qu'il aura été effacé .

S'il est impossible d'ouvrir le lecteur de cartes, on s'endort pendant cinq secondes, avant d'essayer à nouveau. Si le lecteur ne peut être ouvert, c'est qu'un autre programme a déjà ouvert le lecteur, et ne l'a pas encore fermé . Cependant, le programme spool.c sera exécuté dès le lancement de UNIX .

Spool.c sera donc toujours le premier programme à utiliser le lecteur de cartes. Comme ce programme n'a normallement pas de fin, il interdira l'utilisation du lecteur de cartes à tout autre utilisateur .

Remarque: pour que spool.c s'exécute dès le lancement de UNIX, il suffit d'insérer la commande d'exécution du programme objet résultant de la compilation de spool.c dans le fichier /ETC/rc. Le fichier est un fichier de commandes qui seront exécutées au démarrage du système.

Lorsqu'une erreur de lecture se produit, nous avons choisi de reinitialiser le processus et de continuer, c'est-à-dire, de fermer le fichier-disque et de remettre l'entier flag à $\emptyset$, avant de se rebrancher à la lecture. Ce choix a été guidé par le fait qu'une erreur de lecture peut se produire pour une seule carte, et la suite du programme peut ne pas être altérée par cette erreur. Cependant, on ne peut laisser continuer le programme sans reinitialiser le processus, car le fichier qui est en train d'être créé lorsque l'erreur se produit, risque de ne pas être la copie conforme du paquet de cartes. On doit donc fermer le fichier et avertir son propriétaire qu'une erreur s'est produite en envoyant un message à la console principale.

Il faut également faire remarquer que, si le lecteur de cartes n'est pas utilisé, spool.c lancera une lecture qui provoquera l'exécution de la routine crread(), qui, elle-même, appellera crinput() pour effectuer cette lecture.

Le lecteur de cartes n'étant pas prêt (généralement pas de cartes dans le lecteur), crinput s'endormira jusqu'à ce qu'il devienne prêt. (crinput sera réveillé par une interruption). Dès lors, spool.c sera endormi et pourra, si nécessaire, être éjecté de la mémoire.

Tout le temps que le lecteur reste inactif, spool.c ne sera pas un processus concurrent pour l'obtention des différentes ressources (CPU, mémoire, mémoire auxiliaire)

Cependant, un problème n'a pas encore été résolu, Lorsqu'un fichier disque a été crée, soit /usr/cr/propri.name, l'utilisateur voudrait pouvoir disposer de ce fichier dans son directoire, et non pas dans le directoire /usr/cr . Il convient donc de créer une nouvelle commande, dont la syntaxe pourrait être :

```
cr   propri    name       file
```

et dont l'effet serait identique à

```
mv   /usr/cr/propri.name    file
```

.

Mais un autre problème vient se greffer sur celui-ci . Le programme spool.c crée un fichier identique à celui représenté par le paquet de cartes, si ce n'est que les différents caractères sont codés en ASCII . Généralement, on ne dispose pas d'un perforateur de cartes comportant l'ensemble des caractères minuscules . Dès lors, tous les fichiers cartes seront des fichiers dont les lettres sont obligatoirement en majuscules . De plus, UNIX reconnait les caractères majuscules et minuscules (notamment pour les programmes en C) . La commande cr devrait donc résoudre ce problème .

Le même problème se présente d'ailleurs parfois pour l'imprimante et les différents terminaux . Lorsqu'il s'agit d'un périphérique utilisé comme fichier de sortie d'un programme (imprimante, terminal), tous les caractères minuscules sont traduits en majuscules.

Lorsqu'il s'agit, par contre, d'un périphérique d'entrée (un terminal), toutes les majuscules seront traduites en minuscules, excepté si une majuscule est précédée du caractère "\". (la grande majorité des caractères dans UNIX sont minuscules).

En conclusion de toutes ces considérations, on peut donc énoncer un certain nombre de problèmes que devra résoudre le programme exécuté lors d'une commande cr.

- exécuter le MOVE du fichier dans /usr/cr vers le directoire de l'utilisateur, selon les paramètres donnés lors de la commande cr.
- traduire toutes les lettres en minuscules , excepté celles précédées du caractère "\".
- traduire les deux premiers paramètres de la commande cr en majuscules, afin que le fichier-disque puisse être trouvé. En effet, propri et name ont été perforés en majuscules sur la carte et le fichier résultant a donc reçu un nom majuscule.

Cinquième partie :

CONCLUSIONS .

CINQUIEME PARTIE: CONCLUSIONS.

Quelle est la marche à suivre pour quelqu'un qui désire écrire un driver pour un autre périphérique?

A notre avis, la procédure suivante pourrait être conseillée:

- tout d'abord, étudier UNIX, en tant qu'utilisateur, tout en apprenant la programmation en C.
- essayer de dégager la logique de fonctionnement du périphérique choisi, en s'habituant à manipuler le tableau de commande du PDP 11.
- s'intéresser aux entrées-sorties de UNIX et à l'écriture d'un driver, et nous espérons que la lecture de ce travail pourra grandement vous aider sur ce point.
- ensuite, écrire le driver et le tester.
- reconfigurer le système afin d'y inclure le driver. Pour cela, la marche à suivre est de:
  * compiler le driver et placer le code objet en librairie.
  * modifier les fichiers de configuration (c.c,, l.s.).
  * créer un nouveau système en liant les différents programmes en librairie.
- créer un fichier spécial pour le périphérique (programme /etc/mknod).
- enfin, écrire éventuellement des utilitaires, afin de faciliter l'emploi des périphériques.

Nous espérons que certains lecteurs de ce mémoire seront intéressés par la poursuite du travail et que UNIX deviendra bientôt complet pour notre installation.

A tous ceux-la, nous leur souhaitons bonne chance et bon courage car il y a une grande marge entre la connaissance théorique de concepts sur les systèmes d'exploitation et la réalisation pratique d'un travail concernant un O.S. donné .

Annexe 1 :

LES FICHIERS .h

file.h

inode.h

user.h

Oct 12 11:54

file.h

```
 1    /*
 2     * One file structure is allocated
 3     * for each open/creat/pipe call.
 4     * Main use is to hold the read/write
 5     * pointer associated with each open
 6     * file.
 7     */
 8    struct  file
 9    {
10            char    f_flag;
11            char    f_count;        /* reference count */
12            int     f_inode;        /* pointer to inode structure */
13            char    *f_offset[2];   /* read/write character pointer */
14    } file[NFILE];
15
16    /* flags */
17    #define FREAD   01
18    #define FWRITE  02
19    #define FPIPE   04
```

Oct 12 11:55

inode.h

```
/*
 * The I node is the focus of all
 * file activity in unix. There is a unique
 * inode allocated for each active file,
 * each current directory, each mounted-on
 * file, text file, and the root. An inode is 'named'
 * by its dev/inumber pair. (iget/iget.c)
 * Data, from mode on, is read in
 * from permanent inode on volume.
 */
struct  inode
{
        char    i_flag;
        char    i_count;        /* reference count */
        int     i_dev;          /* device where inode resides */
        int     i_number;       /* i number, 1-to-1 with device
                                   adress
        int     i_mode;
        char    i_nlink;        /* directory entries */
        char    i_uid;          /* owner */
        char    i_gid;          /* group of owner */
        char    i_size0;        /* most significant of size */
        char    *i_size1;       /* least sig */
        int     i_addr[8];      /* device addresses constituting fi
        int     i_lastr;        /* last logical block read (for rea
                                   ahead
} inode[NINODE];

/* flags */
#define ILOCK   01              /* inode is locked */
#define IUPD    02              /* inode has been modified */
#define IACC    04              /* inode access time to be updated
#define IMOUNT  010             /* inode is mounted on */
#define IWANT   020             /* some process waiting on lock */
#define ITEXT   040             /* inode is pure text prototype */

/* modes */
#define IALLOC  0100000         /* file is used */
#define IFMT    060000          /* type of file */
#define         IFDIR   040000  /* directory */
#define         IFCHR   020000  /* character special */
#define         IFBLK   060000  /* block special, 0 is regular */
#define ILARG   010000          /* large addressing algorithm */
#define ISUID   04000           /* set user id on execution */
#define ISGID   02000           /* set group id on execution */
#define ISVTX   01000           /* save swapped text even after use
#define IREAD   0400            /* read, write, execute permissions
#define IWRITE  0200
#define IEXEC   0100
```

USER.H

```
/*
 * THE USER STRUCTURE,
 * ONE ALLOCATED PER PROCESS,
 * CONTAINS ALL PER PROCESS DATA
 * THAT DOESN'T NEED TO BE REFERENCED
 * WHILE THE PROCESS IS SWAPPED,
 * THE USER BLOCK IS USIZE*64 BYTES
 * LONG; RESIDES AT VIRTUAL KERNEL
 * LOC 140000; CONTAINS THE SYSTEM
 * STACK PER USER; IS CROSS REFERENCED
 * WITH THE PROC STRUCTURE FOR THE
 * SAME PROCESS,
 */
STRUCT USER
{
        INT     U_RSAV[2];              /* SAVE R5,R6 WHEN EXCHANGING ST
        INT     U_FSAV[25];             /* SAVE FP REGISTERS */
                                        /* RSAV AND FSAV MUST BE FIRST I
        CHAR    U_SEGFLG;               /* FLAG FOR IO; USER OR KERNEL S
        CHAR    U_ERROR;                /* RETURN ERROR CODE */
        CHAR    U_UID;                  /* EFFECTIVE USER ID */
        CHAR    U_GID;                  /* EFFECTIVE GROUP ID */
        CHAR    U_RUID;                 /* REAL USER ID */
        CHAR    U_RGID;                 /* REAL GROUP ID */
        INT     U_PROCP;                /* POINTER TO PROC STRUCTURE */
        CHAR    *U_BASE;                /* BASE ADDRESS FOR IO */
        CHAR    *U_COUNT;               /* BYTES REMAINING FOR IO */
        CHAR    *U_OFFSET[2];           /* OFFSET IN FILE FOR IO */
        INT     *U_CDIR;                /* POINTER TO INODE OF CURRENT D
        CHAR    U_DBUF[DIRSIZ];         /* CURRENT PATHNAME COMPONENT */
        CHAR    *U_DIRP;                /* CURRENT POINTER TO INODE */
        STRUCT  {                       /* CURRENT DIRECTORY ENTRY */
                INT     U_INO;
                CHAR    U_NAME[DIRSIZ];
        } U_DENT;
        INT     *U_PDIR;                /* INODE OF PARENT DIRECTORY OF
        INT     U_UISA[16];             /* PROTOTYPE OF SEGMENTATION ADD
        INT     U_UISD[16];             /* PROTOTYPE OF SEGMENTATION DES
        INT     U_OFILE[NOFILE];        /* POINTERS TO FILE STRUCTURES O
        INT     U_ARG[5];               /* ARGUMENTS TO CURRENT SYSTEM C
        INT     U_TSIZE;                /* TEXT SIZE (*64) */
        INT     U_DSIZE;                /* DATA SIZE (*64) */
        INT     U_SSIZE;                /* STACK SIZE (*64) */
        INT     U_SEP;                  /* FLAG FOR I AND D SEPARATION *
        INT     U_QSAV[2];              /* LABEL VARIABLE FOR QUITS AND
        INT     U_SSAV[2];              /* LABEL VARIABLE FOR SWAPPING *
        INT     U_SIGNAL[NSIG];         /* DISPOSITION OF SIGNALS */
        INT     U_UTIME;                /* THIS PROCESS USER TIME */
        INT     U_STIME;                /* THIS PROCESS SYSTEM TIME */
        INT     U_CUTIME[2];            /* SUM OF CHILDS' UTIMES */
        INT     U_CSTIME[2];            /* SUM OF CHILDS' STIMES */
        INT     *U_ARG;                 /* ADDRESS OF USERS SAVED R0 */
        INT     U_PROF[4];              /* PROFILE ARGUMENTS */
        CHAR    U_INTFLG;               /* CATCH INTR FROM SYS */
                                        /* KERNEL STACK PER USER
                                         * EXTENDS FROM U + USIZE*64
                                         * BACKWARD NOT TO REACH HERE
                                         */
} U;
```

USER.H (suite)

```
/* U_ERROR CODES */
#DEFINE EFAULT  106
#DEFINE EPERM   1
#DEFINE ENOENT  2
#DEFINE ESRCH   3
#DEFINE EINTR   4
#DEFINE EIO     5
#DEFINE ENXIO   6
#DEFINE E2BIG   7
#DEFINE ENOEXEC 8
#DEFINE EBADF   9
#DEFINE ECHILD  10
#DEFINE EAGAIN  11
#DEFINE ENOMEM  12
#DEFINE EACCES  13
#DEFINE ENOTBLK 15
#DEFINE EBUSY   16
#DEFINE EEXIST  17
#DEFINE EXDEV   18
#DEFINE ENODEV  19
#DEFINE ENOTDIR 20
#DEFINE EISDIR  21
#DEFINE EINVAL  22
#DEFINE ENFILE  23
#DEFINE EMFILE  24
#DEFINE ENOTTY  25
#DEFINE ETXTBSY 26
#DEFINE EFBIG   27
#DEFINE ENOSPC  28
#DEFINE ESPIPE  29
#DEFINE EROFS   30
#DEFINE EMLINK  31
#DEFINE EPIPE   32
```

Annexe 2 :

LE FICHIER /USR/SYS/RUN .

/USR/SYS/RUN

```
CHDIR KEN
CC -C -O *.C
AR R ../LIB1
RM *.O

CHDIR ../DMR
CC -C -O *.C
AR R ../LIB2
RM *.O

CHDIR ../CONF
AS M40.S
MV A.OUT M40.O
: AS M45.S
: MV A.OUT M45.O
: CC SYSFIX.C
: MV A.OUT SYSFIX
CC MKCONF.C
MV A.OUT MKCONF

MKCONF
RK
TM
TC
DONE

CC -C C.C
AS L.S
LD -X A.OUT M40.O C.O ../LIB1 ../LIB2
: AS DATA.S L.S
: LD -X -R -O A.OUT M45.O C.O ../LIB1 ../LIB2
: NM -UG
: SYSFIX A.OUT X
: MV X A.OUT
CMP A.OUT /RKUNIX
CP A.OUT /RKUNIX

MKCONF
RP
TM
TC
DONE

CC -C C.C
AS L.S
LD -X A.OUT M40.O C.O ../LIB1 ../LIB2
: AS DATA.S L.S
: LD -X -R -O A.OUT M45.O C.O ../LIB1 ../LIB2
: NM -UG
: SYSFIX A.OUT X
: MV X A.OUT
CMP A.OUT /RPUNIX
CP A.OUT /RPUNIX

MKCONF
HP
TM
TC
DONE
```

/USR/SYS/RUN (suite)

```
CC -C C.C
AS L.S
LD -X A.OUT M40.O C.O ../LIB1 ../LIB2
: AS DATA.S L.S
: LD -X -R -O A.OUT M45.O C.O ../LIB1 ../LIB2
: NM -UG
: SYSFIX A.OUT X
: MV X A.OUT
CMP A.OUT /HPUNIX
CP A.OUT /HPUNIX

RM MKCONF C.C L.S A.OUT *.O
: RM SYSFIX
```

Annexe 3 :

VECTEURS D INTERRUPTIONS ET REGISTRES DE COMMANDES DE PERIPHERIQUES DU PDP 11 .

```
FN  := FPSR<3>                          floating negative
FZ  := FPSR<2>                          floating zero
FV  := FPSR<1>                          floating overflow
FC  := FPSR <0>                         floating carry

Instruction format
OO<3:0>   := i<15:12>                   op code
FOO<3:0>  := i<11:8>                    floating op code
AC<1:0>   := i<7:6>                     accumulator


General Definitions
XL  := ((FD=0) ⇒ 1-2^-24;               largest fraction
       (FD=1) ⇒ 1-2^-56)
XLL := 2^-128                           smallest non-zero number
XUL := 2^127 * XL                       largest number
JL  := ((FL=0) ⇒ 2^15-1;                largest integer
       (FL=1) ⇒ 2^31-1)

Address Calculation
FPS<63:0> := (                          floating point processor source
   (dm=0) ⇒ FAC(dr);
   (dm≠0) ⇒ (
      (FD=0) ⇒ D<15:0>□Mw[PC+2];
      (FD=1) ⇒ D<15:0>□Mw[PC+2]□
        Mw[PC+4]□Mw[PC+6]))
FPS'<63:0> := (                         floating point processor source with
   (dm=0) ⇒ FAC(dr);                     side effects
   (dm≠0) ⇒ (
      (FD=0) ⇒ D'<15:0>□nw'
      (FD=1) ⇒ D'<15:0>□nw'□nw'□nw'))
FPD<63:0> := FPS<63:0>                  floating point processor destination
FPD'<63:0>:= FPS'<63:0>                 floating point processor destination with
                                         side effects
FS<15:0>  := D<15:0>                    floating source, CPU mode
FS'<15:0> := D'<15:0>                   floating source with side effects,
                                         CPU mode
FD<15:0>  := D<15:0>                    floating destination, CPU mode
FD'<15:0> := D'<15:0>                   floating destination with side effects,
                                         CPU mode
Fac := FAC(AC)                          destination floating register
```

[1] a 17 bit result, r, used only for descriptive purposes

[2] A prime is used in S (e.g., S') and D (e.g., D') to indicate that when a word is accessed in this fashion, side effects may occur. That is, registers of R may be changed.

[3] If all 16 bits of result, r = 0, then Z is set to 1 else Z is set to 0.

[4] The 8 least significant bits are used to form a 16-bit positive or negative number by extending bit 7 into 15:8.

[5] a ⇒ b means: if boolean a is true then b is executed.

[6] Mw means the memory taken as a work-organized memory.

APPENDIX B MEMORY MAP

INTERRUPT VECTORS.

000	RESERVED
004	TIME OUT, BUS ERROR
010	RESERVED INSTRUCTION
014	DEBUGGING TRAP VECTOR
020	IOT TRAP VECTOR
024	POWER FAIL TRAP VECTOR
030	EMT TRAP VECTOR
034	"TRAP" TRAP VECTOR
040	SYSTEM SOFTWARE
044	SYSTEM SOFTWARE
050	SYSTEM SOFTWARE
054	SYSTEM SOFTWARE
060	TTY IN-BR4
064	TTY OUT-BR4
070	PC11 HIGH SPEED READER-BR4
074	PC11 HIGH SPEED PUNCH
100	KW11L - LINE CLOCK BR6
104	KW11P - PROGRAMMER REAL TIME CLOCK BR6
120	XY PLOTTER
124	DR11B-(BR5 HARDWIRED)
130	ADO1 BR5-(BR7 HARDWIRED)
134	AFC11 FLYING CAP MULTIPLEXER BR4
140	AA11-A,B,C SCOPE BR4
144	AA11 LIGHT PIN BR5
170	USER RESERVED
174	USER RESERVED
200	LP11 LINE PRINTER CTRL-BR4
204	RF11 DISK CTRL-BR5
210	RC11 DISK CTRL-BR5
214	TC11 DEC TAPE CTRL-BR6
220	RK11 DISK CTRL-BR5
224	TM11 COMPATIBLE MAG TAPE CTRL-BR5
230	CR11/CM11 CARD READER CTRL-BR6
234	UDC11 (BR4, BR6 HARDWIRED)
240	11/45 PIRQ
244	FPU ERROR
254	RP11 DISK PACK CTRL-BR5
260	
264	
270	USER RESERVED
274	USER RESERVED
300	START OF FLOATING VECTORS

DEVICE ADDRESSES

NOTE: XX MEANS A RESERVED ADDRESS FOR THAT OPTION. OPTION MAY NOT USE IT BUT IT WILL RESPOND TO BUS ADDRESS.

777776	CPU STATUS	
777774	STACK LIMIT REGISTER	
777772	11/45 PIRQ REGISTER	
777716	TO 777700 CPU REGISTERS	
777676	TO 777600 11/45 SEGMENTATION REGISTER	
777656	TO 777650 MX11 #6	
777646	TO 777640 MX11 #5	
777636	TO 777630 MX11 #4	
777626	TO 777620 MX11 #3	
777616	TO 777610 MX11 #2	
777606	TO 777600 MX11 #1	
777576	11/45SSR2	
777574	11/45 SSR1	
777572	11/45 SSR0	
777570	CONSOLE SWITCH REGISTER	
777566	KL11 TTY OUT DBR	
777564	KL11 TTY OUT CSR	
777562	KL11 TTY IN DBR	
777560	KL11 TTY IN CSR	
777556	PC11 HSP DBR	
777554	PC11 HSP CSR	
777552	PC11 HSR DBR	
777550	PC11 HSR CSR	
777546	LKS LINE CLOCK KW11-L	
777516	LP11 DBR	
777514	LP11 CSR	
777512	LP11 XX	
777510	LP11 XX	
777476	RF11 DISK RFLA	LOOK AHEAD
777474	RF11 DISK RFMR	MAINTENANCE
777472	RF11 DISK RFDBR	
777470	RF11 DISK RFDAE	
777466	RF11 DISK RFDAR	
777464	RF11 DISK RFCAR	
777462	RF11 DISK RFWC	
777460	RF11 DISK RFDSC	
777456	RC11 DISK RCDBR	
777454	RC11 MAINTENANCE	
777452	RC11 RCCAR	
777450	RC11 RCWC	
777446	RC11 RCCSR-	
777444	RC11 RCCSR1	
777442	RC11 RCER	
777440	RC11 RCLA	

777434	DT11 BUS SWITCH #7	
777432	BUS SWITCH #6	
777430	BUS SWITCH #5	
777426	BUS SWITCH #4	
777424	BUS SWITCH #3	
777422	BUS SWITCH #2	
777420	BUS SWITCH #1	
777416	RKDB	RK11 DISK
777414	RKMR	
777412	RKDA	
777410	RKBA	
777406	RKWC	
777404	RKCS	
777402	RKER	
777400	RKDS	
777356	TCXX	
777354	TCXX	
777352	TCXX	
777350	TCDT	DEC TAPE (TC11)
777346	TCBA	
777344	TCWC	
777342	TCCM	
777340	TCST	
777336	ASH	EAE (KE11-A) #2
777334	LSH	
777332	NOR	
777330	SC	
777326	MUL	
777324	MQ	
777322	AC	
777300	DIV	
777316	ASH	EAE (KE11-A) #1
777314	LSH	
777312	NOR	
777310	SC	
777306	MUL	
777304	MQ	
777302	AC	
777300	DIV	
777166	CR11 XX	
777164	CRDBR2	CR11 CARD READER
777162	CRDBR1	
777160	CRCSR	
776776	ADO1-D XX	
776774	ADO1-D XX	
776772	ADDBR	A/D CONVERTER ADO1-D
776770	ADCSR	

776766	DAC3	DAC AA11
776764	DAC2	
776762	DAC1	
776760	DAC0	
776756	SCOPE CONTROL - CSR	
776754	AA11 XX	
776752	AA11 XX	
776750	AA11 XX	
776740	RPBR3	RP11 DISK
776736	RPBR2	
776734	RPBR1	
776732	MAINTENANCE #3	
776730	MAINTENANCE #2	
776726	MAINTENANCE #1	
776724	RPDA	
776222	RPCA	
776720	RPBA	
776716	RPWC	
776714	RPCS	
776712	RPER	
776710	RPDS	

776676 TO 776500 MULTI TTY FIRST STARTS AT 776500

776476	TO 776406 MULTIPLE AA11'S SECOND STARTS @ 776760
776476	TO 776460 5TH AA11
776456	TO 776440 4TH AA11
776436	TO 776420 3RD AA11
776416	TO 776400 2ND AA11

NOTE 1ST AA11 IS AT 776750

776377 TO 776200 DX11
775600 DS11 AUXILIARY LOCATION
775577 TO 775540 DS11 MUX3
775537 TO 775500 DS11 MUX2
775477 TO 775440 DS11 MUX1
775436 TO 775400 DS11 MUX0
775377 TO 775200 DN11
775177 TO 775000 DM11
774777 TO 774400 DP11
774377 TO 774000 DC11

773777 TO 773000 DIODE MEMORY MATRIX

773000 BM792-YA PAPER TAPE BOOTSTRAP
773100 BM792-YB RC,RK,RP,RF AND TC11 - BOOTSTRAP
773200 BM792-YC CARD READER BOOTSTRAP
773300
773400
773500
773600
773700 RESERVED FOR MAINTENANCE LOADER

772776 TO 772700 TYPESET PUNCH
772676 TO 772600 TYPESET READER

772576	AFC-MAINTENANCE
772574	AFC-MUX ADDRESS
772572	AFC-DBR
772570	AFC-CSR
772546	KW11P XX
772544	KW11P COUNTER
772542	KW11P COUNT SET BUFFER
772540	KW11P CSR
772536	TM11 XX
772534	TM11 XX
772532	TM11 LRC
772530	TM11 DBR
772526	TM11 BUS ADDRESS
772524	TM11 BYTE COUNT
772522	TM11 CONTROL
772520	TM11 STATUS
772512	OST CSR
772510	OST EADRS1,2
772506	OST ADRS2
772504	OST ADRS1
772502	OST MASK2
772500	OST MASK1
772416	DR11B/DATA
772414	DR11B/STATUS
772412	DR11B/BA
772410	DR11B/WC

772136 TO 772110 MEMORY PARITY CSR

772136	15
772120	4
772116	3
772114	2
772112	1
772110	0
771776	UDCS - CONTROL AND STATUS REGISTER
771774	UDSR - SCAN REGISTER
771772	MCLK - MAINTENANCE REGISTER
771766	UDC FUNCTIONAL I/O MODULES
771000	UDC FUNCTIONAL I/O MODULES

770776 TO 770700 KG11 CRC OPTION

770776	KG11A KGNU7
770774	KGDBR7
770772	KGBBC7
770770	KGCSR7
770716	KGNU1
770714	KGBCC1
770712	KGDBR1
770710	KGCSR1
770706	KGNU0
770704	KGDBR0
770702	KGBCC0

770700 KG11A KGCSR0
770676 TO 770500 16 LINE FOR DM11BB
770676 DM11BB #16
770674
770672
770670
770666 DM11BB #15
770664
770662
770660
770656 DM11BB #14
770654
770652
770650
770646 DM11BB #13
770644
770642
770640
770636 DM11BB #12
770634
770632
770630
770626 DM11BB #11
770624
770622
770620
770616 DM11BB #10
770614
770612
770610
770606 DM11BB #9
770604
770602
770600 DM11BB #8
770076 LATENCY TESTER
770074 LATENCY TESTER
770072 LATENCY TESTER
770070 LATENCY TESTER
770056 TO 770000 SPECIAL FACTORY BUS TESTERS
767776 TO 764000 FOR USER and SPECIAL SYSTEMS---DR11A ASSIGNED IN USER AREA-STARTING AT HIGHEST ADDRESS WORKING DOWN
767776 DR11A #0
767774
767772
767770
767766 DR11A #1
767764
767762
767760
767756 DR11A #2
767754
767752
767750

764000 START NORMAL USER ADDRESSES HERE AND ASSIGN UPWARD.
760004 TO 760000 RESERVED FOR DIAGNOSTIC - SHOULD NOT BE ASSIGNED

Annexe 4 :

LES FICHIERS DE CONFIGURATION

L.S
C.C

FICHIER L.S

```
/ LOW CORE

BR4 = 200
BR5 = 240
BR6 = 300
BR7 = 340

. = 0^.
        BR      1F
        4

/ TRAP VECTORS
        TRAP; BR7+0.            / BUS ERROR
        TRAP; BR7+1.            / ILLEGAL INSTRUCTION
        TRAP; BR7+2.            / BPT-TRACE TRAP
        TRAP; BR7+3.            / IOT TRAP
        TRAP; BR7+4.            / POWER FAIL
        TRAP; BR7+5.            / EMULATOR TRAP
        TRAP; BR7+6.            / SYSTEM ENTRY

. = 40^.
.GLOBL  START, DUMP
1:      JMP     START
        JMP     DUMP


. = 60^.
        KLIN; BR4
        KLOU; BR4

. = 70^.
        PCIN; BR4
        PCOU; BR4

. = 100^.
        KWLP; BR6
        KWLP; BR6

. = 114^.
        TRAP; BR7+7.            / 11/70 PARITY

. = 200^.
        LPOU; BR4

. = 220^.
        RKIO; BR5

. = 224^.
        TMIO; BR5

. = 230^.
        CRIN; BR6

. = 240^.
        TRAP; BR7+7.            / PROGRAMMED INTERRUPT
        TRAP; BR7+8.            / FLOATING POINT
        TRAP; BR7+9.            / SEGMENTATION VIOLATION

/ FLOATING VECTORS
                . = 310^.
```

FICHIER L.S (suite)

```
        KLIN; BR4+1.
        KLOU; BR4+1.
. = 320^.
        GTOU; BR4.          /GT STOP
. = 324^.
        LPIN; BR4.          /LIGHT PEN
. = 330^.
        CHOU; BR4.          /CHARACTER AND TIME OUT INTERRUPT VECTOR
. = 340^.
        KLIN; BR4+2.
        KLOU; BR4+2.

////////////////////////////////////////////////////////
/                  INTERFACE CODE TO C
////////////////////////////////////////////////////////

.GLOBL  CALL, TRAP

.GLOBL  _KLRINT
KLIN:   JSR     R0,CALL; _KLRINT
.GLOBL  _KLXINT
KLOU:   JSR     R0,CALL; _KLXINT

.GLOBL _CRINT
CRIN:   JSR     R0,CALL; _CRINT
.GLOBL  _PCRINT
PCIN:   JSR     R0,CALL; _PCRINT
.GLOBL  _PCPINT
PCOU:   JSR     R0,CALL; _PCPINT

.GLOBL  _CLOCK
KWLP:   JSR     R0,CALL; _CLOCK


.GLOBL  _LPINT
LPOU:   JSR     R0,CALL; _LPINT

.GLOBL  _RKINTR
RKIO:   JSR     R0,CALL; _RKINTR

.GLOBL  _TMINTR
TMIO:   JSR     R0,CALL; _TMINTR

.GLOBL _GTINT
GTOU:   JSR     R0,CALL; _GTINT

.GLOBL _PENINT
LPIN:   JSR     R0,CALL; _PENINT

.GLOBL _CHARINT
CHOU:   JSR     R0,CALL; _CHARINT
```

```
4600 /* Used to dissect integer device code
4601  * into major (driver designation) and
4602  * minor (driver parameter) parts.
4603  */
4604 struct      {
4605             char    d_minor;
4606             char    d_major;
4607 };
4608 /* ------------------------        */
4609 /* Declaration of block device
4610  * switch. Each entry (row) is
4611  * the only link between the
4612  * main unix code and the driver.
4613  * The initialization of the
4614  * device switches is in the
4615  * file conf.c.
4616  */
4617 struct      bdevsw  {
4618     int     (*d_open)();
4619     int     (*d_close)();
4620     int     (*d_strategy)();
4621     int     *d_tab;
4622 } bdevsw[];
4623 /* ------------------------        */
4624 /* Nblkdev is the number of entries
4625  * (rows) in the block switch. It is
4626  * set in binit/bio.c by making
4627  * a pass over the switch.
4628  * Used in bounds checking on major
4629  * device numbers.
4630  */
4631 int         nblkdev;
4632
4633 /* Character device switch.
4634  */
4635 struct      cdevsw  {
4636     int     (*d_open)();
4637     int     (*d_close)();
4638     int     (*d_read)();
4639     int     (*d_write)();
4640     int     (*d_sgtty)();
4641 } cdevsw[];
4642 /* ------------------------        */
4643
4644 /* Number of character switch entries.
4645  * Set by cinit/tty.c
4646  */
4647 int         nchrdev;
4648
4649
```



```
4650 /*
4651  * this file is created, along with the file "low.s",
4652  * by the program "mkconf.c", to reflect the actual
4653  * configuration of peripheral devices on a system.
4654  */
4655
4656 int (*bdevsw[])()
4657 {
4658  &nulldev, &nulldev, &rkstrategy, &rktab, /* rk */
4659  &nodev, &nodev, &nodev, 0, /* rp */
4660  &nodev, &nodev, &nodev, 0, /* rf */
4661  &nodev, &nodev, &nodev, 0, /* tm */
4662  &nodev, &nodev, &nodev, 0, /* tc */
4663  &nodev, &nodev, &nodev, 0, /* ns */
4664  &nodev, &nodev, &nodev, 0, /* hr */
4665  &nodev, &nodev, &nodev, 0, /* ht */
4666  0
4667 };
4668
4669 int (*cdevsw[])()
4670 {
4671  &klopen, &klclose, &klread, &klwrite, &klsgtty,
4672                                          /* console */
4673  &pcopen, &pcclose, &pcread, &pcwrite, &nodev,
4674                                          /* pc */
4675  &lpopen, &lpclose, &nodev, &lpwrite, &nodev,
4676                                          /* lp */
4677  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dc */
4678  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dh */
4679  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dp */
4680  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* du */
4681  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* dn */
4682  &nulldev, &nulldev, &mmread, &mmwrite, &nodev,
4683                                          /* mem */
4684  &nulldev, &nulldev, &rkread, &rkwrite, &nodev,
4685                                          /* rk */
4686  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* rp */
4687  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* rf */
4688  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* tm */
4689  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* hs */
4690  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* hp */
4691  &nodev, &nodev, &nodev,  &nodev,  &nodev, /* ht */
4692  0
4693 };
4694
4695 int rootdev ((0<<8)|0);
4696 int swapdev ((0<<8)|0);
4697 int swplo 4000; /* cannot be zero */
4698 int nswap 872;
4699
```

Annexe 5 :

LE FICHIER M45.S

FICHIER M45.S page 1

```
/ MACHINE LANGUAGE ASSIST
/ FOR 11/45 OR 11/70 CPUS

.FPP = 1

/ NON-UNIX INSTRUCTIONS
MFPI    = 6500^TST
MTPI    = 6600^TST
MFPD    = 106500^TST
MTPD    = 106600^TST
SPL     = 230
LDFPS   = 170100^TST
STFPS   = 170200^TST
WAIT    = 1
RTT     = 6
RESET   = 5

HIPRI   = 300
HIGH    = 6

/ MAG TAPE DUMP
/ SAVE REGISTERS IN LOW CORE AND
/ WRITE ALL CORE ONTO MAG TAPE.
/ ENTRY IS THRU 44 ABS

.DATA
.GLOBL  DUMP
DUMP:
        BIT     $1,SSR0
        BNE     DUMP

/ SAVE REGS R0,R1,R2,R3,R4,R5,R6,KIA6
/ STARTING AT ABS LOCATION 4

        MOV     R0,4
        MOV     $6,R0
        MOV     R1,(R0)+
        MOV     R2,(R0)+
        MOV     R3,(R0)+
        MOV     R4,(R0)+
        MOV     R5,(R0)+
        MOV     SP,(R0)+
        MOV     KDSA6,(R0)+

/ DUMP ALL OF CORE (IE TO FIRST MT ERROR)
/ ONTO MAG TAPE. (9 TRACK OR 7 TRACK 'BINARY')

        MOV     $MTC,R0
        MOV     $60004,(R0)+
        CLR     2(R0)
1:
        MOV     $-512.,(R0)
        INC     -(R0)
2:
        TSTB    (R0)
        BGE     2B
        TST     (R0)+
        BGE     1B
        RESET
```

```
/ END OF FILE AND LOOP

        MOV     $60007,-(R0)
        BR      .

.GLOBL  START, _END, _EDATA, _ETEXT, _MAIN

/ 11/45 AND 11/70 STARTUP.
/ ENTRY IS THRU 0 ABS.
/ SINCE CORE IS SHUFFLED,
/ THIS CODE CAN BE EXECUTED BUT ONCE

START:
        INC     $-1
        BNE     .
        RESET
        CLR     PS

/ SET KI0 TO PHYSICAL 0

        MOV     $77406,R3
        MOV     $KISA0,R0
        MOV     $KISD0,R1
        CLR     (R0)+
        MOV     R3,(R1)+

/ SET KI1-6 TO EVENTUAL TEXT RESTING PLACE

        MOV     $_END+63.,R2
        ASH     $-6,R2
        BIC     $!1777,R2
1:
        MOV     R2,(R0)+
        MOV     R3,(R1)+
        ADD     $200,R2
        CMP     R0,$KISA7
        BLOS    1B

/ SET KI7 TO IO SEG FOR ESCAPE

        MOV     $IO,-(R0)

/ SET KD0-7 TO PHYSICAL

        MOV     $KDSA0,R0
        MOV     $KDSD0,R1
        CLR     R2
1:
        MOV     R2,(R0)+
        MOV     R3,(R1)+
        ADD     $200,R2
        CMP     R0,$KDSA7
        BLOS    1B

/ INITIALIZATION
/ GET A TEMP (1-WORD) STACK
/ TURN ON SEGMENTATION
/ COPY TEXT TO I SPACE
/ CLEAR BSS IN D SPACE
```

```
        MOV     $STK+2,SP
        MOV     $65,SSR3                        / 22-BIT, MAP, K+U SEP
        BIT     $20,SSR3
        BEQ     1F
        MOV     $70.,_CPUTYPE
1:
        INC     SSR0
        MOV     $_ETEXT,R0
        MOV     $_EDATA,R1
        ADD     $_ETEXT-8192.,R1
1:
        MOV     -(R1),-(SP)
        MTPI    -(R0)
        CMP     R1,$_EDATA
        BHI     1B
1:
        CLR     (R1)+
        CMP     R1,$_END
        BLO     1B

/ USE KI ESCAPE TO SET KD7 TO IO SEG
/ SET KD6 TO FIRST AVAILABLE CORE

        MOV     $10,-(SP)
        MTPI    *$KDSA7
        MOV     $_ETEXT-8192.+63.,R2
        ASH     $-6,R2
        BIC     $!1777,R2
        ADD     KISA1,R2
        MOV     R2,KDSA6

/ SET UP SUPERVISOR D REGISTERS

        MOV     $6,SISD0
        MOV     $6,SISD1

/ SET UP REAL SP
/ CLEAR USER BLOCK

        MOV     $_U+[USIZE*64.],SP
        MOV     $_U,R0
1:
        CLR     (R0)+
        CMP     R0,SP
        BLO     1B
/       JSR     PC,_ISPROF

/ SET UP PREVIOUS MODE AND CALL MAIN
/ ON RETURN, ENTER USER MODE AT 0R

        MOV     $30000,PS
        JSR     PC,_MAIN
        MOV     $170000,-(SP)
        CLR     -(SP)
        RTT

.GLOBL  TRAP, CALL
.GLOBL  _TRAP

/ ALL TRAPS AND INTERRUPTS ARE
```

```
/ VECTORED THRU THIS ROUTINE.

TRAP:
        MOV     PS,SAVEPS
        TST     NOFAULT
        BNE     1F
        MOV     SSR0,SSR
        MOV     SSR1,SSR+2
        MOV     SSR2,SSR+4
        MOV     $1,SSR0
        JSR     R0,CALL1; _TRAP
        / NO RETURN
1:
        MOV     $1,SSR0
        MOV     NOFAULT,(SP)
        RTT
.TEXT

.GLOBL  _RUNRUN, _SWTCH
CALL1:
        MOV     SAVEPS,-(SP)
        SPL     0
        BR      1F

CALL:
        MOV     PS,-(SP)
1:
        MOV     R1,-(SP)
        MFPD    SP
        MOV     4(SP),-(SP)
        BIC     $!37,(SP)
        BIT     $30000,PS
        BEQ     1F
.IF .FPP
        MOV     $20,_U+4                / FP MAINT MODE
.ENDIF
        JSR     PC,*(R0)+
2:
        SPL     HIGH
        TSTB    _RUNRUN
        BEQ     2F
        SPL     0
        JSR     PC,_SAVFP
        JSR     PC,_SWTCH
        BR      2B
2:
.IF .FPP
        MOV     $_U+4,R1
        BIT     $20,(R1)
        BNE     2F
        MOV     (R1)+,R0
        LDFPS   R0
        MOVF    (R1)+,FR0
        MOVF    (R1)+,FR1
        MOVF    FR1,FR4
        MOVF    (R1)+,FR1
        MOVF    FR1,FR5
        MOVF    (R1)+,FR1
        MOVF    (R1)+,FR2
        MOVF    (R1)+,FR3
```

```
        LDFPS   R0
2:
.ENDIF
        TST     (SP)+
        MTPD    SP
        BR      2F
1:
        BIS     $300000,PS
        JSR     PC,*(R0)+
        CMP     (SP)+,(SP)+
2:
        MOV     (SP)+,R1
        TST     (SP)+
        MOV     (SP)+,R0
        RTT

.GLOBL  _SAVFP
_SAVFP:
.IF .FPP
        MOV     $_U+4,R1
        BIT     $20,(R1)
        BEQ     1F
        STFPS   (R1)+
        MOVF    FR0,(R1)+
        MOVF    FR4,FR0
        MOVF    FR0,(R1)+
        MOVF    FR5,FR0
        MOVF    FR0,(R1)+
        MOVF    FR1,(R1)+
        MOVF    FR2,(R1)+
        MOVF    FR3,(R1)+
1:
.ENDIF
        RTS     PC

.GLOBL  _INCUPC
_INCUPC:
        MOV     R2,-(SP)
        MOV     6(SP),R2        / BASE OF PROF WITH BASE,LENG,OFF,SCALE
        MOV     4(SP),R0        / PC
        SUB     4(R2),R0        / OFFSET
        CLC
        ROR     R0
        MUL     6(R2),R0        / SCALE
        ASHC    $-14.,R0
        INC     R1
        BIC     $1,R1
        CMP     R1,2(R2)        / LENGTH
        BHIS    1F
        ADD     (R2),R1         / BASE
        MOV     NOFAULT,-(SP)
        MOV     $2F,NOFAULT
        MFPD    (R1)
        INC     (SP)
        MTPD    (R1)
        BR      3F
2:
        CLR     6(R2)
3:
        MOV     (SP)+,NOFAULT
```

```
1:
        MOV     (SP)+,R2
        RTS     PC

.GLOBL  _DISPLAY
_DISPLAY:
        DEC     DISPDLY
        BGE     2F
        CLR     DISPDLY
        MOV     PS,-(SP)
        MOV     $HIPRI,PS
        MOV     CSW,R1
        BIT     $1,R1
        BEQ     1F
        BIS     $30000,PS
        DEC     R1
1:
        JSR     PC,FUWORD
        MOV     R0,CSW
        MOV     (SP)+,PS
        CMP     R0,$-1
        BNE     2F
        MOV     $120.,DISPDLY           / 2 SEC DELAY AFTER CSW FAULT
2:
        RTS     PC

/ CHARACTER LIST GET/PUT

.GLOBL  _GETC, _PUTC
.GLOBL  _CFREELIST

_GETC:
        MOV     2(SP),R1
        MOV     PS,-(SP)
        MOV     R2,-(SP)
        SPL     5
        MOV     2(R1),R2        / FIRST PTR
        BEQ     9F              / EMPTY
        MOVB    (R2)+,R0        / CHARACTER
        BIC     $!377,R0
        MOV     R2,2(R1)
        DEC     (R1)+           / COUNT
        BNE     1F
        CLR     (R1)+
        CLR     (R1)+           / LAST BLOCK
        BR      2F
1:
        BIT     $7,R2
        BNE     3F
        MOV     -10(R2),(R1)    / NEXT BLOCK
        ADD     $2,(R1)
2:
        DEC     R2
        BIC     $7,R2
        MOV     _CFREELIST,(R2)
        MOV     R2,_CFREELIST
3:
        MOV     (SP)+,R2
        MOV     (SP)+,PS
        RTS     PC
```

```
9:
        CLR     4(R1)
        MOV     $-1,R0
        MOV     (SP)+,R2
        MOV     (SP)+,PS
        RTS     PC

_PUTC:
        MOV     2(SP),R0
        MOV     4(SP),R1
        MOV     PS,-(SP)
        MOV     R2,-(SP)
        MOV     R3,-(SP)
        SPL     5
        MOV     4(R1),R2        / LAST PTR
        BNE     1F
        MOV     _CFREELIST,R2
        BEQ     9F
        MOV     (R2),_CFREELIST
        CLR     (R2)+
        MOV     R2,2(R1)        / FIRST PTR
        BR      2F
1:
        BIT     $7,R2
        BNE     2F
        MOV     _CFREELIST,R3
        BEQ     9F
        MOV     (R3),_CFREELIST
        MOV     R3,-10(R2)
        MOV     R3,R2
        CLR     (R2)+
2:
        MOVB    R0,(R2)+
        MOV     R2,4(R1)
        INC     (R1)            / COUNT
        CLR     R0
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,PS
        RTS     PC
9:
        MOV     PC,R0
        MOV     (SP)+,R3
        MOV     (SP)+,R2
        MOV     (SP)+,PS
        RTS     PC

.GLOBL  _BACKUP
.GLOBL  _REGLOC
_BACKUP:
        MOV     2(SP),R0
        MOVB    SSR+2,R1
        JSR     PC,1F
        MOVB    SSR+3,R1
        JSR     PC,1F
        MOVB    _REGLOC+7,R1
        ASL     R1
        ADD     R0,R1
        MOV     SSR+4,(R1)
        CLR     R0
```

```
2:
        RTS     PC
1:
        MOV     R1,-(SP)
        ASR     (SP)
```

Annexe 6 :

LE FICHIER TRAP.C

LE FICHIER TRAP.C page1 .

```
#
#INCLUDE "../PARAM.H"
#INCLUDE "../SYSTM.H"
#INCLUDE "../USER.H"
#INCLUDE "../PROC.H"
#INCLUDE "../REG.H"
#INCLUDE "../SEG.H"

#DEFINE EBIT    1               /* USER ERROR BIT IN PS: C-BIT */
#DEFINE UMODE   0170000         /* USER-MODE BITS IN PS WORD */
#DEFINE SETD    0170011         /* SETD INSTRUCTION */
#DEFINE SYS     0104400         /* SYS (TRAP) INSTRUCTION */
#DEFINE USER    020             /* USER-MODE FLAG ADDED TO DEV */

/*
 * STRUCTURE OF THE SYSTEM ENTRY TABLE (SYSENT.C)
 */
STRUCT SYSENT   {
        INT     COUNT;          /* ARGUMENT COUNT */
        INT     (*CALL)();      /* NAME OF HANDLER */
} SYSENT[64];

/*
 * OFFSETS OF THE USER'S REGISTERS RELATIVE TO
 * THE SAVED R0. SEE REG.H
 */
CHAR    REGLOC[9]
{
        R0, R1, R2, R3, R4, R5, R6, R7, RPS
};

/*
 * CALLED FROM L40.S OR L45.S WHEN A PROCESSOR TRAP OCCURS.
 * THE ARGUMENTS ARE THE WORDS SAVED ON THE SYSTEM STACK
 * BY THE HARDWARE AND SOFTWARE DURING THE TRAP PROCESSING.
 * THEIR ORDER IS DICTATED BY THE HARDWARE AND THE DETAILS
 * OF C'S CALLING SEQUENCE. THEY ARE PECULIAR IN THAT
 * THIS CALL IS NOT 'BY VALUE' AND CHANGED USER REGISTERS
 * GET COPIED BACK ON RETURN.
 * DEV IS THE KIND OF TRAP THAT OCCURRED.
 */
TRAP(DEV, SP, R1, NPS, R0, PC, PS)
{
        REGISTER I, A;
        REGISTER STRUCT SYSENT *CALLP;

        SAVFP();
        IF ((PS&UMODE) == UMODE)
                DEV =+ USER;
        U.U_ARG = &R0;
        SWITCH(DEV) {

        /*
         * TRAP NOT EXPECTED.
         * USUALLY A KERNEL MODE BUS ERROR.
         * THE NUMBERS PRINTED ARE USED TO
         * FIND THE HARDWARE PS/PC AS FOLLOWS.
         * (ALL NUMBERS IN OCTAL 18 BITS)
         *      ADDRESS_OF_SAVED_PS =
         *              (KA6*0100) + APS - 0140000;
```

```
 *      ADDRESS_OF_SAVED_PC =
 *              ADDRESS_OF_SAVED_PS - 2;
 */
DEFAULT:
        PRINTF("KA6 = %O\N", *KA6);
        PRINTF("APS = %O\N", &PS);
        PRINTF("TRAP TYPE %O\N", DEV);
        PANIC("TRAP");

CASE 0+USER: /* BUS ERROR */
        I = SIGBUS;
        BREAK;

/*
 * IF ILLEGAL INSTRUCTIONS ARE NOT
 * BEING CAUGHT AND THE OFFENDING INSTRUCTION
 * IS A SETD, THE TRAP IS IGNORED.
 * THIS IS BECAUSE C PRODUCES A SETD AT
 * THE BEGINNING OF EVERY PROGRAM WHICH
 * WILL TRAP ON CPUS WITHOUT 11/45 FPU.
 */
CASE 1+USER: /* ILLEGAL INSTRUCTION */
        IF(FUIWORD(PC-2) == SETD && U.U_SIGNAL[SIGINS] == 0)
                GOTO OUT;
        I = SIGINS;
        BREAK;

CASE 2+USER: /* BPT OR TRACE */
        I = SIGTRC;
        BREAK;

CASE 3+USER: /* IOT */
        I = SIGIOT;
        BREAK;

CASE 5+USER: /* EMT */
        I = SIGEMT;
        BREAK;

CASE 6+USER: /* SYS CALL */
        U.U_ERROR = 0;
        PS =& =EBIT;
        CALLP = &SYSENT[FUIWORD(PC-2)&077];
        IF (CALLP == SYSENT) {  /* INDIRECT */
                A = FUIWORD(PC);
                PC =+ 2;
                I = FUWORD(A);
                IF ((I & =077) != SYS)
                        I = 077;        /* ILLEGAL */
                CALLP = &SYSENT[I&077];
                FOR(I=0; I<CALLP->COUNT; I++)
                        U.U_ARG[I] = FUWORD(A =+ 2);
        } ELSE {
                FOR(I=0; I<CALLP->COUNT; I++) {
                        U.U_ARG[I] = FUIWORD(PC);
                        PC =+ 2;
                }
        }
        U.U_DIRP = U.U_ARG[0];
        TRAP1(CALLP->CALL);
```

```
                IF(U.U_INTFLG)
                        U.U_ERROR = EINTR;
                IF(U.U_ERROR < 100) {
                        IF(U.U_ERROR) {
                                PS =| EBIT;
                                R0 = U.U_ERROR;
                        }
                        GOTO OUT;
                }
                I = SIGSYS;
                BREAK;

        /*
         * SINCE THE FLOATING EXCEPTION IS AN
         * IMPRECISE TRAP, A USER GENERATED
         * TRAP MAY ACTUALLY COME FROM KERNEL
         * MODE, IN THIS CASE, A SIGNAL IS SENT
         * TO THE CURRENT PROCESS TO BE PICKED
         * UP LATER.
         */
        CASE 8: /* FLOATING EXCEPTION */
                PSIGNAL(U.U_PROCP, SIGFPT);
                RETURN;

        CASE 8+USER:
                I = SIGFPT;
                BREAK;

        /*
         * IF THE USER SP IS BELOW THE STACK SEGMENT,
         * GROW THE STACK AUTOMATICALLY.
         * THIS RELIES ON THE ABILITY OF THE HARDWARE
         * TO RESTART A HALF EXECUTED INSTRUCTION.
         * ON THE 11/40 THIS IS NOT THE CASE AND
         * THE ROUTINE BACKUP/L40.S MAY FAIL.
         * THE CLASSIC EXAMPLE IS ON THE INSTRUCTION
         *      CMP     -(SP),-(SP)
         */
        CASE 9+USER: /* SEGMENTATION EXCEPTION */
                A = SP;
                IF(BACKUP(U.U_AR0) == 0)
                IF(GROW(A))
                        GOTO OUT;
                I = SIGSEG;
                BREAK;
        }
        PSIGNAL(U.U_PROCP, I);

OUT:
        IF(ISSIG())
                PSIG();
        SETPRI(U.U_PROCP);
}

/*
 * CALL THE SYSTEM-ENTRY ROUTINE F (OUT OF THE
 * SYSENT TABLE). THIS IS A SUBROUTINE FOR TRAP, AND
 * NOT IN-LINE, BECAUSE IF A SIGNAL OCCURS
 * DURING PROCESSING, AN (ABNORMAL) RETURN IS SIMULATED FROM
 * THE LAST CALLER TO SAVU(QSAV); IF THIS TOOK PLACE
```

```
 * INSIDE OF TRAP, IT WOULDN'T HAVE A CHANCE TO CLEAN UP.
 *
 * IF THIS OCCURS, THE RETURN TAKES PLACE WITHOUT
 * CLEARING U_INTFLG; IF IT'S STILL SET, TRAP
 * MARKS AN ERROR WHICH MEANS THAT A SYSTEM
 * CALL (LIKE READ ON A TYPEWRITER) GOT INTERRUPTED
 * BY A SIGNAL.
 */
TRAP1(F)
INT (*F)();
{

        U.U_INTFLG = 1;
        SAVU(U.U_QSAV);
        (*F)();
        U.U_INTFLG = 0;
}

/*
```

Annexe 7 :

LE FICHIER SYSENT.C

FICHIER SYSENT .C

```
/*
 */

/*
 * THIS TABLE IS THE SWITCH USED TO TRANSFER
 * TO THE APPROPRIATE ROUTINE FOR PROCESSING A SYSTEM CALL.
 * EACH ROW CONTAINS THE NUMBER OF ARGUMENTS EXPECTED
 * AND A POINTER TO THE ROUTINE.
 */
INT     SYSENT[]
{
        0, &NULLSYS,                    /*  0 = INDIR */
        0, &REXIT,                      /*  1 = EXIT */
        0, &FORK,                       /*  2 = FORK */
        2, &READ,                       /*  3 = READ */
        2, &WRITE,                      /*  4 = WRITE */
        2, &OPEN,                       /*  5 = OPEN */
        0, &CLOSE,                      /*  6 = CLOSE */
        0, &WAIT,                       /*  7 = WAIT */
        2, &CREAT,                      /*  8 = CREAT */
        2, &LINK,                       /*  9 = LINK */
        1, &UNLINK,                     /* 10 = UNLINK */
        2, &EXEC,                       /* 11 = EXEC */
        1, &CHDIR,                      /* 12 = CHDIR */
        0, &GTIME,                      /* 13 = TIME */
        3, &MKNOD,                      /* 14 = MKNOD */
        2, &CHMOD,                      /* 15 = CHMOD */
        2, &CHOWN,                      /* 16 = CHOWN */
        1, &SBREAK,                     /* 17 = BREAK */
        2, &STAT,                       /* 18 = STAT */
        2, &SEEK,                       /* 19 = SEEK */
        0, &GETPID,                     /* 20 = GETPID */
        3, &SMOUNT,                     /* 21 = MOUNT */
        1, &SUMOUNT,                    /* 22 = UMOUNT */
        0, &SETUID,                     /* 23 = SETUID */
        0, &GETUID,                     /* 24 = GETUID */
        0, &STIME,                      /* 25 = STIME */
        3, &PTRACE,                     /* 26 = PTRACE */
        0, &NOSYS,                      /* 27 = X */
        1, &FSTAT,                      /* 28 = FSTAT */
        0, &NOSYS,                      /* 29 = X */
        1, &NULLSYS,                    /* 30 = SMDATE; INOPERATIVE */
        1, &STTY,                       /* 31 = STTY */
        1, &GTTY,                       /* 32 = GTTY */
        0, &NOSYS,                      /* 33 = X */
        0, &NICE,                       /* 34 = NICE */
        0, &SSLEP,                      /* 35 = SLEEP */
        0, &SYNC,                       /* 36 = SYNC */
        1, &KILL,                       /* 37 = KILL */
        0, &GETSWIT,                    /* 38 = SWITCH */
        0, &NOSYS,                      /* 39 = X */
        0, &NOSYS,                      /* 40 = X */
        0, &DUP,                        /* 41 = DUP */
        0, &PIPE,                       /* 42 = PIPE */
        1, &TIMES,                      /* 43 = TIMES */
        4, &PROFIL,                     /* 44 = PROF */
        0, &NOSYS,                      /* 45 = TIU */
        0, &SETGID,                     /* 46 = SETGID */
        0, &GETGID,                     /* 47 = GETGID */
```

FICHIER SYSENT.C (suite) .

```
2,  &SSIG,          /* 48 = SIG */
0,  &NOSYS,         /* 49 = X */
0,  &NOSYS,         /* 50 = X */
0,  &NOSYS,         /* 51 = X */
0,  &NOSYS,         /* 52 = X */
0,  &NOSYS,         /* 53 = X */
0,  &NOSYS,         /* 54 = X */
0,  &NOSYS,         /* 55 = X */
0,  &NOSYS,         /* 56 = X */
0,  &NOSYS,         /* 57 = X */
0,  &NOSYS,         /* 58 = X */
0,  &NOSYS,         /* 59 = X */
0,  &NOSYS,         /* 60 = X */
0,  &NOSYS,         /* 61 = X */
0,  &NOSYS,         /* 62 = X */
0,  &NOSYS          /* 63 = X */

};
```

Annexe 8 :

LE FICHIER SYS2.C

read
write
rdwr
open
creat
close

FICHIER SYS2.C page 1

```
#
#INCLUDE "../PARAM.H"
#INCLUDE "../SYSTM.H"
#INCLUDE "../USER.H"
#INCLUDE "../REG.H"
#INCLUDE "../FILE.H"
#INCLUDE "../INODE.H"

/*
 * READ SYSTEM CALL
 */
READ()
{
        RDWR(FREAD);
}

/*
 * WRITE SYSTEM CALL
 */
WRITE()
{
        RDWR(FWRITE);
}

/*
 * COMMON CODE FOR READ AND WRITE CALLS:
 * CHECK PERMISSIONS, SET BASE, COUNT, AND OFFSET,
 * AND SWITCH OUT TO READI, WRITEI, OR PIPE CODE.
 */
RDWR(MODE)
{
        REGISTER *FP, M;

        M = MODE;
        FP = GETF(U.U_AR0[R0]);
        IF(FP == NULL)
                RETURN;
        IF((FP->F_FLAG&M) == 0) {
                U.U_ERROR = EBADF;
                RETURN;
        }
        U.U_BASE = U.U_ARG[0];
        U.U_COUNT = U.U_ARG[1];
        U.U_SEGFLG = 0;
        IF(FP->F_FLAG&FPIPE) {
                IF(M==FREAD)
                        READP(FP); ELSE
                        WRITEP(FP);
        } ELSE {
                U.U_OFFSET[1] = FP->F_OFFSET[1];
                U.U_OFFSET[0] = FP->F_OFFSET[0];
                IF(M==FREAD)
                        READI(FP->F_INODE); ELSE
                        WRITEI(FP->F_INODE);
                DPADD(FP->F_OFFSET, U.U_ARG[1]-U.U_COUNT);
        }
        U.U_AR0[R0] = U.U_ARG[1]-U.U_COUNT;
}

/*
```

```
 * OPEN SYSTEM CALL
 */
OPEN()
{
        REGISTER *IP;
        EXTERN UCHAR;

        IP = NAMEI(&UCHAR, 0);
        IF(IP == NULL)
                RETURN;
        U.U_ARG[1]++;
        OPEN1(IP, U.U_ARG[1], 0);
}

/*
 * CREAT SYSTEM CALL
 */
CREAT()
{
        REGISTER *IP;
        EXTERN UCHAR;

        IP = NAMEI(&UCHAR, 1);
        IF(IP == NULL) {
                IF(U.U_ERROR)
                        RETURN;
                IP = MAKNODE(U.U_ARG[1]&07777&(~ISVTX));
                IF (IP==NULL)
                        RETURN;
                OPEN1(IP, FWRITE, 2);
        } ELSE
                OPEN1(IP, FWRITE, 1);
}

/*
 * COMMON CODE FOR OPEN AND CREAT.
 * CHECK PERMISSIONS, ALLOCATE AN OPEN FILE STRUCTURE,
 * AND CALL THE DEVICE OPEN ROUTINE IF ANY.
 */
OPEN1(IP, MODE, TRF)
INT *IP;
{
        REGISTER STRUCT FILE *FP;
        REGISTER *RIP, M;
        INT I;

        RIP = IP;
        M = MODE;
        IF(TRF != 2) {
                IF(M&FREAD)
                        ACCESS(RIP, IREAD);
                IF(M&FWRITE) {
                        ACCESS(RIP, IWRITE);
                        IF((RIP->I_MODE&IFMT) == IFDIR)
                                U.U_ERROR = EISDIR;
                }
        }
        IF(U.U_ERROR)
                GOTO OUT;
        IF(TRF)
```

```
                ITRUNC(RIP);
        PRELE(RIP);
        IF ((FP = FALLOC()) == NULL)
                GOTO OUT;
        FP->F_FLAG = M&(FREAD+FWRITE);
        FP->F_INODE = RIP;
        I = U.U_AR0[R0];
        OPENI(RIP, M&FWRITE);
        IF(U.U_ERROR == 0)
                RETURN;
        U.U_OFILE[I] = NULL;
        FP->F_COUNT--;

OUT:
        IPUT(RIP);
}

/*
 * CLOSE SYSTEM CALL.
 */
CLOSE()
{
        REGISTER *FP;

        FP = GETF(U.U_AR0[R0]);
        IF(FP == NULL)
                RETURN;
        U.U_OFILE[U.U_AR0[R0]] = NULL;
        CLOSEF(FP);
}

/*
 * SEEK SYSTEM CALL.
 */
SEEK()
{
        INT N[2];
        REGISTER *FP, T;

        FP = GETF(U.U_AR0[R0]);
        IF(FP == NULL)
                RETURN;
        IF(FP->F_FLAG&FPIPE) {
                U.U_ERROR = ESPIPE;
                RETURN;
        }
        T = U.U_ARG[1];
        IF(T > 2) {
                N[1] = U.U_ARG[0]<<9;
                N[0] = U.U_ARG[0]>>7;
                IF(T == 3)
                        N[0] =& 0777;
        } ELSE {
                N[1] = U.U_ARG[0];
                N[0] = 0;
                IF(T!=0 && N[1]<0)
                        N[0] = -1;
        }
        SWITCH(T) {
```

```
        CASE 1:
        CASE 4:
                N[0] =+ FP->F_OFFSET[0];
                DPADD(N, FP->F_OFFSET[1]);
                BREAK;

        DEFAULT:
                N[0] =+ FP->F_INODE->I_SIZE0&0377;
                DPADD(N, FP->F_INODE->I_SIZE1);

        CASE 0:
        CASE 3:
                ;
        }
        FP->F_OFFSET[1] = N[1];
        FP->F_OFFSET[0] = N[0];
}

/*
 * LINK SYSTEM CALL
 */
LINK()
{
        REGISTER *IP, *XP;
        EXTERN UCHAR;

        IP = NAMEI(&UCHAR, 0);
        IF(IP == NULL)
                RETURN;
        IF(IP->I_NLINK >= 127) {
                U.U_ERROR = EMLINK;
                GOTO OUT;
        }
        IF((IP->I_MODE&IFMT)==IFDIR && !SUSER())
                GOTO OUT;
        /*
         * UNLOCK TO AVOID POSSIBLY HANGING THE NAMEI
         */
        IP->I_FLAG =& ~ILOCK;
        U.U_DIRP = U.U_ARG[1];
        XP = NAMEI(&UCHAR, 1);
        IF(XP != NULL) {
                U.U_ERROR = EEXIST;
                IPUT(XP);
        }
        IF(U.U_ERROR)
                GOTO OUT;
        IF(U.U_PDIR->I_DEV != IP->I_DEV) {
                IPUT(U.U_PDIR);
                U.U_ERROR = EXDEV;
                GOTO OUT;
        }
        WDIR(IP);
        IP->I_NLINK++;
        IP->I_FLAG =| IUPD;

OUT:
        IPUT(IP);
}
```

FICHIER SYS2.C Page 5

```
/*
 * MKNOD SYSTEM CALL
 */
MKNOD()
{
        REGISTER *IP;
        EXTERN UCHAR;

        IF(SUSER()) {
                IP = NAMEI(&UCHAR, 1);
                IF(IP != NULL) {
                        U.U_ERROR = EEXIST;
                        GOTO OUT;
                }
        }
        IF(U.U_ERROR)
                RETURN;
        IP = MAKNODE(U.U_ARG[1]);
        IF (IP==NULL)
                RETURN;
        IP->I_ADDR[0] = U.U_ARG[2];

OUT:
        IPUT(IP);
}

/*
 * SLEEP SYSTEM CALL
 * NOT TO BE CONFUSED WITH THE SLEEP INTERNAL ROUTINE.
 */
SSLEP()
{
        CHAR *D[2];

        SPL7();
        D[0] = TIME[0];
        D[1] = TIME[1];
        DPADD(D, U.U_ARG0[R0]);

        WHILE(DPCMP(D[0], D[1], TIME[0], TIME[1]) > 0) {
                IF(DPCMP(TOUT[0], TOUT[1], TIME[0], TIME[1]) <= 0 ||
                   DPCMP(TOUT[0], TOUT[1], D[0], D[1]) > 0) {
                        TOUT[0] = D[0];
                        TOUT[1] = D[1];
                }
                SLEEP(TOUT, PSLEP);
        }
        SPL0();
}
```

Annexe 9:

LE FICHIER RDWRI.C

readi

writei

FICHIER RDWRI.C

```
#
/*
 */

#INCLUDE "../PARAM.H"
#INCLUDE "../INODE.H"
#INCLUDE "../USER.H"
#INCLUDE "../BUF.H"
#INCLUDE "../CONF.H"
#INCLUDE "../SYSTM.H"

/*
 * READ THE FILE CORRESPONDING TO
 * THE INODE POINTED AT BY THE ARGUMENT.
 * THE ACTUAL READ ARGUMENTS ARE FOUND
 * IN THE VARIABLES:
 *      U.U_BASE            CORE ADDRESS FOR DESTINATION
 *      U.U_OFFSET          BYTE OFFSET IN FILE
 *      U.U_COUNT           NUMBER OF BYTES TO READ
 *      U.U_SEGFLG          READ TO KERNEL/USER
 */
READI(AIP)
STRUCT INODE *AIP;
{
        INT *BP;
        INT LBN, BN, DN;
        REGISTER DN, N;
        REGISTER STRUCT INODE *IP;

        IP = AIP;
        IF(U.U_COUNT == 0)
                RETURN;
        IP->I_FLAG =| IACC;
        IF((IP->I_MODE&IFMT) == IFCHR) {
                (*CDEVSW[IP->I_ADDR[0].D_MAJOR].D_READ)(IP->I_ADDR[0]);
                RETURN;
        }

        DO {
                LBN = BN = LSHIFT(U.U_OFFSET, -9);
                DN = U.U_OFFSET[1] & 0777;
                N = MIN(512-DN, U.U_COUNT);
                IF((IP->I_MODE&IFMT) != IFBLK) {
                        DN = DPCMP(IP->I_SIZE0&0377, IP->I_SIZE1,
                                U.U_OFFSET[0], U.U_OFFSET[1]);
                        IF(DN <= 0)
                                RETURN;
                        N = MIN(N, DN);
                        IF ((BN = BMAP(IP, LBN)) == 0)
                                RETURN;
                        DN = IP->I_DEV;
                } ELSE {
                        DN = IP->I_ADDR[0];
                        RABLOCK = BN+1;
                }
                IF (IP->I_LASTR+1 == LBN)
                        BP = BREADA(DN, BN, RABLOCK);
                ELSE
                        BP = BREAD(DN, BN);
                IP->I_LASTR = LBN;
```

```
                IOMOVE(BP, ON, N, B_READ);
                BRELSE(BP);
        } WHILE(U.U_ERROR==0 && U.U_COUNT!=0);
}

/*
 * WRITE THE FILE CORRESPONDING TO
 * THE INODE POINTED AT BY THE ARGUMENT.
 * THE ACTUAL WRITE ARGUMENTS ARE FOUND
 * IN THE VARIABLES:
 *      U_BASE          CORE ADDRESS FOR SOURCE
 *      U_OFFSET        BYTE OFFSET IN FILE
 *      U_COUNT         NUMBER OF BYTES TO WRITE
 *      U_SEGFLG        WRITE TO KERNEL/USER
 */
WRITEI(AIP)
STRUCT INODE *AIP;
{
        INT *BP;
        INT N, ON;
        REGISTER DN, BN;
        REGISTER STRUCT INODE *IP;

        IP = AIP;
        IP->I_FLAG =| IACC|IUPD;
        IF((IP->I_MODE&IFMT) == IFCHR) {
                (*CDEVSW[IP->I_ADDR[0].D_MAJOR].D_WRITE)(IP->I_ADDR[0]);
                RETURN;
        }
        IF (U.U_COUNT == 0)
                RETURN;

        DO {
                BN = LSHIFT(U.U_OFFSET, -9);
                ON = U.U_OFFSET[1] & 0777;
                N = MIN(512-ON, U.U_COUNT);
                IF((IP->I_MODE&IFMT) != IFBLK) {
                        IF ((BN = BMAP(IP, BN)) == 0)
                                RETURN;
                        DN = IP->I_DEV;
                } ELSE
                        DN = IP->I_ADDR[0];
                IF(N == 512)
                        BP = GETBLK(DN, BN); ELSE
                        BP = BREAD(DN, BN);
                IOMOVE(BP, ON, N, B_WRITE);
                IF(U.U_ERROR != 0)
                        BRELSE(BP); ELSE
                IF ((U.U_OFFSET[1]&0777)==0)
                        BAWRITE(BP); ELSE
                        BDWRITE(BP);
                IF(DPCMP(IP->I_SIZE0&0377, IP->I_SIZE1,
                  U.U_OFFSET[0], U.U_OFFSET[1]) < 0 &&
                  (IP->I_MODE&(IFBLK&IFCHR)) == 0) {
                        IP->I_SIZE0 = U.U_OFFSET[0];
                        IP->I_SIZE1 = U.U_OFFSET[1];
                }
                IP->I_FLAG =| IUPD;
        } WHILE(U.U_ERROR==0 && U.U_COUNT!=0);
}
```

Annexe 10:

LE FICHIER SUBR.C.

passc

cpass

LE FICHIER SUBR.C.

```
        IF(I < 255)
                RABLOCK = BAP[I+1];
        RETURN(NB);
}

/*
 * PASS BACK  C  TO THE USER AT HIS LOCATION U.BASE;
 * UPDATE U.BASE, U.COUNT, AND U.OFFSET.  RETURN -1
 * ON THE LAST CHARACTER OF THE USER'S READ.
 * U.BASE IS IN THE USER ADDRESS SPACE UNLESS U.SEGFLG IS SET.
 */
PASSC(C)
CHAR C;
{

        IF(U.U.SEGFLG)
                *U.U.BASE = C; ELSE
                IF(SUBYTE(U.U.BASE, C) < 0) {
                        U.U.ERROR = EFAULT;
                        RETURN(-1);
                }
        U.U.COUNT--;
        IF(++U.U.OFFSET[1] == 0)
                U.U.OFFSET[0]++;
        U.U.BASE++;
        RETURN(U.U.COUNT == 0? -1: 0);
}

/*
 * PICK UP AND RETURN THE NEXT CHARACTER FROM THE USER'S
 * WRITE CALL AT LOCATION U.BASE;
 * UPDATE U.BASE, U.COUNT, AND U.OFFSET.  RETURN -1
 * WHEN U.COUNT IS EXHAUSTED.  U.BASE IS IN THE USER'S
 * ADDRESS SPACE UNLESS U.SEGFLG IS SET.
 */
CPASS()
{
        REGISTER C;

        IF(U.U.COUNT == 0)
                RETURN(-1);
        IF(U.U.SEGFLG)
                C = *U.U.BASE; ELSE
                IF((C=FUBYTE(U.U.BASE)) < 0) {
                        U.U.ERROR = EFAULT;
                        RETURN(-1);
                }
        U.U.COUNT--;
        IF(++U.U.OFFSET[1] == 0)
                U.U.OFFSET[0]++;
        U.U.BASE++;
        RETURN(C&0377);
}

/*
 * ROUTINE WHICH SETS A USER ERROR; PLACED IN
 * ILLEGAL ENTRIES IN THE BDEVSW AND CDEVSW TABLES.
 */
NODEV()
{
```

Annexe 11 :

NOTES CONCERNANT LE LECTEUR DE CARTES CR.11

DANS LE "PERIPHERALS HANDBOOK" DE DEC.

CONTROLS & INDICATORS

Front Panel

Control or Indicator	Type	Function
POWER switch	alternate-action pushbutton/ indicator switch	Controls application of all power to the card reader. When indicator is off, depressing switch applies power to reader and causes associated indicator to light. When indicator is lit, depressing switch removes all power from reader and causes indicator to go out.
READ CHECK indicator	white light	When lit, this light indicates that the card just read may be torn on the leading or trailing edges, or that the card may have punches in the 0 or 81st column positions. Because READ CHECK indicates an error condition, whenever this indicator is lit, it causes the card reader to stop operation and extinguishes the RESET indicator.
PICK CHECK indicator	white light	When lit, this light indicates that the card reader failed to move a card into the read station after it received a READ COMMAND from the controller. Stops card reader operation and extinguishes RESET indicator.
STACK CHECK indicator	white light	When lit, this light indicates that the previous card was not properly seated in the output stacker and therefore may be badly mutilated. Stops card reader operation and extinguishes RESET indicator.
HOPPER CHECK indicator	white light	When lit, this light indicates that either the input hopper is empty or that the output stacker is full.

		In either case, the operator must manually correct the condition before card reader operation can continue.
STOP switch	momentary pushbutton/ indicator switch (red light)	When depressed, immediately lights and drops the READY line, thereby extinguishing the RESET indicator. Card reader operation then stops as soon as the card currently in the read station has been read. This switch has no effect on the system power; it only stops the current operation.
RESET switch	momentary pushbutton/ indicator switch (green light)	When depressed and released, clears all error flip-flops and initializes card reader logic. Associated RESET indicator lights to indicate that the READY signal is applied to the controller. The RESET indicator goes out whenever the STOP switch is depressed or whenever an error indicator lights (READ CHECK, PICK CHECK, STACK CHECK, or HOPPER CHECK).

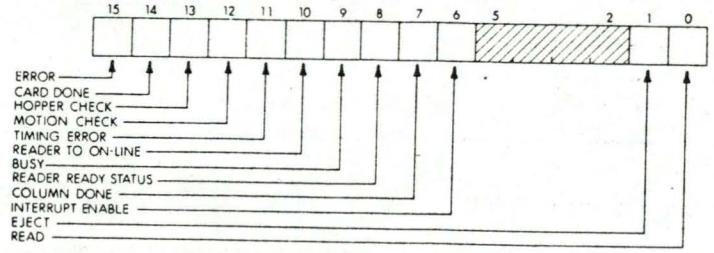
Rear Panel

Control	Type	Function
LAMP TEST switch	pushbutton	When depressed, illuminates all indicators on the front control panel to determine if any of the indicator lamps are faulty.
SHUTDOWN switch	2-position toggle	Controls automatic operation of the input hopper blower. MAN position—blower operates continuously whether or not cards are in the input hopper. AUTO position—causes the blower to shut down automatically whenever the input hopper is emptied. Blower automatically restarts when

		cards are loaded into the hopper and the RESET switch is depressed.
		Blower activates approximately three seconds after RESET is depressed.
MODE switch	2-position toggle	Permits selection of either on-line or off-line operation.
		LOCAL position—removes the READ COMMAND input from the controller to allow the operator to run the reader off-line by using the RESET and STOP switches on the front control panel.
		REMOTE position—enables the READ COMMAND input from the controller to allow normal on-line operation under program control once RESET is depressed.

REGISTERS

Card Reader Status Register (CRS) 777 160



Effect of the Initialize (INIT) signal: clear bits 15, 14, 11, 10, 7, 6, 1, and 0.

Read only: bits 15 through 7
Write only: bit 0

BIT	NAME	FUNCTION
15	Error	Set when an error occurs.
14	Card Done	Set when one card has passed through the read station and another one may be demanded from the input hopper.

13	Hopper Check	Set when the input hopper is empty or output stack is full. This signal is provided by mark sense card readers and later models of the punched card units.
12	Motion Check	Set to indicate abnormal condition in the card reader. Three conditions can cause this bit to be set: a) Feed error b) Motion error c) Stack Fail These signals are available from the mark sense readers and later models of the punched card units.
11	Timing Error	Set when a new column of data arrived into the CRB before the previously loaded column was attended to by a program.
10	Reader To on Line	Set when the reader is on-line. Sensing an error or operating the stop switch on the card reader panel causes the reader to go off-line. Operating the start switch brings the reader on-line providing no error causing condition exists.
9	Busy	Set when a card is being read.
8	Reader Ready Status	Set when the reader is off line; O indicates on-line and hence ready to accept read commands.
7	Column Done	Set when a column of data is ready in CRB.
6	Interrupt Enable	Set to allow Card Done, Column Done, or Error = 1 to cause an interrupt.
1	Eject	When set, column ready flag is inhibited from setting. However, data transfers between card reader and data buffer do take place.
0	Read	Set to allow the feed mechanism to deliver a card to the read station.

A program can load and read information from the Card Reader Status (CRS) register using appropriate instructions and considering the following limitations:

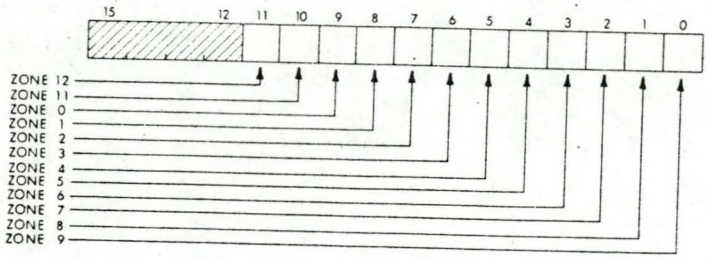
a. Bits 15-7 can only be read on the bus.

b. COLUMN DONE bit is automatically cleared by reading the Data Buffer.

c. Bits 15-8 are automatically cleared when an attempt to load the status register is made. However, if this loading is to read a card, and an error condition requiring manual intervention has not been attended by the operator, appropriate error bit will be set again to cause an interrupt. Commands to READ CARD under these circumstances is not honored.

d. BIT 0 is always read as zero on the bus.

Card Reader Data Buffer Register (CRB1, CRB2) 777 162, 777 164



Read only: all bits

Data from one column at a time of the card is loaded into this register.

BIT	FUNCTION
11	ZONE 12
10	ZONE 11
9	ZONE 0
8	ZONE 1
7	ZONE 2
6	ZONE 3
5	ZONE 4
4	ZONE 5
3	ZONE 6
2	ZONE 7
1	ZONE 8
0	ZONE 9

If the data buffer is addressed at CRB2, the 12-bit content is compressed into an 8-bit character by an encoding network before getting on to the bus as low order byte. The 8-bit code is:

BIT	FUNCTION
7	ZONE 12
6	ZONE 11
5	ZONE 0
4	ZONE 9
3	ZONE 8
2-0	DATA encoded as follows:

000 = no punches, ZONE 1-7

001 = ZONE 1
010 = ZONE 2
011 = ZONE 3
100 = ZONE 4
101 = ZONE 5
110 = ZONE 6
111 = ZONE 7

In case of multiple zones twice, bits will be the inclusive OR of the octal codes of the zones.

PROGRAMMING EXAMPLE

The following example shows a typical method of programming the CR11 Card Reader System. In this example, the card reader is used to read a bootstrap loader program from punched cards and load the program into core memory.

```
        CRS=777160          ;CARD READER STATUS REGISTER
        CRB1=777162         ;12-BIT DATA BUFFER
        .=1000              ;STARTING ADDRESS FOR MEMORY

        R1=%1
        R2=%2
        R3=%3
        R4=%4

START:  MOV   #CRS, R1      ;SET UP ADDRESS OF CRS IN R1
        MOV   #CRB1, R3     ;ADD DATA BUFFER ADDRESS IN R3

RTST:   BIT   @R1, #1400    ;IS READER ON-LINE?
        BNE   RTST          ;NO, SO MAY AS WELL WAIT.

RDCD:   INC   @R1           ;O.K., READ A CARD

RCHK:   BIT   @R1, #140000  ;SPECIAL CONDITION OR CARD
                            ;DONE SET?
        BGT   RDCD          ;SPECIAL CONDITION OFF BUT CARD
                            ;DONE ON.
        BEQ   GOGO          ;BOTH OFF
```

```
END:    RESET
        JMP     @R2             ;SPECIAL CONDITION ON, ASSUME
                                ;HOPPER EMPTY AND BRANCH TO
                                ;PROGRAM

GOGO:   TSTB    @R1             ;COLUMN READY?
        BPL     RCHK            ;NO, KEEP LOOKING.
        BIT     @R3,    #400    ;ROW 1 IN THIS BYTE?
        BEQ     GO2             ;NO, MUST BE DATA
        MOVB    @R3,    R2      ;YES, IS FIRST ADDRESS BYTE
        SWAB    R2              ;MOVE TO HIGH-ORDER BYTE
        MOV     R1,     R4      ;AND SET SECOND-ADD.-BYTE FLAG
        BR      RCHK            ;AND GET NEXT COLUMN

GO2:    MOV     R4,     R4      ;TEST SECOND-ADD.-BYTE FLAGS
        BNE     GO3             ;IF ON, USE THIS FOR ADD. BYTE
        MOVB    @R3,    (R2)+   ;OTHERWISE, STORE IT IN MEMORY
        BR      RCHK            ;AND GET NEXT BYTE.

GO3:    ADD     @R3,    R2      ;COMPLETE ADDRESS MAKEUP
        CLR     R4              ;RESET SECOND-ADD.-BYTE FLAG
        BR      RCHK            ;AND GO AROUND

        .END
```

SPECIFICATIONS

Main Specifications

Input medium:	80-column punched cards
Speed:	285 cards/minute
Hopper capacity:	550 cards

Register Addresses

Card Reader Status (CRS)	777 160	
Card Reader Buffer (CRB1)	777 162	(12-bit characters)
Card Reader Buffer (CRB2)	777 164	(8-bit char, compressed)

UNIBUS Interface

Interrupt vector address:	230
Priority level:	BR6
Bus loading:	1 bus load

Mechanical

Mounting:	1 table-top unit + 1 SPC slot (quad module)
Size:	11"H x 19"W x 14"D
Weight:	60 lbs

Power

Starting current:	12 A at 115 VAC
Running current:	6 A at 115 VAC
Current for Control Unit:	1.5 A at +5 V
Heat dissipation:	600 W

Environmental

Operating temperature:	15°C to 32°C
Relative humidity:	20% to 80%

Models

CR11:	Punched card reader and control, 115 VAC, 60 Hz
CR11-A:	Punched card reader and control, 230 VAC, 50 Hz
CM11-FA:	Mark-sense card reader and control, 115 VAC, 60 Hz
CM11-FB:	Mark-sense card reader and control, 230 VAC, 50 Hz

Annexe 12:

LE DRIVER DU LECTEUR DE CARTES CR.C.

```
#

/*

                CR11 PUNCHED CARD READER DRIVER

*/


#INCLUDE "../PARAM.H"
#INCLUDE "../CONF.H"
#INCLUDE "../USER.H"

#DEFINE         CRADDR          0777160

#DEFINE         READ            01
#DEFINE         EJECT           02
#DEFINE         IENABLE         0100
#DEFINE         DONE            0200
#DEFINE         READY           0400
#DEFINE         BUSY            01000
#DEFINE         ONLINE          02000
#DEFINE         TIMING          04000
#DEFINE         MOTION          010000
#DEFINE         HOPPER          020000
#DEFINE         CDONE           040000
#DEFINE         ERROR           0100000

#DEFINE         CLOSED          0
#DEFINE         WAITING         1
#DEFINE         READING         2
#DEFINE         EOF             3

#DEFINE         CRPRI           30

STRUCT {
        INT CRSR;
        INT CRNCHR;
        INT CRCHR;
       } ;

CHAR BUFFER[81] ;
CHAR ARRAY[256]
{
040,    061,    062,    063,    064,    065,    066,    067,
070,    0377,   072,    043,    0100,   047,    075,    042,
071,    0377,   026,    0377,   0377,   036,    0377,   04,
0377,   0377,   0377,   0377,   024,    025,    0377,   032,
060,    057,    0123,   0124,   0125,   0126,   0127,   0130,
0131,   0377,   0134,   054,    045,    055,    076,    077,
0132,   0377,   054,    0377,   0377,   012,    027,    033,
0377,   0377,   0377,   0377,   0377,   05,     06,     07,
055,    0112,   0113,   0114,   0115,   0116,   0117,   0120,
0121,   0377,   041,    044,    052,    051,    073,    0377,
0122,   021,    022,    0377,   0377,   0377,   010,    0377,
030,    031,    0377,   0377,   0377,   0377,   0377,   0377,
0175,   0176,   0163,   0164,   0165,   0166,   0167,   0170,
0171,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0172,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
```

```
0377,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
046,    0101,   0102,   0103,   0104,   0105,   0106,   0107,
0110,   0377,   0377,   056,    074,    050,    053,    0174,
0111,   01,     02,     03,     0377,   011,    0377,   0177,
0377,   0377,   0377,   0377,   014,    015,    016,    017,
0173,   0141,   0142,   0143,   0144,   0145,   0146,   0147,
0150,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0151,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0377,   0,      0377,   0377,   0377,   0377,   0377,   0377,
0377,   0152,   0153,   0154,   0155,   0156,   0157,   0160,
0161,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0162,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0377,   020,    0377,   0377,   0377,   0377,   0377,   0377,
0377,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0377,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0377,   0377,   0377,   0377,   0377,   0377,   0377,   0377,
0377,   0377,   0174,   0377,   0377,   0377,   0377,   0377
};

;

INT I ;

INT CRSTATE ;


CROPEN()
{
IF(CRSTATE != CLOSED)
        {
        U.U_ERROR = ENXIO ;
        RETURN;
        }
I = 0 ;
CRSTATE = WAITING ;
CRADDR->CRSR = IENABLE ;
CRINPUT() ;
SLEEP(&CRSTATE,CRPRI) ;
}

CRINT()
{

IF(CRADDR->CRSR&DONE)
        {
        BUFFER[I] = CRADDR->CRCBR ;
        I++ ;
        RETURN ;
        }

IF(CRADDR->CRSR&CDONE)
        {
        IF(CRSTATE == WAITING)
                {
                CRSTATE = READING ;
                WAKEUP(&CRSTATE) ;
                }
        BUFFER[80] = 065 ;
        I=0 ;
        WAKEUP(&BUFFER);
        RETURN ;
        }
```

```
IF(CRADDR->CRSR&ERROR)
        {
        IF(CRADDR->CRSR&(MOTION + TIMING))
                {
                U.U_ERROR = EIO ;
                CRSTATE = EOF ;
                }

        CRADDR->CRSR =& 011111 ;
        RETURN ;
        }
WAKEUP(&I) ;
}

CRREAD()
{

INT Y ;
DO
{
WHILE(I > 80 )
        {
        IF(CRSTATE == EOF)
                RETURN ;
       I= 0 ;
        CRINPUT() ;

        SLEEP(&BUFFER,CRPRI) ;
        }
Y = BUFFER[I++] ;
Y =& 0377 ;
}
WHILE(PASSC(ARRAY[Y]) >= 0) ;
}

CRCLOSE()
{
CRSTATE = CLOSED ;
}


CRINPUT()
{
WHILE(1)
        {
        IF((CRADDR->CRSR & READY) == 0 )
                {
                CRADDR->CRSR++;
                RETURN ;
                }
        CRADDR->CRSR =& 0175777 ;
        SLEEP(&I,CRPRI);
        }
}
```

Annexe 13:

LE FICHIER SPOOL.C.

```
#

# DEFINE        MP      0177777

# DEFINE        BEG     1
# DEFINE        END     2
# DEFINE        ERROR   3


# DEFINE        CLOSED  0
# DEFINE        OPENED  1

INT II,CARTE, FILE, FDOCR, FDRCR, FDWSF, FDOSF, I, J ;
CHAR BUFFER[81] ;
CHAR BEGIN[5] "FILE " ;
CHAR ENDCARD[4] "END " ;



/*  ATTENTION : FILE ET END SONT EN MAJUSCULES */
CHAR FILENAME[26] "/USR/CR/" ;


/* LE PROGRAMME SPOOL.C SERT A LIRE UN DECK DE CARTE ET DE
   CREER UN FICHIER DISQUE QUI EST UNE COPIE DE CES CARTES
   LE PROGRAMME SERA EN EXECUTION PERMANENTE ; IL SUFFIRA DE
   PLACER SES CARTES DANS LE LECTEUR ET CELLES-CI SERONT LUES .
   L ENSEMLE DES CARTES DOIVENT ETRE PRECEDEE D UNE CARTE DEBUT
   ET SUIVI D UNE CARTE FIN ( VOIR SPECIFICATIONS DU PROGRAMME )
*/

MAIN()
{


FILE = CLOSED ;

                /* FILE EST UN ENTIER QUI VAUT 1 ( OPENED )  SI ON A OU
                                FICHIER DANS LE SPOOL
                */

OPE:
FDOCR = OPEN("/DEV/CR",0);
IF(FDOCR == -1)
                {
                PRINTF(" CANNOT OPEN /DEV/CR  ---  CR SPOOL PROGRAM ");
                SLEEP(5);
                GOTO OPE;
                }

/* ON A PU OUVRIR /DEV/CR ON LANCE UNE LECTURE  */
```

```
READCARD:
FDRCR = READ(FDOCR,BUFFER,81);
IF(FDRCR == 81) GOTO N81 ;

PRINTF("  READ ERROR ---- CARD READER  ") ;
IF(FILE == 1 )
        {
        CLOSE(FDOSF);
        FILE = CLOSED ;
        }

GOTO READCARD ;
/* LE NOMBRE DE CARACTERE LU = 81        */


N81:
II = BUFFER[0] ;
IF(II == MP) GOTO ANALYSE ;
IF(FILE==1)
        {

        WRITE(FDOSF,BUFFER,81);
        }
GOTO READCARD ;

/*
SI ON A LU UNE CARTE SPECIALE ( 1ER CARACTERE = MP )
ON L ANALYSE SINON, DANS LE CAS OU UN FICHIER EST OUVERT
ON ECRIT LA CARTE ET ON RETOURNE A LA LECTURE
*/


ANALYSE:
/* ON A LU UNE CARTE SPECIALE, ON DOIT L ANALYSER AFIN
DE VOIR S IL S AGIT D UNE CARTE DEBUT,FIN OU D UNE CARTE
   ERRONEE. POUR CELA,ON APPELLE UNE ROUTINE (ANA) QUI
   RENVOIT DANS L ENTIER CARTE LA VALEUR BEG END OU ERROR
   (DEBUT,FIN,ERREUR) SELON LE CAS
*/

ANA();
IF(CARTE == BEG)
                {
                IF(FILE == OPENED) CLOSE(FDOSF) ;
                FDOSF = CREAT(FILENAME,0666);
                IF(FDOSF == -1)
                                {
                                PRINTF("CANNOT OPEN FILE IN SPOOL \N");
                                GOTO OUT;
                                }
                FILE = OPENED ;
                GOTO READCARD ;
                }


IF(CARTE == END)
                {
                IF(FILE == OPENED)
                                {
```

```
                    CLOSE(FDOSF);
                    FILE = CLOSED;
                    }
                GOTO READCARD;
                }


IF(CARTE == ERROR)
        {
        IF(FILE == OPENED) WRITE(FDOSF,BUFFER,81) ;
        GOTO READCARD ;
        }


OUT: ;

}



ANA()
{

/*

LA ROUTINE ANA ANALYSE UNE CARTE AFIN D'ESSAYER D'Y
RECONNAITRE UNE CARTE DEBUT, UNE CARTE FIN .
S'IL NE S AGIT NI DE L'UNE NI DE L'AUTRE IL S AGIT
D UNE CARTE ERREUR.
ON APPELLE ANA LORSQUE LE 1ER CARACTERE DE LA CARTE
EST MULTI PUNCH .
ANA POSITIONNE LA VAR GLOBALE CARTE AUX VALEURS
BEG END ERROR  SELON QU IL S AGIT D UNE CARTE DEBUT FIN OU ERRONEE

*/

INT X ;
I=1;
J=8;
EJECT();

IF(CARTE == ERROR) RETURN ;

FOR(X=I;X <= I+4; X++)
        IF(BEGIN[X-I] != BUFFER[X])
                GOTO TESTEND;

/*
LA PREMIERE ZONE DE LA CARTE EST FILE ON S ATTEND DONC A AVOIR
UNE CARTE DEBUT
*/


I =+ 5 ;
CARTE = BEG ;
                    EJECT();
```

```
IF(CARTE == ERROR) RETURN ;
TRANSF();
IF(CARTE == ERROR) RETURN ;
FILENAME[J++] = ',' ;
EJECT();
IF(CARTE == ERROR) RETURN ;
TRANSF();
IF(CARTE == ERROR) RETURN ;
FILENAME[J] ='\0' ;
RETURN ;


TESTEND:
FOR(X=I;X <= I+3;X++)
        IF(ENDCARD[X-I] != BUFFER[X])
                GOTO ERREUR;

                /* LA PREMIERE ZONE DE LA CARTE EST END , ON A
                   DONC UNE CARTE FIN
                */


CARTE = END ;
RETURN ;


ERREUR:


/*
   LA PREMIERE ZONE DE LA CARTE N EST NI FILE NI END IL S AGIT
   DONC D UNE CARTE ERRONEE .

   N.B. : UNE CARTE ERRONEE PEUT AUSSI ETRE DECELEE DANS LE PREMIER CAS
          (FILE) DANS L APPEL DES ROUTINES EJECT ET TRANSF .

*/


CARTE = ERROR ;
RETURN;


}


EJECT()
{


/*
EJECT A POUR BUT DE POSITIONNER L'INDICE I AU PROCHAIN
BYTE DIFFERENT DE SPACE DANS LE BUFFER
*/

INT K ;
WHILE(BUFFER[I] == ' ')
        {
        I++ ;
        IF(I == 80)
```

```
                {
                CARTE = ERROR ;
                RETURN ;
                }
        }

}

TRANSF()
{

/*
TRANSF A POUR BUT DE TRANSFERER DANS LE TABLEAU FILENAME INDICE
PAR J LES CARACTERE DU BUFFER INDICE PAR I JUSQU'A CE QUE
L ON RENCONTRE UN SPACE
*/

INT L ;
L = 0;
WHILE(BUFFER[I] != ' ')
        {
        L++;
        IF(L > 8)
                {
                CARTE = ERROR ;
                RETURN ;
                }
        FILENAME[J] = BUFFER[I] ;
        I++ ;
        IF(I == 80)
                {
                CARTE = ERROR ;
                RETURN ;
                }

        J++ ;
        }
RETURN ;
}
```

BIBLIOGRAPHIE

- Unix Programmer's Manual
  K. Thompson, D.M. Ritchie Mai 1975

- The Unix Time-Sharing System
  D.M. Ritchie

- Unix For Beginners
  B.W. Kernighan

- A Tutorial Introduction To The Unix Text Editor
  B.W. Kernighan

- Programming in C - A Tutorial
  B.W. Kernighan

- C Reference Manual
  D.M. Ritchie

- Setting Up Unix

- The Unix I/O System
  D.M. Ritchie

- A Commentary on The Unix O.S.
  J. Lions 1977

- UNIX O.S. System Source Code Level Six
  J. Lions juin 1977

- Gestion des Terminaux dans UNIX
A. Wupit décembre 1976

- PDP 11/40 Processor Handbook
DEC 1972

- PDP 11 Peripherals Handbook
DEC 1976