5.2 Le port parallèle *Revision* : 1.9

Nous allons étudier dans cette section les outils fournis par le système pour manipuler le port parallèle. L'ensemble du code relatif à ce système se trouve dans le répertoire `drivers/parport`.

Cette section est organisée de la façon suivante

- Nous allons dans un premier temps décrire en 5.2.1 le fonctionnement du port parallèle, ainsi que ses différentes évolutions.
- Nous observerons ensuite dans 5.2.2 le sous-système de gestion des ports parallèle dans le système Linux du point de vue de l'administrateur.
- La sous-section suivante décrira l'architecture logicielle générale de ce sous-système. Nous verrons qu'elle se décompose en plusieurs couches reposant les unes sur les autres.
- Nous étudierons alors dans les sous-sections 5.2.4 et 5.2.5 les deux couches les plus hautes de cette architecture, consacrées à l'implantation des pilotes de périphériques sur port parallèle et à la gestion de ces périphériques.
- Nous observerons alors dans la sous-section 5.2.6 l'ensemble des fonctions disponibles pour les pilotes de périphériques et par lesquelles ces derniers peuvent dialoguer avec les périphériques mais également, si nécessaire, avec les ports parallèle.
- La sous-section suivante décrira ensuite les pilotes de port parallèle, qui constituent la couche la plus basse de l'architecture du sous-système Linux des ports parallèle.
- Nous consacrerons alors une sous-section à l'étde du pilote *parport_pc*, à titre d'exemple d'implantation d'un pilote de bas niveau.
- La couche d'interfaçage entre les pilotes de périphériques et les pilotes de port parallèle sera enfin décrite dans la dernière sous-section.

5.2.1 Le port parallèle

Le port parallèle constitue un dispositif d'entrée/sortie extrèmement répandu sur les systèmes informatiques. C'est un dispositif relativement vieux (il date du début des années 80) qui a subi quelques évolutions. Il en existe donc différentes versions, définies notemment par la norme IEEE 1284.

Le mode ESP , ou mode unidirectionnel ou Centronics, permet à l'ordinateur d'émettre des données vers un périphérique (à l'origine, une imprimante) à une vitesse maximale de 150 ko/s. Il faut noter qu'une utilisation un peut détournée de ce mode de fonctionnement (nommée mode entrelacé, ou "nibble") permet de réaliser des communications bidirectionnelles, grâce à un cablage particulier.

Le mode SPP , apparru au début des années 90, permet une communication bidirectionnelle à un débit maximal de 2 Mo/s.

Le mode ECP permet l'utilisation d'un canal DMA, ce qui permet de soulager le processeur, et offre la possibilité de gérér les périphériques "plug and play".

L'interface physique

La figure 5.1 montre l'interface classique d'un port parallèle. Il s'agit d'une interface DB de 25 broches, même si elle est généralement nommée simplement "interface parallèle".

Dans l'architecture PC, le port parallèle est manipulé par le biais de trois registres d'entrée-sortie. Si nous appelons *base* l'adresse de base des registres liés

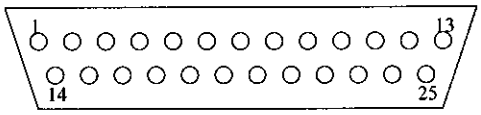

FIG. 5.1 – L'interface du port parallèle

à un port parallèle (traditionnellement, `base` vaut 0x378 pour le premier port parallèle sur un PC), alors :

le registre `base` est le registre de données, par le biais duquel les données réelles circulent ;

le registre `base + 1` est le registre d'état, par le biais duquel une imprimante peut informer l'ordinateur de son état (absence de papier, ...) ;

le registre `base + 2` est le registre de contrôle par le biais duquel l'ordinateur peut dialoguer avec l'imprimante.

La table suivante donne le rôle de chacune des broches de l'interface et de chacun des bits des trois registres décrits ci-dessus (où D représente le registre de données, C celui de contrôle et E celui d'état).

Numéro	Bit	Rôle	Numéro	Bit	Rôle
1	C_0	Strobe	10	E_6	Accusé de réception
2	D_0	Données, bit 0	11	E_7	Occupé
3	D_1	Données, bit 1	12	E_5	Plus de papier
4	D_2	Données, bit 2	13	E_4	Select
5	D_3	Données, bit 3	14	C_1	Alimentation automatique
6	D_4	Données, bit 4	15	E_3	Erreur
7	D_5	Données, bit 5	16	C_2	Initialisation
8	D_6	Données, bit 6	17	C_3	Sélectionné
9	D_7	Données, bit 7	18-25		Masse
	C_4	Interruption activée			

Ajoutons que le mode EPP utilise des registres supplémentaires, afin de permettre des transferts plus rapides tout en conservant une compatibilité totale avec le port parallèle standard. Les registres `base + 3` et `base + 4` permettent l'ecriture et la lecture d'adresses (pour le `base + 3`) et de données (pour le `base + 4`). Les adresses sont en fait des données de paramétrage de la communication entre le périphérique parallèle (scanner, disque dur externe, ...) et la machine. Trois registres supplémentaires peuvent encore être utilisés, mais ne sont pas normalisés.

5.2.2 Gestion par le noyau Linux

Comme de nombreux sous-systèmes du noyau Linux, la gestion des ports parallèle peut être intégrée de façon statique dans le noyau, ou bien de façon dynamique, ou bien pas du tout en fonction des choix réalisés par l'utilisateur lors de la compilation du noyau.

C'est le module `parport` qui constitue le cœur du sous-système auquel viennent se rajouter des drivers de port spécifiques au matériel, `parport_pc`, `parport_atari`, ...

De façon également très classique dans le noyau Linux, des paramètres peuvent être fournis par l'utilisateur lors du démarrage du système.

Si la gestion des ports parallèle est compilée statiquement dans le noyau, les options sont passées au démarrage de la façon suivante sur la ligne de commande du chargeur (LILO, *Grub*, ...) :

```
parport=0x378,7
```

Plusieurs déclarations de ce type peuvent figurer sur la ligne de démarrage, chacune permettant de définir un nouveau port parallèle. La première valeur donne l'adresse du port d'entrée/sortie, et la seconde le numéro d'interrution utilisée par le port.

Si la gestion des ports parallèle est chargée dynamiquement, les paramètres seront placés sur la ligne de commande de chargement du module ou dans le fichier de configuration du démon de chargement automatique.

En ce qui concerne un chargement manuel, on utilisera par exemple les commandes suivantes sur une architecture PC :

```
# insmod parport
# insmod parport_pc io=0x378 irq=7
```

La première commande permet de charger la gestion des ports parallèle, la seconde charge le pilote de bas niveau.

Ici, plusieurs adresses de port peuvent figurer après le paramètres `io=` en les séparant par des virgules ; il en est de même pour les numéros d'interruption.

Je vous invite à consulter la documentation du chargeur dynamique de module et de votre outil de démarrage de Linux pour plus de précisions sur l'automatisation de ce chargement.

5.2.3 Architecture générale

La figure 5.2 montre l'architecture générale du système de gestion des ports parallèle sous Linux. L'ensemble des fichiers implantant cette architecture se trouve dans le répertoire `drivers/parport`. Décrivons brièvement cette architecture de bas en haut.

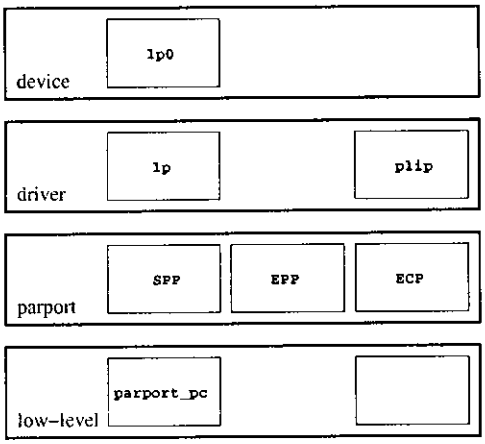

FIG. 5.2 – Architecture du système de gestion des ports parallèle

- Au niveau le plus bas se trouve la couche dite "low-level", qui intègre les pilotes de port parallèle, ce sont ces pilotes qui dialoguent réellement avec les interfaces d'entrée-sortie de type parallèle. C'est par exemple à ce niveau que se situe le pilote *parport_pc* chargé de gérer les ports parallèle que l'on trouve traditionnellement sur les PC. Nous décrirons brièvement cette couche dans la sous-section 5.2.7 et observerons l'exemple de *parport_pc* dans la sous-section 5.2.8.
- Le niveau supérieur, nommé "parport", réalise l'interface entre la couche "low-level" que nous venons de décrire et les pilotes de périphériques connectés

sur un port parallèle. C'est au sein de cette couche que sont définies toutes les fonctions permettant d'utiliser un port parallèle.
- Le troisième niveau ("driver") est le niveau des pilotes de périphériques sur port parallèle. C'est à ce niveau que se situe, par exemple, le pilote `lp` de gestion d'imprimantes sur port parallèle. Nous le décrirons dans la sous-section 5.2.4.
- Le dernier niveau ("device") permet d'identifier chaque périphérique réel en cours d'utilisation via un port parallèle. Nous le décrirons dans la sous-section 5.2.5.

Cette structure peut paraître un peut complexe, surtout pour un élément aussi simple que le port parallèle. Mais elle permet une grande souplesse, en séparant clairement la gestion du port parallèle lui-même, quelle qu'en soit la version et le modèle, de la gestion des périphériques eux-mêmes (scanner, imprimante, ...).

Elle permet de plus un partage à la fois simple et dynamique des ports parallèle entre les différents périphériques.

Le programmeur se lançant dans l'écriture d'un pilote de pépriphérique utilisant le port parallèle n'a donc pas à se préoccuper des détails techniques de la gestion du port parallèle, et peut se focaliser sur la gestion de son périphérique.

5.2.4 Les pilotes de périphériques sur port parallèle

Nous allons nous intéresser ici à l'écriture de pilotes de périphériques utilisant le port parallèle. Nous nous contenterons de décrire ici un minimum des fonctions permettant de manipuler un port parallèle. Une étude plus exhaustive de ces fonctions sera réalisée dans la sous-section suivante.

Le programmeur qui souhaite écrire un pilote de périphériques utilisant le port parallèle doit dans un premier temps enregistrer ce pilote auprès du système ; ceci lui permet d'être informé des différents ports parallèle disponibles sur le système, et ce dynamiquement, c'est-à-dire non seulement lors de son enregistrement, mais également à chaque fois qu'un port parallèle est configuré ou supprimé du système. Nous allons donc commencer par décrire comment réaliser un tel enregistrement.

En toute logique, lorsque le pilote de périphérique n'est plus utile et doit donc être supprimé du système, il doit d'abord être désenregistré, nous verrons comment en 5.2.4.

Lorsqu'un pilote de périphériques est chargé (et enregistré) dans le système, il est amené à piloter des périphériques (c'est la moindre des choses !). Nous verrons en 5.2.5 puis 5.2.5 comment chaque périphérique doit alors être à son tour enregistré puis désenregistré avant et après sa prise en charge.

Ce n'est malheureusement pas tout ! Un même port parallèle peut éventuellement être utilisé par plusieurs périphériques différents. Il est alors nécessaire à chacun d'entre eux de réclamer l'accès au port lorsqu'il en a bseoin. Nous décrirons donc les fonctions permettant cela en 5.2.5.

Enregistrement d'un pilote

Un pilote de périphériques sur port parallèle est identifié par une instance de la structure suivante, déclarée dans le fichier `linux/parport.h` :

```
struct parport_driver {
        const char *name;
        void (*attach) (struct parport *);
        void (*detach) (struct parport *);
        struct parport_driver *next;
};
```

Les champs de cette structure ont la signification suivante

name est, naturellement, le nom du pilote ;

attach est la fonction invoquée lors de la détection d'un nouveau port, c'est à elle de prendre en compte ces nouveaux ports ;

detach est appelée lors de la désactivation d'un port (parce que le pilote de bas niveau de ce port a été supprimé, par exemple) ;

next permet de chaîner les pilotes, il doit être initialisé à NULL.

Un pilote est alors enregistré auprès du noyau grâce à la fonction suivante

```
int parport_register_driver(struct parport_driver * pd);
```

qui renvoie 0 en cas de succès et un code d'erreur sinon.

La fonction `attach()` de la structure ainsi enregistrée est alors invoquée une fois pour chaque port disponible sur le système (et le sera à nouveau par la suite si un nouveau port est activé, lors du chargement d'un module, par exemple) ; le paramètre qui lui est passé est de type `struct parport` dont voici certains champs parmi les plus intéressants :

```
const char *name;
int portnum;
```

représentent, respectivement, le nom symbolique du port et le numéro du port physique correspondant ;

```
unsigned int modes;
```

permet de connaître les capacités du port, c'est une conjonction des macros suivantes

PARPORT_MODE_PCSPP stipule que les registres traditionnels de l'IBM PC sont disponibles ;

PARPORT_MODE_TRISTATE précise que le port peut fonctionner en entrée (périphérique vers ordinateur) ;

PARPORT_MODE_EPP signifie que le matériel peut fonctionner en mode EPP ;

PARPORT_MODE_ECP signifie que le matériel peut fonctionner en mode ECP ;

PARPORT_MODE_COMPAT stipule que le matériel peut fonctionner en mode compatible ;

PARPORT_MODE_DMA précise que le matériel peut utiliser le DMA ;

PARPORT_MODE_SAFEININT assure que les registres peuvent être utilisés pendant le traitement d'une interruption.

Les deux derniers champs de la structure `struct parport` susceptibles de nous intéresser ici sont

```
int irq;
int dma;
```

Qui donnent, respectivement, le numéro d'interruption et le canal DMA utilisés par le port ; le champ `irq` est égal à -1 si aucune interruption n'est utilisée, et le champ `modes` précise si le port peut utiliser un canal DMA.

Voici par exemple comment la fonction `lp_init` du fichier `drivers/char/lp.c` enregistre le pilote d'impression sur port parallèle. La structure décrivant le pilote `lp` est définie de la façon suivante :

```
static struct parport_driver lp_driver = {
    "lp",
    lp_attach,
    lp_detach,
    NULL
};
```

La fonction `lp_init` enregistre alors cette structure par les quelques lignes suivantes :

```
if (parport_register_driver (&lp_driver)) {
    printk (KERN_ERR "lp:␣unable␣to␣register␣with␣parport\n");
    return -EIO;
}
```

Désenregistrement d'un pilote

Un pilote de périphérique parallèle peut alors être désenregistré grâce à la fonction suivante

```
void parport_unregister_driver (struct parport_driver * ppdrv);
```

Voici par exemple comment la fonction `lp_cleanup_module` du fichier `drivers/char/lp.c` désenregistre le pilote `lp` :

```
parport_unregister_driver (&lp_driver);
```

Notons que cette fonction n'est invoquée que lors du déchargement du module de gestion des imprimantes sur port parallèle. Si ce pilote a été compilé statiquement dans le noyau, c'est en effet inutile.

5.2.5 Les clients de port parallèle

Un client de port paralèlle, c'est-à-dire un périphérique spécifique dialoguant par le biais d'un port parallèle est décrit par une instance de la structure suivante :

```
struct pardevice;
```

Nous ne décrirons pas ici en détail cette structure, mais citons les champs suivants :

```
const char *name;
```

Ce champ donne le nom du périphérique.

```
struct parport *port;
```

Ce champ permet de retrouver le port parallèle effectivement utilisé, ou plus précisément le pilote de bas niveau de ce port.

```
int (*preempt)(void *);
void (*wakeup)(void *);
```

Ces deux fonctions définissent le comportement pour ce périphérique afin d'assurer un accès cohérent au port parallèle. Nous allons les décrire très bientôt.

```
void *private;
```

Ce champ permet d'associer à un périphérique des données privées.

Ces différents champs n'ont pas besoin d'être initialisés "à la main" par le pilote de gestion du périphérique. Cette initialisation sera réalisée par la fonction d'enregistrement que nous allons observer maintenant.

Enregistrement d'un client

Lorsqu'un pilote s'est enregistré auprès du système de ports parallèle, il est informé de l'apparition et de la disparition de chaque port via l'appel des fonctions `attach()` et `detach()` qu'il a fournit. C'est donc généralement dans le code de la fonction `attach()` qu'il va s'enregistrer en temps que client du port.

Un client (c'est-à-dire une portion de code qui utilise un port parallèle) s'enregistre alors grâce à la fonction suivante :

```
struct pardevice *parport_register_device(struct parport *port,
                          const char *name,
                          int (*pf)(void *),
                          void (*kf)(void *),
                          void (*irq_func)(int, void *, struct pt_regs *),
                          int flags, void *handle);
```

Les paramètres de cette fonction ont la signification suivante

name est, bien entendu, le nom du client (par exemple lp pour le pilote d'imprimante) ;

pf est la fonction de préemption, cette fonction sera invoquée si un autre client souhaite obtenir le port parallèle alors qu'il est en cours d'utilisation par ce client ;

kf est la fonction de réveil, elle est invoquée lorsqu'un client a libéré le port parallèle et qu'aucun autre client n'en a fait la demande ;

irq_func est la fonction d'interruption, elle sera invoquée lorsque le port parallèle déclenchera une interruption si ce client a obtenu l'accès au port ;

flags peut prendre la valeur *PARPORT_DEV_EXCL* qui stipule qu'un accès exclusif est souhaité ;

handle permet de faire passer des données spécifiques aux fonctions de traitement (stockées dans le champ *private*).

Les fonctions peuvent être *NULL* si on ne souhaite pas de traitement particulier.

La fonction *parport_register_device()* renvoie un pointeur sur une structure définissant le client ainsi créé, ou *NULL* en cas d'impossibilité à le créer.

Désenregistrement d'un client

Un client doit être désenregistré par un appel à la fonction

```
void parport_unregister_device(struct pardevice *dev);
```

Attention, cette fonction doit être utilisée même si le pilote concerné à été lui-même désinscrit par un appel à *parport_unregister_driver()*.

Obtention de l'accès au port

Puisqu'un même port peut être partagé par plusieurs pilotes de périphérique parallèle, il est nécessaire à chacun d'entre eux d'obtenir un accès exclusif au port afin de pouvoir y émettre ou lire des données ; le programmeur dispose pour cela des fonctions suivantes :

```
int parport_claim(struct pardevice *dev);
int parport_claim_or_block(struct pardevice *dev);
```

Ces deux fonctions tentent d'obtenir l'accès à un port pour un client passé en paramètre, elles renvoient 0 en cas de succès et une valeur négative en cas d'erreur (seule l'erreur *EAGAIN* est possible, elle signifie que le port est momentanément indisponible, et ce uniquement pour *parport_claim()*). La première de ces fonctions n'attend pas que le port soit disponible pour se terminer, contrairement à la seconde qui renvoie alors 1 si elle a pu obtenir l'accès au port après une phase d'attente. C'est la fonction *sleep_on()* qui est utilisée pour réaliser l'attente.

Il ne faut pas invoquer une de ces fonctions sur un port dont l'accès a déjà été obtenu.

Un port peut ensuite être libéré par un appel à

```
void parport_release(struct pardevice *dev);
```

Il ne faut pas libérer un port dont l'accès n'a pas été préalablement obtenu.

Les deux fonctions suivantes

```
int parport_yield(struct pardevice *dev);
int parport_yield_blocking(struct pardevice *dev);
```

permettent de « prêter » un port, c'est-à-dire de le libérer un bref instant afin de le rendre disponible pour d'autres client puis de le ré-acquérir.

La seconde est éventuellement bloquante. En fait, ces fonctions sont équivalentes à un appel à `parport_release()` suivi d'un appel à `parport_claim()` pour la première et à `parport_claim_or_block()` pour la seconde. Comme ces dernières, elles renvoient donc 0 en cas de succès et une valeur négative en cas d'erreur.

Préemption

Observons maintenant les deux fonctions `preempt()` et `wakeup()` passées en paramètre à la fonction d'enregistrement d'un client.

La fonction de préemption du client `c1` est invoquée par le système lorsqu'il a besoin (pour un client `c2`) d'accéder à un port parallèle actuellement utilisé par ce client. En fait, c'est donc l'utilisation de la fonction `parport_claim()` par `c2` qui va provoquer l'invocation de la fonction `preempt()` associé au client `c1`.

L'interface de cette fonction est la suivante :

```
int preempt(void * private);
```

Lorsqu'elle est invoquée, ce sont les données privées passées en dernier paramètre à `parport_register_device()` (et stockées dans le champ `private` de la structure `pardevice`) qui lui sont fournies en paramètre. Cette fonction doit renvoyer 0 si le client accepte d'abandonner l'accès au port, une valeur non nulle sinon.

Il est inutile d'invoquer la fonction `parport_release()` , cela sera réalisé automatiquement par le système si (et seulement si) la fonction `preempt()` accèpte de libérer le port (en renvoyant une valeur nulle).

Disponibilité du port

La fonction de "réveil" passée en paramètre lors de l'enregistrement d'un client permet à ce dernier d'être informé chaque fois que le port parallèle est libéré par un autre client (qui a donc utilisé la fonction `parport_release()`.

L'interface de cette fonction est la suivante :

```
void wakeup(void * private);
```

Lorsqu'elle est invoquée, ce sont les données privées passées en dernier paramètre à `parport_register_device()` (et stockées dans le champ `private` de la structure `pardevice`) qui sont fournies en paramètre.

Cette fonction ne doit réaliser aucune opération bloquante. En fait elle ne doit rien faire si le client ne souhaite pas accéder au port, et elle peut (et doit) utiliser la fonction `parport_claim()` si le client souhaite accéder au port. Le système assure que la fonction `parport_claim()` ne sera pas blocante ici.

5.2.6 Utilisation d'un port parallèle

Maintenant que nous savons comment un pilote de périphériques s'intègre dans le sous-sytsème Linux de gestion des ports parallèle, nous allons observer l'ensemble des fonctions dont il peut disposer pour manipuler le port parallèle et dialoguer avec les périphériques qu'il gère.

Ces fonctions se décomposent en plusieurs ensembles

- Les fonctions de haut niveau sont les plus générales. Les fonctions d'entrée/sortie de ce niveau sont particulièrement confortables puisqu'elles permettent de lire ou écrire tout un bloc de données en un seul appel de fonction. Elles ne permettent cependant pas de prendre en compte les spécificités du port parallèle utilisé. Ce sont ces fonctions que l'on préfèrera utiliser pour toute utilisation "banale" du port parallèle.
- Les fonctions du mode standard sont des fonctions plus élémentaires de manipulation d'un port parallèle fonctionnant en mode SPP. Elles permettent donc une manipulation plus fine du port parallèle, mais moins confortable, puisqu'elles ne permettent, en particulier, que de lire ou écrire un octet à chaque appel.
- Les fonctions du mode EPP permettent de la même façon de manipuler les spécificités de ce mode.
- Il en est de même pour les fonctions du mode ECP.
- Quelques fonctions particulières permettent de réaliser quelques manipulations complémentaires.

Seules les fonctions du premier ensemble sont de "véritables fonctions" définies globalement dans le noyau et utilisables par une invocation tout à fait classique. Les autres foncions sont en fait stockées, pour chaque port, dans le champs *ops* de la structure caractérisant le port.

Nous n'allons pourtant pas décrire ces fonctions en suivant cette classification "technique", mais plutôt en suivant une classification basée sur les services fournis par les fonctions. Au sein de chaque groupe de fonctions liées à un service donné, nous ferons alors la différences entre les différents niveaux que nous venons de citer.

Nous allons donc commencer par les fonctions permettant de réaliser des entrées/sorties. Nous décrirons ensuite la configuration du port parallèle.

Nous observerons alors comment un pilote de périphériques peut, si nécessaire, manipuler des registres d'un port parallèle.

Nos étudierons enfin comment gérer les interruptions que peut générer un port parallèle.

Lecture/écriture

Lorsqu'un client a obtenu l'accès à un port, il peut y envoyer des données, éventuellement en lire, et consulter l'état du port.

Sur un port parallèle, les entrées/sorties peuvent être réalisées grâce à des fonctions de haut niveau, relativement confortables, mais également par des fonctions liées au mode standard.

Nous allons maintenant passer en revue ces ensembles de fonctions.

Les fonctions de haut niveau Lorsqu'un client a obtenu l'accès à un port, il peut y envoyer des données, éventuellement en lire.

Les deux premières fonctions permettant une telle communication sont les suivantes :

```
ssize_t parport_write(struct parport * port, const void *buf, size_t len);
ssize_t parport_read(struct parport * port, void *buf, size_t len);
```

Elles permettent, respectivement, d'écrire et de lire un certain nombre d'octets vers ou depuis un port. Leurs paramètres sont sans grande surprise :

`port` est le port sur lequel les données doivent être écrites ou lues ;

`buf` est un pointeur vers la zone mémoire contenant les données ;

`len` est le nombre maximal d'octets à lire ou écrire ;

la valeur de retour est le nombre d'octets transférés ou un code d'erreur (négatif).

Ces fonctions ne sont en fait que des aiguilleurs vers les différentes fonctions d'entrée/sortie applicables selon le mode dans lequel le port a été configuré.

Les fonctions du mode standard Sur un port parallèle *port* en mode standard, il est possible de lire ou écrire un octet grâce, respectivement, aux fonctions suivantes :

```
unsigned char port->ops->read_data(struct parport * port);
void port->ops->write_data(struct parport * port, unsigned char o);
```

L'interface de ces fonctions est suffisament explicite ; précisons simplement que la fonction de lecture n'est, bien sûr, utilisable que si le port est capable de fonctionner dans les deux sens (ce qui est déterminé par la présence du drapeau *PARPORT_MODE_TRISTATE* dans le champs *modes* du port.

Les fonctions du mode EPP Pour profiter explicitement du mode EPP sur un port *port*, les fonctions suivantes sont disponibles :

```
size_t port->ops->epp_read_data(struct parport *port, void *buf, size_t len,
                                int flags);
size_t port->ops->epp_write_data(struct parport *port, const void *buf,
                                 size_t len, int flags);
size_t port->ops->epp_read_addr(struct parport *port, void *buf, size_t len,
                                int flags);
size_t port->ops->epp_write_addr(struct parport *port, const void *buf,
                                 size_t len, int flags);
```

Pour chacune de ces fonctions, sans surprise :

port est le port par lequel on souhaite transférer les données ou les adresses ;

buf est un buffer dans lequel sont les données/adresses ;

len est le nombre d'octets que l'on souhaite transférer ;

flags peut prendre la valeur nulle ou *PARPOT_EPP_FAST* qui utilise un transfert rapide (sur 16 ou 32 bits) lorsque le matériel le permet ;

la valeur de retour est le nombre d'octets effectivement transférés.

Les fonctions du mode CPP Pour profiter explicitement du mode ECP sur un port *port*, les fonctions suivantes sont disponibles :

```
size_t port->ops->ecp_read_data(struct parport *port, void *buf, size_t len,
                                int flags);
size_t port->ops->ecp_write_data(struct parport *port, const void *buf,
                                 size_t len, int flags);
size_t port->ops->ecp_write_addr(struct parport *port, const void *buf,
                                 size_t len, int flags);
```

Les paramètres ont le même rôle que dans le mode EPP, à l'exception du dernier paramètre *flags*, qui ne sert à rien et n'apparait donc que dans un soucis d'uniformisation.

Les fonctions annexes La fonction suivante permet de lire un bloc de données en mode entrelacé ("nibble") :

```
size_t port->ops->nibble_read_data(struct parport * port,
                                   void * buf, size_t len,
                                   int flags);
```

Le paramètre *flags* ne sert ici à rien ...

Il est également possible de lire un bloc de données en mode bidirectionnel grâce à la fonction suivante

```
size_t port->ops->byte_read_data(struct parport * port,
                                 void * buf, size_t len,
                                 int flags);
```

Le paramètre `flags` ne sert ici à rien ...

Enfin, la fonction suivanet permet d'écrire un bloc de données en mode standard

```
size_t port->ops->compat_write_data(struct parport * port,
                                    const void * buf, size_t len,
                                    int flags);
```

Une fois de plus, le paramètre `flags` ne sert à rien !

Les fonctions de contrôle

Certaines fonctions permettent de manipuler un port parallèle. De la même façon que pour les fonctions d'entrée/sortie, ces fonctions se déclinent selon le mode de fonctionnement du port parallèle.

Mode de communication Si le mode IEEE 1284 a été activé dans le noyau lors de sa compilation, il est possible de négocier avec un port le mode dans lequel il doit être utilisé grâce à la fonction

```
int parport_negotiate (struct parport *port, int mode);
```

où

`port` est, bien sûr, le port à configurer ;

`mode` est le mode dans lequel on souhaite communiquer avec ce port, la valeur de ce paramètre doit être choisie parmi les macros suivantes

- ***`IEEE1284_MODE_NIBBLE`*** pour le mode par demi octet ;
- ***`IEEE1284_MODE_BYTE`*** pour le mode bidirectionnel ;
- ***`IEEE1284_MODE_COMPAT`*** pour le mode compatibilité, ou standard ;
- ***`IEEE1284_MODE_BECP`*** ce mode n'est pas encore géré ;
- ***`IEEE1284_MODE_ECP`*** pour le mode EPP ;
- ***`IEEE1284_MODE_ECPRLE`*** pour le mode ECPRLE ;
- ***`IEEE1284_MODE_ECPSWE`*** pour le mode ECP émulé par logiciel ;
- ***`IEEE1284_MODE_EPP`*** pour le mode EPP ;
- ***`IEEE1284_MODE_EPPSL`*** pour le mode EPPSL
- ***`IEEE1284_MODE_EPPSWE`*** pour le mode EPP émulé par logiciel.

La valeur de retour de cette fonction est -1 en cas d'erreur, 0 si la négociation a été réussie et qu'un périphérique est présent, et 1 si un périphérique IEEE 1284 est présent mais le mode souhaité n'est pas disponible.

Manipulation des registres

En mode standard, il est possible de consulter l'état des registres d'état et de contrôle d'un port parallèle `port` grâce aux fonctions suivantes :

```
unsigned char port->ops->read_status(struct parport * port);
unsigned char port->ops->read_control(struct parport * port);
```

La fonction `read_control` permet d'obtenir la dernière valeur écrite, elle ne procède à aucune manipulation du port.

La fonction `read_status` permet de consulter l'état du port, elle renvoie une valeur qui est une conjonction des valeurs suivantes (définies dans `linux/parport.h`)

PARPORT_STATUS_ERROR stipule que le périphérique a positionné le bit d'erreur.

PARPORT_STATUS_SELECT stipule que le périphérique a positionné le bit "on line".

PARPORT_STATUS_PAPEROUT stipule que le périphérique a positionné le bit "plus de papier".

PARPORT_STATUS_ACK stipule que le périphérique a positionné le bit d'accusé de réception.

PARPORT_STATUS_BUSY stipule que le périphérique a positionné le bit "occupé".

La fonctions suivante permet quand à elle de modifier le registre de contrôle du port parallèle en mode standard :

```
void port->ops->write_control(struct parport * port, unsigned c);
```

Le paramètre `c` doit être une conjonction des masques suivants :

PARPORT_CONTROL_STROBE pour positionner le bit " strobe".

PARPORT_CONTROL_AUTOFD pour positionner le bit "autofd".

PARPORT_CONTROL_INIT pour positionner le bit "init".

PARPORT_CONTROL_SELECT pour positionner le bit "select".

Notons que la fonction suivante permet de ne modifier que certains bits du registre de contrôle

```
void port->ops->frob_control(struct parport * port,
                             unsigned char mask,
                             unsigned char val);
```

Seuls les bits définis dans le `mask` seront touchés, ils prendront alors la valeur fournie dans `val`.

Gestion des interruptions

Lorsqu'une interruption est déclenchée par un port parallèle, c'est donc la fonction passé en paramètre à `parport_register_device()` lors de l'enregistrement d'un client qui est invoquée. Il s'agira bien sûr en l'occurence du client qui utilise actuellement le port (celui qui en a obtenu l'accès par `parport_claim()`)..

Le prototype d'une telle fonction est le suivant

```
void irq_func(int irq, void * private, struct pt_regs * foo);
```

Le premier paramètre de cette fonction est le numéro d'interruption et le deuxième est le pointeur vers les données privées passé en paramètre lors de l'enregistrement du client. Le dernier paramètre n'est pas utile ici.

Il est possible de demander au port parallèle `port` de ne pas générer d'interruption grâce à la fonction suivante

```
void port->ops->disable_irq(struct parport * port);
```

La fonction suivante permet alors d'autoriser à nouveau le port paralèle à générer des interruptions :

```
void port->ops->enable_irq(struct parport * port);
```

Notons que ces fonctions agissent au niveau du port parallèle, pas au niveau du système. Cela signifie en particulier qu'aucune interruption n'est masquée ou démasquée.

5.2.7 Les pilotes de port parallèle

Nous allons maintenant étudier dans cette sous-section la structure des pilotes de bas niveau, c'est-à-dire ceux permettant de gérer les ports parallèle physiques.

Description d'un pilote de bas niveau

Un pilote de port parallèle (un pilote de la couche de bas niveau) est identifé par une instance de la structure suivante, définie dans le fichier `linux/parport.h` :

```
struct parport;
```

Nous ne rentrerons pas dans le détail de cette structure qui contient diverses informations permettant la gestion des ports parallèle.

L'un des champs intéressants de cette structure est le suivant

```
struct parport_operations *ops;
```

C'est lui qui permet de définir l'ensemble des fonctions permettant de manipuler effectivement le port parallèle.

Enregistrement d'un port parallèle

Chaque fois qu'un pilote de port parallèle découvre un nouveau port parallèle (par exemple lors de sa phase d'initialisation), il doit invoquer la fonction suivante :

```
struct parport *parport_register_port(unsigned long base, int irq, int dma,
                                      struct parport_operations *ops)
```

C'est cette fonction qui est chargée d'allouer et d'initialiser la structure de type `struct parport` décrivant le port.

Initialisation du système de gestion des ports parallèle

Cette initialisation est réalisée par la fonction `parport_setup()` du fichier `drivers/parport/init.c`.

Si, par exemple, le système intègre le pilote `parport_pc`, implanté dans le fichier `drivers/parport/parport_pc.c`, alors la fonction `parport_pc_init()` de ce dernier est invoquée.

5.2.8 Le pilote *parport_pc*

Le pilote `parport_pc` est chargé de la gestion des ports parallèle que l'on trouve clasiquement sur les machines d'architecture PC. Il est décrit et codé dans les fichiers `linux/parport_pc.h` et `drivers/parport/parport_pc.c`.

Initialisation

Les opérations de manipulation d'un port parallèle de type PC sont identifées dans la structure `parport_pc_ops` définie et initialisée dans le fichier `drivers/parport/parport_pc.c`.

La fonction d'initialisation Comme nous l'avons vu précédemment, c'est la fonction `parport_pc_init()` qui est chargée de l'initialisation du système de ports de type PC.

Observons donc le code de cette fonction. Elle commence par enregistrer le système `parport_pc` auprès du système de gestion du "plug and play" :

```
int __init parport_pc_init (int *io, int *io_hi, int *irq, int *dma)
{
        int count = 0, i = 0;
        /* try to activate any PnP parports first */
        pnp_register_driver(&parport_pc_pnp_driver);
```

Ensuite, si l'utilisateur a spécifier les ports d'entrée-sortie à initialiser, alors la fonction `parport_pc_probe_port()` est invoquée pour chacun de ces ports :

```
        if (io && *io) {
                /* Only probe the ports we were given. */
                user_specified = 1;
                do {
                        if ((*io_hi) == PARPORT_IOHI_AUTO)
                                *io_hi = 0x400 + *io;
                        if (parport_pc_probe_port(*(io++), *(io_hi++),
                                                  *(irq++), *(dma++), NULL))
                                count++;
                } while (*io && (++i < PARPORT_PC_MAX_PORTS));
```

Si l'utilisateur n'a rien spécifié, alors la fonction `parport_pc_find_ports()` est utilisée pour découvrir les ports existants sur le système :

```
        } else {
                count += parport_pc_find_ports (irq[0], dma[0]);
        }
```

Enfin, le nombre de ports configurés est renvoyé à l'appelant :

```
        return count;
}
```

La fonction `parport_pc_probe_port()` C'est donc la fonction `parport_pc_probe_port()` qui a la charge de configurer un port parallèle spécifique de type PC. Nous n'allons pas décrire entièrement le code de cette fonction dont une majeure partie est liée au matériel et ne présente donc qu'un intéret limité.

Notons cependant que dans ce code figure les lignes suivantes :

```
        if (!(p = parport_register_port(base, PARPORT_IRQ_NONE,
                                        PARPORT_DMA_NONE, ops)))
                goto errout;
```

C'est donc à cet instant que le port est identifié auprès du système et que les opérations de manipulation lui sont associées (la variable `ops` est une copie de `parport_pc_ops`).

```
        parport_proc_register(p);
```

La dernière action de `parport_pc_probe_port()` à laquelle nous nous intéresserons est la suivante, située vers la fin du code :

```
        parport_announce_port (p);
```

C'est ainsi que les pilotes de périphériques parallèle (qui se sont enregistré par le biais de la fonction `parport_register_driver()`) seront prévenus de la présence de ce port parallèle.

5.2.9 La couche d'interfaçage

L'ajout de nouveaux ports

La fonction `parport_register_port()` Cette fonction doit être invoquée par chaque pilote de port parallèle pour chaque port parallèle identifié. Nous ne la décrirons pas car son code se résume à quelques initialisations purement administratives, comme l'attribution d'un nom au port.

La fonction `parport_announce_port()` C'est cette fonction qui est invoquée lorsqu'un nouveau port parallèle est découvert. Elle invoque la fonction `attach_driver_chain()` qui se charge d'invoquer la fonction d'attachement `attach` déclarée par chaque pilote de périphérique parallèle.