TELECOM Paris

IP PARIS

Interface avec l'espace utilisateur

M2 SETI B4 / MS SE SE758

Guillaume Duc
guillaume.duc@telecom-paris.fr
2021–2022

Institut Mines-Télécom

Plan

Classification des périphériques

- Vus depuis l'espace utilisateur
  - Périphériques caractères (*char device*) : unité de base de transfert = un caractère (un octet), interaction à l'aide d'appels systèmes standards (`open`, `read`, `write`, `ioctl`...) sur un fichier spécial (classiquement dans `/dev`)
  - Périphériques blocs (*block device*) : unité de base d'un transfert = bloc de données, accessibles également par l'intermédiaire d'un fichier spécial
  - Les périphériques réseaux (*network device*) : les applications peuvent les manipuler par l'intermédiaire de la pile réseau (`socket`)
- La majorité des périphériques étant vus comme des périphériques caractères, ce sont eux que nous étudierons par la suite

Fichiers spéciaux

- L'interaction entre une application et un périphérique caractère ou bloc se passe par l'intermédiaire d'un fichier spécial identifié par
  - Un mode (caractère ou bloc)
  - Un numéro majeur (*major*)
  - Un numéro mineur (*minor*)
- La liste des numéros majeurs/mineurs est disponible dans

  `Documentation/admin-guide/devices.txt`
- Ces fichiers sont traditionnellement placés dans le répertoire `/dev` (mais ce n'est pas obligatoire)
- Ils sont crées soit par l'appel de la commande `mknod` (manuellement, par un script, ou par un démon tel `udev`), soit automatiquement par le noya (`devtmpfs`)

File operations

- Lorsqu'un pilote réagit à la détection d'un périphérique supporté, il annonce au noyau qu'il supporte un nouveau triplet (mode, majeur, mineur) et déclare un ensemble de fonctions de rappel (*callback*)
- Lorsqu'une application réalise un appel système (exemple : `read`) sur un fichier spécial, le noyau va appeler une des fonctions de rappel du pilote associé au périphérique concerné en lui passant les arguments transmis par l'application
- La fonction de rappel appelée dans le pilote aura la charge de répondre à la demande de l'application

struct file_operations

<include/linux/fs.h>

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
  ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
  int (*iterate) (struct file *, struct dir_context *);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*mremap)(struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *, fl_owner_t id);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, loff_t, loff_t, int datasync);
  int (*aio_fsync) (struct kiocb *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
  unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
  int (*check_flags)(int);
  int (*flock) (struct file *, int, struct file_lock *);
  ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_
  ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *
  int (*setlease)(struct file *, long, struct file_lock **, void **);
  long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len
  void (*show_fdinfo)(struct seq_file *m, struct file *f); };
```

`struct file_operations`

- Cette structure regroupe les fonctions de rappel fournies par le pilote de périphérique
- Cette structure est passée par le pilote au noyau lors de la détection d'un nouveau périphérique
- Certains champs peuvent être mis à la valeur `NULL` car le noyau fournit un comportement par défaut pour la plupart de ces fonctions

Plan

Code d'erreurs

- Les appels systèmes (par exemple ceux utilisés pour effectuer des opérations sur des fichiers) peuvent échouer (problème de permissions, fichier inexistant, etc.)
- Cet échec est indiqué à l'espace utilisateur par un entier négatif représentant un code d'erreur
- Au niveau de la bibliothèque standard, ce code d'erreur est ensuite stocké dans la variable `errno` et la fonction appelée pour faire l'appel système renvera -1
- Ces codes d'erreur sont définis au niveau du noyau dans `include/linux/errno.h`

Code d'erreurs (extrait)

Attention, entiers positifs !

```
#define EPERM            1      /* Operation not permitted */
#define ENOENT           2      /* No such file or directory */
#define ESRCH            3      /* No such process */
#define EINTR            4      /* Interrupted system call */
#define EIO              5      /* I/O error */
#define ENXIO            6      /* No such device or address */
#define E2BIG            7      /* Argument list too long */
#define ENOEXEC          8      /* Exec format error */
#define EBADF            9      /* Bad file number */
#define ECHILD          10      /* No child processes */
#define EAGAIN          11      /* Try again */
#define ENOMEM          12      /* Out of memory */
#define EACCES          13      /* Permission denied */
#define EFAULT          14      /* Bad address */
#define ENOTBLK         15      /* Block device required */
#define EBUSY           16      /* Device or resource busy */
#define EEXIST          17      /* File exists */
#define EXDEV           18      /* Cross-device link */
#define ENODEV          19      /* No such device */
#define ENOTDIR         20      /* Not a directory */
#define EISDIR          21      /* Is a directory */
#define EINVAL          22      /* Invalid argument */
#define ENFILE          23      /* File table overflow */
#define EMFILE          24      /* Too many open files */
/* ... */
```

open

```
int foo_open(struct inode *inode, struct file *file);
```

- Appelée *par le noyau* lorsqu'une application fait un appel système `open` sur le fichier spécial représentant un des périphérique géré par le pilote
- Arguments
  - `struct inode *inode` : représente de façon unique le fichier spécial sur lequel l'appel système a été fait
  - `struct file *file` : représente le fichier ouvert (plusieurs instances de cette structure peuvent exister pour un unique `inode` si ce dernier est ouvert par plusieurs applications ou par la même application plusieurs fois). Cette structure est passée ultérieurement à chaque fonction (exemple : `read`...). Elle contient un certain nombre d'informations importantes (voir slide suivant)

open

`struct file` (extrait)

- `fmode_t f_mode` : Fichier ouvert en lecture (`FMODE_READ`) ou en écriture (`FMODE_WRITE`)
- `unsigned int f_flags` : Divers drapeaux dont le plus important est `O_NONBLOCK` qui est positionné quand l'application demande à ce que les opérations sur le fichier soient non bloquantes
- `const struct file_operations *f_op` : Pointeur vers la structure contenant les opérations pour ce fichier
- `void *private_data` : Pointeur que votre pilote peut utiliser pour y stocker des informations. Elles seront alors accessibles via ce champ à chaque fois qu'une opération est appelée pour ce fichier

open

```
int foo_open(struct inode *inode, struct file *file);
```

- Valeur renvoyée : `0` pour indiquer que l'ouverture s'est bien passée, une valeur négative pour indiquer une erreur (exemple `-EPERM`)
- Cette fonction peut ne pas être définie dans la structure `struct file_operations`. Dans ce cas le noyau considère que l'ouverture du fichier par une application réussira toujours (sauf problème de permission)

release

```
int foo_release(struct inode *, struct file *);
```

- Appelée en cas de fermeture du fichier
- Mêmes arguments que open
- Peut ne pas être définie, auquel cas la fermeture réussira toujours

read

```
ssize_t foo_read(struct file *file, char __user *buf,
                 size_t count, loff_t *f_pos);
```

- Appelée lors d'une lecture (appel système `read`)
- Si elle n'est pas définie, la lecture échouera avec l'erreur `-EINVAL`
- Arguments
  - `struct file *file` : La structure représentant le fichier
  - `char __user *buf` : Le tampon, dans l'espace utilisateur, à remplir avec les données en provenance du périphérique
  - `size_t count` : Le nombre *maximum* d'octets à lire
  - `loff_t *f_pos` : Pointeur vers la la position courante dans le fichier, à mettre à jour

read

```
ssize_t foo_read(struct file *file, char __user *buf,
                 size_t count, loff_t *f_pos);
```

- Lit au plus `count` octets depuis le périphérique et les écrit dans le tampon dans l'espace utilisateur `buf`
- Renvoie le nombre réel d'octets écrits dans `buf` (qui peut être inférieur à `count`), `0` pour indiquer la fin du fichier ou un nombre négatif pour indiquer une erreur

read

```
ssize_t foo_read(struct file *file, char __user *buf,
                 size_t count, loff_t *f_pos);
```

- Dans le cas normal, la fonction doit bloquer jusqu'à ce qu'au moins un octet soit disponible et dès que c'est le cas, renvoyer tout de suite les octets disponibles (même s'il y en a moins que `count`)
- Si le drapeau `O_NONBLOCK` est positionné dans `file->f_flags`, si aucune donnée n'est disponible, la fonction doit renvoyer `-EAGAIN` tout de suite (sans bloquer) ou, si des données sont disponibles, les renvoyer tout de suite (même s'il y en a moins que `count`)

read

```
ssize_t foo_read(struct file *file, char __user *buf,
                 size_t count, loff_t *f_pos);
```

- L'utilisation du tampon en espace utilisateur `buf` sera expliquée par la suite
- Dans le cas des périphériques, la notion de position courante dans le fichier `f_pos` n'a souvent pas de sens
  - Dans un fichier *classique*, l'appel `read` (ainsi que `write`, `seek`, etc.) modifie cette position courante, d'où cet argument

write

```
ssize_t foo_write(struct file *file,
                  const char __user *buf,
                  size_t count, loff_t *f_pos);
```

- Lit *au plus* `count` octets depuis le tampon en espace utilisateur `buf`, les envoie au périphérique
- Renvoie le nombre réel d'octets transmis (qui peut être inférieur à `count`), ou un nombre négatif pour indiquer une erreur

write

```
ssize_t foo_write(struct file *file,
                  const char __user *buf,
                  size_t count, loff_t *f_pos);
```

- Dans le cas normal, la fonction doit bloquer jusqu'à ce qu'au moins un octet ait été écrit (transmis au périphérique) et dès que c'est le cas, renvoyer tout de suite le nombre d'octets transmis (même s'il y en a moins que `count`)
- Si le drapeau `O_NONBLOCK` est positionné dans `file->f_flags`, s'il n'est pas possible de transmettre tout de suite des données au périphérique, la fonction doit renvoyer `-EAGAIN` immédiatement (sans bloquer)

ioctl

```
long foo_unlocked_ioctl(struct file *file,
                        unsigned int cmd,
                        unsigned long arg);
```

- La métaphore lecture/écriture est appropriée pour transférer des données entre un périphérique et une application mais elle ne permet pas aisément à une application de modifier ou de récupérer la configuration d'un périphérique (modifier la vitesse d'un port série, etc.)
- Cette fonction, appelée suite à l'appel système `ioctl`, permet justement d'échanger des informations qui ne sont pas des données "utiles" entre le pilote et l'application

ioctl

```
long foo_unlocked_ioctl(struct file *file, unsigned int cmd, unsigned long arg);
```

- L'application fournit à l'appel système `ioctl` un entier qui identifie l'opération à effectuer (argument `cmd`) ainsi qu'éventuellement un autre argument, entier ou pointeur (argument `arg`)
- La signification de `cmd` et de `arg` est spécifique à chaque pilote
- `Documentation/driver-api/ioctl.rst` contient des informations supplémentaires importantes sur `ioctl`, notamment le choix des valeurs pour `cmd` ainsi que sur `compat_ioctl`

Plan

Transferts applications/noyau

- Le tampon dont l'adresse est fournie aux fonctions `read` et `write` est situé dans la partie espace utilisateur de l'espace d'adressage de l'application
- Le code tournant dans l'espace noyau (ce qui concerne donc le pilote de périphérique) n'a pas le droit d'accéder directement à ce tampon (en déréférençant directement le pointeur ou en utilisant des fonction comme `memcpy`)
  - Impossible sur certaines architectures
  - Problème de sécurité potentiel (la mémoire référencée par le pointeur peut être volontairement ou involontairement invalide, ce qui peut mener à une faute de pagination dans le noyau, ce qui est interdit, ou pire)

__user

- La macro `__user` décore les pointeurs contenant des adresses qui pointent vers l'espace utilisateur
- En fonction de la configuration du noyau, `gcc` s'en sert et vérifie qu'aucun déréférencement de ces pointeurs n'est réalisé

Fonctions de transfert

<asm/uaccess.h> et <linux/uaccess.h>

- `get_user(x, p)` : la variable `x` dans l'espace noyau reçoit la valeur pointée par le pointeur `p` en provenance de l'espace utilisateur
- `put_user(x, p)` : le contenu de la variable `x` dans l'espace noyau est copié vers l'adresse pointée par le pointeur `p` dans l'espace utilisateur
- `unsigned long copy_to_user(void __user *to, const void *from, unsigned long n)`
- `unsigned long copy_from_user(void *to, const void __user *from, unsigned long n)`
  - Transfèrent `n` octets depuis/vers l'espace utilisateur

Plan

Frameworks

- Des *frameworks* existent afin d'unifier et de rationaliser l'interface entre les applications et les pilotes pour des périphériques similaires
- En fonction du framework, l'abstraction est plus ou moins importante
- Exemples
  - Framework *input* : regroupe beaucoup de périphériques d'entrée (clavier, souris, joystick...). Fournit une abstraction importante à base d'événements
  - Framework *misc* : très léger, simplifie juste la gestion des fichiers spéciaux

Framework *misc*

- Dans la suite, nous allons décrire un peu plus le framework *misc*
- Framework très léger
- Simplifie notamment la gestion des fichiers spéciaux (il est possible de s'enregistrer directement auprès du noyau en tant que périphérique caractère mais c'est plus compliqué)
- C'est le premier framework que vous allez utiliser pour interface entre votre pilote d'accéléromètre et l'espace utilisateur (on aurait pu utiliser également le framework *input*)

Framework misc

```
struct miscdevice {
  int minor;
  const char *name;
  const struct file_operations *fops;
  struct list_head list;
  struct device *parent;
  struct device *this_device;
  const struct attribute_group **groups;
  const char *nodename;
  umode_t mode;
};
```

Framework misc

`struct miscdevice`

- Décrit un pilote utilisant le framework *misc*
- Principaux champs
  - `minor` : le numéro *minor* désiré pour le périphérique (le numéro *major* sera automatiquement celui des périphériques *misc*, c'est-à-dire 130) ou `MISC_DYNAMIC_MINOR` pour en obtenir un dynamiquement
  - `name` : le nom du périphérique, utilisé pour créer plus ou moins automatiquement le fichier spécial correspondant dans `/dev`
  - `fops` : pointeur vers la structure `struct file_operations` qui déclare les fonctions de rappel utilisées pour répondre aux opérations de lecture, écriture, etc.
  - `parent` : le périphérique associé

Framework misc
Enregistrement

- Un pilote s'enregistre auprès du framework *misc* grâce à la fonction `int misc_register(struct miscdevice *misc)`
  - L'appel à cette fonction est traditionnellement effectué dans la fonction `probe` du pilote
- Lorsque le périphérique n'est plus présent, il faut penser à appeler la fonction `void misc_deregister(struct miscdevice *misc)`
  - L'appel à cette fonction est traditionnellement effectué dans la fonction `remove`

Plan

Problématique

- Prenons le cas d'un pilote de périphérique I^2C utilisant le framework *misc*
- Chaque périphérique physique géré par le pilote est représenté par la structure `struct i2c_client`
  - Les fonctions utilisées pour effectuer un transfert I^2C (`i2c_master_send` et `i2c_master_recv` par exemple), c'est-à-dire communiquer avec le périphérique pour transférer des données par exemple, ont besoin de cette structure `struct i2c_client`

Problématique (suite)

- Le pilote va également avoir besoin de stocker des informations pour chaque périphérique physique présent qu'il gère (par exemple : état courant, `struct miscdevice` associée, etc.)
  - Souvent cette structure s'appelle `foo_device` où `foo` est le nom du périphérique (ex. `adxl345_device`)
  - Elle est instanciée dynamiquement (via un appel à `kmalloc`), par la fonction `probe`, pour chaque nouveau périphérique supporté

Problématique (suite)

- Enfin, lorsqu'une opération est effectuée sur le fichier spécial représentant un des périphérique physique supporté, le noyau va appeler la fonction de rappel correspondante et passer la structure `struct file` représentant le fichier
- Le problème suivant se pose
  - Dans la fonction `foo_read(struct file *file, char __user *buf, size_t count, loff_t *f_pos)`, comment retrouver la structure `struct i2c_client` (nécessaire pour faire des transferts I^2C) ou la structure `struct foo_device` (dont on peut avoir besoin) ?

Résumé de la situation

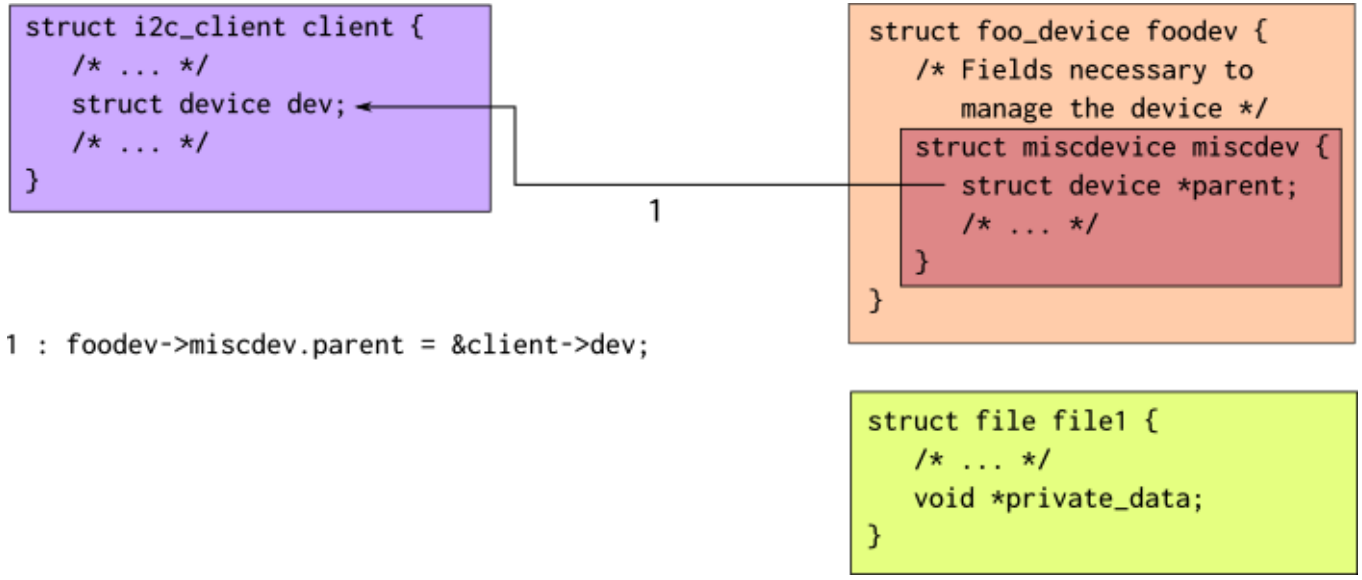
```
struct i2c_client client {
   /* ... */
   struct device dev;
   /* ... */
}
```

```
struct foo_device foodev {
   /* Fields necessary to
      manage the device */
   struct miscdevice miscdev {
      struct device *parent;
      /* ... */
   }
}
```

```
struct file file1 {
   /* ... */
   void *private_data;
}
```

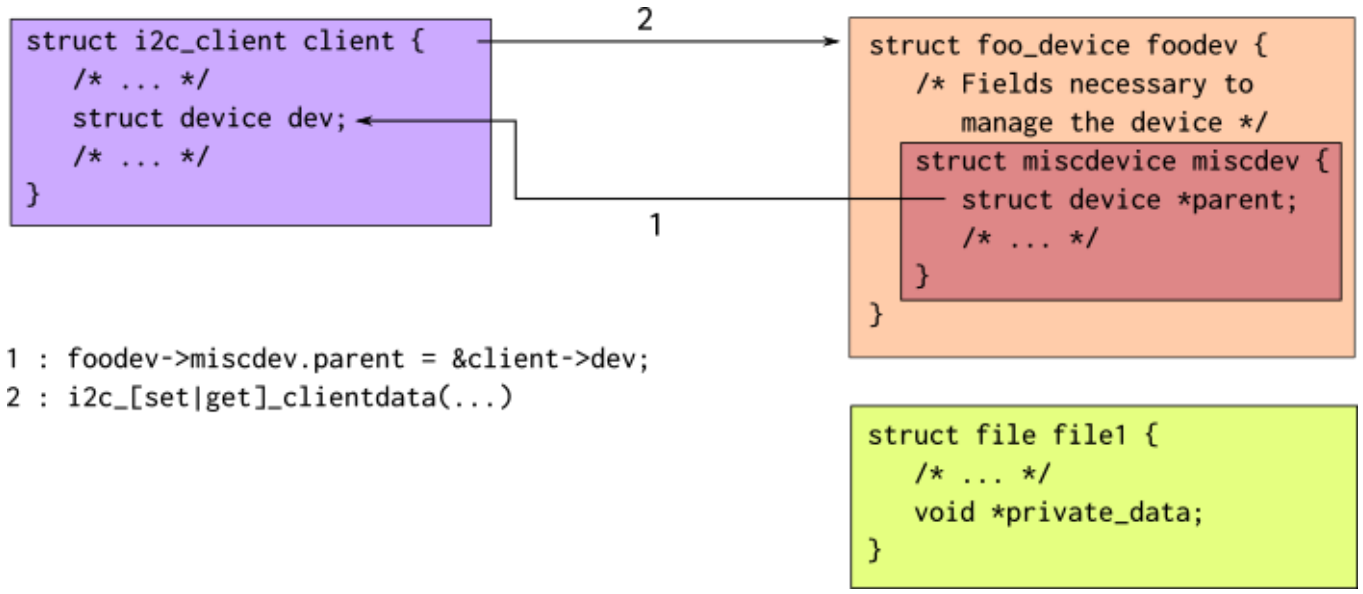
- Comment, à partir d'une de ces structures, accéder aux autres ?

`struct miscdevice` vers `struct device`



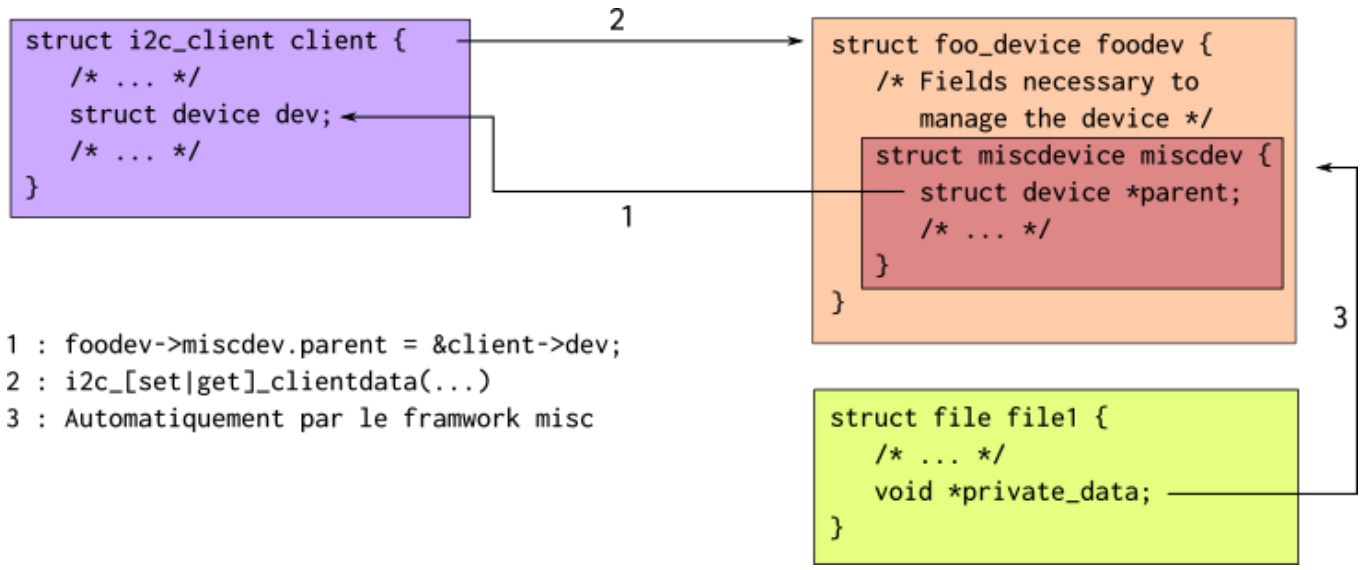
- Dans la fonction `probe`, on initialise le champ `parent` de la structure `struct miscdevice` avec l'adresse du champ `dev` de la structure `struct i2c_client`

`struct i2c_client` vers `struct foo_device`



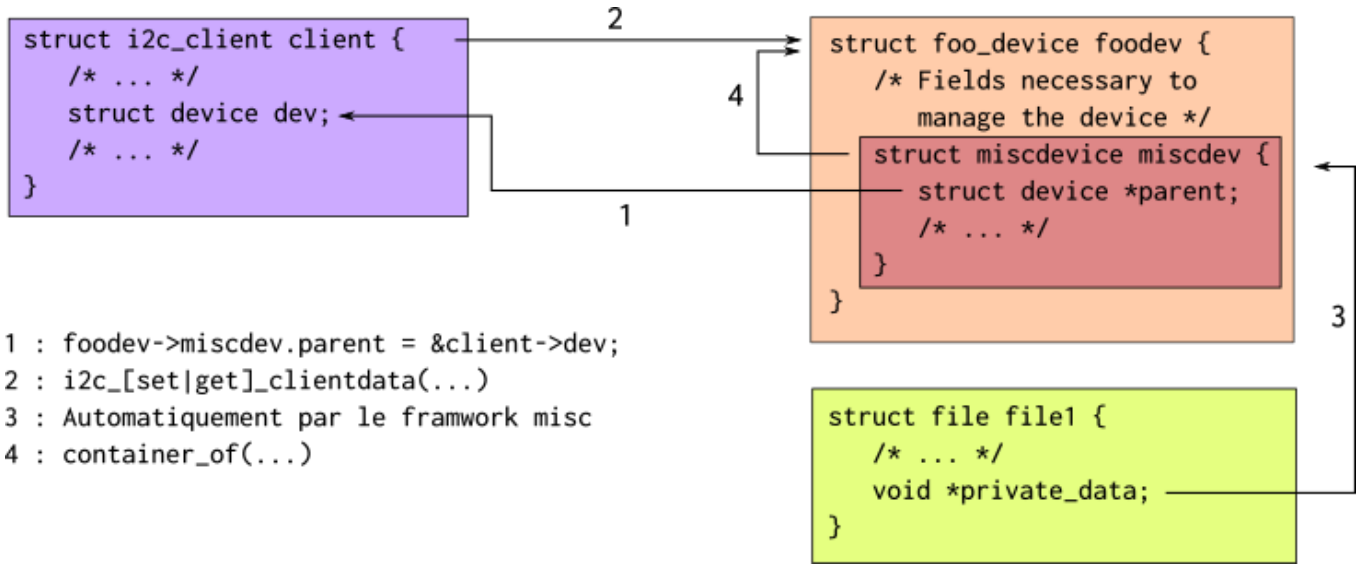
- La fonction `i2c_set_clientdata` permet de stocker un pointeur quelconque dans la structure `i2c_client`
- Ce pointeur peut être récupéré par `i2c_get_clientdata`
- Dans la fonction `probe`, on utilise cette fonction pou stocker l'adresse de `foodev` dans `client`

`struct file` vers `struct miscdevice`



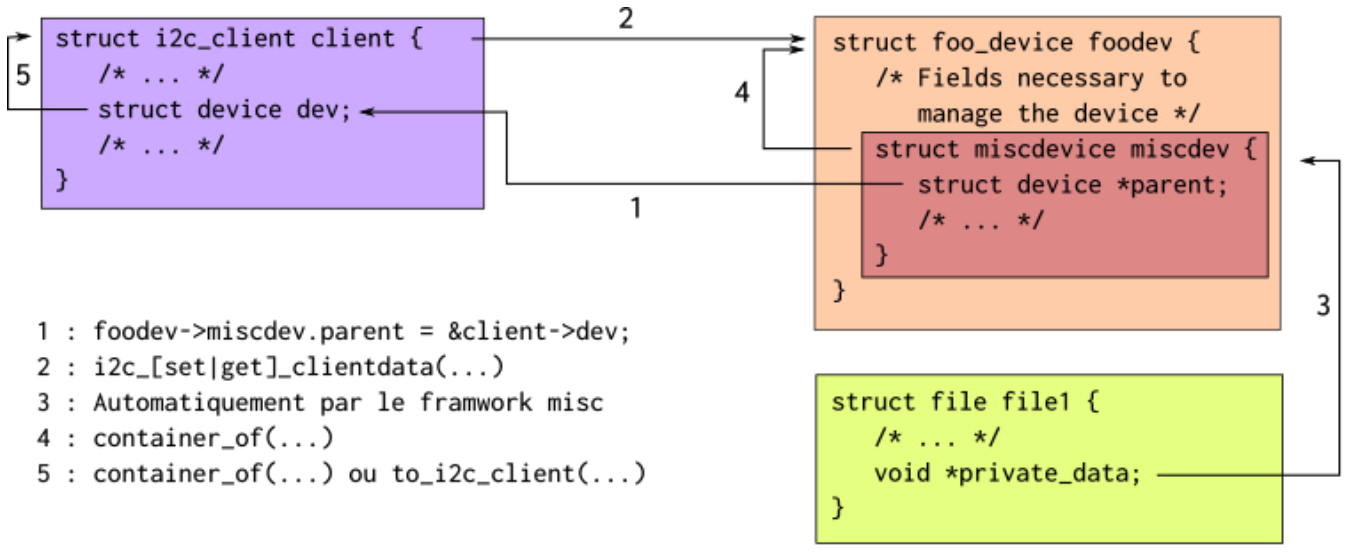
- Le framework *misc* stocke automatiquement dans le champ `private_data` de la structure `struct file` l'adresse de la structure `struct miscdevice` correspondant au fichier ouvert

`struct miscdevice` vers `struct foo_device`



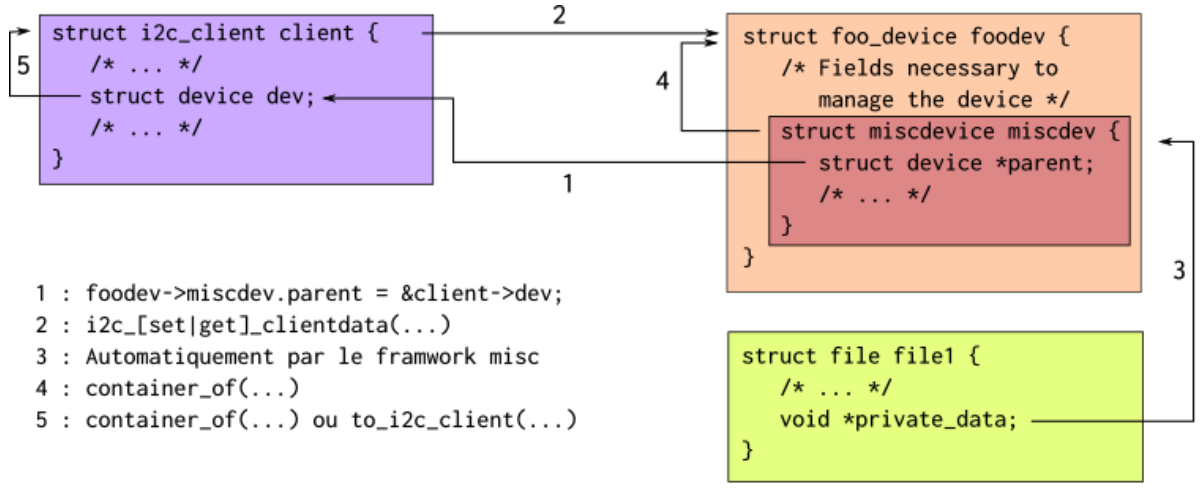
- À partir de l'adresse du champ `miscdev` de la structure `struct foo_device`, il est possible de retrouver l'adresse de la structure qui le contient (`foodev`) à l'aide de la macro `container_of`
- `ptr_foodev = container_of(ptr, struct foo_device, miscdev)`

`struct device` vers `struct i2c_client`



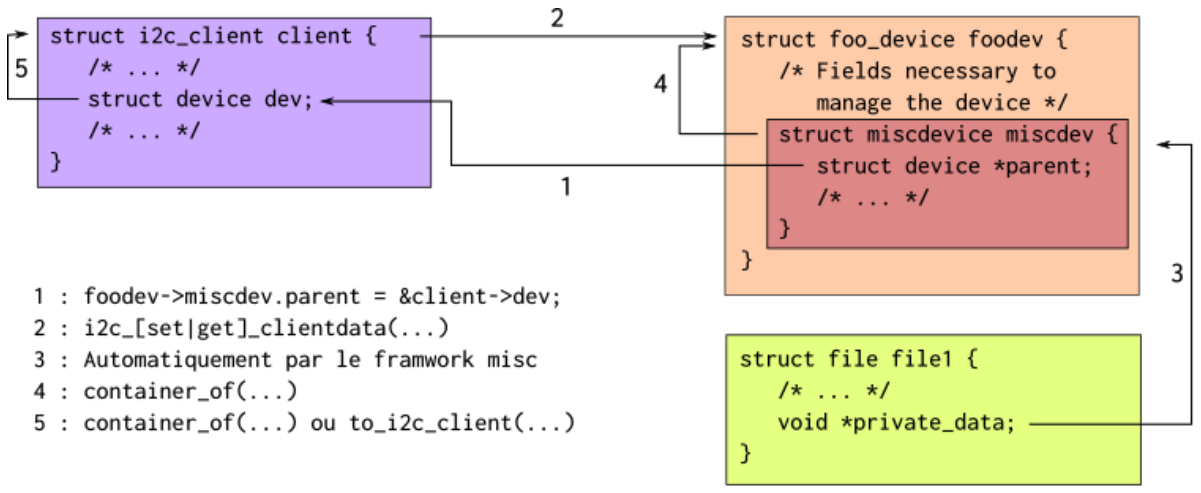
- On peut faire de même pour remonter à `client` à partir de l'adresse du champ `dev` à l'aide de la macro `container_of`
- On peut également utiliser la macro `to_i2c_client` plus simple d'utilisation

Pour résumer



- Dans les fonctions recevant en argument un `struct file` (les fonctions de rappel en cas d'opération sur le fichier spécial), on peut retrouver
  - La structure `struct foo_device` grâce à 3 et 4
  - La structure `struct i2c_client` grâce à 3, 1 et 5

Pour résumer



- Dans les fonctions recevant en argument un `struct i2c_client` (la fonction `remove` par exemple), on peut retrouver
  - La structure `struct foo_device` grâce à 2

Plan

Conclusion

- On viens de voir, pour les périphériques caractères
  - Les fonctions principales à implémenter pour réagir aux appels systèmes effectués par les applications sur les fichiers spéciaux
  - Le transfert de données entre l'espace noyau et l'espace utilisateur
  - Les liens entre les principales structures de données que le pilote devra manipuler
- Il nous reste un point important à voir concernant la mise en veille d'un processus (qui, par exemple, demande une lecture bloquante alors qu'aucune donnée n'est pour le moment disponible)