

# Spécification des fonctions d'un pilote de périphérique de type capteur

Paul ADENOT

Étienne BRODU

## Table des matières

<b>1</b>	<b>Documentation de l'API</b>	<b>2</b>
<b>2</b>	<b>Structure des données</b>	<b>5</b>
2.1	table_capteur . . . . .	5
2.2	table_buffer . . . . .	5
<b>3</b>	<b>Conception graphique</b>	<b>6</b>
3.1	Phase d'initialisation . . . . .	6
3.2	Handler d'interruption . . . . .	6
3.3	open . . . . .	6
3.4	read . . . . .	7
3.5	close . . . . .	7
3.6	ioctl . . . . .	7
<b>4</b>	<b>Plan de test</b>	<b>8</b>

# 1 Documentation de l'API

## open

---

### Synopsis

```
int open(const char* filename, int flags, int perms)
```

### Description

Ouvre le capteur désigné par `filename`, et renvoie un descripteur de fichier (*file descriptor*), qui l'identifie au sein du programme. `flags` indique le mode d'ouverture, et doit être fixé à `O_RDONLY`, les capteurs étant en lecture seule. D'autres valeurs, possiblement passées par l'utilisateur, provoque une erreur, et `errno` est fixé à `EARG`. L'argument `perms` dénote les permission qui seront utilisée sur le fichier. Plusieurs capteurs peuvent être ouvert au sein du même programme. Si un même capteur est ouvert plusieurs fois au sein du même programme, alors plusieurs descripteurs de fichiers seront disponibles pour lire sur un même capteur. Si le fichier précisé dans le premier paramètre (`filename`) n'existe pas, l'appel échoue, et `open` retourne immédiatement, avec la valeur -1.

### Valeur de retour

Si l'appel réussi, un descripteur de fichier (entier positif). Sinon, -1, et `errno` est fixé à l'une des valeurs suivantes :

**EARG :** L'appel a été effectué avec de mauvais arguments, avec une valeur autre que `O_RDONLY` pour `flags`.

**ENEXIST :** Premier argument invalide, le fichier n'existe pas.

**EALREADYOPENED :** Le périphérique est déjà ouvert.

## creat

---

### Synopsis

```
int creat(const char *pathname, int mode);
```

### Description

Le comportement de cette fonction est similaire à celui de la fonction `open`

### Valeur de retour

Les valeurs de retours sont les mêmes que celles de la fonction `open`.

## close

---

### Synopsis

```
int close(int fd);
```

### Description

Ferme le capteur désigné par le descripteur de fichier `fd`. Celui-ci ne sera plus utilisable dans le programme. Si le paramètre `fd` est invalide (i.e. négatif ou ne correspondant pas à un descripteur de fichier valide), `close` retourne -1, et `errno` est positionné à `EARG`.

Si le capteur est en cours d'utilisation, l'appel échoue en renvoyant -1, et `errno` est positionné à `ECPTBUSY`.

### Valeur de retour

Si l'appel réussi, 0 est renvoyé, -1 sinon, et `errno` est positionné aux valeurs suivantes :

**EARG :** L'appel a été effectué avec de mauvais arguments, le `fd` spécifié est invalide.

**ECPTBUSY :** Le capteur est en cours d'utilisation.

## remove

---

### Synopsis

```
int remove(const char *pathname);
```

### Description

Ferme le capteur désigné par `pathname`. Il ne sera plus utilisable au sein du programme. Si `pathname` est invalide (le fichier n'existe pas, ou n'est pas ouvert au sein du programme), alors l'appel échoue en renvoyant -1, et `errno` est positionné à `ENEXIST`. Si le capteur est en cours d'utilisation, l'appel échoue en renvoyant -1, et `errno` est positionné à `ECPTBUSY`.

### Valeur de retour

Si l'appel réussi, 0 est renvoyé, -1 sinon, `errno` est positionné à l'une des valeurs suivantes :

**EARG :** Le fichier précisé n'existe pas.

**ECPTBUSY :** Le capteur est en cours d'utilisation.

## read

---

### Synopsis

```
int read (int fd, char *buffer, size_t maxbytes);
```

### Description

Lit un message d'un capteur désigné par `fd`, et le place dans l'adresse pointé par `buffer`. Si un message est disponible, alors il est placé dans à l'adresse `buffer`, mais n'est pas *consommé*, la lecture étant non destructive. Un message lu sur un capteur est du type `capt_msg`, qui est défini de la manière suivante :

```
1 struct capt_msg
2 {
3     unsigned ID;
4     timestamp date;
5     char msg[TAILLE_MAX];
6 };
```

L'entier `ID` est commun à tous les capteurs, et est incrémenté à chaque message. Lors de l'initialisation du driver, il est fixé à zéro. En cas de dépassement de capacité, la valeur de `ID` redeviendra 0, et continuera normalement.

La date `date` est un entier, qui correspond au nombre de périodes de 20ms qui se sont écoulés depuis le démarrage du système. Il sert donc à ordonner temporellement les message, et non à déterminer leur date d'arrivée.

### Valeur de retour

Un entier positif, correspondant à la taille lue (`TAILLE_MSG`) est renvoyée. En cas d'erreur, -1 est renvoyé, et `errno` est positionné aux valeurs suivantes :

**EARG :** L'appel a été effectué avec de mauvais arguments.

**ENOAVAL :** Aucun message n'est disponible.

## write

---

### Synopsis

```
int write (int fd, char *buffer, size_t maxbytes);
```

### Description

Appel non supporté, les capteurs sont en lecture seule. Pour faire une opération sur un capteur en fonctionnement, utiliser `ioctl`.

### Valeur de retour

N.A.

# ioctl

---

## Synopsis

```
int ioctl(int fd, int request, int value);
```

## Description

Configuration du pilote. Le paramètre **request** doit être égal à la constant **CHANGEMENT\_CAPTEUR**. La valeur de **value** doit alors être inférieur ou égale à 255, et correspond au nouveau numéro de capteur pour le descripteur de fichier passé en premier argument.

## Valeur de retour

**ioctl** renvoie 0 en cas de succès, -1 sinon, et **errno** est alors positionné à l'une des valeurs suivantes :

**EARG :** L'appel a été effectué avec de mauvais arguments.

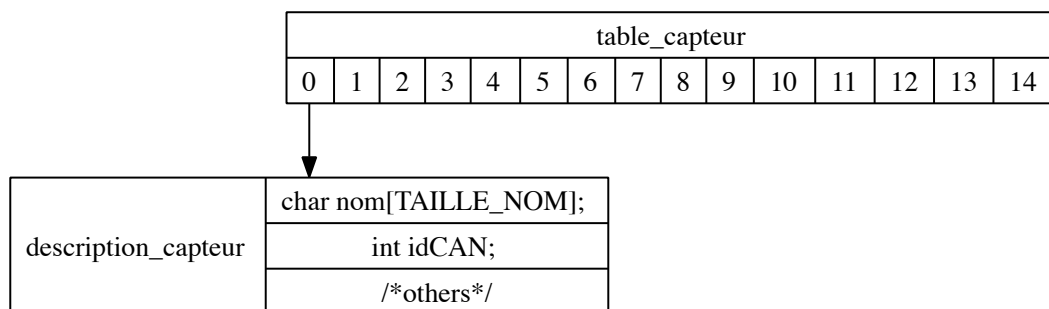
**ECPTBUSY :** Le capteur est occupé, il doit être possible de recommencer l'appel avec succès dans un futur proche.

## 2 Structure des données

### 2.1 table\_capteur

Ce tableau contient 15 pointeurs vers des structures décrivant chaque capteur. L'index du tableau servant d'identifiant logique au sein du driver. Structure décrivant un capteur :

```
1 struct description_capteur
2 {
3     char nom[TAILLE_NOM];
4     int idCAN;
5     /* others */
6 };
```



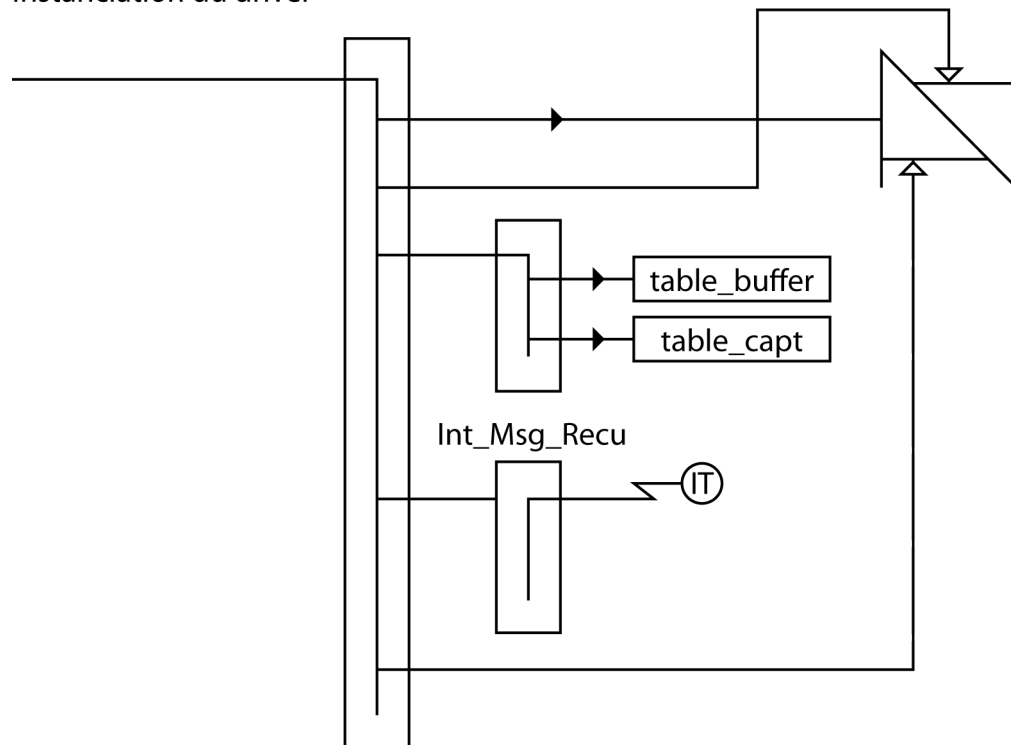
### 2.2 table\_buffer

Ce tableau contient 15 pointeurs vers le dernier message du capteur dont l'index du tableau est l'identifiant logique.

### 3 Conception graphique

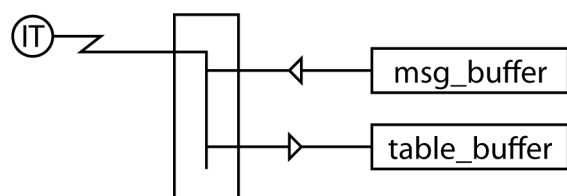
#### 3.1 Phase d'initialisation

Instanciation du driver

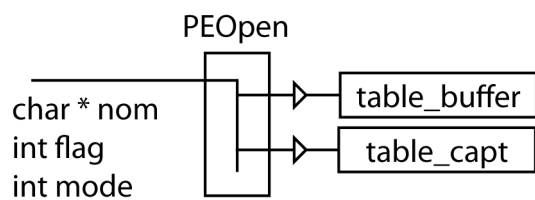


#### 3.2 Handler d'interruption

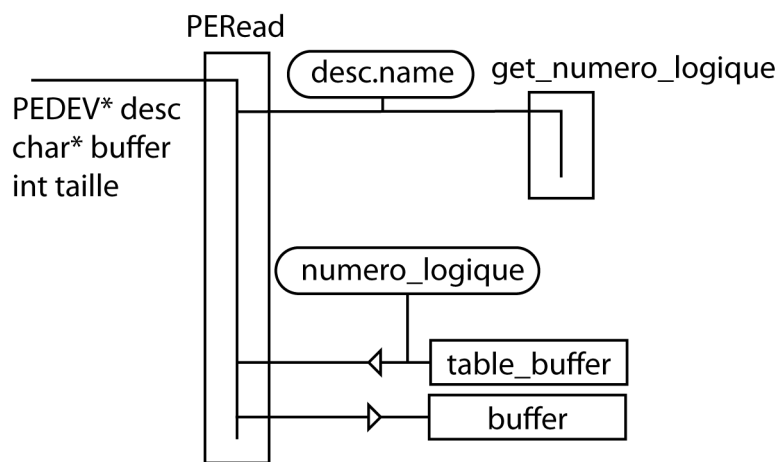
Int\_Msg Recu



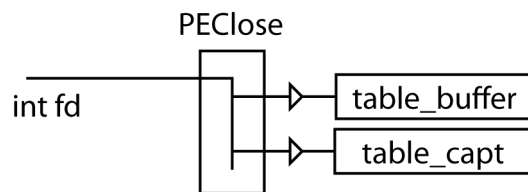
#### 3.3 open



### 3.4 read



### 3.5 close



### 3.6 ioctl

TODO

## 4 Plan de test

### Test 1 – Installation d'un driver

---

#### Description

Installer le driver alors qu'il n'est pas installé

#### Resultat attendu

La valeur de retour doit être positive, et correspond au numéro du driver. Il doit être possible de le retrouver en utilisant la fonction `iosDrvShow`.

### Test 2 – Installation d'un driver déjà installé

---

#### Description

Installer le driver alors qu'il est déjà installé : appeler deux fois `iosDrvInstall`.

#### Resultat attendu

La première installation doit bien se passer (valeur de retour positive). Le second appel de `iosDrvInstall` doit renvoyer `ERROR`.

### Test 3 – Retrait d'un driver

---

#### Description

Utilisation de la fonction `iosDrvRemove`, alors que le pilote est installé sur le système, et qu'il n'est pas utilisé.

#### Resultat attendu

La valeur de retour doit être égale à `OK`.

### Test 4 – Retrait du driver alors qu'il n'est pas installé

---

#### Description

Utilisation de la fonction `iosDrvRemove`, alors que le pilote n'est pas installé sur le système.

#### Resultat attendu

La valeur de retour doit être `ERROR`.

### Test 5 – Retrait du driver alors qu'un périphérique est ouvert

---

#### Description

Alors qu'un capteur a été ouvert en lecture, retirer le driver, à l'aide de la fonction `iosDrvRemove`.

#### Resultat attendu

La fonction doit retourner `ERROR`, et `errno` doit être positionné à `ECPTBUSY`. Le driver ne doit pas être retiré.



## Test 6 – Ajout d’un périphérique

---

### Description

Utilisation de la fonction `iosDevAdd`, une seule fois, avec des paramètres valides.

### Resultat attendu

La valeur de retour doit être OK, le périphérique doit être trouvable en utilisant `iosDevFind`, qui ne doit pas renvoyer NULL.

## Test 7 – Retrait d’un périphérique

---

### Description

Utilisation de la fonction `iosDevDelete`, avec des paramètres valides.

### Resultat attendu

Il ne doit plus être possible d’ouvrir le périphérique : un appel à `open` sur ce périphérique doit échouer (il doit retourner `ERROR`), et `errno` doit être positionné à `ENEXIST`.

## Test 8 – Ajout d’un périphérique alors que 15 périphériques ont été ajoutés.

---

### Description

Utilisation de la fonction `iosDevAdd`, 16 fois, avec des paramètres valides.

### Resultat attendu

Le 16<sup>e</sup> appel à `iosDevAdd` doit provoquer une erreur, et renvoyer `ERROR`.

## Test 9 – Ouverture d’un capteur

---

### Description

Appeler `open` sur un capteur valide (le fichier existe et est accessible en écriture), avec des options valide (`O_RDONLY`), une seule fois.

### Resultat attendu

La valeur de retour doit être un entier positif.

## Test 10 – Ouverture d’un capteur déjà ouvert

---

### Description

Appeler `open` sur un capteur valide (le fichier existe, et est accessible en lecture), alors qu’il vient d’être ouvert avec succès.

### Resultat attendu

`open` doit renvoyer `ERROR`, et `errno` doit être positionné à `EALREADYOPENED`.

## Test 11 – Ouverture d’un capteur avec des paramètres invalides

---

### Description

Appeler `open` sur un capteur valide (le fichier existe, et est accessible en lecture/écriture, en passant un mode différent de `O_RDONLY`).

### Resultat attendu

L’appel doit échouer, et donc renvoyer `ERROR`. De plus, `errno` doit être positionné à `EARG`.

## Test 12 – Fermeture d'un capteur

---

### Description

Appeler `close` sur un descripteur de fichier valide (qui a été ouvert avec succès précédemment), et qui n'a pas été fermé depuis.

### Resultat attendu

La valeur de retour doit être égale à `OK`

## Test 13 – Lecture d'une valeur dans un capteur

---

### Description

Utiliser `read` sur un capteur ouvert avec succès

### Resultat attendu

La valeur de retour doit être un nombre positif, et doit être cohérente par rapport aux paramètres d'appel de `read`.

## Test 14 – Utilisation de `read` avec une taille de lecture invalide

---

### Description

Utilisation de l'appel système `read` avec un descripteur de fichier valide, mais avec une taille de lecture négative.

### Resultat attendu

L'appel doit échouer en renvoyant `-1`, et `errno` doit être positionné à `EARG`.

## Test 15 – Utilisation de `ioctl` avec des paramètres corrects

---

### Description

Utilisation de `ioctl` avec des paramètres corrects : un descripteur de fichier valide, une valeur pour `request` égale à la constante `CHANGEMENT_CAPTEUR`, et une valeur pour `value` inférieure ou égale à 255, correspondant bien à un capteur valide.

### Resultat attendu

La valeur de retour doit être égale à `OK`, ou alors elle doit être égale à `ERROR`, mais alors `errno` doit être positionné à `ECPTBUSY`, et le même appel effectué ultérieurement doit renvoyer `OK`.

## Test 16 – Utilisation de `ioctl` avec de mauvais arguments

---

### Description

Utilisation de la fonction `ioctl` avec un second paramètre correspondant à une fonction non-implémentée. Le descripteur de fichier passé en tant que premier paramètre doit être valide, et correspondre à un périphérique géré par le driver.

### Resultat attendu

L'appel doit échouer (la valeur de retour doit être égale à `ERROR`), et `errno` doit être positionné à `EARG`.

### Test 17 – Utilisation de `ioctl` pour associer le même capteur à deux descripteur de fichiers

#### Description

On utilise la fonction `ioctl` deux fois, avec des descripteurs de fichier différents, mais avec le même numéro de capteur.

#### Resultat attendu

L'appel doit échouer, et renvoyer `ERROR`. `errno` doit alors être positionné à `EARG`.

### Test 18 – Utilisation de `write` \_\_\_\_\_

#### Description

Appel de `write` sur un capteur ouvert avec succès.

#### Resultat attendu

L'appel doit échouer, et `errno` doit être positionné à `ENOTSUP`.

### Test 19 – Retrait d'un périphérique ouvert \_\_\_\_\_

#### Description

Tentative de retrait d'un périphérique sur lequel un appel de `open` a été effectué avec succès précédemment, et sur lequel on n'a pas fait d'appel à `close` ou `remove`.

#### Resultat attendu

L'appel doit réussir : la valeur de retour doit être égale à `OK`.

### Test 20 – Retrait d'un périphérique qui n'existe pas \_\_\_\_\_

#### Description

Tentative de retrait d'un périphérique qui n'existe pas.

#### Resultat attendu

L'appel doit échouer, et retourner `ERROR`. `errno` doit être positionné à `ENEXIST`.