Code d'un pilote de périphérique de type capteur

Paul ADENOT Étienne BRODU

Table des matières

1 Fichier d'entête du pilote

```
1  /**
2   * Auteur : Paul ADENOT & Etienne Brodu
3   * Ce fichier fournit les en-têtes pour le pilote implémenté dans le fichier
4   * driver.c
5   */
6
7  #include <iosLib.h>
8  #include <time.h>
9
10
11 /* La taille maximale des données d'un message (\0 compris)*/
12 #define MAX_DATA_SIZE 64
13 /* La taille maximale d'un nom de capteur (chemin complet, et \0 compris) */
14 #define NAME_SIZE 32
15
16 /**
17  * L'état d'un capteur : il peut être fermé, ouvert, ou pas encore
18  * crée.
19  */
20 typedef enum
21 {
22        closed,  //!< closed Le capteur est fermé.
23        opened,  //!< opened Le capteur est ouvert.
24        notcreated//!< notcreated L'indice n'a pas de capteur associé.
25 } EtatCapteur;
26
27
28 /**
29  * Les données spécifiques d'une structure PEDEV.
30  */
31 typedef struct
32 {
33        char name[NAME_SIZE]; ///< name Le nom du capteur.
34        EtatCapteur state; ///< state L'état du capteur.
35        int address; ///< address L'adresse CAN du capteur.
36 } specific_t;
37
38 /**
39  * Membre d'une liste chainée gérée par l'ios. Contient des
40  * des informations de gestion dans le membre specific.
41  */
42 typedef struct
43 {
44        DEV_HDR header;
45        specific_t specific;
46 } PEDEV;
```

```
47
48  /**
49   * Structure représentant un message.
50   */
51  typedef struct
52  {
53          unsigned id; ///< id identifiant du message
54          struct timespec date; ///< date date de reception
55          char msg[MAX_DATA_SIZE]; ///< msg contenu
56  } Message;
57
58
59  /**
60   * Valeurs possibles de errno pour ce pilote.
61   */
62  typedef enum {
63          EARG, /** Problème d'arguments */
64          ENEXIST, /** Le capteur n'existe pas */
65          EALREADYOPENED, /** Le capteur est déjà ouvert */
66          ENOTOPENED, /** Le capteur n'est pas ouvert */
67          ECPTBUSY, /** Le capteur est en cours d'utilisation */
68          ECPTALREADYUSED, /** Le capteur est deja associe a un autre peripherique */
69          ENOAVAIL, /** Aucun message n'est disponible */
70          EINSTALLED, /** Le pilote est déjà installé */
71          ENINSTALLED, /** Le pilote n'est pas installé */
72          ETOOMUCHDEV, /** Nombre de périphériques demandé superieur à ceux disponible */
73          EUNKNOW, /** Erreur inconnue */
74          ECANNOTADD /** Impossible d'ajouter un capteur */
75  };
76
77  /**
78   * Valeurs possibles de request pour ioctl.
79   */
80  typedef enum {
81          SET_CPT_ADDRESS /** Changer l'adresse d'un capteur */
82  };
83
84  int pe_driverInstall(int dev_count);
85  int pe_driverUninstall();
86  int pe_deviceAdd(int i);
87  int pe_deviceRemove(int i);
88
89  void hardware_mockup(int address, char* data);
```

2 Fichier d'implémentation du pilote

```
1   /**
2    * Auteurs : Paul ADENOT & Etienne BRODU
3    * Ce fichier fournit l'implémentation d'un pilote de périphérique VxWorks, capable
4    * de fournir un moyen d'interroger des capteurs sur un bus CAN.
5    */
6
7
8   #include "driver.h"
9   #include <vxWorks.h>
10  #include <string.h>
11  #include <stdlib.h>
12  #include <errnoLib.h>
13  #include <stdio.h>
14  #include <semLib.h>
15  #include <timers.h>
16  #include <time.h>
17  #include <sysLib.h>
18  #include <msgQLib.h>
19  #include <taskLib.h>
20  #include <errnoLib.h>
21
22  /**
23   * Constantes internes.
24   */
```

```
25  /* Nom d'un périphérique. Doit être utilisé avec sprintf pour
26   * indiquer son numéro */
27  #define DEVICE_BASENAME "/dev/capteur%d"
28
29  /* Nombre maximal de périphérique installables */
30  #define DEVICE_MAX_COUNT (15)
31
32  #define DISPATCHER_PRIORITY 200
33
34  /***
35   * == Globaux
36   ***/
37  /** driver_id vaut -1 si le driver n'est pas installé.
38   * Il est égal au numéro du driver sinon.
39   */
40  static int driver_id = -1;
41  /**
42   * Buffer pour les messages. Chaque capteur a la place pour
43   * un seul message.
44   */
45  static Message table_buffer[DEVICE_MAX_COUNT];
46  /**
47   * Table d'information sur les périphériques.
48   */
49  static PEDEV table_capt[DEVICE_MAX_COUNT];
50  /**
51   * Sémaphore d'exclusion mutuelle pour la table des capteurs.
52   */
53  static SEM_ID mut_table_capt;
54  /**
55   * Sémaphore d'exclusion mutuelle pour les buffers de messages.
56   */
57  static SEM_ID mut_table_buffer;
58
59  /**
60   * File de messages receptionnés par le handler d'it.
61   */
62  static MSG_Q_ID msgQ_dispatcher;
63
64  static int id_pe_task_dispatcher;
65
66  /**
67   * Fausse mémoire de la carte réseau pour le bus can.
68   */
69  static struct
70  {
71          unsigned char address;
72          char data[MAX_DATA_SIZE];
73  } mock_can_device;
74
75  static Message msg_buffer;
76
77  /***
78   * ////////////////////////////////////////////////////////////////
79   * /// Fonctions du driver : implémentation des appels systèmes. ///
80   * ////////////////////////////////////////////////////////////////
81   ***/
82
83  /**
84   *
85   * @param desc
86   * @param remainder
87   * @param flag
88   * @return desc
89   */
90  static int pe_open(PEDEV* desc, char* remainder, int flag)
91  {
92          if( *remainder != '\0' )
93          {
94                  errnoSet(EARG);
95                  return ERROR;
96          }
97
```

```
98          if( flag != O_RDONLY )
99          {
100                 errnoSet(EARG);
101                 return ERROR;
102         }
103
104         // On ouvre le capteur
105         semTake(mut_table_capt, WAIT_FOREVER);
106
107         switch (desc->specific.state)
108         {
109                 case opened :
110                         semGive(mut_table_capt);
111                         errnoSet(EALREADYOPENED);
112                         return ERROR;
113
114                 case notcreated :
115                         semGive(mut_table_capt);
116                         errnoSet(ENEXIST);
117                         return ERROR;
118
119                 default:
120                         break;
121         }
122
123         desc->specific.state = opened;
124         semGive(mut_table_capt);
125
126         return ((int) desc);
127 }
128
129 /**
130  *
131  * @param desc
132  * @param remainder
133  * @param flag
134  * @return OK
135  */
136 static int pe_close(PEDEV* desc, char* name)
137 {
138         // On ferme le capteur
139         semTake(mut_table_capt, WAIT_FOREVER);
140         if(desc->specific.state == opened)
141         {
142                 desc->specific.state = closed;
143                 semGive(mut_table_capt);
144                 return OK;
145         }
146         else
147         {
148                 semGive(mut_table_capt);
149                 errnoSet(ENOTOPENED);
150                 return ERROR;
151
152         }
153 }
154
155 /**
156  *
157  * @param desc
158  * @param buff
159  * @param nBytes
160  * @return
161  */
162 static int pe_read (PEDEV* desc, char* buff, int nBytes)
163 {
164         int device;
165
166         if(nBytes < 0)
167         {
168                 errnoSet(EARG);
169                 return ERROR;
170         }
```

```
171
172         /*
173          * Pour trouver l'index du capteur dans le tableau de capteur,
174          * 2 solutions sont possible :
175          *
176          * - chercher une correspondance entre l'adresse
177          * renvoyé par le descripteur de fichier et chaucun des capteurs du tableau de
                 capteur.
178          * - calculer l'indice en soustrayant l'adresse du descripteur de fichier a l'
                 adresse
179          * du tableau de capteur.
180          *
181          * Pour une question d'efficacité, nous avons préféré calculer l'indice,
182          * cependant, cette methode ne peut pas être considéré comme sûr, nous avons donc
                 laissé
183          * l'autre solution imaginé.
184          */
185
186         //semTake(mut_table_capt, WAIT_FOREVER);
187         // On cherche si
188         /*for(i=0; i < DEVICE_MAX_COUNT; ++i)
189         {
190                 if(table_capt[i].specific.state == opened &&
191                    table_capt[i].specific.address == desc->specific.address)
192                 {
193
194                         semTake(mut_table_buffer, WAIT_FOREVER);
195                         memcpy(buff, &(table_buffer[i]), nBytes);
196                         semGive(mut_table_buffer);
197
198                         break;
199                 }
200         }
201         semGive(mut_table_capt);
202         */
203
204         // On clacule l'index du capteur en comparant l'adresse de desc a celle du tableau
                 .
205         device = ((int)desc - (int)&(table_capt))/sizeof(PEDEV);
206
207         semTake(mut_table_buffer, WAIT_FOREVER);
208         if(table_buffer[device].id == -1)
209         {
210                 semGive(mut_table_buffer);
211                 errnoSet(ENOAVAIL);
212                 return ERROR;
213         }
214         memcpy(buff, &(table_buffer[device]), nBytes);
215         semGive(mut_table_buffer);
216
217         return sizeof(Message);
218 }
219
220 /**
221  *
222  * @param desc
223  * @param request
224  * @param value
225  * @return OK en cas de succès, ERROR sinon.
226  */
227 static int pe_ioctl_change_cpt(PEDEV* desc, int request, int value)
228 {
229         int i;
230         if(value >= 0 && value <= 255)
231         {
232                 semTake(mut_table_capt, WAIT_FOREVER);
233
234                 for(i=0; i < DEVICE_MAX_COUNT; ++i)
235                 {
236                         if(table_capt[i].specific.state == opened &&
237                            table_capt[i].specific.address == value)
238                         {
239                                 errnoSet(ECPTALREADYUSED);
```

```
240                             return ERROR;
241                     }
242             }
243
244             desc->specific.address = value;
245             semGive(mut_table_capt);
246             return OK;
247     }
248     else
249     {
250             errnoSet(EARG);
251             return ERROR;
252     }
253 }
254
255 /**
256  *
257  * @param desc
258  * @param request
259  * @param value
260  * @return OK en cas de succès, ERROR sinon.
261  */
262 static int pe_ioctl (PEDEV* desc, int request, int value)
263 {
264     switch (request)
265     {
266             case SET_CPT_ADDRESS:
267                     return pe_ioctl_change_cpt(desc, request, value);
268             default:
269                     errnoSet(EARG);
270                     return ERROR;
271     }
272 }
273
274 /**
275  * Cette fonction est appelée quand un message est disponible sur le réseau.
276  * Elle numérote et date le message, et le place dans le buffer correspondant a son
277  * addresse, si un device logiciel est associé à un capteur.
278  * Sinon, le message est perdu.
279  */
280 void pe_interrupt_handler()
281 {
282     /* Les id des messages sont globales au driver */
283     static unsigned id_msg = 0;
284     Message msg;
285
286     msg.id = id_msg++;
287     // On fait passer l'adresse par la date non initialisé pour éviter des champs de
        mémoire supplémentaire inutile.
288     msg.date.tv_nsec = mock_can_device.address;
289     strncpy(msg.msg, mock_can_device.data, MAX_DATA_SIZE);
290
291     msgQSend( msgQ_dispatcher, (char*)&msg, sizeof(msg), WAIT_FOREVER, MSG_PRI_NORMAL)
        ;
292 }
293
294 void pe_dispatcher()
295 {
296     int i;
297     Message msg;
298
299     for(;;)
300     {
301             msgQReceive(msgQ_dispatcher, (char*)&msg, sizeof(msg), WAIT_FOREVER);
302
303             semTake(mut_table_capt, WAIT_FOREVER);
304             // On cherche si
305             for(i=0; i < DEVICE_MAX_COUNT; ++i)
306             {
307                     if(table_capt[i].specific.state == opened &&
308                        table_capt[i].specific.address == msg.date.tv_nsec)
309                     // On fait passer l'adresse par la date non initialisé pour éviter
                        des champs de mémoire supplémentaire inutile.
```

```
310                         {
311                                 // Trouvé : on ne drop pas le message.
312                                 clock_gettime(CLOCK_REALTIME, &(msg.date));
313 
314                                 semTake(mut_table_buffer, WAIT_FOREVER);
315                                 memcpy((char*)&(table_buffer[i]), (char*)&msg, sizeof(Message
                                        ));
316                                 semGive(mut_table_buffer);
317 
318                                 break;
319                         }
320                 }
321                 semGive(mut_table_capt);
322         }
323 }
324 
325 
326 /**
327  * @brief Détruire les ressources systèmes.
328  * @return OK en cas de succès, ERROR sinon.
329  */
330 int pe_cleanup_resources()
331 {
332         int errtest = 0;
333 
334         // Destruction des ressources.
335         if(semDelete(mut_table_capt))
336         {
337                 printf("Erreur␣:␣Problème␣de␣destruction␣du␣sémaphore␣pour␣table_capt.\n");
338                 errtest = 1;
339         }
340         if(semDelete(mut_table_buffer))
341         {
342                 printf("Erreur␣:␣Problème␣de␣destruction␣du␣sémaphore␣pour␣table_buffer.\n"
                        );
343                 errtest = 1;
344         }
345         if(msgQDelete(msgQ_dispatcher))
346         {
347                 printf("Erreur␣:␣mauvais␣numéro␣de␣file␣de␣message.\n");
348                 errtest = 1;
349         }
350 
351         if(taskDelete(id_pe_task_dispatcher))
352         {
353                 printf("Erreur␣:␣mauvais␣numéro␣de␣tache␣pour␣la␣tache␣dispatcher.\n");
354                 errtest = 1;
355         }
356 
357         if (errtest)
358         {
359                 return ERROR;
360         }
361         return OK;
362 }
363 
364 /***
365  * ////////////////////////////////////////////////////
366  * /// Fonction d'administration du driver : interface ///
367  * ////////////////////////////////////////////////////
368  *
369  * Ces fonctions permettent d'installer / désinstaller le pilote, et
370  * d'enlever / ajouter des périphériques.
371  *
372  * Elles encapsulent les appels classiques de l'ios (iosDrvInstall, etc.),
373  * et doivent être utilisées pour manipuler ce driver.
374  ***/
375 
376 /**
377  * @brief Installation du périphérique.
378  *
379  * @param dev_count Le nombre de périphériques à créer.
380  *
```

```
381  * @return OK si la fonction s'est executé normalement, ERROR sinon.
382  */
383 int pe_driverInstall(int dev_count)
384 {
385         /* iteration pour la création des périphériques */
386         int i = 0;
387         struct timespec initial_time = {0 , 0};
388
389
390         if(dev_count > DEVICE_MAX_COUNT)
391         {
392                 errnoSet(ETOOMUCHDEV);
393                 return ERROR;
394         }
395
396         if(driver_id != -1)
397         {
398                 errnoSet(EINSTALLED);
399                 return ERROR;
400         }
401
402         /* Driver installation */
403         driver_id = iosDrvInstall(pe_open,pe_close, pe_open, pe_close, pe_read, (FUNCPTR)
                NULL, pe_ioctl);
404         if(driver_id < 0)
405         {
406                 errnoSet(EUNKNOW);
407                 return ERROR;
408         }
409
410         /* Création du sémaphore d'exclusion mutuelle pour les deux tables (table_capt
411          * et table_buffer, qui sont susceptibl
412          * Les tâches s'enfilent dans l'ordre d'arrivée avec SEM_Q_FIFO, et
413          * SEM_DELETE_SAFE garantie que la tâche ayant verrouillé un sémaphore ne soit
414          * pas détruite avant de le libérer.
415          */
416         mut_table_capt = semMCreate(SEM_Q_FIFO | SEM_DELETE_SAFE);
417
418         if(mut_table_capt == NULL)
419         {
420                 return ERROR;
421         }
422
423         mut_table_buffer = semMCreate(SEM_Q_FIFO | SEM_DELETE_SAFE);
424
425         if(mut_table_buffer == NULL)
426         {
427                 return ERROR;
428         }
429
430         /* Création de la file de message de communication entre le serveur d'interuption
431          * et la tâche dispatcher
432          */
433         msgQ_dispatcher = msgQCreate(10, sizeof(Message), MSG_Q_FIFO );
434
435         if(msgQ_dispatcher == NULL)
436         {
437                 return ERROR;
438         }
439
440         /* Création de la tâche dispatcher
441          */
442         id_pe_task_dispatcher = taskSpawn("pe_task_dispatcher", DISPATCHER_PRIORITY, 0,
                512, (FUNCPTR) pe_dispatcher, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
443
444         if(id_pe_task_dispatcher == ERROR)
445         {
446                 return ERROR;
447         }
448
449         for(i=0 ; i < dev_count; ++i)
450         {
451                 if(pe_deviceAdd(i) == ERROR)
```

```
452                 {
453                         pe_cleanup_resources();
454                         errnoSet(ECANNOTADD);
455                         return ERROR;
456                 }
457         }
458
459         for(; i < DEVICE_MAX_COUNT; i++)
460         {
461                 table_capt[i].specific.address = -1;
462                 table_capt[i].specific.state = notcreated;
463                 table_buffer[i].id = -1;
464         }
465
466         /* Initialisation de l'horloge système : elle est placée arbitrairement
467          * à 0:0 à l'installation du pilote. */
468         sysClkRateSet(100);
469         clock_settime(CLOCK_REALTIME, &initial_time);
470
471         return driver_id;
472 }
473
474 /**
475  * @brief Retirer le driver.
476  * @return OK en cas de succès, ERROR sinon.
477  */
478 int pe_driverUninstall()
479 {
480         int i;
481         if(driver_id != -1)
482         {
483                 /* L'appel ne renvoit pas d'erreur bien que des capteurs soient ouverts.
484                  * Nous vérifions donc qu'aucun capteur ne soit ouvert.
485                  */
486
487                 semTake(mut_table_capt, WAIT_FOREVER);
488                 for(i=0; i < DEVICE_MAX_COUNT; ++i)
489                 {
490                         if(table_capt[i].specific.state == opened)
491                         {
492                                 errnoSet(ECPTBUSY);
493                                 return ERROR;
494                         }
495                 }
496                 semGive(mut_table_capt);
497
498                 if(iosDrvRemove(driver_id, FALSE) == ERROR)
499                 {
500                         // The driver has open files.
501                         errnoSet(ECPTBUSY);
502                         return ERROR;
503                 }
504
505                 driver_id = -1;
506
507                 if(pe_cleanup_resources() == ERROR)
508                 {
509                         return ERROR;
510                 }
511         }
512         else
513         {
514                 errnoSet(ENINSTALLED);
515                 return ERROR;
516         }
517
518         return OK;
519 }
520
521 /**
522  * @brief Ajouter un périphérique.
523  * @param i
524  * @return OK en cas de succès, ERROR sinon.
```

```
525  */
526 int pe_deviceAdd(int i)
527 {
528         // Buffer pour le nom de périphérique.
529         char dev_name[NAME_SIZE];
530 
531         if(driver_id == -1)
532         {
533                 errnoSet(S_ioLib_NO_DRIVER);
534                 return ERROR;
535         }
536 
537         sprintf(dev_name, DEVICE_BASENAME, i);
538         if(iosDevAdd((DEV_HDR*)&(table_capt[i]),dev_name,driver_id) == ERROR)
539         {
540                 errnoSet(ECANNOTADD);
541                 return ERROR;
542         }
543 
544         semTake(mut_table_capt, WAIT_FOREVER);
545         // Le capteur est fermé
546         table_capt[i].specific.state = closed;
547         // Il n'a pas encore de capteur associé, au niveau CAN.
548         // utiliser ioctl pour lui en founir une.
549         table_capt[i].specific.address = -1;
550         semGive(mut_table_capt);
551         return OK;
552 }
553 
554 /**
555  * @brief Enlever un périphérique.
556  * @param i L'index du device à enlever
557  * @return OK pour un succès, ERROR sinon.
558  */
559 int pe_deviceRemove(int i)
560 {
561         semTake(mut_table_capt, WAIT_FOREVER);
562         if(i < DEVICE_MAX_COUNT && table_capt[i].specific.state != notcreated)
563         {
564                 iosDevDelete((DEV_HDR*)&(table_capt[i]));
565                 table_capt[i].specific.state = notcreated;
566                 table_capt[i].specific.address = -1;
567                 semGive(mut_table_capt);
568 
569                 semTake(mut_table_capt, WAIT_FOREVER);
570                 table_buffer[i].id = -1;
571                 semGive(mut_table_buffer);
572 
573                 return OK;
574         }
575         else
576         {
577                 // Le capteur n'a pas été crée :
578                 // " If the device was never added to the device list,
579                 //   unpredictable results may occur."
580                 semGive(mut_table_capt);
581                 errnoSet(ENEXIST);
582                 return ERROR;
583         }
584 }
585 
586 /**
587  * Cette fonction mime le comportement du hardware :
588  * Elle place un message dans le faux buffer de la carte réseau,
589  * et simule une interruption, en appelant le handler.
590  *
591  * @param address L'adresse du capteur où est envoyée le message.
592  */
593 void hardware_mockup(int address, char* data)
594 {
595         strncpy(mock_can_device.data, data, MAX_DATA_SIZE);
596         mock_can_device.address = address;
597         pe_interrupt_handler();
```

```
598 }
```

3 Fichier d'implémentation des tests du pilote

```
1  /**
2   * Auteurs : Paul ADENOT & Étienne Brodu.
3   * Ce fichier fournit la suite de test pour le pilote
4   * implémenté dans le fichier driver.c
5   */
6
7  #include "driver.h"
8  #include <vxWorks.h>
9  #include <errnoLib.h>
10 #include <stdio.h>
11 #include <string.h>
12
13 /* Écrit un message d'erreur sur la sortie d'erreur */
14 #define ERR_MSG(x) printf("Test␣%d␣failed.␣errno␣:␣%d\n",(x), errnoGet());
15
16 /**
17  * Test 1 - Installation d'un driver
18  * Description
19  * Installer le driver alors qu'il n'est pas installé
20  * Resultat attendu
21  * La valeur de retour doit être positive, et correspond au numéro du driver. Il doit
        être possible
22  * de le retrouver en utilisant la fonction iosDrvShow.
23  **/
24 int test_1()
25 {
26         int success = 0;
27         if(pe_driverInstall(10) >= 0)
28         {
29                 success++;
30                 iosDrvShow();
31                 getchar();
32                 // Call this to clean the system.
33         }
34         else
35         {
36                 ERR_MSG(1);
37         }
38         pe_driverUninstall();
39         return success;
40 }
41
42  /**
43  * Test 2 - Installation d'un driver déjà installé
44  * Description
45  * Installer le driver alors qu'il est déjà installé : appeler deux fois iosDrvInstall.
46  * Resultat attendu
47  * La première installation doit bien se passer (valeur de retour positive). Le second
        appel de
48  * iosDrvInstall doit renvoyer ERROR.
49  **/
50 int test_2()
51 {
52         int success = 0;
53         if(pe_driverInstall(10))
54         {
55                 if( pe_driverInstall(10) == -1)
56                 {
57                         success++;
58                 }
59                 else
60                 {
61                         ERR_MSG(2)
62                 }
63         }
64         else
```

```
65          {
66                  ERR_MSG(2)
67          }
68
69          // Call this to clean the system.
70          pe_driverUninstall();
71          return success;
72  }
73   /**
74   * Test 3 - Retrait d'un driver
75   * Description
76   * Utilisation de la fonction iosDrvRemove, alors que le pilote est installé sur le
         système, et qu'il
77   * n'est pas utilisé.
78   * Resultat attendu
79   * La valeur de retour doit être égale à OK.
80   **/
81  int test_3()
82  {
83          int success = 0;
84          pe_driverInstall(10);
85          if(pe_driverUninstall() == OK)
86          {
87                  success++;
88          }
89          else
90          {
91                  ERR_MSG(3);
92          }
93          return success;
94  }
95   /**
96   * Test 4 - Retrait du driver alors qu'il n'est pas installé
97   * Description
98   * Utilisation de la fonction iosDrvRemove, alors que le pilote n'est pas installé sur le
         système.
99   * Resultat attendu
100  * La valeur de retour doit être ERROR.
101  **/
102 int test_4()
103 {
104         int success = 0;
105         if(pe_driverUninstall() == ERROR)
106         {
107                 success++;
108         }
109         else
110         {
111                 ERR_MSG(4)
112         }
113
114         return success;
115 }
116
117 /**
118  * Test 5 - Retrait du driver alors qu'un périphérique est ouvert
119  * Description
120  * Alors qu'un capteur a été ouvert en lecture, retirer le driver, à l'aide de la
         fonction iosDrvRemove.
121  * Resultat attendu
122  * La fonction doit retourner ERROR, et errno doit être positionné à ECPTBUSY. Le driver
         ne doit
123  * pas être retiré.
124  */
125 int test_5()
126 {
127         int success = 0;
128         int fd = 0;
129         Message buffer;
130
131         pe_driverInstall(10);
132
133         // Ouvertude d'un capteur.
```

```
134         fd = open("/dev/capteur5", O_RDONLY, 0);
135         ioctl(fd, SET_CPT_ADDRESS, 42);
136         hardware_mockup(42, "test 5\0");
137 
138 
139         // l'attente est obligatoire sinon le message n'est pas encore traité.
140         sleep(1);
141         read(fd,(char*)&(buffer), sizeof(buffer));
142 
143         if(strncmp(buffer.msg, "test 5\0", 7) != 0)
144         {
145                 printf("Erreur a la lecture du message\n");
146         }
147 
148         if( fd != ERROR )
149         {
150                 // Tentative de retrait du driver : cela doit être un échec.
151                 if(pe_driverUninstall() == ERROR)
152                 {
153                         if(errnoGet() == ECPTBUSY)
154                                 success++;
155                         else
156                                 printf("Pas le bon code errno : test_5\n");
157                 }
158                 else
159                 {
160                         printf("driver enleve avec succes : Echec\n");
161                 }
162 
163                 close(fd);
164         }
165         else
166         {
167                 printf("Erreur a l'ouverture du capteur\n");
168         }
169         pe_driverUninstall();
170 
171         return success;
172 }
173 
174 /**
175  * Test 6 - Ajout d'un périphérique
176  * Description
177  * Utilisation de la fonction iosDevAdd, une seule fois, avec des paramètre valides.
178  * Resultat attendu
179  * La valeur de retour doit être OK, le périphérique doit être trouvable en utilisant
        iosDevFind, qui
180  * ne doit pas renvoyer NULL.
181  **/
182 int test_6()
183 {
184         int success = 0;
185         int fd = 0;
186 
187         // on n'installe pas de device par défaut.
188         pe_driverInstall(0);
189 
190         // On utilise la procédure qui encapsule iosDevAdd.
191         if(pe_deviceAdd(0) == OK)
192         {
193                 if(iosDevFind("/dev/capteur0",NULL) != NULL)
194                 {
195                         success++;
196                 }
197                 else
198                 {
199                         ERR_MSG(6);
200                 }
201         }
202         else
203         {
204                 ERR_MSG(6);
205         }
```

```
206
207         pe_driverUninstall();
208
209         return success;
210 }
211 /**
212  * Test 7 - Retrait d'un périphérique
213  * Description
214  * Utilisation de la fonction iosDevDelete, avec des paramètres valides.
215  * Resultat attendu
216  * Il ne doit plus être possible d'ouvrir le périphérique : un appel à open sur ce
        périphérique doit
217  * échouer (il doit un nombre négatif), et errno doit être positionné à ENOENT.
218  **/
219 int test_7()
220 {
221         int success = 0;
222         int fd = 0;
223
224         // on n'installe pas de device par défaut.
225
226         pe_driverInstall(0);
227
228         // On utilise la procédure qui encapsule iosDevAdd.
229         if(pe_deviceAdd(0) == OK)
230         {
231                 if(pe_deviceRemove(0) == OK)
232                 {
233                         if(open("/dev/capteur0", O_RDONLY, 0) == ERROR)
234                         {
235                                 if(errnoGet() == ENOENT)
236                                 {
237                                         success++;
238                                 }
239                                 else
240                                 {
241                                         ERR_MSG(7);
242                                 }
243                         }
244                         else
245                         {
246                                 printf("Il␣est␣possible␣d'ouvrir␣le␣device␣alors␣qu'il␣a␣été␣
                                        retiré␣:␣7\n");
247                         }
248                 }
249                 else
250                 {
251                         printf("L'ajout␣a␣échoué␣:␣7\n");
252                 }
253         }
254         else
255         {
256                 printf("Erreur␣a␣l'ajout␣du␣peripherique\n");
257         }
258
259         pe_driverUninstall();
260
261         return success;
262 }
263 /**
264  * Test 8 - Ajout d'un périphérique alors que 15 périphériques ont été ajoutés.
265  * Description
266  * Utilisation de la fonction iosDevAdd sur un périphérique déjà ajouté.
267  * L'appel à iosDevAdd doit provoquer une erreur, et renvoyer ERROR.
268  **/
269 int test_8()
270 {
271         int success = 0;
272
273         if(pe_driverInstall(15) == ERROR)
274         {
275                 return success;
276         }
```

```
277
278         if(pe_deviceAdd(0) == ERROR)
279         {
280                 success++;
281         }
282
283         pe_driverUninstall();
284
285         return success;
286 }
287 /**
288  * Test 9 - Ouverture d'un capteur
289  * Description
290  * Appeler open sur un capteur valide (le fichier existe et est accessible en écriture), avec des options
291  * valide (O_RDONLY), une seule fois.
292  * Resultat attendu
293  * La valeur de retour doit être un entier positif.
294  **/
295 int test_9()
296 {
297         int success = 0;
298         int fd = 0;
299
300         pe_driverInstall(10);
301
302         // Ouvertude d'un capteur.
303         fd = open("/dev/capteur0", O_RDONLY, 0);
304         if( fd == ERROR )
305         {
306                 printf("Erreur d'ouverture du peripherique\n");
307         }
308         else
309         {
310                 success++;
311                 if(close(fd))
312                 {
313                         printf("Erreur pendant la fermeture du capteur\n");
314                 }
315         }
316
317         if(pe_driverUninstall())
318         {
319                 printf("Erreur de désinstallation du driver\n");
320         }
321
322         return success;
323 }
324 /**
325  * Test 10 - Ouverture d'un capteur déjà ouvert
326  * Description
327  * Appeler open sur un capteur valide (le fichier existe, et est accessible en lecture), alors qu'il vient
328  * d'être ouvert avec succès.
329  * Resultat attendu
330  * open doit renvoyer ERROR, et errno doit être positionné à EALREADYOPENED.
331  **/
332 int test_10()
333 {
334         int success = 0;
335         int fd1 = 0;
336         int fd2 = 0;
337
338         pe_driverInstall(10);
339
340         // Ouvertude d'un capteur.
341         fd1 = open("/dev/capteur0", O_RDONLY, 0);
342         if( fd1 == ERROR )
343         {
344                 printf("Erreur d'ouverture du peripherique\n");
345                 if( pe_driverUninstall() == ERROR )
346                 {
347                         printf("Erreur de désinstallation du driver\n");
```

```
348                 }
349                 return success;
350         }
351
352         fd2 = open("/dev/capteur0", O_RDONLY, 0);
353         if( fd2 == ERROR )
354         {
355                 success++;
356         }
357
358         close(fd2);
359
360         if(close(fd1))
361         {
362                 printf("Erreur pendant la fermeture du capteur\n");
363         }
364
365         if(pe_driverUninstall())
366         {
367                 printf("Erreur a l'enlevement du driver\n");
368         }
369
370         return success;
371 }
372 /**
373  * Test 11 - Ouverture d'un capteur avec des paramètres invalides
374  * Description
375  * Appeler open sur un capteur valide (le fichier existe, et est accessible en lecture/
        écriture, en
376  * passant un mode différent de O_RDONLY.
377  * Resultat attendu
378  * .
379  * L'appel doit échouer, et donc renvoyer ERROR. De plus, errno doit être positionné à
        EARG.
380  **/
381 int test_11()
382 {
383         int success = 0;
384         int fd = 0;
385
386         pe_driverInstall(1);
387
388         // Ouvertude d'un capteur.
389         fd = open("/dev/capteur0", O_RDWR, 0);
390         if( fd == ERROR )
391         {
392                 if(errnoGet() == EARG)
393                 {
394                         success++;
395                 }
396                 else
397                 {
398                         printf("Mauvais errno");
399                 }
400         }
401
402         close(fd);
403
404         if(pe_driverUninstall())
405         {
406                 printf("Erreur a l'enlevement du driver\n");
407         }
408
409         return success;
410 }
411 /**
412  * Test 12 - Fermeture d'un capteur
413  * Description
414  * Appeler close sur un descripteur de fichier valide (qui a été ouvert avec succès
        précédemment),
415  * et qui n'a pas été fermé depuis.
416  * Resultat attendu
417  * La valeur de retour doit être égale à OK
```

```
 **/
int test_12()
{
	int success = 0;
	int fd = 0;

	pe_driverInstall(10);

	// Ouvertude d'un capteur.
	fd = open("/dev/capteur0", O_RDONLY, 0);
	if( fd != ERROR )
	{
		if(close(fd) == ERROR)
		{
			printf("Erreur a la fermeture du capteur\n");
		}
		else
		{
			success++;
		}
	}
	else
	{
		printf("Erreur a l'ouverture du capteur\n");
	}

	if(pe_driverUninstall())
	{
		printf("Erreur a l'enlevement du driver\n");
	}

	return success;
}
/**
 * Test 13 - Lecture d'une valeur dans un capteur
 * Description
 * Utiliser read sur un capteur ouvert avec succès avant puis après avoir
 * envoyé un message
 * Resultat attendu
 * La valeur de retour doit être :
 * - ERROR avec errno sur ENOAVAIL
 * - un nombre positif, et doit être cohérente par rapport aux paramètre
 * d'appel de read.
 **/
int test_13()
{
	int success = 0;
	int fd = 0;

	Message buffer;
	char* data = "fake DATA :)\0";
	static int id = -1;
	struct timespec date;
	pe_driverInstall(10);

	// Ouvertude d'un capteur.
	fd = open("/dev/capteur2", O_RDONLY, 0);
	if(fd != ERROR)
	{
		ioctl(fd, SET_CPT_ADDRESS, 42);

		if(read(fd,(char*)&(buffer), sizeof(buffer)) < 0)
		{
			if(errnoGet() == ENOAVAIL)
			{
				success++;
			}
			else
			{
				printf("Erreur : mauvais message d'erreur\n");
			}
		}
		else
```

```
491                 {
492                         printf("Erreur␣:␣lecture␣avec␣succés␣sur␣un␣capteur␣vide\n");
493                 }
494
495                 hardware_mockup(42, data);
496                 id++;
497                 clock_gettime(CLOCK_REALTIME, &(date));
498                 // l'attente est obligatoire sinon le message n'est pas encore traité.
499                 sleep(1);
500                 if(read(fd,(char*)&(buffer), sizeof(buffer)) < 0)
501                 {
502                         printf("Erreur␣a␣la␣lecture␣%d\n", errnoGet());
503                 }
504                 else
505                 {
506                         if(date.tv_sec != buffer.date.tv_sec)
507                         {
508                                 printf("Erreur␣:␣mauvaise␣date␣\n");
509                         }
510                         else
511                         {
512                                 if(strncmp(buffer.msg, data, MAX_DATA_SIZE) == 0)
513                                 {
514                                         success++;
515                                 }
516                                 else
517                                 {
518                                         //printf("Message:\n- id : %d\n- date : %ld:%ld\n-
                                              Message : %s\n", buffer.id, buffer.date.tv_sec,
                                              buffer.date.tv_nsec, buffer.msg);
519
520                                         printf("Erreur␣:␣le␣message␣recus␣n'est␣pas␣le␣
                                              message␣envoye\n");
521                                         //printf("envoye : %s\n", data);
522                                         //printf("recus : %s\n", buffer.msg);
523
524                                 }
525                         }
526                 }
527
528                 if(close(fd) == ERROR)
529                 {
530                         printf("Erreur␣a␣la␣fermeture␣du␣capteur\n");
531                 }
532         }
533         else
534         {
535                 printf("Erreur␣a␣l'ouverture␣du␣capteur\n");
536         }
537
538
539         if(pe_driverUninstall() == ERROR)
540         {
541                 printf("Erreur␣a␣l'enlevement␣du␣driver\n");
542         }
543
544         return (success == 2) ? 1 : 0;
545 }
546 /**
547  * Test 14 - Utilisation de read avec une taille de lecture invalide
548  * Description
549  * Utilisation de l'appel système read avec un descripteur de fichier valide, mais avec
          une taille
550  * de lecture négative.
551  * Resultat attendu
552  * L'appel doit échouer en renvoyant ERROR, et errno doit être positionné à EARG.
553  **/
554 int test_14()
555 {
556         int success = 0;
557         int fd = 0;
558         Message buffer;
559
```

```
560         pe_driverInstall(1);
561
562         // Ouvertude d'un capteur.
563         fd = open("/dev/capteur0", O_RDONLY, 0);
564         if(fd != ERROR)
565         {
566                 hardware_mockup(0, "fake DATA :)\0");
567
568                 if(read(fd, (char*)&(buffer), -42) == ERROR)
569                 {
570                         success++;
571                 }
572
573                 if(close(fd) == ERROR)
574                 {
575                         printf("Erreur a la fermeture du capteur\n");
576                 }
577         }
578         else
579         {
580                 printf("Erreur a l'ouverture du capteur\n");
581         }
582
583
584         if(pe_driverUninstall() == ERROR)
585         {
586                 printf("Erreur a l'enlevement du driver\n");
587         }
588
589         return success;
590 }
591 /**
592  * Test 15 - Utilisation de ioctl avec des paramètres corrects
593  * Description
594  * Utilisation de ioctl avec des paramètres corrects : un descripteur de fichier valide,
        une valeur
595  * pour request égale à la constante SET_CPT_ADDRESS, et une valeur pour value inférieur
        ou
596  * égale à 255, correspondant bien à un capteur valide.
597  * Resultat attendu
598  * La valeur de retour doit être égale à OK, ou alors elle doit être égale à ERROR, mais
        alors errno
599  * doit être positionné à ECPTBUSY, et le même appel effectué ultérieurement doit
        renvoyer OK.
600  **/
601 int test_15()
602 {
603         int success = 0;
604         int fd = 0;
605
606         pe_driverInstall(10);
607
608         // Ouvertude d'un capteur.
609         fd = open("/dev/capteur0", O_RDONLY, 0);
610         if(fd != ERROR)
611         {
612                 if(ioctl(fd, SET_CPT_ADDRESS, 42) == OK)
613                 {
614                         success++;
615                 }
616                 else
617                 {
618                         printf("Erreur lors de l'appel a ioctl\n");
619                 }
620
621                 if(close(fd) == ERROR)
622                 {
623                         printf("Erreur a la fermeture du capteur\n");
624                 }
625         }
626         else
627         {
628                 printf("Erreur a l'ouverture du capteur\n");
```

```
629         }
630
631
632         if(pe_driverUninstall() == ERROR)
633         {
634                 printf("Erreur␣a␣l'enlevement␣du␣driver\n");
635         }
636
637         return success;
638 }
639 /**
640  * Test 16 - Utilisation de ioctl avec de mauvais arguments
641  * Description
642  * Utilisation de la fonction ioctl avec un second paramètre correspondant à une fonction
        nonimplémenté.
643  * Le descripteur de fichier passé en tant que premier paramètre doit être valide, et
644  * correspondre à un périphérique géré par le driver.
645  * Resultat attendu
646  * L'appel doit échouer (la valeur de retour doit être égale à ERROR), et errno doit être
        positionné
647  * à EARG.
648  **/
649 int test_16()
650 {
651         int success = 0;
652         int fd = 0;
653
654         pe_driverInstall(10);
655
656         // Ouvertude d'un capteur.
657         fd = open("/dev/capteur0", O_RDONLY, 0);
658         if(fd != ERROR)
659         {
660                 if(ioctl(fd, 42, 42) == ERROR)
661                 {
662                         if(errnoGet() == EARG)
663                         {
664                                 success++;
665                         }
666                         else
667                         {
668                                 printf("Erreur␣:␣Mauvais␣code␣d'erreur\n");
669                         }
670                 }
671                 else
672                 {
673                         printf("Erreur␣lors␣de␣l'appel␣a␣ioctl\n");
674                 }
675
676                 if(close(fd) == ERROR)
677                 {
678                         printf("Erreur␣a␣la␣fermeture␣du␣capteur\n");
679                 }
680         }
681         else
682         {
683                 printf("Erreur␣a␣l'ouverture␣du␣capteur\n");
684         }
685
686
687         if(pe_driverUninstall() == ERROR)
688         {
689                 printf("Erreur␣a␣l'enlevement␣du␣driver\n");
690         }
691
692         return success;
693 }
694 /**
695  * Test 17 - Utilisation de ioctl pour associer le même capteur à deux descripteur
696  * de fichiers
697  * Description
698  * On utilise la fonction ioctl deux fois, avec des descripteurs de fichier différents,
        mais avec le
```

```
699  * même numéro de capteur.
700  * Resultat attendu
701  * L'appel doit échouer, et renvoyer ERROR. errno doit alors être positionné à
         ECPTALREADYUSED.
702  **/
703 int test_17()
704 {
705         int success = 0;
706         int fd0 = 0;
707         int fd1 = 0;
708 
709         pe_driverInstall(10);
710 
711         // Ouvertude d'un capteur.
712         fd0 = open("/dev/capteur0", O_RDONLY, 0);
713         fd1 = open("/dev/capteur1", O_RDONLY, 0);
714         if(fd0 != ERROR && fd1 != ERROR)
715         {
716                 if(ioctl(fd0, SET_CPT_ADDRESS, 42) == OK)
717                 {
718                         if(ioctl(fd0, SET_CPT_ADDRESS, 42) == ERROR)
719                         {
720                                 if(errnoGet() == ECPTALREADYUSED)
721                                 {
722                                         success++;
723                                 }
724                                 else
725                                 {
726                                         printf("Erreur : Mauvais code d'erreur\n");
727                                 }
728                         }
729                         else
730                         {
731                                 printf("Erreur : l'appel a ioctl n'a pas echoue, il existe
                                     deux capteur pointant vers la même adresse CAN\n");
732                         }
733                 }
734                 else
735                 {
736                         printf("Erreur lors de l'appel a ioctl sur le premier capteur\n");
737                 }
738 
739                 if(close(fd0) == ERROR || close(fd1) == ERROR)
740                 {
741                         printf("Erreur a la fermeture d'un des capteurs\n");
742                 }
743         }
744         else
745         {
746                 printf("Erreur a l'ouverture d'un des capteurs\n");
747         }
748 
749 
750         if(pe_driverUninstall() == ERROR)
751         {
752                 printf("Erreur a l'enlevement du driver\n");
753         }
754 
755         return success;
756 }
757 /**
758  * Test 18 - Utilisation de write
759  * Description
760  * Appel de write sur un capteur ouvert avec succès.
761  * Resultat attendu
762  * L'appel doit échouer, et errno doit être positionné à ENSUP.
763  **/
764 int test_18()
765 {
766         int success = 0;
767         int fd = 0;
768         Message buffer;
769 
```

```
770         pe_driverInstall(1);
771
772         // Ouvertude d'un capteur.
773         fd = open("/dev/capteur0", O_RDONLY, 0);
774         if(fd != ERROR)
775         {
776                 if(write(fd,(char*)&(buffer), sizeof(Message)))
777                 {
778                         if(errnoGet() == ENOTSUP)
779                         {
780                                 success++;
781                         }
782                         else
783                         {
784                                 printf("Mauvais numero d'erreur\n");
785                         }
786                 }
787                 else
788                 {
789                         printf("Erreur : lecture avec succes");
790                 }
791
792                 if(close(fd) == ERROR)
793                 {
794                         printf("Erreur a la fermeture du capteur\n");
795                 }
796         }
797         else
798         {
799                 printf("Erreur a l'ouverture du capteur\n");
800         }
801
802
803         if(pe_driverUninstall() == ERROR)
804         {
805                 printf("Erreur a l'enlevement du driver\n");
806         }
807
808         return success;
809 }
810
811 /**
812  * Test 19 - Retrait d'un périphérique ouvert
813  * Description
814  * Tentative de retrait d'un périphérique sur lequel un appel de open a été
815  * effectué avec succès précédemment, et sur lequel on n'a pas fait d'appel
816  * à close ou remove.
817  * Resultat attendu
818  * L'appel doit réussir : la valeur de retour doit être égale à OK.
819  **/
820 int test_19()
821 {
822         int success = 0;
823         int fd = 0;
824         pe_driverInstall(15);
825         fd = creat("/dev/capteur0", O_RDONLY);
826
827         if(pe_deviceRemove(0) == OK)
828         {
829                 success++;
830         }
831         else
832         {
833                 ERR_MSG(19);
834         }
835         pe_driverUninstall();
836         return success;
837 }
838 /**
839  * Test 20 - Retrait d'un périphérique qui n'existe pas
840  * Description
841  * Tentative de retrait d'un périphérique qui n'existe pas.
842  * Resultat attendu
```

```
843  * L'appel doit échouer, et retourner ERROR. errno doit être positionné à
844  * ENEXIST.
845  **/
846 int test_20()
847 {
848         int success = 0;
849         int fd = 0;
850         pe_driverInstall(15);
851         // Le capteur 67 n'existe pas.
852         if(pe_deviceRemove(67) == ERROR && errnoGet() == ENEXIST)
853         {
854                 success++;
855         }
856         else
857         {
858                 ERR_MSG(20);
859         }
860         pe_driverUninstall();
861         return success;
862 }
863 void run_suite()
864 {
865         int count = 0;
866         test_1() ? count++ : printf("test_1 : ERROR\n");
867         test_2() ? count++ : printf("test_2 : ERROR\n");
868         test_3() ? count++ : printf("test_3 : ERROR\n");
869         test_4() ? count++ : printf("test_4 : ERROR\n");
870         test_5() ? count++ : printf("test_5 : ERROR\n");
871         test_6() ? count++ : printf("test_6 : ERROR\n");
872         test_7() ? count++ : printf("test_7 : ERROR\n");
873         test_8() ? count++ : printf("test_8 : ERROR\n");
874         test_9() ? count++ : printf("test_9 : ERROR\n");
875         test_10() ? count++ : printf("test_10 : ERROR\n");
876         test_11() ? count++ : printf("test_11 : ERROR\n");
877         test_12() ? count++ : printf("test_12 : ERROR\n");
878         test_13() ? count++ : printf("test_13 : ERROR\n");
879         test_14() ? count++ : printf("test_14 : ERROR\n");
880         test_15() ? count++ : printf("test_15 : ERROR\n");
881         test_16() ? count++ : printf("test_16 : ERROR\n");
882         test_17() ? count++ : printf("test_17 : ERROR\n");
883         test_18() ? count++ : printf("test_18 : ERROR\n");
884         test_19() ? count++ : printf("test_19 : ERROR\n");
885         test_20() ? count++ : printf("test_20 : ERROR\n");
886
887
888         if(count == 20)
889         {
890                 printf("All tests passed successfully.\n");
891         }
892         else
893         {
894                 printf("%d/20 tests passed.\n", count);
895         }
896 }
```