Le Linux Serial Programming HOWTO

par Peter H. Baumann, *Peter.Baumann@dlr.de*
Adaptation française Etienne BERNARD *eb@via.ecp.fr*

v1.0, 22 janvier 1998

Ce document décrit comment programmer sous Linux la communication avec des périphériques sur port série.

Contents

1 Introduction

Voici le Linux Serial Programming HOWTO, qui explique comment programmer sous Linux la communication avec des périphériques ou des ordinateurs via le port série. Différentes techniques sont abordées : Entrées/Sorties canoniques (envoi ou réception ligne par ligne), asynchrones, ou l'attente de données depuis de multiples sources.

Ce document ne décrit pas comment configurer les ports séries, puisque c'est décrit par Greg Hankins dans le Serial-HOWTO.

Je tiens à insister sur le fait que je ne suis pas un expert dans ce domaine, mais j'ai eu à réaliser un projet utilisant la communication par le port série. Les exemples de code source présentés dans ce document sont dérivés du programme `miniterm`, disponible dans le *Linux programmer's guide* (`ftp://sunsite.unc.edu/pub/Linux/docs/LDP/programmers-guide/lpg-0.4.tar.gz` et les miroirs, par exemple `ftp://ftp.lip6.fr/pub/linux/docs/LDP/programmers-guide/lpg-0.4.tar.gz`) dans le répertoire contenant les exemples.

Depuis la dernière version de ce document, en juin 1997, j'ai dû installer Windows NT pour satisfaire les besoins des client, et donc je n'ai pas pu investiguer plus en avant sujet. Si quelqu'un a des commentaires à me faire, je me ferai un plaisir de les inclure dans ce document (voyez la section sur les commentaires). Si vous désirez prendre en main l'évolution de ce document, et l'améliorer, envoyez moi un courrier électronique.

Tous les exemples ont été testés avec un i386, utilisant un noyau Linux de version 2.0.29.

1.1 Copyright

1.2 Nouvelles versions de ce document.

Les nouvelles version du Serial-Programming-HOWTO seront disponibles à *ftp://sunsite.unc.edu:/pub/Linux/docs/HOWTO/Serial-Programming-HOWTO* et les sites miroir, comme par exemple *ftp://ftp.lip6.fr/pub/linux/docs/HOWTO/Serial-Programming-HOWTO* . Il existe sous d'autres formats, comme PostScript ou DVI dans le sous répertoire `other-formats`. Le Serial-Programming-HOWTO est également disponible sur *http://sunsite.unc.edu/LDP/HOWTO/Serial-Programming-HOWTO.html* , et sera posté dans *comp.os.linux.answers* tous les mois (NdT : la version française de ce document est également postée dans *fr.comp.os.linux.annonce* tous les mois).

1.3 Commentaires

Envoyez moi, s'il vous plaît toute correction, question, commentaire, suggestion ou complément. Je désire améliorer cet HOWTO ! Dites moi exactement ce que vous ne comprenez pas, ou ce qui pourrait être plus clair. Vous pouvez me contacter à *Peter.Baumann@dlr.de* par courrier électronique. Veuillez inclure le numéro de version de ce document pour tout courrier. Ce document est la version 0.3.

2 Démarrage

2.1 Débuggage

Le meilleur moyen de débugguer votre code est d'installer une autre machine sous Linux et de connecter les deux ordinateurs par un câble null-modem. Utilisez `miniterm` (disponible dans le Linux programmers guide – `ftp://sunsite.unc.edu/pub/Linux/docs/LDP/programmers-guide/lpg-0.4.tar.gz` – dans le répertoire des exemples) pour transmettre des caractères à votre machine Linux. `Miniterm` est très simple à compiler et transmettra toute entrée clavier directement sur le port série. Vous n'avez qu'à adapter la commande `#define MODEMDEVICE "/dev/ttyS0"` à vos besoins. Mettez `ttyS0` pour COM1, `ttyS1` for COM2, etc... Il est essentiel, pour les tests, que *tous* les caractères soient transmis bruts (sans traitements) au travers de la ligne série. Pour tester votre connexion, démarrez `miniterm` sur les deux ordinateurs et taper au clavier. Les caractères écrit sur un ordinateur devraient apparaître sur l'autre ordinateur, et vice versa. L'entrée clavier sera également recopiée sur l'écran de l'ordinateur local.

Pour fabriquer un câble null-modem, pour devez croiser les lignes TxD (*transmit*) et RxD (*receive*). Pour une description du câble, référez vous à la section 7 du Serial-HOWTO.

Il est également possible de faire cet essai avec uniquement un seul ordinateur, si vous disposez de deux ports série. Vous pouvez lancez deux `miniterms` sur deux consoles virtuelles. Si vous libérez un port série en déconnectant la souris, n'oubliez pas de rediriger `/dev/mouse` si ce fichier existe. Si vous utilisez une carte série à ports multiples, soyez sûr de la configurer correctement. La mienne n'était pas correctement configurée, et tout fonctionnait correctement lorsque je testais sur mon ordinateur. Lorsque je l'ai connecté à un autre, le port a commencé à perdre des caractères. L'exécution de deux programmes sur un seul ordinateur n'est pas totalement asynchrone.

2.2 Configuration des ports

Les périphériques `/dev/ttyS*` sont destinés à connecter les terminaux à votre machine Linux, et sont configurés pour cet usage après le démarrage. Vous devez vous en souvenir lorsque vous programmez la communication avec un périphérique autre. Par exemple, les ports sont configurés pour afficher les caractères envoyés vers lui-même, ce qui normalement doit être changé pour la transmission de données.

Tous les paramètres peuvent être facilement configuré depuis un programme. La configuration est stockée dans une structure de type `struct termios`, qui est définie dans `<asm/termbits.h>` :

```
#define NCCS 19
struct termios {
        tcflag_t c_iflag;               /* Modes d'entrée */
        tcflag_t c_oflag;               /* Modes de sortie */
        tcflag_t c_cflag;               /* Modes de contrôle */
        tcflag_t c_lflag;               /* Modes locaux */
        cc_t c_line;                    /* Discipline de ligne */
        cc_t c_cc[NCCS];                /* Caractères de contrôle */
};
```

Ce fichier inclus également la définition des constantes. Tous les modes d'entrée dans `c_iflag` prennent en charge le traitement de l'entrée, ce qui signifie que les caractères envoyés depuis le périphérique peuvent être traités avant d'être lu par `read`. De la même façon, `c_oflags` se chargent du traitement en sortie. `c_cflag` contient les paramètres du port, comme la vitesse, le nombre de bits par caractère, les bits d'arrêt, etc... Les modes locaux, stockés dans `c_lflag` déterminent si les caractères sont imprimés, si des signaux sont envoyés à votre programme, etc... Enfin, le tableau `c_cc` définit les caractères de contrôle pour la fin de fichier, le caractère stop, etc... Les valeurs par défaut pour les caractères de contrôle sont définies dans

<asm/termios.h>. Les modes possibles sont décrits dans la page de manuel de `termios(3)`. La structure `termios` contient un champ `c_line` (discipline de ligne), qui n'est pas utilisé sur les systèmes conformes à POSIX.

2.3 Façons de lire sur les périphériques série

Voici trois façons de lire sur les périphériques série. Le moyen approprié doit être choisi pour chaque application. Lorsque cela est possible, ne lisez pas les chaînes caractère par caractère. Lorsque j'utilisais ce moyen, je perdais des caractères, alors qu'un `read` sur la chaîne complète ne donnait aucune erreur.

2.3.1 Entrée canonique

C'est le mode de fonctionnement normal pour les terminaux, mais peut également être utilisé pour communiquer avec d'autres périphériques. Toutes les entrées sont traitées lignes par lignes, ce qui signifie qu'un `read` ne renverra qu'une ligne complète. Une ligne est terminée par défaut avec un caractère `NL` (ACII `LF`), une fin de fichier, ou un caractère de fin de ligne. Un `CR` (le caractère de fin de ligne par défaut de DOS et Windows) ne terminera pas une ligne, avec les paramètres par défaut.

L'entrée canonique peut également prendre en charge le caractère erase, d'effacement de mot, et de réaffichage, la traduction de `CR` vers `NL`, etc...

2.3.2 Entrée non canonique

L'entrée non canonique va prendre en charge un nombre fixé de caractère par lecture, et autorise l'utilisation d'un compteur de temps pour les caractères. Ce mode doit être utilisé si votre application lira toujours un nombre fixé de caractères, ou si le périphérique connecté envoit les caractères par paquet.

2.3.3 Entrée asynchrone

Les deux modes ci-dessus peut être utilisé en mode synchrone ou asynchrone. Le mode synchrone est le mode par défaut, pour lequel un appel à `read` sera bloquant, jusqu'à ce que la lecture soit satisfaite. En mode asynchrone, un appel à `read` retournera immédiatement et lancera un signal au programme appelant en fin de transfert. Ce signal peut être reçu par un gestionnaire de signal.

2.3.4 Attente d'entrée depuis de multiples sources

Cela ne constitue pas un mode d'entrée différent, mais peut s'avérer être utile, si vous prenez en charge des périphériques multiples. Dans mon application, je traitais l'entrée depuis une socket TCP/IP et depuis une connexion série sur un autre ordinateur quasiment en même temps. L'exemple de programme donné plus loin attendra des caractères en entrée depuis deux sources. Si des données sur l'une des sources deviennent disponibles, elles seront traitées, et le programme attendra de nouvelles données.

L'approche présentée plus loin semble plutôt complexe, mais il est important que vous vous rappeliez que Linux est un système multi-tâche. L'appel système `select` ne charge pas le processeur lorsqu'il attend des données, alors que le fait de faire une boucle jusqu'à ce que des caractères deviennent disponibles ralentirait les autres processus.

3 Exemples de programmes

Tous les exemples ont été extraits de `miniterm.c`. Le tampon d'entrée est limité à 255 caractères, tout comme l'est la longueur maximale d'une ligne en mode canonique (`<linux/limits.h>` ou `<posix1_lim.h>`).

Référez-vous aux commentaires dans le code source pour l'explication des différents modes d'entrée. J'espère que le code est compréhensible. L'exemple sur l'entrée canonique est la plus commentée, les autres exemples sont commentés uniquement lorsqu'ils diffèrent, afin de signaler les différences.

Les descriptions ne sont pas complètes, mais je vous encourage à modifier les exemples pour obtenir la solution la plus intéressante pour votre application.

N'oubliez pas de donner les droits corrects aux ports série (par exemple, `chmod a+rw /dev/ttyS1`) !

3.1 Traitement canonique

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

/* les valeurs pour la vitesse, baudrate, sont définies dans <asm/termbits.h>, qui est inclus
dans <termios.h> */
#define BAUDRATE B38400
/* changez cette définition pour utiliser le port correct */
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* code source conforme à POSIX */

#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

main()
{
  int fd,c, res;
  struct termios oldtio,newtio;
  char buf[255];
/*
  On ouvre le périphérique du modem en lecture/écriture, et pas comme
  terminal de contrôle, puisque nous ne voulons pas être terminé si l'on
  reçoit un caractère CTRL-C.
*/
 fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
 if (fd <0) {perror(MODEMDEVICE); exit(-1); }

 tcgetattr(fd,&oldtio); /* sauvegarde de la configuration courante */
 bzero(&newtio, sizeof(newtio)); /* on initialise la structure à zéro */

/*
  BAUDRATE: Affecte la vitesse. vous pouvez également utiliser cfsetispeed
            et cfsetospeed.
  CRTSCTS : contrôle de flux matériel (uniquement utilisé si le câble a
            les lignes nécessaires. Voir la section 7 du Serial-HOWTO).
```

```
   CS8     : 8n1 (8 bits,sans parité, 1 bit d'arrêt)
   CLOCAL  : connexion locale, pas de contrôle par le modem
   CREAD   : permet la réception des caractères
 */
  newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;

 /*
   IGNPAR  : ignore les octets ayant une erreur de parité.
   ICRNL   : transforme CR en NL (sinon un CR de l'autre côté de la ligne
             ne terminera pas l'entrée).
   sinon, utiliser l'entrée sans traitement (device en mode raw).
 */
  newtio.c_iflag = IGNPAR | ICRNL;

 /*
  Sortie sans traitement (raw).
 */
  newtio.c_oflag = 0;

 /*
   ICANON  : active l'entrée en mode canonique
   désactive toute fonctionnalité d'echo, et n'envoit pas de signal au
   programme appelant.
 */
  newtio.c_lflag = ICANON;

 /*
   initialise les caractères de contrôle.
   les valeurs par défaut peuvent être trouvées dans
   /usr/include/termios.h, et sont données dans les commentaires.
   Elles sont inutiles ici.
 */
  newtio.c_cc[VINTR]    = 0;     /* Ctrl-c */
  newtio.c_cc[VQUIT]    = 0;     /* Ctrl-\ */
  newtio.c_cc[VERASE]   = 0;     /* del */
  newtio.c_cc[VKILL]    = 0;     /* @ */
  newtio.c_cc[VEOF]     = 4;     /* Ctrl-d */
  newtio.c_cc[VTIME]    = 0;     /* compteur inter-caractère non utilisé */
  newtio.c_cc[VMIN]     = 1;     /* read bloquant jusqu'à l'arrivée d'1 caractère */
  newtio.c_cc[VSWTC]    = 0;     /* '\0' */
  newtio.c_cc[VSTART]   = 0;     /* Ctrl-q */
  newtio.c_cc[VSTOP]    = 0;     /* Ctrl-s */
  newtio.c_cc[VSUSP]    = 0;     /* Ctrl-z */
  newtio.c_cc[VEOL]     = 0;     /* '\0' */
  newtio.c_cc[VREPRINT] = 0;     /* Ctrl-r */
  newtio.c_cc[VDISCARD] = 0;     /* Ctrl-u */
  newtio.c_cc[VWERASE]  = 0;     /* Ctrl-w */
  newtio.c_cc[VLNEXT]   = 0;     /* Ctrl-v */
  newtio.c_cc[VEOL2]    = 0;     /* '\0' */

 /*
   à présent, on vide la ligne du modem, et on active la configuration
   pour le port
 */
  tcflush(fd, TCIFLUSH);
```

```
 tcsetattr(fd,TCSANOW,&newtio);

/*
  la configuration du terminal est faite, à présent on traite
  les entrées
  Dans cet exemple, la réception d'un 'z' en début de ligne mettra
  fin au programme.
*/
 while (STOP==FALSE) {     /* boucle jusqu'à condition de terminaison */
 /* read bloque l'exécution du programme jusqu'à ce qu'un caractère de
    fin de ligne soit lu, même si plus de 255 caractères sont saisis.
    Si le nombre de caractères lus est inférieur au nombre de caractères
    disponibles, des read suivant retourneront les caractères restants.
    res sera positionné au nombre de caractères effectivement lus */
    res = read(fd,buf,255);
    buf[res]=0;       /* on termine la ligne, pour pouvoir l'afficher */
    printf(":%s:%d\n", buf, res);
    if (buf[0]=='z') STOP=TRUE;
 }
 /* restaure les anciens paramètres du port */
 tcsetattr(fd,TCSANOW,&oldtio);
}
```

3.2 Entrée non canonique

Dans le mode non canonique, les caractères lus ne sont pas assemblés ligne par ligne, et ils ne subissent pas de traitement (erase, kill, delete, etc...). Deux paramètres contrôlent ce mode : `c_cc[VTIME]` positionne le *timer* de caractères, et `c_cc[VMIN]` indique le nombre minimum de caractères à recevoir avant qu'une lecture soit satisfaite.

Si MIN $>$ 0 et TIME $=$ 0, MIN indique le nombre de caractères à recevoir avant que la lecture soit satisfaite. TIME est égal à zéro, et le *timer* n'est pas utilisé.

Si MIN $=$ 0 et TIME $>$ 0, TIME est utilisé comme une valeur de *timeout*. Une lecture est satisfaite lorsqu'un caractère est reçu, ou que la durée TIME est dépassée (t = TIME * 0.1s). Si TIME est dépassé, aucun caractère n'est retourné.

Si MIN $>$ 0 et TIME $>$ 0, TIME est employé comme *timer* entre chaque caractère. La lecture sera satisfaite si MIN caractères sont reçus, ou que le *timer* entre deux caractères dépasse TIME. Le *timer* est réinitialisé à chaque fois qu'un caractère est reçu, et n'est activé qu'après la réception du premier caractère.

Si MIN $=$ 0 et TIME $=$ 0, le retour est immédiat. Le nombre de caractères disponibles, ou bien le nombre de caractères demandé est retourné. Selon Antonino (voir le paragraphe sur les participations), vous pouvez utiliser un `fcntl(fd, F_SETFL, FNDELAY);` avant la lecture pour obtenir le même résultat.

Vous pouvez tester tous les modes décrit ci-dessus en modifiant `newtio.c_cc[VTIME]` et `newtio.c_cc[VMIN]`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* code source conforme à POSIX */
```

```
#define FALSE 0
#define TRUE 1

volatile int STOP=FALSE;

main()
{
  int fd,c, res;
  struct termios oldtio,newtio;
  char buf[255];

 fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY );
 if (fd <0) {perror(MODEMDEVICE); exit(-1); }

 tcgetattr(fd,&oldtio); /* sauvegarde de la configuration courante */

 bzero(&newtio, sizeof(newtio));
 newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
 newtio.c_iflag = IGNPAR;
 newtio.c_oflag = 0;

 /* positionne le mode de lecture (non canonique, sans echo, ...) */
 newtio.c_lflag = 0;

 newtio.c_cc[VTIME]    = 0;   /* timer inter-caractères non utilisé */
 newtio.c_cc[VMIN]     = 5;   /* read bloquant jusqu'à ce que 5 */
                              /* caractères soient lus */
 tcflush(fd, TCIFLUSH);
 tcsetattr(fd,TCSANOW,&newtio);


 while (STOP==FALSE) {       /* boucle de lecture */
   res = read(fd,buf,255);   /* retourne après lecture 5 caractères */
   buf[res]=0;               /* pour pouvoir les imprimer... */
   printf(":%s:%d\n", buf, res);
   if (buf[0]=='z') STOP=TRUE;
 }
 tcsetattr(fd,TCSANOW,&oldtio);
}
```

3.3 Lecture asynchrone

```
#include <termios.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/signal.h>
#include <sys/types.h>

#define BAUDRATE B38400
#define MODEMDEVICE "/dev/ttyS1"
#define _POSIX_SOURCE 1 /* code source conforme à POSIX */
#define FALSE 0
#define TRUE 1
```

```
volatile int STOP=FALSE;

void signal_handler_IO (int status);   /* le gestionnaire de signal */
int wait_flag=TRUE;              /* TRUE tant que reçu aucun signal */

main()
{
  int fd,c, res;
  struct termios oldtio,newtio;
  struct sigaction saio;        /* définition de l'action du signal */
  char buf[255];

  /* ouvre le port en mon non-bloquant (read retourne immédiatement) */
  fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY | O_NONBLOCK);
  if (fd <0) {perror(MODEMDEVICE); exit(-1); }

  /* installe le gestionnaire de signal avant de passer le port en
     mode asynchrone */
  saio.sa_handler = signal_handler_IO;
  saio.sa_mask = 0;
  saio.sa_flags = 0;
  saio.sa_restorer = NULL;
  sigaction(SIGIO,&saio,NULL);

  /* permet au processus de recevoir un SIGIO */
  fcntl(fd, F_SETOWN, getpid());
  /* rend le descripteur de fichier asynchrone (la page de manuel
     indique que seuls O_APPEND et O_NONBLOCK fonctionnent avec
     F_SETFL...) */
  fcntl(fd, F_SETFL, FASYNC);

  tcgetattr(fd,&oldtio); /* sauvegarde de la configuration courante */
  /* positionne les nouvelles valeurs pour lecture canonique */
  newtio.c_cflag = BAUDRATE | CRTSCTS | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR | ICRNL;
  newtio.c_oflag = 0;
  newtio.c_lflag = ICANON;
  newtio.c_cc[VMIN]=1;
  newtio.c_cc[VTIME]=0;
  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);

  /* on boucle en attente de lecture. généralement, on réalise
     des traitements à l'intérieur de la boucle */
  while (STOP==FALSE) {
    printf(".\n");usleep(100000);
    /* wait_flag = FALSE après réception de SIGIO. Des données sont
       disponibles et peuvent être lues */
    if (wait_flag==FALSE) {
      res = read(fd,buf,255);
      buf[res]=0;
      printf(":%s:%d\n", buf, res);
      if (res==1) STOP=TRUE; /* on arrête la boucle si on lit une
                                ligne seule */
```

```
      wait_flag = TRUE;      /* on attend de nouvelles données */
    }
  }
  /* restaure les anciens paramètres du port */
  tcsetattr(fd,TCSANOW,&oldtio);
}

/***************************************************************************
* gestionnaire de signal. Positionne wait_flag à FALSE, pour indiquer à    *
* la boucle ci-dessus que des caractères ont été reçus.                    *
***************************************************************************/

void signal_handler_IO (int status)
{
  printf("réception du signal SIGIO.\n);
  wait_flag = FALSE;
}
```

3.4 Multiplexage en lecture

Cette section est réduite au minimum, et n'est là que pour vous guider. Le code source d'exemple présenté est donc réduit au strict minimum. Il ne fonctionnera pas seulement avec des ports série, mais avec n'importe quel ensemble de descripteurs de fichiers.

L'appel système select et les macros qui lui sont attachées utilisent un `fd_set`. C'est un tableau de bits, qui dispose d'un bit pour chaque descripteur de fichier valide. `select` accepte un `fd_set` ayant les bits positionnés pour les descripteurs de fichiers qui conviennent, et retourne un `fd_set`, dans lequel les bits des descripteurs de fichier où une lecture, une écriture ou une exception sont positionnés. Toutes les manipulations de `fd_set` sont faites avec les macros fournies. Reportez vous également à la page de manuel de `select(2)`.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
   int    fd1, fd2;  /* entrées 1 et 2 */
   fd_set readfs;    /* ensemble de descripteurs */
   int    maxfd;     /* nombre max des descripteurs utilisés */
   int    loop=1;    /* boucle tant que TRUE */

   /* open_input_source ouvre un périphérique, configure le port
      correctement, et retourne un descripteur de fichier. */
   fd1 = open_input_source("/dev/ttyS1");   /* COM2 */
   if (fd1<0) exit(0);
   fd2 = open_input_source("/dev/ttyS2");   /* COM3 */
   if (fd2<0) exit(0);
   maxfd = MAX (fd1, fd2)+1; /* numéro maximum du bit à tester */

   /* boucle d'entrée */
   while (loop) {
     FD_SET(fd1, &readfs);  /* test pour la source 1 */
     FD_SET(fd2, &readfs);  /* test pour la source 2 */
```

```
    /* on bloque jusqu'à ce que des caractères soient
       disponibles en lecture */
    select(maxfd, &readfs, NULL, NULL, NULL);
    if (FD_ISSET(fd1))         /* caractères sur 1 */
      handle_input_from_source1();
    if (FD_ISSET(fd2))         /* caractères sur 2 */
      handle_input_from_source2();
  }

}
```

L'exemple ci-dessus bloque indéfiniment, jusqu'à ce que des caractères venant d'une des sources soient disponibles. Si vous avez besoin d'un *timeout*, remplacez juste l'appel à select par :

```
int res;
struct timeval Timeout;

/* fixe la valeur du timeout */
Timeout.tv_usec = 0;  /* millisecondes */
Timeout.tv_sec  = 1;  /* secondes */
res = select(maxfd, &readfs, NULL, NULL, &Timeout);
if (res==0)
/* nombre de descripteurs de fichiers avec caractères
   disponibles = 0, il y a eu timeout */
```

Cet exemple verra l'expiration du delai de *timeout* après une seconde. S'il y a *timeout*, select retournera 0, mais faîtes attention, `Timeout` est décrémenté du temps réellement attendu par `select`. Si la valeur de *timeout* est 0, select retournera immédiatement.

4 Autres sources d'information

- Le Linux Serial-HOWTO décrit comment mettre en place les ports série et contient des informations sur le matériel.
- Le *Serial Programming Guide for POSIX Compliant Operating Systems* `http://www.easysw.com/~mike/serial` , par Michael Sweet. Ce lien est périmé et je n'arrive pas à trouver la nouvelle adresse du document. Quelqu'un sait-il où je peux le retrouver ? C'était un très bon document !
- La page de manuel de `termios(3)` décrit toutes les constantes utilisées pour la structure `termios`.

5 Contributions

Comme je l'ai dit dans l'introduction, je ne suis pas un expert dans le domaine, mais j'ai rencontré des problèmes, et j'ai trouvé les solutions avec l'aide d'autres personnes. Je tiens à remercier pour leur aide M. Strudthoff du European Transonic WindTunnel, Cologne, Michael Carter (`mcarter@rocke.electro.swri.edu`) et Peter Waltenberg (`p.waltenberg@karaka.chch.cri.nz`).

Antonino Ianella (`antonino@usa.net` a écrit le Serial-Port-Programming Mini HOWTO, au même moment où je préparais ce document. Greg Hankins m'a demandé d'inclure le Mini-HOWTO d'Antonino dans ce document.

La structure de ce document et le formattage SGML ont été dérivés du Serial-HOWTO de Greg Hankins. Merci également pour diverses corrections faites par : Dave Pfaltzgraff (`Dave_Pfaltzgraff@patapsco.com`),

Sean Lincolne (`slincol@tpgi.com.au`), Michael Wiedmann (`mw@miwie.in-berlin.de`), et Adrey Bonar (`andy@tipas.lt`).