Petit guide de programmation des ports d’entrées / sorties sous Linux

Adaptation française du Linux I/O port programming mini-HOWTO

Riku Saikkonen

<Riku POINT Saikkonen CHEZ hut POINT fi>

Adaptation française: Jean-François Prévost

Relecture de la version française: Guillaume Lelarge, Jean-Philippe Guérard

Version : 3.0.fr.1.1

24 avril 2006

Historique des versions		
Version 3.0.fr.1.1	2006-04-24	JPG
Conversion en XML. Mise à jour de l’en-tête du document pour respecter les conventions actuelles du projet Traduc.org.		
Version 3.0.fr.1.0	2003-02-04	JFP, GL, JPG
Première traduction française.		
Version 3.0	2000-12-13	RS

Résumé

Ce document présente les différentes façons de programmer des entrées / sorties pour les architectures Intel x86 ainsi que de les différentes méthodes permettant l’utilisation de temporisations très courtes pour les applications Linux tournant en mode utilisateur.

Table des matières

1. Introduction

Ce document traite de la façon de programmer des entrées / sorties matérielles sur une architecture Intel x86 ainsi que de l'utilisation de temporisations très courtes pour des applications s'exécutant en mode utilisateur sous Linux. Ce document est un descendant du très court Petit guide des ports d'entrées-sorties (*IO-Port mini-HOWTO*) du même auteur.

1.1. Droits d'utilisation

1.2. Commentaires et corrections

Si vous avez des remarques ou des corrections, n'hésitez pas à m'écrire en anglais à l'adresse <`Riku POINT Saikkonen CHEZ hut POINT fi`> ...

N'hésitez pas à faire parvenir vos commentaires et suggestions concernant l'adaptation française de ce document au projet Traduc.org à l'adresse : <`commentaires CHEZ traduc POINT org`>.

2. Utilisation des ports d'entrées / sorties en langage C

2.1. La méthode normale

Les routines pour accéder aux ports d'entrées / sorties sont situées dans le fichier d'en-tête `/usr/include/asm/io.h` (ou `linux/include/asm-i386/io.h` dans les sources du noyau Linux). Ces routines sont des macros, il suffit donc de déclarer `#include <asm/io.h>`; dans votre code source sans avoir besoin de bibliothèques additionnelles.

À cause d'une limitation de gcc (présente dans toutes les versions que je connais, egcs y compris) vous *devez* compiler le code source qui fait appel à ces routines avec le drapeau d'optimisation (**gcc -O1** ou plus), ou alternativement en déclarant `#define extern static` avant la ligne `#include <asm/io.h>` (n'oubliez pas de rajouter ensuite `#undef extern`).

Pour le débogage, vous pouvez compiler avec les drapeaux suivants (tout du moins avec les versions les plus récentes de gcc) : **gcc -g -O**. Il faut savoir que l'optimisation engendre un comportement parfois bizarre de la part du débogueur. Si cela vous pose un réel problème, vous pouvez toujours utiliser les routines d'accès aux ports d'entrées / sorties dans un fichier source séparé, et ne compiler que ce fichier avec le drapeau d'optimisation activé.

2.1.1. Les permissions

Avant d'accéder aux ports, vous devez donner à votre programme la permission de le faire. Pour cela, il vous faut faire appel à la fonction `ioperm()` (déclarée dans `unistd.h` et définie dans le noyau) quelque part au début de votre programme (avant tout accès aux ports d'entrées / sorties). La syntaxe est la suivante : `ioperm(`*`premier_port`*`, , `*`nombre`*`, , `*`activer`*`)`, où *`premier_port`* est le numéro du premier port auquel on souhaite avoir accès et *`nombre`* le nombre de ports consécutifs auxquels on veut avoir la permission d'accéder. Par exemple, `ioperm(0x300, 5, 1)` donnerait accès aux ports `0x300` jusqu'à `0x304` (au total 5 ports). Le dernier argument est une valeur booléenne spécifiant si on autorise l'accès aux ports (vrai [1]) ou si on le restreint (faux [0]). Pour activer l’accès à plusieurs ports non consécutifs, vous pouvez faire plusieurs appels à `ioperm()`. Reportez vous à la page de manuel ioperm(2) pour plus de détails sur la syntaxe.

L’appel à `ioperm()` dans votre programme nécessite les privilèges de super utilisateur (root). Il faut donc que votre programme soit exécuté en tant qu’utilisateur root, ou qu’il soit rendu setuid root. Vous pouvez abandonner les privilèges d’utilisateur root après l’appel à `ioperm()`. Il n’est pas impératif d’abandonner de façon explicite les privilèges d’accès aux ports en utilisant `ioperm( ... , 0 )` à la fin de votre programme, ceci est fait automatiquement lorsque le processus se termine.

L’utilisation de `setuid()` par un utilisateur non privilégié ne supprime pas l’accès accordé aux ports par `ioperm()`. En revanche, lors d’un `fork()`, le processus fils n’hérite pas des permissions de son père (qui lui les garde).

La fonction `ioperm()` permet de contrôler l’accès aux ports de `0x000` à `0x3ff` uniquement. Pour les ports supérieurs, vous devez utiliser `iopl()` qui ouvre un accès à tous les ports d’un coup. Pour donner à votre programme l’accès à *tous* les ports d’entrées / sorties (soyez certains de ce que vous faites car l’accès à des ports inappropriés peut avoir des conséquences désastreuses pour votre système), il suffit de passer à la fonction un argument de valeur `3` (`iopl(3)`). Reportez-vous à la page de manuel iopl(2) pour plus de détails.

2.1.2. L’accès aux ports

Pour lire un octet (8 bits) sur un port, un appel à `inb(`*`port`*`)` retourne la valeur de l’octet lu. Pour l’écriture d’un octet, il suffit d’appeler la fonction `outb(`*`valeur, , port`*`)` (attention à l’ordre des paramètres). La lecture d’un mot (16 bits) sur les ports *x* et *x*+1 (un octet sur chaque port pour constituer un mot grâce à l’instruction assembleur `inw`), faites appel à `inw(`*`x`*`)`. Enfin, pour l’écriture d’un mot sur les deux ports, utilisez `outw(`*`value, , x`*`)`. Si vous n’êtes pas certain quant à la fonction à utiliser (octet ou mot), il est sage de se cantonner à l’appel de `inb()` et `outb()`. La plupart des périphériques sont conçus pour des accès sur un octet. Notez que toutes les instructions d’accès aux ports nécessitent un temps d’exécution d’au minimum une microseconde.

Les macros `inb_p()`, `outb_p()`, `inw_p()` et `outw_p()` fonctionnent de manière identique à celles évoquées précédemment à l’exception du fait qu’elles effectuent un court temps de pause additionnel après l’accès au port (environ une microseconde). Vous avez la possibilité d’allonger ce temps de pause à quatre microsecondes avec la directive `#define REALLY_SLOW_IO` avant de déclarer `#include <asm/io.h>`. Ces macros utilisent normalement une écriture sur le port `0x80` pour leur temps de pause (sauf en déclarant un `#define SLOW_IO_BY_JUMPING`, qui est en revanche moins précis). Vous devez donc au préalable autoriser l’accès au port `0x80` avec `ioperm()` (l’écriture sur le port `0x80` ne devrait avoir aucun effet indésirable sur votre système).

Si vous êtes à la recherche de méthodes plus souples d’utilisation, lisez la suite …

Des pages de manuel pour ioperm(2), iopl(2) et les macros décrites ci-dessus sont disponibles dans les collections assez récentes des pages de manuel Linux.

2.2. Une méthode alternative : `/dev/port`

Un autre moyen d’accéder aux ports d’entrées / sorties est d’ouvrir en lecture ou en écriture le périphérique `/dev/port` (un périphérique en mode caractère, numéro majeur `1`, mineur `4`) au moyen de la fonction `open()`. Notons que les fonctions en `f*()` de la bibliothèque stdio font appel à des tampons mémoires internes, il vaut donc mieux les éviter. Il suffit ensuite, comme dans le cas d’un fichier, de se positionner sur l'octet approprié au moyen de la fonction `lseek()` (l'octet `0` du fichier équivaut au port `0x00`, l'octet `1` au port `0x01`, et cætera) et d'en lire (`read()`) ou écrire (`write()`) un octet ou un mot.

Il est évident que l'application doit avoir la permission d'accéder au périphérique `/dev/port` pour que cette méthode fonctionne. Cette façon de faire reste certainement plus lente que la première, mais elle ne nécessite ni optimisation lors de la compilation ni appel à `ioperm()`. L'accès aux privilèges de super-utilisateur n'est pas impératif non plus, si vous donnez les permissions adéquates à un utilisateur ou un groupe pour accéder à `/dev/port` (cela reste tout de même une très mauvaise idée du point de vue de la sécurité du système, puisqu'il devient possible de porter atteinte au système, peut-être même d'obtenir le statut de root en utilisant `/dev/port` pour accéder directement aux disques durs, cartes réseaux, et cætera).

Il n'est pas possible d'utiliser les fonctions select(2) ou poll(2) pour lire `/dev/port` puisque l'électronique du système n'a pas la possibilité d'avertir le microprocesseur qu'une valeur a changé sur un port d'entrée.

3. Interruptions (IRQ) et accès DMA

Vous ne pouvez tout simplement pas utiliser directement les interruptions ou l'accès DMA depuis un processus en mode utilisateur. Pour cela, il vous faut développer un pilote pour le noyau. Reportez-vous au Linux Kernel Hacker's Guide pour plus de détails et au code source du noyau pour des exemples.

Vous avez cependant la possibilité de désactiver les interruptions depuis une application en mode utilisateur, mais cela peut s'avérer dangereux (même les pilotes du noyau ne le font que pour des périodes de temps très brèves). Après appel à `iopl(3)`, vous pouvez désactiver les interruptions en utilisant `asm("cli");` et les réactiver avec `asm("sti");`.

4. Temporisation de haute précision

4.1. Temporisations

Avant toutes choses, il est important de préciser l'impossibilité de garantir un contrôle précis des temps d'exécution de processus en mode utilisateur du fait de la nature multitâche du noyau Linux. Votre processus peut être mis en sommeil à n'importe quel moment pour une durée allant de 10 millisecondes à quelques secondes (sur un système dont la charge est très importante). Malgré tout, pour la plupart des applications utilisant les ports d'entrées / sorties, cela n'est pas très important. Si vous voulez minimiser cet inconvénient, vous pouvez donner à votre processus une priorité plus haute (reportez-vous à la page de manuel de nice(2)) ou faire appel à l'ordonnancement temps-réel (voir ci-après).

Si vous souhaitez obtenir une précision de temporisation plus élevée que celle qu'offre les processus en mode utilisateur usuels, sachez qu'il existe des possibilités de support « temps-réel » en mode utilisateur. Les noyaux Linux de la série 2.x permettent de travailler en quasi temps-réel. Pour les détails, reportez-vous à la page de manuel de sched_setscheduler(2). Il existe également des versions spéciales du noyau offrant un vrai ordonnancement temps-réel.

4.1.1. Avec `sleep()` et `usleep()`

Commençons tout d'abord par les appels de temporisation les plus simples. Pour des temporisation de plusieurs secondes, le meilleur choix est probablement la fonction `sleep()`. Pour des durées au minimum de l'ordre de dizaines de millisecondes (10 millisecondes semblent être la durée minimum), `usleep()` devrait s'avérer suffisant. Ces fonctions libèrent l'accès au microprocesseur pour d'autres processus, évitant ainsi le gaspillage du temps machine. Les pages de manuel de sleep(3) et usleep(3) vous donneront plus de précisions.

Pour des temporisations de moins de 50 millisecondes (en fonction de la cadence du microprocesseur, de la machine ainsi que de la charge du système), redonner le processeur aux autres processus prend énormément de temps. En effet l'ordonnanceur des tâches du noyau Linux (en tout cas pour les microprocesseurs de la famille x86) prend généralement au moins 10 à 30 millisecondes avant de rendre le contrôle au processus. De ce fait, dans les temporisations de courte durée, usleep(3) effectue en réalité une pause plus longue que celle spécifiée en paramètre, prenant au moins 10 millisecondes supplémentaires.

4.1.2. `nanosleep()`

Dans les noyaux Linux de la série 2.0.x est apparu l'appel système `nanosleep()` (voir la page de manuel de nanosleep(2)) permettant d'endormir ou de retarder un processus pendant un laps de temps très court (quelques microsecondes ou plus).

Pour des attentes ≤ 2 millisecondes, si (et seulement si) votre processus fonctionne en ordonnancement quasi temps-réel (au moyen de `sched_setscheduler()`), `nanosleep()` fait appel à une boucle d'attente ; si tel n'est pas le cas, le processus s'endort simplement tout comme avec `usleep()`.

La boucle d'attente utilise `udelay()` (une fonction interne au noyau utilisée par beaucoup de pilotes), la durée de celle-ci étant calculée en fonction du nombre de `BogoMips`. La vitesse de ce type de boucle d'attente est une des grandeurs que les `BogoMips` permettent de mesurer de façon précise. Voyez `/usr/include/asm/delay.h` pour plus de détails quant à son fonctionnement.

4.1.3. Temporisations grâce aux ports d'entrée / sortie

Les accès aux ports d'entrée / sortie sont un autre moyen d'obtenir des temporisations. L'écriture ou la lecture d'un octet sur le port `0x80` (voir ci-dessus la procédure à suivre) devrait avoir pour conséquence un retard d'une microseconde, indépendamment du type et de la cadence du microprocesseur. Vous pouvez donc procéder de la sorte afin d'obtenir un retard de quelques microsecondes. L'écriture sur ce port ne devrait pas avoir d'effets secondaires sur une machine classique, pour preuve certains pilotes du noyau font appel à cette méthode. C'est également de cette manière que `{in|out}[bw]_p()` effectue une pause (voir `asm/io.h`).

Plus précisément, une opération de lecture ou d'écriture sur la plupart des ports dans l'intervalle `0x000-0x3ff` prend 1 microseconde. Par exemple, si vous utilisez directement le port parallèle, il suffit d'utiliser des `inb()` additionnels sur ce port pour obtenir une temporisation.

4.1.4. Temporisations en assembleur

Si vous connaissez le type et la fréquence du processeur de la machine sur laquelle votre programme va s'exécuter, vous pouvez coder en dur les temporisations en faisant appel à certaines instructions assembleur. Rappelez-vous cependant qu'à tout moment votre processus peut-être mis en attente par l'ordonnanceur et, de ce fait, les temporisations peuvent s'avérer plus longues que souhaitées.

Pour les données du tableau ci-dessous, la fréquence interne du microprocesseur détermine le nombre de cycles d'horloge consommés. Par exemple, pour un microprocesseur à 50 Mhz (un 486DX-50), un cycle d'horloge dure 1/50000000 de seconde (soit 200 nanosecondes).

Instruction	cycles d'horloge i386	cycles d'horloge i486
`xchg %bx,%bx`	3	3
`nop`	3	1
`or %ax,%ax`	2	1
`mov %ax,%ax`	2	1
`add %ax,0`	2	1

Les cycles d'horloges du Pentium devraient être les mêmes que ceux du 486, à l'exception du Pentium Pro / II, dont l'instruction `add %ax, 0` peut ne consommer qu'un demi cycle d'horloge. Cette instruction peut-être parfois être combinée avec une autre (cependant, du fait de l'algorithme d'exécution hors de séquence (*out-of-order*), il n'est pas nécessaire qu'il s'agisse d'une instruction consécutive dans le flot).

Les instructions `nop` et `xchg` du tableau ne devraient pas avoir d'effets secondaires. Les autres, en revanche, peuvent modifier le registre d'état, mais cela reste sans gravité puisque gcc devrait le détecter. `xchg %bx,%bx` reste un bon compromis comme instruction de temporisation.

Pour utiliser ces instructions, placez un appel `asm("`*`instruction`*`")` dans votre programme. La syntaxe d'appel de ces instructions est telle qu'énumérée dans le tableau ci-dessus. Si vous préférez grouper plusieurs instructions dans le même appel à `asm`, il vous suffit de les séparer par des points-virgules. Par exemple `asm("nop ; nop ; nop ; nop")` exécute quatre fois l'instruction `nop`, effectuant une temporisation de quatre cycles d'horloge sur un i486 ou un Pentium (ou douze cycles sur un i386).

Les instructions `asm()` sont directement intégrées au code par gcc, évitant ainsi la perte de temps que pourrait engendrer un appel de fonction classique.

Les temporisations de moins d'un cycle d'horloge sont impossibles sur les architectures x86 d'Intel.

4.1.5. `rdtsc` pour Pentium

Avec les microprocesseurs Pentium, vous avez la possibilité de connaître le nombre de cycles d'horloge écoulés depuis le dernier redémarrage avec le code C suivant (qui fait appel à l'instruction appelée RDTSC) :

```
extern __inline__ unsigned long long int rdtsc()
{
unsigned long long int x;
__asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
return x;
}
```

Vous pouvez scruter cette valeur dans une boucle d'attente afin d'obtenir un retard correspondant au nombre de cycles d'horloge que vous souhaitez.

4.2. Mesure du temps

Pour des durées de la précision d'une seconde, il est certainement plus simple d'utiliser la fonction `time()`. Pour obtenir plus de précision, `gettimeofday()` a une précision d'environ une microseconde (mais rappelez-vous de l'ordonnancement déjà évoqué précédemment). Pour les Pentium, le fragment de code `rdtsc` ci-dessus est précis au cycle d'horloge près.

Si vous souhaitez que votre processus reçoive un signal après un certain laps de temps, utilisez `setitimer()` ou `alarm()`. Voyez les pages de manuel de ces fonctions pour plus de détails.

5. D'autres langages de programmation

Les descriptions ci-dessus se concentrent principalement sur le langage C. Cependant, ces exemples devraient être directement applicables au C++ et à l'objective C. En assembleur, il vous suffit de faire appel à `ioperm()` ou `iopl()`, comme en C, et d'utiliser ensuite directement les ports d'entrée / sortie.

Avec d'autres langages, à moins de pouvoir insérer du code assembleur, du code C ou utiliser les appels systèmes, il est probablement plus simple d'écrire un programme C très simple offrant des fonctions prenant en charge les temporisations et les accès aux ports d'entrées / sorties souhaités, de le compiler et de le lier avec le reste de votre application. Ou vous pouvez utiliser `/dev/port` comme décrit plus haut.

6. Quelques ports utiles

Voici quelques informations utiles pour la programmation des ports les plus utilisés. Ces ports peuvent être directement interfacés avec une électronique logique de type TTL (ou CMOS).

Si vous avez l'intention d'utiliser ces ports ou des ports classiques en vue d'applications pour lesquels ils ont été conçus (par exemple pour contrôler une imprimante ou un modem ordinaire), vous auriez intérêt à utiliser les pilotes déjà existants (qui sont généralement inclus dans le noyau) plutôt que de programmer directement les ports comme décrit dans ce document. Cette section s'adresse principalement aux personnes désireuses de connecter aux ports d'entrée / sortie standards de leur PC des écrans à cristaux liquides, des moteurs pas-à-pas ou d'autres montages électroniques faits maison.

En revanche, si votre but est de piloter un périphérique grand public, tel qu'un scanner, disponible sur le marché depuis quelques temps déjà, essayez plutôt de savoir si un pilote a déjà été développé. Le Hardware-HOWTO est un bon point de départ pour mener cette investigation.

http://www.hut.fi/Misc/Electronics/ est également une excellente source d'information sur la connexion de périphériques à un ordinateur et sur l'électronique en générale.

6.1. Le port parallèle

L'adresse de base du port parallèle (appelée *`BASE`* ci-dessous) est `0x3bc` pour `/dev/lp0`, `0x378` pour `/dev/lp1` et `0x278` pour `/dev/lp2`. Si vous souhaitez piloter un matériel fonctionnant comme une imprimante classique, voyez le Printing-HOWTO.

En plus du mode standard d'écriture-seule décrit ci-dessous, il existe un mode « étendu » bidirectionnel sur la plupart des ports parallèles. Pour des informations à ce sujet ainsi que sur les récents modes ECP / EPP (et le standard IEEE 1284 en général), voici deux adresses : http://www.fapo.com/ et http://www.beyondlogic.org/. Rappelez-vous cependant que puisque vous ne pouvez pas utiliser les requêtes d'interruptions (IRQ) ou l'accès DMA dans un programme en mode utilisateur, vous serez certainement obligé d'écrire un pilote pour le noyau afin d'exploiter les modes ECP / EPP. Il me semble que quelqu'un est déjà en train de développer un tel pilote, mais je n'en sais pas plus à ce sujet.

Le port `BASE+0` (port de données) contrôle les signaux de données du port (respectivement D0 à D7 pour les bits 0 à 7; états : 0 = bas (0V), 1 = haut (5V)). Une écriture sur ce port positionne l'état des broches correspondantes. La lecture retourne la dernière valeur écrite en mode standard ou étendu, sinon les données présentes sur les broches connectées à un autre périphérique en mode de lecture étendue.

Le port `BASE+1` (port d'état) est en lecture seule et retourne l'état des signaux d'entrée suivants :

- Bits 0 et 1 sont réservés.
- Bit 2 état d'IRQ (ne correspond pas à une broche, je ne sais pas comment il fonctionne)
- Bit 3 ERROR (1 = état haut)
- Bit 4 SLCT (1 = état haut)
- Bit 5 PE (1 = état haut)
- Bit 6 ACK (1 = état haut)
- Bit 7 -BUSY (0 = état haut)

Le port `BASE+2` (port de contrôle) est en écriture seule (une lecture sur celui-ci retourne la dernière valeur écrite) et contrôle les signaux d'état suivants :

- Bit 0 -STROBE (0 = état haut)
- Bit 1 -AUTO_FD_XT (0 = état haut)
- Bit 2 INIT (1 = état haut)
- Bit 3 -SLCT_IN (0 = état haut)
- Bit 4 active la requête d'interruption (IRQ) du port parallèle (qui survient lors d'une transition bas-vers-haut de la broche ACK) lorsqu'il est positionné à 1.
- Bit 5 contrôle la direction du mode étendu (0 = écriture, 1 = lecture) et est en écriture seule (une lecture sur ce bit ne retournera pas de valeur significative).
- Bits 6 et 7 sont réservés.

Brochage (d'un connecteur femelle sub-D 25 sur le port) (e = entrée, s = sortie) :

```
 1(e/s) -STROBE,   2(e/s) D0,          3(e/s) D1,          4(e/s) D2,
 5(e/s) D3,        6(e/s) D4,          7(e/s) D5,          8(e/s) D6,
 9(e/s) D7,       10(e) ACK,          11(e) -BUSY,        12(e) PE,
13(e) SLCT,       14(s) -AUTO_FD_XT,  15(e) ERROR,        16(s) INIT,
17(s) -SLCT_IN,   18-25 masse
```

Les spécifications données par IBM indiquent que les broches 1, 14, 16 et 17 (les sorties de contrôle) sont de type collecteur ouvert ramené à 5,0 V au moyen d'une résistance de 4,7 kohm (intensité de fuite de 20 mA, de source 0,55 mA, état haut de sortie 5,0 V moins tension de *pull-up*). Le reste des broches ont une intensité de fuite de 24 mA, de source de 15 mA et leur voltage à l'état haut est au minimum de 2,4 V. L'état bas pour toutes les broches est au maximum de 0,5 V. Les ports parallèles d'autres types que celui d'IBM peuvent s'écarter de ce standard. Pour plus d'informations à ce sujet, voyez l'adresse : http://www.hut.fi/Misc/Electronics/circuits/lptpower.html.

Avertissement

Faites très attention aux masses ! J'ai déjà grillé à plusieurs reprises des ports parallèles en les connectant alors que l'ordinateur était sous tension. Utiliser un port parallèle non intégré à la carte mère peut s'avérer une bonne solution pour éviter de trop grands désagréments. Vous pouvez obtenir un second port parallèle pour votre machine au moyen d'une carte multi-entrées / sorties à petit prix. Il vous suffit de désactiver les ports dont vous n'avez pas besoin, puis de configurer le port parallèle de la carte sur une adresse libre. Vous n'avez pas besoin de vous préoccuper de l'IRQ du port parallèle si vous n'y faites pas appel.

6.2. Le port de manette de jeu

Le port de manette de jeu est accessible aux adresses `0x200-0x207`. Si vous souhaitez contrôler une manette de jeu ordinaire, vous serez probablement mieux servi en utilisant les pilotes distribués avec le noyau.

Brochage (pour un connecteur sub-D 15 femelle) :

- 1,8,9,15 : +5 V (Alimentation)
- 4,5,12 : masse
- 2,7,10,14 : entrées numériques, respectivement BA1, BA2, BB1 et BB2
- 3,6,11,13 : entrées « analogiques », respectivement AX, AY, BX et BY

Les broches fournissant une tension de +5 V semblent souvent être connectées directement sur l'alimentation de la carte mère, ce qui peut leur permettre d'obtenir pas mal de puissance, en fonction de la carte mère, du bloc d'alimentation et du port de manette de jeu.

Les entrées numériques sont utilisées pour les boutons des deux manettes de jeu (manette A et manette B, avec deux boutons chacune) que vous pouvez connecter au port. Ces entrées devraient être de niveau TTL classique, ainsi vous pouvez lire directement leurs valeurs sur le port d'état (voir plus bas). Une véritable manette de jeu retourne un état bas (0 V) lorsque le bouton est pressé et un état haut dans l'autre cas (une tension de 5 V des broches de d'alimentation au travers d'une résistance de 1 kohm).

Les pseudo entrées analogiques mesurent en réalité la résistance. Le port de manette de jeu comporte un quadruple monostable (une puce de type NE558) connecté aux quatre entrées. Pour chaque entrée, il y a une résistance de 2,2 kohm entre la broche d'entrée et la sortie du monostable, et un condensateur de 0,01 µF entre la sortie du monostable et la masse. Une véritable de manette de jeu a un potentiomètre pour chaque axe (X et Y), connecté entre le +5 V et la broche d'entrée appropriée (AX ou AY pour la manette A, ou BX ou BY pour la manette B).

Lorsqu'il est activé, le monostable initialise ses lignes de sortie à un état haut (5 V) et attend que chaque condensateur de temporisation atteigne une tension de 3,3 V avant de mettre la sortie correspondante à un état bas. La durée de l'état haut de la sortie du temporisateur est proportionnelle à la résistance du potentiomètre de la manette de jeu (en clair, la position du manche de la manette de jeu de l'axe approprié) comme expliqué ci-dessous :

$$R = (t - 24.2) / 0.011$$

où `R` est la résistance en ohm, du potentiomètre et `t` la durée de l'état haut de la sortie, en microsecondes.

Pour effectuer une lecture sur les entrées analogiques, vous devez tout d'abord activer le monostable (au moyen d'une écriture sur le port, voir plus bas), puis scruter l'état des quatre axes au moyen de lectures répétées jusqu'à la transition à un état bas, permettant ainsi de mesurer la durée de l'état haut. Cette scrutation est très gourmande en temps machine. De plus, sur un système multitâche non temps-réel tel que Linux (en mode utilisateur normal) le résultat n'est pas très précis, du fait de l'impossibilité de scruter le port constamment (à moins d'utiliser un pilote noyau et de désactiver la gestion des interruptions pendant la scrutation, mais sachez que vous consommerez encore plus de temps machine). Si vous savez à l'avance que le signal va mettre un certain temps (plusieurs dizaines de millisecondes) avant de basculer, vous pouvez faire appel à `usleep()` avant de procéder à la scrutation afin de laisser un peu de temps machine aux autres processus.

Le seul port d'entrées / sorties auquel vous avez besoin d'accéder est le port `0x201` (les autres ports se comportent de façon identique ou ne réagissent pas). Toute écriture sur ce port, peu importe la valeur envoyée, active le temporisateur. Une lecture retourne l'état des signaux d'entrée :

- Bit 0 : AX (état de la sortie du temporisateur, 1 = état haut)
- Bit 1 : AY (état de la sortie du temporisateur, 1 = état haut)
- Bit 2 : BX (état de la sortie du temporisateur, 1 = état haut)
- Bit 3 : BY (état de la sortie du temporisateur, 1 = état haut)
- Bit 4 : BA1 (entrée numérique, 1 = état haut)
- Bit 5 : BA2 (entrée numérique, 1 = état haut)
- Bit 6 : BB1 (entrée numérique, 1 = état haut)
- Bit 7 : BB2 (entrée numérique, 1 = état haut)

6.3. Le port série

Si le périphérique avec lequel vous communiquez est à peu près compatible avec le standard RS-232, vous devriez être en mesure d'utiliser pour cela le port série. Le pilote série du noyau Linux devrait suffire pour la plupart des applications (vous ne devriez pas avoir à programmer le port directement, de plus vous devriez probablement écrire un pilote pour le noyau si vous souhaitiez le faire), il est en effet très polyvalent et l'usage de débits non-standards ne devrait pas poser de problèmes particuliers.

Voyez la page de manuel de termios(3), le code source du pilote (`linux/drivers/char/serial.c`) ainsi que la page http://www.easysw.com/~mike/serial/ pour plus d'informations sur la programmation des ports séries des systèmes Unix.

7. Conseils

Si vous souhaitez obtenir une bonne acquisition analogique, vous pouvez connecter un CAN ou un CNA au port parallèle (conseil : pour l'alimentation, utilisez soit le port de manette de jeu, soit un connecteur d'alimentation de disque que vous relierez à l'extérieur de votre boîtier, à moins que vous n'utilisiez un périphérique peu consommateur d'énergie, et que vous puissiez utiliser le port parallèle lui-même pour l'alimenter, sinon utilisez tout simplement une source d'alimentation externe). Vous pouvez également investir dans l'achat d'une carte d'acquisition numérique / analogique ou analogique / numérique (la plupart des cartes d'acquisition les plus anciennes et les plus lentes s'appuient sur l'usage de ports de communication externes). Sinon, si vous pouvez vous satisfaire de seulement un ou deux canaux d'acquisition, de résultats peu précis et sans doute d'un mauvais niveau du zéro, l'utilisation d'une carte son bas de gamme supportée par le noyau Linux devrait répondre à votre attente (et vous permettre de faire des acquisitions rapides).

Avec des périphériques analogiques de précision, une mauvaise mise à la masse peut générer de mauvaises mesures lors de lectures ou d'écritures analogiques. Si vous rencontrez ce type de désagrément, vous pouvez isoler électriquement le périphérique de votre ordinateur au moyen d'optocoupleurs (sur *toutes* les lignes entre le périphérique et l'ordinateur). Vous pouvez essayer de tirer l'alimentation électrique des optocoupleurs de votre ordinateur (les signaux de broches non utilisées du port peuvent peut-être vous fournir une puissance suffisante) afin d'obtenir une meilleure isolation.

Si vous êtes à la recherche d'un logiciel de conception de circuits imprimés sous Linux, sachez qu'il existe une application libre pour X11 appelée Pcb™ qui devrait vous rendre de grands services, tout du moins si vous n'avez pas de projets trop compliqués. Ce logiciel est inclus dans la plupart des distributions Linux et est disponible à l'adresse suivante :
ftp://sunsite.unc.edu/pub/Linux/apps/circuits/ (nom de l'archive `pcb-*`).

8. Problèmes et solutions

Q1.

J'obtiens une erreur de segmentation lorsque j'essaye d'accéder aux ports.

R1.

Soit votre programme n'a pas les privilèges de super utilisateur, soit l'appel à `ioperm()` n'a pas réussi pour une raison ou une autre. Vérifiez la valeur de retour de `ioperm()`. Vérifiez également que vous accédez bien aux ports que vous avez activés préalablement avec `ioperm()`

(voir Q3). Si vous faites appel aux macros de temporisation (`inb_p()`, `outb_p()`, et cætera), pensez aussi à utiliser `ioperm()` pour obtenir l'accès au port `0x80`.

Q2.

Je n'arrive à trouver nulle part la définition de `in*()`, `out*()`, de plus gcc se plaint de références non-définies.

R2.

Vous n'avez pas lancé la compilation avec les drapeaux d'optimisation (`-O`), en conséquence gcc n'a pas pu trouver les macros définies dans `asm/io.h`. Ou alors, vous n'avez pas inclus du tout la ligne `#include <asm/io.h>` dans votre code.

Q3.

`out*()` ne fait rien ou ne retourne que des valeurs bizarres.

R3.

Vérifiez l'ordre des paramètres, ceux-ci devraient être comme ce qui suit : `outb(`*`valeur`*`, , `*`port`*`)` et non pas `outportb(`*`port`*`, , `*`valeur`*`)` comme en MS-DOS.

Q4.

Je souhaite contrôler un périphérique standard, tel qu'un périphérique RS-232, une imprimante parallèle, une manette de jeu, …

R4.

Vous auriez plutôt intérêt à utiliser les pilotes déjà existants (dans le noyau Linux ou un serveur X ou ailleurs) pour ce faire. Ces pilotes sont généralement assez polyvalents. Ainsi ils arrivent même en général à faire fonctionner les périphériques sortant légèrement des standards. Voyez la note d'information ci-dessus sur les ports standards pour de la documentation à leur sujet.

9. Code d'exemple

Voici un exemple de programme simple permettant l'accès aux ports d'entrées / sorties :

```
/*
* exemple.c : un exemple très simple d'accès aux ports d'E/S
*
* Ce programme ne fait rien d'utile, juste une écriture sur le port,
* une pause, puis une lecture sur le même port.
* À compiler avec « gcc -O2 -o exemple exemple.c »et à
* exécuter en tant que root avec « ./exemple ».
*/

#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

#define BASEPORT 0x378 /* lp1 */

int main()
```

```
{
/* Obtention de l'accès aux ports */
if (ioperm(BASEPORT, 3, 1)) {perror("ioperm"); exit(1);}

/* Initialisation de tous les signaux de données (D0-D7) à l'état bas (0) */
outb(0, BASEPORT);

/* Dormons pendant un moment (100 ms) */
usleep(100000);

/* Lecture sur le port d'état (BASE+1) et affichage du résultat */
printf("status : %d\n", inb(BASEPORT + 1));

/* Nous n'avons plus besoin de l'accès aux ports */
if (ioperm(BASEPORT, 3, 0)) {perror("ioperm"); exit(1);}

exit(0);
}

/* fin d'exemple.c */
```

10. Remerciements

Beaucoup trop de personnes ont contribué à l'écriture de ce document pour que je puisse en faire la liste ici, mais merci à tous. Je n'ai pas pu répondre à toutes les contributions reçues, je m'en excuse, mais encore merci pour votre aide.

11. Adaptation française

11.1. Traduction

La traduction française de ce document a été réalisée par Jean-François Prévost <prevost CHEZ 2cse TIRET group POINT com>.

11.2. Relecture

La relecture de ce document a été réalisée par Guillaume Lelarge <gleu CHEZ wanadoo POINT fr> et Jean-Philippe Guérard <fevrier CHEZ tigreraye POINT org>.