Mémoire de fin d'étude

GNU/Linux : systèmes embarqués

Nicolas Ferre - M5

Mars - Juin 2000

Proposé par *IRTS* (*Integrated Real Time Systems*)

Remerciements

Je tiens à remercier les personnes suivantes pour avoir rendu ce stage possible :

- Monsieur *Claude Silv*e qui, malgré ses occupations, a toujours répondu à mes questions,
- Monsieur *Michel Dumas* pour m'avoir fait profité d'explications à propos de leur développement en cours,
- Monsieur *Hugues Perony* sans qui ce stage n'aurait pas été possible,
- et enfin toutes les autres personnes de la société *IRTS* pour avoir été si accueillantes avec moi.

Table des matières

Table des figures

Première partie .

SYNTHÈSE

1. Présentation de l'entreprise *IRTS*

Integrated Real Time Systems, désignée par le sigle *IRTS* a été créée en 1995 sous la forme d'une S.A.R.L. et dispose d'un capital de 2 340 000 F, entièrement détenu par ses fondateurs.

La société a réalisé 30 millions de francs de chiffre d'affaires au titre de son dernier exercice. Elle comptera 19 personnes dont 16 ingénieurs et réalisera un chiffre d'affaires de 32 millions de francs au titre de l'exercice 2000.

1.1. Stratégie

La société *IRTS* est spécialisée en ingénierie des systèmes dans le domaine des applications temps réel, de l'informatique embarquée, et de l'informatique industrielle, et se positionne sur deux secteurs d'activités complémentaires :

- l'ingénierie d'étude et de réalisation, d'assistance technique et de conseil
- la réalisation et la fourniture de systèmes informatiques dédiés et la commercialisation de produits standards appartenant au catalogue de divers constructeurs (micro stations embarquées, cartes et systèmes *VME*, *PCI*, *CompactPCI*, *PICMG*, moniteurs *LCD* de classe industrielle, *panel PC*)

1.2. Ingénierie d'Etude et de Développement, d'Assistance Technique et de Conseil

Créée par des ingénieurs possédant une grande expérience dans l'étude et la conception de systèmes informatiques pour les domaines de la défense et de l'industrie, *IRTS* propose des services d'expertise et de conseils couvrant les phases d'études, de définition de besoins, d'analyse, de conception et de réalisation de ces systèmes.

1.2.1. Assistance Technique et Conseil

Répondant aux besoins de grands systémiers, les services proposés incluent la rédaction de cahier des charges, l'assistance au développement et à la conduite de projet concernant la réalisation de grands systèmes d'information à logiciel prépondérant.

1.2.2. Ingénierie d'Etude et de Développement

Le savoir-faire de la société dans ce domaine réside dans sa capacité à assembler des systèmes multi-cartes, parfois multi-CPU et multi-OS. Sous l'aspect matériel, les standards maîtrisés par *IRTS* sont relatifs aux architectures *VME* et *PCI* déclinées sous leurs différents facteurs de formes (simple et double Europe, *Compact PCI*, *PMC*, *PICMG*).

Sous l'aspect logiciel, l'intégration de solutions s'opère sous des systèmes d'exploitation *Unix* (*AIX*, *Solaris*, *Linux*), *Windows/NT* ou temps réel (*LynxOS*, *VxWorks*). Les développements correspondants concernent la réalisation de drivers unitaires de cartes (E/S, réseaux, video, etc...), l'implantation de protocoles spécialisés ou encore l'assistance à l'intégration d'applications.

La société a pour objectif de développer ses activités autour des logiciels libres, il lui faut donc évaluer de tels produits pour pouvoir proposer à ses clients une solution logicielle à base de RTLinux ou BlueCat Linux... C'est dans cette partie de la société que s'est déroulé mon stage chez *IRTS*.

1.2.3. Réalisation de systèmes dédiés et Technologies Informatiques

En complément de ses activités de services et d'ingénierie, *IRTS* se positionne comme équipementier informatique et fournisseur de produits *COTS* (*COTS Products - Commercial On The Shelf*) ou de systèmes 'temps réel' dédiés répondant aux exigences du client, en termes de contraintes d'exploitation, de fonctionnalités ou de performances.

Cette activité consiste à étudier, concevoir, produire et assembler, en fonction d'un besoin client, des systèmes composés de produits standards du commerce ou développés à la demande ; ces systèmes informatiques sont adaptés aux besoins du client en terme de standard choisi, de caractéristiques physiques ou électriques, de configurations mono ou multiprocesseurs, d'entrées-sorties, de systèmes d'exploitation et de *"packagings"*.

En terme de marché, ces systèmes sont mis en oeuvre dans des domaines pour lesquels les calculateurs classiques ne sont pas toujours les mieux adaptés ou les plus compétitifs.

La société a par ailleurs développé et commercialise depuis mai 1998, une gamme de moniteurs à écrans plats *LCD* (du 14" *VGA* au 20" *UXGA*) répondant aux exigences du marché de l'informatique embarquée et industrielle (matériaux, tenue aux chocs et aux vibrations, étanchéité à l'eau et à la poussière, ergonomie...).

1.3. Positionnement Marché

IRTS travaille en partenariat avec des sociétés de services, de grands intégrateurs et des systémiers afin de satisfaire dans les meilleures conditions à la demande de ses clients et de proposer les solutions les mieux adaptées à leurs besoins.

Le marché du logiciel libre

Le logiciel libre repose essentiellement sur la *GNU*[1] *General Public License* (*GPL*) de la *Free Software Fundation* (*FSF*). Cette licence, qui régit le système d'exploitation GNU/Linux, stipule entre autre que chacun est libre de copier, distribuer et modifier gratuitement les logiciels qu'elle protège. La seule condition requise est que les codes sources soient également fournis en cas de redistribution et que les noms des précédents auteurs perdurent lors de modifications.

Ce nouveau marché pour IRTS permet non seulement de *gagner* des parts auprès de nouvelles entreprises mais aussi de conforter leurs actuelles relations commerciales. En effet, les grands intégrateurs et systémiers s'intéressent de plus en plus aux possibilités qu'offre le système d'exploitation GNU/Linux. Son intégration comme système embarqué prend de l'envergure.

D'autre part, le code source, des pilotes de périphériques notamment, est utilisé pour effectuer des portages vers d'autres systèmes d'exploitation (LynxOS, AIX, QNX...). Un développement sous un système d'exploitation autre que GNU/Linux peut donc bénéficier du travail fait par la Communauté du logiciel libre car un portage est plus aisé qu'une création. Il sera alors nécessaire de suivre tout de même les termes de la *GPL* (Licence publique GNU) ; entre autre, de mettre les modifications apportées à un code source libre à la disposition de la Communauté.

[1] *GNU* qui signifie *'GNU is not UNIX'* est le nom d'un projet de la *Free Software Fundation* qui a pour but de créer et de maintenir toute une panoplie de programmes utilitaires compatibles *UNIX* et de les mettre gratuitement à disposition de tous. On appelle abusivement *Linux*, le système d'exploitaiton *GNU/Linux* qui est constitué du noyau *Linux* et des outils *GNU*.

2. Le projet

2.1. Présentation du sujet

Le projet qui m'a été confié fait suite au développement de l'utilisation des logiciels libres, au cours de ces dernières années. L'une des composantes les plus populaires du monde du logiciel libre est le système d'exploitation GNU/Linux. En plus d'être populaire dans les milieux universitaires et de recherche, ce système, de par ses innombrables qualités qui seront exposées ultérieurement, séduit les entreprises.

La société *IRTS* voit dans cette montée en puissance des perspectives de nouveaux marchés et a donc décidé d'ajouter GNU/Linux et ses dérivés, temps réel notamment, à ses domaines de compétences.

Le sujet de base de ce stage était la mise en oeuvre et l'évaluation du noyau RTLinux en terme de fonctionnalités et de performances. Mais, RTLinux, de par son architecture, est utilisé en interaction forte avec le noyau Linux et les outils *GNU* qui l'accompagnent. Le noyau temps réel RTLinux n'est rien par lui même. J'ai donc pris l'initiative d'élargir le sujet de mon stage, d'autant qu'à plusieurs reprises, j'ai dû répondre à des questions générales relatives aux systèmes à base de logiciels libres. Je pense que cette approche permettra à *IRTS* de **choisir** la bonne solution logicielle en réponse à une demande client dans ce domaine.

2.2. Environnement de développement

Afin de mettre en oeuvre ses solutions à base de logiciels libres, la société IRTS a créé une plate-forme de développement de type PC. Julien Gaulmin y a installé GNU/Linux, lors d'un stage précédent.

Ce PC à base de processeur *AMD K6II* 300MHz tourne sous une distribution *SuSE Linux 6.1*. J'y ai ajouté les différents outils logiciels nécessaires à un projet sous *RTLinux, RTAI, Linux et BlueCat Linux*. La distribution *Red Hat Linux 6.1* a dû être installée sur ce PC pour pouvoir utiliser le kit de développement de Lynx : BlueCat Linux.

Tous les outils d'ingénierie logicielle (éditeurs de textes, compilateurs, éditeurs de liens...) ainsi que les pages de manuel et les codes sources de l'OS et des drivers existants sont fournis dans les distributions et sont disponibles sur de nombreux sites Internet. J'ai disposé ainsi de nombreuses sources d'informations et d'une communauté toujours à l'écoute des problèmes de l'un de ses membres.

Le rack CPCI inova sert de machine cible dans le développement d'applications embarquées. Le cycle de mise en oeuvre d'un système embarqué, résumé sur le schéma 2.2, est facilité par l'utilisation des outils mis à la disposition du développeur par BlueCat Linux.

2.3. Déroulement de l'activité technique

Mon stage a commencé par deux semaines de travail intense : je devais répondre à un appel d'offre lancé par le Service Technique de la Navigation Aérienne. La consultation portait sur un ensemble de PC industriels tournant sous GNU/Linux opérant dans un système critique du contrôle aérien. Rentrant dans un mécanisme de gestion de calculateurs redondants, ces PC devaient contrôler d'autres machines et faire basculer le travail, d'un calculateur à l'autre, en cas de panne. Robustesse, fiabilité et temps de réponse sont demandés à de tels systèmes. Ceci m'a permis d'aborder directement le monde de l'embarqué sous GNU/Linux.

Puis, j'ai abordé la partie principale de mon stage : les Linux temps réel et RTLinux en particulier. Le choix de RTLinux est discuté dans le dossier technique en 5.1.2.

La partie de mon projet concernant le son s'est avérée être ardue car elle mobilisait beaucoup de concepts, nouveaux pour moi. Il fallait me familiariser avec les notions relatives au signal sonore, aux driver PCI, à la mesure de performance d'un système d'exploitation temps réel. Donc, cette partie avait aussi pour but de mesurer les performances temps réel de RTLinux soit la latence, la gîte et le temps de réponse à une interruption.

Durant toutes les phases de ce stage, j'ai dû me tenir au courant de tous les développements relatifs au monde de l'informatique embarquée. En effet, en s'intéressant aux Linux temps réel, on est plongé dans ce milieu en permanence. Actuellement, son activité est débordante et il ne faut rater aucune annonce sous peine de vite voir ses connaissances périmées (par exemple : la sortie, pendant ces quatre mois, de RTLinux V3beta et de RTAI V1.3 avec à chaque fois, de nouvelles fonctionnalités, de nouveaux drivers USB plus performants...). Cette activité de *veille technologique* me semble très importante : l'élargissement du sujet de mon stage et ce rapport en sont le fruit.

2.4. Techniques et connaissances mises en oeuvre

Ce projet m'a fortement intéressé car, tout au long de son déroulement, j'avais toujours quelque chose de nouveau à apprendre. Il m'a permis de suivre au jour le jour les activités d'une communauté de programmeurs, d'utiliser et d'alimenter l'énorme source de développements communs. En plus de connaissances techniques, il m'a fallu développer mon esprit de synthèse, en ne gardant que l'essentiel et le pertinent de tous les renseignements présents sur Internet.

Systèmes d'exploitation - noyaux : GNU/Linux (SuSE, Red Hat), RTLinux, RTAI, BlueCat Linux (Lynx Inc.)

Développement : C/C(GNU), outils GNU : tar, make, API POSIX, API RTLinux, modules noyau Linux.

Pilotes : carte TOR(ISA), carte son (PCI), port parallèle.

Performances : signal sonore et driver carte son, API son, caractérisation temps réel (benchmark)

FIG. 2.1. – Matériel mis à ma disposition.

FIG. 2.2. – Cycle de développement d'un logiciel embarqué

FIG. 2.3. – Planning de déroulement du stage.

3. Conclusion

Autonomie pourrait être le maître mot de mon stage car j'ai pu organiser, sans contrainte, ma progression dans le monde des Linux temps réel et du logiciel libre embarqué. La société IRTS m'a donc laissé toute latitude pour mener à bien une expédition dans cette jungle et défricher le terrain. J'espère lui avoir permis d'y voir plus clair et d'utiliser ces outils plus facilement avec l'aide de ce rapport.

Ce stage m'a appris à prendre mes responsabilités et à organiser le déroulement de mon activité technique. Je pense que ce stage sera riche d'enseignements pour mes futurs projets.

Sur le plan humain, j'ai eu la chance d'évoluer dans une jeune société dynamique et dans une excellente ambiance de travail. Ceci m'a permis de choisir la structure dans laquelle je voulais débuter ma vie professionnelle.

Cette expérience m'a aussi permis de découvrir le travail communautaire si vital au développement de GNU/Linux. J'ai ainsi obtenu 'gratuitement' et quasiment instantanément de l'aide très pointue auprès des milliers de développeurs systèmes passionnés de cet OS, ceci via les *newsgroups*, les forums et les nombreux articles et livres disponibles en ligne.

Deuxième partie .

Dossier Technique

4. GNU/Linux comme système d'exploitation embarqué

4.1. Embarqué ?

De plus en plus d'objets quotidiens embarquent avec eux de l'intelligence. Un ordinateur embarqué se définit généralement par le fait qu'il n'est pas visible en tant que tel, mais est intégré dans un équipement doté d'une autre fonction ; on dit aussi que le système est enfoui, ce qui traduit plus fidèlement le terme anglais *embedded*.

Un inventaire à la Prévert serait nécessaire pour citer les systèmes informatiques embarqués que nous utilisons quotidiennement : gestion de l'ascenseur, auto radio, calculateur d'"air bag", distributeur de boissons, routeur Internet ou téléphonique, téléphone mobile, distributeur de billets, une console de jeux, une carte graphique (bientôt un raton laveur car le chien électronique existe déjà !!))... Nous voyons bien que les contraintes imposées à tous ces systèmes ne sont pas les mêmes : qu'y a t'il de commun entre un satellite et une imprimante, à part le fait de contenir des processeurs ?

On peut classer les acteurs de l'informatique embarquée suivant de multiples critères :

Secteur d'activité	Contraintes
Équipements scientifiques	Performances, fiabilité, coût
Équipements militaires et aérospatiaux	Performances, fiabilité, pérennité, intégration
Transports	Fiabilité, coût, interactivité
Informatique industrielle	Fiabilité, coût, pérennité
Matériel de bureau	Performance, coût, standardisation
Réseau et télécommunications	Performance, fiabilité, intégration
Électronique grand public	Performance, coût, design / intégration

Ceci implique une diversité certaine des architectures matérielles...

4.1.1. Architectures matérielles

On peut s'amuser à découper une station de travail classique suivant cinq blocs fonctionnels. Puis, en éliminant tour à tour un ou plusieurs blocs, on obtient différentes configurations matérielles pouvant s'appliquer à un équipement embarqué. [1]

[1] Cette excellente idée est tirée du livre "Java embarqué" [10].

FIG. 4.1. – Typologie des architectures

En utilisant le schéma 4.1 on peut construire :

1. avec tous les blocs une station de travail industrielle connectée à un réseau
2. avec l'unité centrale, une IHM, et de la mémoire de masse, un PC domestique
3. avec l'unité centrale, et des entrées / sorties spécifiques, un système industriel indépendant que l'on peut relier à un réseau si on lui adjoint le bloc #5.
4. ...

Il y a ainsi 14 configurations possibles (car certaines des 27 sont absurdes) décrivant chacune un équipement informatique. GNU/Linux est pensé à l'origine comme un système d'exploitation pour station de travail de type PC connectée à un réseau (blocs 1,2,3,4). Il faut donc voir s'il est assez configurable pour pouvoir animer les autres agencements de blocs sans trop de modifications et / ou de pertes de performance.

D'autre part, l'augmentation de la proportion de matériel standard (et notamment de type PC) dans les équipements embarqués est tout à fait remarquable. En effet,

- leur fiabilité,
- leur disponibilité immédiate,
- leur facilité d'intégration sur des standards ouverts (PCI, USB...),
- la multiplicité des interfaces développées (réseau, entrées / sorties numériques ou analogiques, son, image, processeurs de signal...),
- et surtout leur coût,

font que ce type de matériel prend une place de plus en plus importante sur ce marché. Ces avantages proviennent du fait qu'il est produit en masse, par de multiples entreprises, tout autour du globe.

De part son gain de notoriété, GNU/Linux s'installe sur de nombreuses plates formes différentes de son "nid" d'origine PC/x86. On peut aussi trouver Linux codé pour les processeurs PPC, 68k, Alpha, MIPS, certains microcontrôleurs, ARM , SPARC, travaillant sur des architectures VME, CompactPCI, PC/104.

4.1.2. Plates-formes logicielles

Nous avons vu que les caractéristiques et besoins d'un système embarqué variaient beaucoup selon leur utilisation. Néanmoins, on peut décrire les composants essentiels dont un système embarqué a besoin :

- un utilitaire de lancement (LILO, BlueCat OS loader, GRUB...)

- un noyau composé d'un gestionnaire de mémoire, de processus et de temps (le noyau Linux lui même, réduit à sa plus simple expression)
- un processus d'initialisation (*init* sous Linux qui peut très bien être remplacé par un script de lancement voir le miniHOWTO Embedded Linux de Paul Moody)

Afin que ce système puisse faire quelque chose d'utile pour le monde réel, on doit ajouter :

- les pilotes de périphériques à utiliser
- une ou plusieurs applications, éventuellement un(des) processus interactif(s) (*shell*) pour commencer le travail (il n'y a pas souvent de processus de login sur un système embarqué)

Et de nos jours, on utilise assez souvent :

- un système de fichier (Sur disque dur ou Flash, ROM, disque RAM)
- une pile réseau TCP/IP
- une interface graphique
- ...

Linux peut apporter d'une part, un noyau répondant à toutes les exigences d'un système embarqué (réduction de taille jusqu'à 300 Koctets), d'autre part, l'ensemble des services utilisateurs imaginables. Linux peut convenir aussi bien à un équipement à taille ultra réduite qu'à un gros serveur d'applications mais l'adaptation à un pico-système demande beaucoup plus de malice et de travail : mais il faut aller chercher les outils de taille réduite car ils ne sont pas habituellement fournis dans les distributions standards. `http://www.embedded-linux.org` constitue un excellent "portail" pour les recherches d'outils de taille réduite (voir 4.3.3)

4.2. Pourquoi GNU/Linux ?

4.2.1. Avantages :

Gratuité à l'achat et pendant l'exploitation (r*untime license*), c'est l'avantage le plus populaire des logiciels libres et tout le travail de la Communauté est de rappeler que libre $\neq$ gratuit !!!

Code modifiable à volonté du moment que les modifications sont mises dans le domaine public. Ceci permet d'adapter son système à ses besoins, d'avoir des exemples pour un développement similaire. Mais aussi, cette caractéristique est importante pour améliorer sécurité et fiabilité (les failles et erreurs sont rapidement repérées et corrigées par la Communauté).

Coûts de formation et maintenance faibles car GNU/Linux est basé sur POSIX (les administrateurs UNIX ont peu de mal à l'aborder).

Système d'exploitation fiable (mémoire protégée).

Un large éventail de protocoles, langages, pilotes, systèmes de fichiers, interfaces graphiques... sont supportés par cet OS. La liste des langages est par exemple absolument impressionnante (il manque seulement Visual Basic (!!) :-)

4.2.2. Inconvénients :

Linux n'est pas un micro-noyau, il en résulte que la taille minimale qu'il peut atteindre n'est pas aussi impressionnante que celle du noyau de QNX : Neutrino (quelques dizaines de Koctets). Mais, d'une part, les contraintes d'empreinte mémoire sont moins drastiques qu'il y a quelques années car on utilise maintenant des composants standards et grand public. La RAM est souvent bien supérieure a 4 Moctets et des Flash de 32 Moctets atteignent des prix raisonnables (développement des appareils photos numériques, balladeurs MP3...). De plus, Un micro-noyau a besoin de nombreux modules périphériques pour fonctionner, la taille totale augmente alors très vite...
Pour les adeptes du concept, il faut s'intéresser de près à *Hurd*, le micro-noyau libre ou *ELKS* pour les machines à base de 286 `http://www.elks.ecs.soton.ac.uk/cgi-bin/ELKS/`.

Les processeurs et / ou architectures différents de x86/PC demandent plus de travail de configuration. Les utilisateurs étant moins nombreux, on ne dispose pas d'une aide aussi rapide et les documents sont plus difficiles à trouver.

Aucune société n'est à l'origine du développement de GNU/Linux. Je vois plutôt ça comme un avantage et un gage de pérennité mais certains décideurs semblent le craindre. De plus, ceci est une opportunité pour des sociétés de service qui peuvent s'engager comme responsables de développements logiciels libres auprès de leurs clients (Je pense bien sûr ici à IRTS).

4.2.3. Embarqué - temps réel

Un logiciel embarqué n'a pas forcément de contraintes temps réel. On peut s'attendre à ce qu'une machine doive suivre un tel comportement si elle est reliée au monde réel, via des entrées / sorties. Les événements extérieurs doivent être traités en un temps déterminé pour produire une réaction appropriée.

Un système fonctionne en temps réel lorsqu'il est capable d'absorber toutes les informations d'entrée sans qu'elles soient trop vieilles pour l'intérêt qu'elles présentent, et par ailleurs, de réagir à celles-ci suffisamment vite pour que cette réaction ait un sens.[2]

ou

Un logiciel ou un sous-ensemble de logiciel est dit temps réel ou déterministe si son exécution a lieu en un temps déterminé et connu.
Un système d'exploitation est dit temps réel ou déterministe s'il permet d'exécuter des tâches logicielles en temps réel. Cette notion est indépendante de la vitesse de calcul.

Pour assurer le déterminisme des tâches, un système d'exploitation doit être soit monotâche, soit multitâche et totalement préemptif. La préemptivité induit un autre concept, la notion de priorité. En effet, un noyau multitâche préemptif[3] recevant une

[2] ABRIAL et BOURGNE
[3] Dont la tâche courante peut être interrompue.

interruption, doit répartir de nouveau le temps CPU entre les tâches. Cette répartition est effectuée par le moniteur système (*scheduler*) selon un critère :

- pour un OS temps partagé, c'est le temps CPU déjà accordé à la tâche,
- pour un OS temps réel "mou" c'est l'ordre d'arrivée de l'interruption,
- pour un OS temps réel "dur" c'est l'ordre d'arrivée de l'interruption mais aussi la priorité des différentes tâches s'exécutant.

Linux à été implémenté à partir des spécifications POSIX pour construire un système d'exploitation temps partagé. En revanche, la norme POSIX.1b spécifie des extensions temps réel au système de base (POSIX.1). Ces extensions, disponibles dans Linux, sont entre autres : sémaphores, timers, queues de messages et scheduler préemptif gérant les priorités. Il y a là tout ce qu'il nous faut!!!

Mais, **le noyau Linux est non préemptif**... Une tâche peut être interrompue et mise en attente par le moniteur, selon sa priorité. En revanche, un appel système ne peut être interrompu, il peut durer jusqu'à quelques centaines de milli-secondes (ex : fork()). L'implémentation d'un comportement temps réel "dur" avec le noyau Linux n'est possible qu'au prix de sévères modifications du code du noyau. Cependant, des projets ont vu le jour pour améliorer les performances temps réel "mou" de Linux notamment dans le domaine du son. On peut trouver de tels projets en partant de `http://www.crosswinds.net/~linuxmusic/lowlatency.html` ou depuis la société MontaVista `http://www.mvista.com`.

L'approche "bas niveau" sera discutée dans la section 5.2 consacrée à l'architecture de RTLinux...

4.3. Distributions et outils

4.3.1. Divers projets libres

AlfaLinux est basée sur la distribution Slackware et tient sur deux disquettes. `http://alfalinux.sourceforge.net/alfaeng.php3`

BYLD Build Your Linux Disk : un paquet qui permet de générer des mini distributions Linux sur une disquette. `http://get.to/byld/`

CClinux tenant sur une disquette, il utilise un maximum d'outils GNU.`http://www.CosmicChaos.com/CClinux/`

DosLinux : un petit système d'exploitation Linux qui peu être installé au dessus d'un système Dos existant. `http://gwyn.tux.org/pub/people/kent-robotti/index.html`

Emblin : Le but de ce projet est de créer un système Linux le plus petit possible contenant le maximum de fonctionnalités (serveur HTTP + CGI, FTP, Telnet ...), le tout tenant sur une seule disquette. `http://www.master.cit.be/c27/index.html`

LEM : Linux EMbedded est une petite distribution basée sur la Mandrake. Elle fait tenir sur moins de 8Moctets un système complet contenant même Xwindow. Sur son site, il est décrit comment ajouter le navigateur Netscape 4 et un serveur web...

Cette distribution, que j'ai utilisée lors de l'appel d'offre STNA, est formidable car elle met tous les outils nécessaires à la disposition du développeur. La liste de diffusion proposée est assez active et le groupe est un acteur majeur des Linux embarqués. Je pense qu'un système graphique plus petit serait en outre plus adapté que Xwindow. `http://www.embedded-linux.org`

Linux Router Project Un projet centré sur le réseau, comme son nom l'indique ! `http://www.linuxrouter.org`

Tomsrtbt disquette de boot/secours/circonstances critiques (mauvais paramétrage de LILO, crash disque...). Son insertion sur une disquette de 1,77Moctets lui permet de contenir beaucoup d'outils (avec les pages de man). Tournant depuis un disque mémoire il permet d'utiliser tout PC comme une station de travail sous GNU/Linux. Il contient une configuration réseau, un éditeur de textes, les utilitaires mount, gzip... La trousse de secours indispensable. Il donne de plus une bonne idée de la construction d'une mini-distribution. `ttp://www.toms.net/rb/`

4.3.2. Solutions d'entreprises commerciales

Les entreprises proposant une solution Linux embarqué :

BlueCat de Lynx Real-Time Systems Inc. entre dans la stratégie d'éditeur de se rapprocher du monde des logiciels libres. Lynx nous dévoile sa stratégie Lynx Linux Initiative et LynuxWorks... Lynx Inc. met aussi en place une compatibilité binaire entre Linux (format ELF) et son RTOS LynxOS 4.0. Sur le diagramme suivant, on peut se rendre compte des possibilités qu'offre BlueCat Linux et des outils qu'il fournit. Le manuel utilisateur est extrêmement précis et guide le concepteur à travers des exemples pertinents.

Organigramme du développement sous BlueCat Linux. `http://www.bluecat.com`

Embedix : Lineo développe et vend des composants logiciels et des solutions de systèmes embarqués en interaction avec Internet. Lineo détient Embedix (e*mbedded Linux OS*), Embrowser (*embedded web browser*), DR DOS et de nombreuses technologies de l'embarqué. En rachetant Zentropix, il s'impose comme un acteur majeur de l'enfoui et du temps réel à base de Linux.

Hard Hat Linux est un produit commercial de la société Monta Vista qui s'attache à faciliter le développement de systèmes embarqués. Elle initie de nombreux travaux pour améliorer les performances temps réel du noyau Linux (modification du scheduler). `http://www.mvista.com` .

Les entreprises proposant une solution Linux temps réel :

- Finite State Machine Labs `http://www.fsmlabs.com`
- Lineo `http://www.lineo.com` un des acteurs majeurs du Linux embarqué en très forte expansion. Produit : Embedix `http://www.lineo.com/products/embedix.html`. Lineo rachète de nombreuses sociétés en relation avec les Linux embarqués dont Zentropix. `http://www.zentropix.com` un acteur principal ou une petite société française INUP `http://www.inup.com` spécialisée dans la mise en place de clusters haute disponibilité sous Linux.
- Cygnus `http://sourceware.cygnus.com` pour les produits EL/IX `http://sourceware.cygnus.com/elix/` une API se voulant unificatrice des Linux temps-réels et eCos `http://sourceware.cygnus.com/ecos/` un micro noyau temps-réel configurable.
- Synergy Microsystems `http://www.synergymicro.com/linux.html` et sa distribution basée sur RTLinux du FSM Labs : Industial Linux.
- ...

4.3.3. Les utilitaires de taille réduite

ae petit éditeur de textes.

ash très petit Bourne *shell*.

Busybox petit programme binaire qui fournit la quasi totalité des commandes POSIX de base i.e. cat, cp, mv, ls, cd, mount... Il est employé dans la plupart des mini-distributions : Linux Router Project, LEM et par l'installation de Debian... Son développement est sponsorisé par Lineo.

elvis-tiny un autre éditeur de textes de style vi.

iproute remplaçant de ifconfig, route, etc. Petit, il apporte aussi des fonctions avancées.

TinyLogin suite d'utilitaires permettant de se "loger" sur le système, et faire la maintenance des utilisateurs. Sécurisé il utilise le mécanisme de "*shadow password*".

4.3.4. Outils documentaires

Il est fort intéressant, lorsque l'on développe une solution à base de logiciels libres, de disposer d'une liaison Internet pour avoir accès aux forums de discussions et aux listes de diffusions. En effet, on trouve rapidement une réponse en posant une question sur un des forums/listes dédiés aux Linux temps-réels car ce milieu est passionné et enthousiaste. Attention tout de même de ne pas poser une question à réponse triviale : lire la documentation avant toute requête.

Internet est une base de documentation inépuisable et effervescente ! Il est impossible de faire le tour de tous les papiers mis à la disposition du développeur. Il faut donc avoir quelques flotteurs pour ne pas se noyer et "surfer" confortablement d'un renseignement à l'autre...

Les outils les plus efficaces, à la disposition du développeur sont principalement :

- Les livres, peu nombreux et assez chers, ils ne correspondent pas toujours aux besoins et sont souvent édités avec un retard certain par rapport à ce que l'on peut trouver sur Internet. Mais, certaines maisons d'édition n'hésitent pas à publier d'excellents ouvrages éclairant les développements récents de la Communauté ; je pense aux éditions *O'Reilly* `http://www.oreilly.com` ou `http://www.editions-oreilly.fr`. Un livre sur les Linux temps réel doit sortir chez eux, avant le quatrième trimestre 2000... Les éditions *Eyrolles* `http://www.eyrolles.com` ou *CampusPress* `http://www.campuspress.fr` produisent aussi de très bons ouvrages.
- Internet et les "portails" classiques : `http://www.linuxfr.org` `http://www.slashdot.org` `http://www.linuxdevices.com` `http://www.linuxembedded.com`.
- Les listes de diffusion ou "*mailing lists*" auxquelles on peut souscrire à partir de nombreux sites Internet.
- Les nouvelles ou "*news*" (un groupe comp.os.linux.embedded est sur le point de se créer !).

Exemple : Pour le temps réel...

Realtimelinux.org a une "*mailing lists*" permettant de se tenir au courant des nouveautés dans le monde des Linux temps réel. Chacun peut émettre un point de vue, poser une question ou demander de l'aide... Pour souscrire à cette liste, il suffit de se rendre sur le site `http://realtimelinux.org` et de suivre la procédure décrite (Archives des mailing lists `http://www.realtimelinux.org/archives/`). La plus intéressante des ces listes et la plus active est celle intitulée "realtime".

Un autre groupe, plus proche du produit RTLinux et extrêmement actif est la liste de diffusion maintenue par rtlinux.org. Elle apporte de bons renseignements au jour le jour, il me semble indispensable d'y souscrire : `http://www.rtlinux.org`

5. Les Linux Temps Réel

5.1. Introduction

Le monde des Linux temps réel profite de la popularité du système d'exploitation GNU/Linux. Il bénéficie donc de l'immense base de documentation que représente Internet. D'autre part, sa particularité et son exploitation dans des conditions bien précises de l'informatique industrielle lui confère une place bien à part du grand tumulte Linux ! Cette place à l'écart ne l'empêche pas de faire parler de lui... Plusieurs initiatives ont vu le jour pour adapter Linux au temps réel "dur".

5.1.1. Les signes avant-coureurs d'un succès à venir :

- Le marché de l'embarqué en pleine explosion du fait de la croissance des "*Internet appliances*"[1]. Mais attention, embarqué ne veut pas dire à coup sûr temps réel !
- Les grandes entreprises historiques du monde Linux s'intéressent de près au temps réel : RedHat `http://www.redhat.com` rachète Cygnus `http://sourceware.cygnus.com`, Lineo `http://www.lineo.com` rachète Zentropix `http://www.zentropix.com`...
- Pareillement, Lynx `http://www.lynx.com` s'inscrit dans une politique fortement orientée vers Linux : BlueCat `http://www.bluecat.com` et LynxOS 4.0
- Les autres acteurs de l'informatique embarquée / temps réel plaident en faveur d'un rapprochement ou d'une complémentarité entre leur système propriétaire et Linux[5]. On peut à cette occasion citer QNX `http://www.qnx.com` qui a, ces derniers mois, rapproché sa stratégie de celle des logiciels libres : libération du code source de son noyau Neutrino et mise à la disposition de certaines applications... Cette politique est appuyée par une volonté de produire une API commune aux Linux temps réel / embarqués se recadrant sur les normes POSIX.

5.1.2. Le choix...

L'une des caractéristiques du Logiciel Libre est... sa liberté ! Dans le domaine émergeant des Linux temps réel, la multitude des offres requiert une étude approfondie des avantages et inconvénients de chaque système. Cette étude doit aussi prendre en compte

[1] *Internet appliances* : divers appareils connectés à Internet mais ne correspondant pas au schéma classique d'un ordinateur (téléphone mobile, minitel web, refrigérateur...)

la notoriété d'un produit, ses soutiens éventuels et leur solidité, enfin le support de mise en oeuvre et de documentation disponible sur Internet.

Le choix de RTLinux par rapport à RTAI s'est imposé dans cet environnement industriel pour plusieurs raisons :

- **Notoriété** : RTLinux a été imaginé en 1997 dans le département informatique du *New Mexoco Institute of Mining and Technology*. Il a été utilisé dans de nombreux projets universitaires et industriels et bénéficie donc d'une large confiance. Ainsi, sa base de documentation et d'aide en ligne est conséquente. Le nombre de programmes actuellement disponibles sur Internet et les listes de diffusion dédiées spécialement à RTLinux ont aussi orienté le choix. On peut partager ce point de vue avec `http://www.student.dtu.dk/~u990873/`.
  RTAI est créé par l'équipe de Paolo Mantegazza au département d'ingénierie aérospatiale du *Politecnico di Milano* à partir de RTLinux. Ce noyau est lui aussi utilisé en entreprise (Zentropix/Lineo) et tend à rattraper son retard de notoriété.
- **Rapidité de développement** : RTLinux est composé d'une API de moins de 70 fonctions dans sa version 2.2. Il est donc facile de se familiariser avec ce système et de développer rapidement une application. De plus, depuis la version 2, RTLinux se rapproche d'une implémentation des normes POSIX (voir 5.2.2).
  RTAI offre des fonctions beaucoup plus complexe et se rapprochant de celles d'un RTOS classique (mailbox, mutex, variables condition, mémoire partagée et RPC...). De ce fait, il est plus difficile d'approche. D'autre part, RTAI est resté sur la base d'une API propre (proche de celle de RTLinux V1) avec seulement une compatibilité POSIX.1c qui correspond au modèle de Threads POSIX.
- **Processeurs supportés** : RTLinux intègre depuis sa version 2 le support de machines multiprocesseurs (SMP) et est en phase de validation sur des architectures PPC (Power PC). Un portage sur MIPS est aussi envisagé.
  RTAI a été le premier des deux systèmes à avoir un support SMP mais reste cantonné à une architecture x86.

Du fait des ses nombreuses fonctionnalités, de son support du temps réel "mou" depuis l'espace utilisateur, de sa gestion du système de fichier /proc (information système), et du développement de modules PERL, RTAI est un système extrêmement intéressant et particulièrement adapté à un développement dans un cadre universitaire ou de recherche et développement.

RTLinux apparaît plus adapté à l'industrie bien que, une fois encore, RTAI prenne de plus en plus d'importance dans ce secteur.

Finalement, les deux systèmes sont excellents et il ne faut en négliger aucun ! La compétition acharnée qu'ils se livrent parfois ne fait que stimuler la créativité de leurs "contributeurs".

5.1.3. Portail web pour le monde des Linux temps réel

- LinuxDevices The embedded Linux portal `http://www.linuxdevices.com`
- realtimelinux.org `http://www.realtimelinux.org`

5.1.4. Les sites des "distributions" historiques

- RTLinux Home Site `http://www.rtlinux.org` aussi présent sur ...
- RealTimeLinux `http://www.rtlinux.com` développement et support assuré par Finite State Machine Labs `http://www.fsmlabs.com`.
- Pour RTAI : `http://www.rtai.org`

5.2. RTLinux : Architecture

RTLinux est un système d'exploitation dans lequel cohabitent un micro-noyau temps réel et le noyau Linux. L'intention des concepteurs est de continuer d'utiliser les services de haut niveau du système à temps partagé, tout en permettant de fournir un comportement déterministe dans un environnement où le temps de latence est extrêmement faible. En effet, les systèmes temps réel ont souvent besoin d'un support réseau, d'une interface graphique, d'un système de fichier et éventuellement d'un accès à une base de données. Ceci peut être fourni par un système d'exploitation "monobloc" où tout service est intégré dans un environnement temps réel, comme LynxOs. Une autre solution est de consacrer une machine aux applications temps réel et y ajouter des services non temps réel comme cela est géré dans VXworks.

La troisième voie adoptée par RTLinux est celle qui permet à un micro-noyau d'exécuter le véritable noyau Linux comme sa tâche de plus faible priorité. RTLinux implémente en fait une sorte de machine virtuelle pour que le noyau Linux standard soit complètement préemptif. Dans cet environnement, toutes les interruptions sont initialement prises en compte par le noyau temps réel et sont passées à Linux seulement s'il n'y a pas de tâche temps réel à exécuter.

FIG. 5.1. – Les différentes couches d'abstraction : RTLinux comme machine virtuelle pour Linux.

Pour minimiser les changements dans le noyau Linux, les concepteurs ont *émulé le contrôleur d'interruptions matériel*. Ainsi, quand Linux désactive une interruption, celle-ci est quand même prise en compte par RTLinux et aiguillée vers une file d'attente si elle n'est pas utile au niveau du micro-noyau. Lorsque Linux rétablit la possibilité de traiter les interruptions, celles arrivées entre temps sont disponibles et peuvent être traitées par la routine de gestion d'interruptions de Linux (*handler* d'IT Linux dans la figure 5.1). Les codes assembleurs des processeurs i486 *cli*, *sti* et *iret* sont remplacés dans le noyau Linux par des macro-instructions *S_CLI*, *S_STI* et *S_IRET*. Toutes les interruptions

matérielles sont attrapées par l'émulateur qui distribue des interruptions logicielles (s*oft interrupts*).

Les tâches temps réel suivent les spécifications de la norme POSIX.1c décrivant les *threads*. Elles peuvent êtres application utilisateur, service fourni par l'environnement RTLinux ou pilote de périphérique. Chargées comme module noyau, elles s'exécutent en suivant les ordres dictés par le *scheduler*. Le temps CPU est fourni à la tâche de plus haute priorité (*scheduler* de base RTLinux).

Ces "*kernel threads*" disposent des droits entiers sur la machine et peuvent accéder à toutes les ressources. Ils occupent tous le même espace d'adressage (celui du noyau !) et se chargent dynamiquement. Il résulte de ces trois points qu'il est délicat de développer un tel module. En effet, la moindre erreur de programmation, vite advenue en C du fait de la manipulation de pointeurs, est fatale au système. En revanche, cette architecture permet d'augmenter les performances en éliminant les temps de changement de niveau de protection et en raccourcissant le temps de changement de contexte. Toutes les ressources d'une tâche RTLinux sont allouées statiquement (exemples : mémoire, fifo).

La règle primordiale des développeurs sous RTLinux doit être :

Si un service est intrinsèquement non temps réel, il doit être fourni par Linux et non par les modules RTLinux[1].

Ceci s'applique naturellement aux pilotes de périphériques (*device drivers*) qui, s'ils n'ont aucune contrainte temps réel à tenir, restent inchangés. Ils doivent tout de même être recompilés dans l'environnement RTLinux (-D__RTL__ et rtl.mk) s'ils emploient les routines *cli / sti*.

5.2.1. Composants du système

RTLinux version 2 est structuré comme une partie centrale minimale sur laquelle vient se charger tout une collection de modules qui fournissent des services optionnels ou des niveaux d'abstraction. Ces modules sont par exemple :

1. *rtl_sched* un moniteur fonctionnant sur un mode exclusivement prioritaire.
2. *rtl_time* qui gère l'horloge processeur et fournit des services autour de l'utilisation du temps (timers, routine de traitement d'interruptions...).
3. *rtl_posixio* fournit une API de style POSIX (read/write/open...) pour interfacer les pilotes de périphérique.
4. *rtl_fifo* connecte les mondes RTLinux et Linux au travers d'une interface "device", permet l'échange de données et commandes.
5. *semaphore* est un paquetage permettant d'utiliser mutex, sémaphores et IPC bloquantes avec/sans échéances.
6. *mbuff* permet d'utiliser des composants de mémoire partagée entre les processus Linux et RTLinux.

Le système est donc pensé pour être transparent, modulaire et extensible. Par exemple, on peut trouver un scheduler implémantant les modes RoundRobin ou FIFO de la norme POSIX, comme module de remplacement, codé par un membre de la Communauté.

5.2.2. RTLinux et POSIX

Le consortium de normalisation POSIX entretient des relations d'attraction / répulsion avec le monde du temps réel et les concepteurs de RTOS. Il est souvent dit par la communauté temps réel que la normalisation de l'interface est un frein à la performance. Le processus est initié par les grands éditeurs d'UNIX qui se préoccupent de gros systèmes ou de systèmes de bureau. L'API proposée, même pour les extensions temps réel (POSIX.4 *Draft* 14 = P1003.1b), est assez lourde à mettre en oeuvre : de nombreux éditeurs s'en démarque. Par exemple QNX se contente de POSIX.1 (base : processus, fichiers, pipes...) et .2 (shell et utilitaires) ; LynxOS V2.2 y ajoute seulement un support de POSIX.4 *Draft* 10.

De plus, de part l'architecture originale de RTLinux, de nombreuses implémentations du standard sont impossibles (système de fichier). Un compromis à été trouvé par les concepteurs de RTLinux à savoir l'utilisation du "*Minimal Realtime System Profile*" (POSIX.13 *Draft* 9) reprenant les spécifications propres aux threads (POSIX.4a *Draft* 8 = P1003.1c) mais remplace la gestion d'un système de fichier par une simple interface d'entrées / sorties, suffisant pour un noyau de cette taille. RTLinux ajoute aussi un support SMP compatible avec cette norme.

Le principe de base de RTLinux reste, selon ses concepteur, de ne pas céder à la compatibilité une quelconque perte de performance.

Cependant cet effort de normalisation est fait pour facilité le portage d'applications dans l'environnement RTLinux, pour bénéficier du "vivier" de programmateurs déjà habitués à POSIX et enfin, pour simplifier la compréhension du code et l'intégration dans l'environnement GNU/Linux. Le module *rtl_posixio* est donc à utiliser sans modération lors d'écriture de pilotes de périphériques !

5.3. Installation

L'installation se passe généralement sans encombre du moment où l'on a déjà compilé un noyau Linux. Les documents livrés avec la "distribution" sont assez bien faits. Une attention toute particulière va aux guides d'installation commentés par Phil Wilshire <philwil@on-ramp.ior.com> que l'on peut trouver à la racine de l'archive (du moins à partir de la version 2). Ce document est présent dans le recueil joint en annexes.

5.4. Prise en main

5.4.1. Exemples

Une attention particulière sera portée à l'interface avec le matériel (exemples de drivers). Sachant qu'il est demandé d'étudier des drivers pour carte ISA puis PCI, les modes de programmation de telles cartes seront observés avec attention.
Exemples et drivers permettent d'aborder un aspect de la programmation dans un environnement RTLinux :

Ex. hello : Le traditionnel "Hello World" pour RTLinux. Il permet de comprendre les mécanismes de base de programmation et chargement des modules dans le kernel temps réel. L'installation d'un module dans le noyau passe par l'exécution de la fonction *init_module()*, contenant l'allocation et l'initialisation des ressources systèmes. Son pendant est la fonction *cleanup_module()*.
*Document*s : $RTLDIR/exemples/hello/README. Si c'est le premier exemple étudié, il est bon de regarder le Makefile et le fichier, définissant des constantes de compilation, généré par RTLinux : rtl.mk.

Ex. frank : Illustre le mode de passage de messages entre une application temps réel et une application utilisateur fonctionnant sous Linux. Cette communication, concernant des données et des commandes, s'effectue par l'intermédiaire de RT-FIFO. Cet exemple est beaucoup moins trivial qu'il en a l'air ! Il permet en effet de comprendre en détail le fonctionnement des RT-FIFO et du mécanisme *d'interruption logicielle* et de *"handler"* associés. Du coté Linux, le programme de récupération des données utilise l'interface POSIX implémentée pour les RT-FIFO (open, read,write, close). Cette application est aussi intéressante car elle implémente un *select()* : mécanisme d'écoute sur les deux RT-FIFO ouvertes, chargées de communiquer avec ce processus Linux.

Ex. mesurements : Calcule la latence min et max (donc la gîte(jitter) = max(ensemble des max-min)). Une tâche temps réel périodique est lancée, on récupère le temps prévu pour l'ordonnancement et le temps d'horloge, au moment de ce changement de contexte. En faisant la différence des deux, on trouve le temps de latence.

Ex. regression : Comme dans l'exemple précédant, il s'agit ici de mesurer le temps de latence de différentes façons. Le module est chargé en prenant pour base de temps, soit un timer RTLinux, soit directement l'horloge temps réel (RTC) du PC. Le temps écoulé fait claquer une interruption prise en compte par le handler qui calcule la latence et l'affiche ou la transmet à un processus Linux de présentation. Les interruptions prises en compte sont soit matérielles dans le cas de l'utilisation de la RTC, soit logicielles : utilisation de timers temps réel.

Ex. sound : Mise en place d'un handler pour les interruptions provenant de l'horloge temps réel du PC (RTC real time clock), préalablement programmée. Le handler se charge d'allumer ou éteindre le haut parleur du PC en fonction du niveau des échantillons reçus par la RT-FIFO. En effet, un fichier au format *.au est envoyé au processus temps réel qui récupère, dans la fifo, un échantillon de 8 bits à chaque

interruption de la RTC programmée à #8kHz. Un petit seuillage permet de déterminer si le HP doit être allumé ou non. C'est ainsi qu'en prêtant l'oreille, on entend Linus Torvalds parler...

Ex. mmap : Permet d'accéder à la mémoire physique, depuis l'espace noyau RTLinux ou l'espace utilisateur Linux. C'est une sorte de driver du périphérique mémoire physique.

Ex. mutex : Une tâche RTLinux et un processus Linux se disputent un sémaphore d'exclusion mutuelle. La tâche Linux est en fait implémentée dans la fonction init_module() qui crée et prend le mutex avant de lancer la tâche temps réel. Cette dernière essaie de prendre le sémaphore, sans succès, jusqu'à ce que la tâche Linux l'ait lâché.

Ex. fp : Décrit les capacités de calcul en virgule flottante de RTLinux. Nous avons besoin d'un module indépendant pour effectuer de tels calculs.

Ex. v1api : Non étudié : permet d'utiliser du code écrit pour RTLinux version 1. Il faut pour cela compiler RTLinux avec l'option CONFIG_RTL_USE_V1_API activée dans include/rtl_conf.h.

Driver rt_com : Communication par ligne série proposant une API POSIX. Permet de se familiariser avec la construction d'un driver complet et l'utilisation du module posixio. La communication avec les processus Linux se fait par l'intermédiaire de RT-FIFO, l'intégration de leur gestion dans un driver est typique de ce mode d'utilisation. Des tests sont disponibles : testcom.c et com_posix.c.
*document*s : $RTLDIR/drivers/rt_com/rt_com.html + README.RTL.

Driver mbuff : pilotant /dev/mbuff qui est le périphérique de mémoire partagée.
document : $RTLDIR/drivers/mbuff/MANUAL.

Module IPC : Permettant d'utiliser sémaphores (d'exclusion mutuelle et à compte), queues de messages et supplément de fonctionnalités pour les RT-FIFO : mode bloquant et *timeouts*.
document : $RTLDIR/sémaphores/README.

5.4.2. Utilisation du port parallèle

La meilleure façon de se familiariser avec un système temps réel est quand même de l'utiliser pour sa raison d'être : l'interaction avec le "monde réel". Or, quelle interface avec l'extérieur est mieux adaptée que le port parallèle pour faire des entrées / sorties numériques ? Disponible sur la plupart des PC il est simple à programmer ; on peut néanmoins s'initier à l'utilisation de deux modes de fonctionnement : *polling* et interruptions tout en mettant en pratique les fonctions primordiales de l'API RTLinux. Attention toutefois à ne pas endommager le composant car il est implanté sur la carte mère : un court circuit ou une mauvaise alimentation en 5V pourrait gravement l'endommager !

La manière la plus simple de mesurer l'état de la broche ACK du port parallèle est de créer une tâche périodique qui se charge de lire l'état du bit. Quand on détecte que la valeur a changé, on envoie un message à un processus Linux par l'intermédiaire d'une

FIG. 5.2. – Les registres de contrôle / commande du port parallèle.

Numéro de bit	Nom registre statut	Nom registre commande
7	BUSY	Unused
6	/ACK	Unused
5	NOPAPER	Input mode enable
4	SELECTED	Interrupt enable
3	/ERROR	/SELECT
2	Undefined	/INITIALIZE
1	Undefined	/AUTOFEED
0	Undefined	/STROBE

TAB. 5.1. – Nom des bits des registres de contrôle et de statut

RT-FIFO. Celui-ci doit être en attente de lecture sur cette FIFO. Le processus Linux peut alors afficher le changement d'état de la broche.

Un autre moyen de mesurer les changements sur la broche ACK consiste à installer un gestionnaire d'interruptions sur l'interruption #7. Cette autre solution dispense de l'usage d'une tâche périodique.

Dans les deux cas, la fonction init_module s'occupe de lancer la tâche. Il suffit de créer les FIFO, la routine de gestion d'interruption (éventuellement), la tâche et enfin de définir sa priorité et périodicité (s'il y a lieu). La valeur de la période est exprimée, comme toutes les grandeurs de RTLinux, en nano-secondes.

Ceci permet de s'initier au fonctionnement de l'API RTLinux pour ce qui est de la gestion du temps, de la création de tâches périodiques. Pour l'autre méthode on aborde la prise en compte d'interruptions matérielles et la mise en place de routines de traitement d'interruptions. Comme les applications communiquent par RT-FIFO, ce mécanisme est aussi... maîtrisé !

5.5. Codage, test d'un pilote carte TOR

L'objectif est ici de mener un développement en interaction forte avec le matériel, de bout en bout. En effet, j'avais à apprendre comment construire un driver, à utiliser les ressources de RTLinux et à coder les programmes de tests de ce driver. Le document présent en annexe est le manuel d'utilisation de ce driver. Pour pouvoir être utilisé par la Communauté, ce document est rédigé en *anglais* ; l'ensemble des programmes de tests, versions de drivers et documentations sont disponibles sur Internet pour permettre

l'utilisation et l'amélioration de mon travail. `http://www.chez.com/noglitch/`

Ce que l'on peut tirer de ce premier développement complet sous RTLinux est que cet environnement est tout à fait homogène, stable et simple d'utilisation. Il est très important, comme il est mentionné dans tout manuel sur RTLinux, de bien délimiter la partie temps réel et la partie qui ne l'est pas. En effet, Il s'agit de minimiser la partie de code "critique" et d'utiliser au maximum les facilités offertes par un développement dans l'espace utilisateur de Linux (déboguage, protection mémoire, librairies, graphismes...) .

L'API de style POSIX disponible dans l'espace RTLinux, facilite la compréhension du code écrit pour utiliser le driver. Toutefois, elle peut entraîner des confusions avec l'utilisation dans l'espace Linux (noyau ou utilisateur). Je conseille d'avoir les idées claires sur la situation du code que l'on est en train d'écrire (espace RT ou Linux, noyau/utilisateur). Un schéma est toujours utile dans de telles situations...

FIG. 5.3. – Fonctionnement du pilote de carte PDISO16 dans l'environnemnt RTLinux.

5.6. Portage et test d'un pilote carte son

Pour ce driver il a fallu mettre en oeuvre beaucoup plus de notions de temps réel. La partie documentation a donc été conséquente pour l'approche du signal sonore traité par une carte son dans un environnement GNU/Linux puis pour les changements à apporter en vue d'un comportement déterministe.

5.6.1. Le son sous GNU/Linux

Théorie :

Le son est un phénomène analogique ; il se traduit par une valeur quelconque sur une échelle continue. Les ordinateurs fonctionnent en numérique : ils utilisent des valeurs discrètes.

Les cartes son fonctionnent grâce à un composant appelé Convertisseur Analogique-Numérique (A/N ou ADC en anglais) afin de convertir les tensions correspondantes aux ondes sonores analogiques en valeurs numériques qui peuvent alors être stockées dans la mémoire de l'ordinateur. De même, un Convertisseur Numérique-Analogique (N/A ou DAC en anglais) convertit les valeurs numériques en une tension analogique qui peut alors être amplifiée pour "attaquer" un haut-parleur et produire du son...

La conversion analogique-numérique (échantillonnage ou *"sampling"* en anglais) génère des erreurs. Deux facteurs déterminent la qualité du signal échantillonné par rapport

au signal initial.

La fréquence d'échantillonnage (*sampling rate*) est le nombre d'échantillons réalisés par unité de temps (exprimée en Hertz). Une petite fréquence d'échantillonnage produira une représentation moins fine du signal analogique d'origine, dans le sens où les fréquences aiguës ne seront pas ou mal restituées. La théorie (Théorèmes d'échantillonnage ou de Shannon) nous enseigne que pour restituer correctement une fréquence sonore de Fm Hz, il faut effectuer un échantillonnage au moins égal à 2 x Fm Hz. Ainsi, l'étendue du spectre sonore audible par un humain (qqHz à 20 kHz) sera bien restituée par un échantillonnage à 44100 Hz (norme CD-AUDIO) .

La dynamique est le nombre de bits d'échantillonnage qui conditionne l'étendue des valeurs utilisables pour représenter chaque échantillon. Il s'agit du rapport entre le niveau sonore le plus fort restituable sans distorsion, et le niveau de souffle ou de bruit inhérent au matériel de restitution. Cette plage dynamique est exprimée de manière logarithmique en décibels (dB). En théorie, lorsque l'on ajoute un nouveau bit de codage pour chaque échantillon, on double cette plage, ce qui correspond à un gain en dynamique de 6 dB.

Pratique :

Les cartes son utilisent typiquement un codage sur 8 ou 16 bits pour des fréquences d'échantillonnage allant de 4000 à 44100 (ou même 48000 Hz). L'échantillonnage pouvant être réalisé sur une voie (mono) ou deux (stéréo) (voire plus (8, 16...) pour des cartes professionnelles).

Les pilotes de carte son utilisent des mécanismes bien particuliers pour fournir, au bon moment, les échantillons nécessaires au convertisseur de la carte...

5.6.2. pourquoi le son requiert un comportement temps réel ?

Quelques petits calculs s'imposent à cet endroit de la discussion ; en effet, il faut savoir quel est le débit de données à transférer dans la carte son, pour que celle-ci puisse avoir l'échantillon qu'il lui faut au bon moment. Un échantillon manque, se produit alors un "crépitement" du signal sonore.

Débit = dynamique * fréquence d'échantillonnage * nombre de voies

- Signal à 8kHz/codé sur 8 bits/mono = 8000*1*1 = 8000 octets/s [2]
- Signal à 44,1kHz/codé sur 16 bits/stéréo = 44100*2*2 = 176400 octets/s (format qualité CD) [3]

Avec le débit de données et la taille du buffet[4] alloué au stockage, on peut déduire la fréquence à laquelle il faut remplir le *buffer* (problème classique de la baignoire qui se vide à débit constant, cher à tous les collégiens).

[2] soit 4,5Mo pour 10mn de musique

[3] Soit 100Mo pour 10mn de musique (!) Heureusement qu'il existe des systèmes de compression pour les fichiers sonores ! Mais ceci n'est pas notre propos !

[4] *buffer* en anglais et dans la suite de mon propos.

Il ne faut pas de délai entre la fin d'un paquet d'échantillons contenus dans un *buffer* et l'échantillon suivant d'où, l'emploi de la technique des *buffer* multiples (*Multi-buffering*). L'application tient toujours un *buffer* rempli à la disposition du driver, qui peut s'en emparer lorsqu'il a vidé le précédent : on parle en fait d'un *buffer* découpé en plusieurs *fragments* (en général 3). Il faut tout de même savoir que l'application a souvent des traitements à faire sur les échantillons avant de les fournir au driver.

Exemple : avec un débit de données de 176400 octets/s et un *buffer* de 2 fragments de 4Koctets, l'application dispose de 4096 octets soit de 23ms pour effectuer son traitement et remplir le *buffer* ce qui est peu ! Il faut alors l'agrandir. Mais, le chargement d'un fragment plus grand demande du temps car il faut transmettre des données, de l'espace utilisateur, au noyau (fonctions *write()* de l'API son). Cela implique une augmentation du temps de latence pour toutes les autres opérations du système. Il faut donc augmenter la fragmentation du *buffer* : les transferts de données seront alors plus nombreux mais de plus petite taille.

Le pilote sous Linux calcule en général la taille et le nombre des fragments tel que la latence soit de 0.5s pour une restitution et de 0.1s pour une acquisition. Sous RTLinux, les données sonores résidant sur le disque sont envoyées, via une RT-FIFO, à une tâche temps réel. Cette dernière utilise l'API son classique (OSS : Open Sound System) pour communiquer avec le driver modifié. Je conseille, avant tout développement dans le domaine du son, de lire attentivement les documents mis à notre disposition sur ce site `http://www.opensound.com` : Open Sound System TM Programmer's Guide.

5.6.3. Le portage du pilote carte son *Sound Blaster 128 PCI*

Le driver pour la carte son *Sound Blaster 128 PCI* est celui développé pour la puce d'Ensoniq es1371. Il se trouve dans le répertoire consacré au son dans l'arborescence du noyau Linux, sous le nom es1371.c (!). Le piège à éviter est d'utiliser le driver es1370, ou de chercher celui s'appelant comme la puce présente sur la carte son i.e. : es1373!!! Pour utiliser ce driver sous Linux, il suffit de le sélectionner comme module lors de la configuration du noyau. Il peut aussi être utilisé quand on travaille sous RTLinux, en agissant exactement comme décrit précédemment, lors de la compilation du noyau "patché" pour RTLinux. Le driver ne fonctionnera que dans l'espace Linux et l'on ne pourra lui soumettre aucune contrainte temps réel : il a les désavantages de toutes les tâches tournant sous Linux et peut aussi être préempté par n'importe quelle tâche RTLinux.

Pour que ce driver ait un comportement déterministe, il faut l'adapter à l'environnement RTLinux. Ce qui signifie modifier :

- les mécanismes de gestion d'interruptions (s*ynchronise_ irq, signal_ pending...*)
- ce qui concerne la protection de ressources dans le noyau Linux (*spin_ lock, spin_ lock_ irq...*)
- la communication de données entre espace utilisateur et noyau (*copy_ to_ user, put_ user...*)
- l'interface style POSIX (*open, read...*)

Pour faciliter le travail et en plus permettre au driver modifié de fonctionner aussi bien sous RTLinux que sous Linux (voire sous RTAI), plusieurs initiatives ont vu le jour.

Driver Programming Interface (DPI Version 0.1.0 beta) pour des drivers RTLinux/Linux développée par David Olofson `mailto:david_olofson@hotmail.com`. C'est l'API que j'ai utilisée car le portage du driver es1370 était fait. Il a donc suffi de suivre l'exemple et de modifier le nom des fonctions... Le résultat semble fonctionnel mais le système se bloque au déchargement du module, je n'ai pas corrigé ce défaut car un tel problème est long à résoudre. `http://www.angelfire.com/or/audiality/`

Comedi : *Control and Measurement Device Interface* for Linux, développé par de nombreux contributeurs dont le principal est David Schleef `mailto:ds@schleef.org`. Cette interface permet d'utiliser le driver de n'importe quel espace utilisateur, noyau, temps réel et depuis n'importe quel système Linux, RTLinux, RTAI. Cette solution bien que plus complexe semble extrêmement élégante et mérite un bon coup d'oeil ! Des driver sont déjà développés pour des cartes d'entrées / sorties analogiques. `http://stm.lbl.gov/comedi/`

Bibliographie

[1] RTLinux Version Two, Victor Yodaiken and Michael Barabanov VJY Associates LLC.

[2] Getting started with Real-Time Linux, Michael Barabanov.

[3] Real Time Linux Application and Use, Phil Wilshire.

[4] Real Time Linux 2.0 Installation & Examples, Mirko Holler.

[5] RTC Europe, V4#1, Janvier 2000, `http://www.rtcgroup.com/rtcmagazine/rtc100.html/rtcmag100.html`.

[6] GNU/Linux magazine France, #17, mai 2000.

[7] Electronique International, #359, 9 septembre 1999.

[8] Electronique International, #378, 10 février 2000.

[9] BlueCat Linux User's Guide, BlueCat Version 1.0.0 DOC-0329, Lynx Real-Time Systems Inc.

[10] Java embarqué, Y. Bossu, C. Nicolas, A. Proust, J.B. Blanchet, Eyrolles.

[11] Le Linux Sound-HOWTO, Jeff Tranter, `mailto:tranter@pobox.com`, traduction française de Gaël Duval (version 1.18) `mailto:gael@linuxmandrake.com`

Glossaire

API : *Application Programming Interface*, Interface matérialisée par des primitives, permettant à une application d'accéder à des programmes de plus bas niveau.

ELF : *Executable and Linkable Format*, Format d'exécutables binaires principalement utilisés sous GNU/Linux (permet l'édition dynamique de liens).

FIFO : *First-In-First-Out* ici, méthode de communication entre processus. RT-FIFO moyen d'échange de données non bloquant entre tâches temps réel ou entre une tâche temps réel et un processus utilisateur Linux.

PCI : *Peripheral Component Interconnect*, spécification de fonctionnement d'un bus.PPC : *Power PC* processeur développé conjointement par IBM et Motorola.

RTAI : *Real Time Application Interface*, interface applicative temps réel...

RTLinux : *Real Time Linux*, Linux temps réel...

RTOS : *Real Time Operating System*, Système d'exploitation Temps Réel...

SMP : *Symmetric Multi-Processing*, Architecture d'ordinateur à base de multiples processeurs travaillant en parallèle.

x86 : Processeur compatible avec le jeu d'instruction de l'Intel 8086.

Troisième partie .

Annexes

RTLinux device driver for Computer Boards CIO-PDISO16 digital I/O board

Introduction

The CIO- PDISO16 is a digital I/O board that has 2*8 relay channels and 2*8 digital inputs. It is designed for control and sensing applications where a few points of high voltage need to be sensed or controlled. The 16 outputs are electromechanical relays. The contacts are rated at 6A @ 120V A. C. or 28V D. C., resistive load. The relays are controlled by writing to two 8 -bit ports. The state of the relays may be determined by a read from the control port address. There are 16 individual, optically isolated (500V) inputs that may be read back as two 8- bit bytes. The inputs are not polarity sensitive and may be driven by either A. C. (50 - 1000 Hz) or D. C. in the range 5V - 24V. Programming is accomplished by writes and reads to the 8- bit ports. Each bit indicates the state of an input or controls an output. Pdiso16 is the RTLinux device driver written by Integrated Real Time Systems (*IRTS*). The driver has been implemented as a Linux loadable module for RTLinux 2.2a (2.2.14 Linux kernel). This document explains the functionalities of the pdiso16 device driver and the programmer's C-interface library for it.

http ://www.computerboards.com `http://www.computerboards.com`

The pdiso16 device driver main features are :

– Access from RTLinux thread.

A RTLinux thread can use the pdiso16 device driver through the posixio API. Thus, calling well known *open(), close(), read(), write()* and *ioctl()* functions from a rt-thread, the user can access all the resources of the pdiso16 board.

– Access from Linux user space.

A Linux user space process can access the pdiso16 device driver using 2 RT-FIFO : one to send instructions, the other to receive a response or acknowledgment. It just have to use the communication structure described in the *com.h* header file : *Message_ struct*. The response or acknowledgment to its answers are also sent in such a structure.

Note that the process have to use specialized command codes and ioctl() command constants to communicate with the device driver.

– Byte-wise (8 bits) command and sensing on relay channels.

User can set/read any relay state using the *read() write()* functions. The port number is specified by a call to *ioctl()* function with the CHANGE_PORT command and the PORT1/PORT2 argument. A precise channel can be accessed through logical opérations.

Note that the LSB of the byte is always associated with the lowest channel of the group.

– Word-wise (16 bits) command and sensing on relay channels.

User can set/read all 16 channels' state using the *read() write()* functions. This mode can be specified by a call to *ioctl()* function with the BOTH_PORTS command.

Note that the LSB of the byte is always associated with the lowest channel of the group, the lowest byte is associated with PORT1 and the higher with PORT2 (!).

– Byte-wise and word-wise sensing on optically isolated digital input channels.

User can read any differential digital input channel's state using an *ioctl()* call with the DIGITAL_INPUT command. Result is returned in the *arg* field of the *ioctl()* function. User can ask a single port or a word-wise reading using the same ioctl() commands : CHANGE_PORT or BOTH_PORTS.

Note that the LSB of the byte is always associated with the lowest channel of the group, the lowest byte is associated with PORT1 and the higher with PORT2 (!).

pdiso16 kernel module

Loading (*init_module*)

Mechanism Linux kernel modules are specially made to be pieces of kernel that can be loaded and unloaded dynamically, while the kernel is running. These appear as object files (modname.o) and are loaded with the command *insmod modname.o [arguments]*. This operation runs the initialization of the device(s) and gives a major number to the device driver. This one can be found in the */proc/devices* file. The command can receive many arguments specific to the module.

After being loaded, the device driver module must be associated with devices files which will be used by user programs. This is made with the command : *mknod devname c major minor*. This action can be avoided, if this file is already created with its proper major and minor numbers.

Parameters An optional parameters can be specified when you load the pdiso16 device driver with the *insmod* command. The loading command will looks like (run being root) :

insmod pdiso16.o [PDISO16_base_adr=adr | PDISO16_major=major]

PDISO16_base_adr : As The cio-pdiso16 board can be installed at several address location, the base address switch sets the starting I/O location where the CPU

can access the registers of the board. The factory default is 300 hex (768 decimal). If you already have a board installed at address 300 hex, choose a new address from those available on your computer and set the switches. User can specify board address by adding PDISO16_base_adr=*adr* at the end of the loading command. It must be presented in a hex format (e.g. : PDISO16_base_adr=0x300. In case of no PDISO16_base_adr parameter, a board with the default address *IO_BOARD_ADR* (specified in the *pdiso16.h* header file) will try to be loaded. To see which I/O space is already use by devices you can look at the */proc/ioports* file.

PDISO16_major : The major number for the pdiso16 device driver can be set to another value. In case of no PDISO16_major parameter, PDISO16_MAJOR is taken from the *pdiso16.h* header file.

Initialization The device driver has to register itself with Linux and RTLinux. The loading process begins with a call to *pdiso16_init()* function dedicated to general purpose initialization (Linux I/O region checking, register device...). Then, it has to deal with RTLinux stuff : register using posixio API, set up fifo's and initialize its base address and communication state (PORT1 byte-wise behavior)

Errors Error can appear during the module loading. This error may be caused by invalid loading parameters or by the fact that module is already running on system.

In case of error, you should check if the module is not already running and if all the required resources are free (see the *Special files* section). You can also use the *dmesg* command to see debug or error messages (see the *Debug options* section).

Unloading (*cleanup_module*)

Mechanism As you can dynamically load your kernel module, you can also unload it when you want using the command *rmmod modname*. You can check its name calling the useful *lsmod* command.

Errors Nevertheless, the module will be unloaded only if all the processes/threads have been closed before. This is done with the driver's *release()* function call which is called by the generic *close()* function. Finally, the call may look like *close(FD)*. If some processes/threads are still using the device driver when you try to unload it, the kernel will display a 'busy device or resource' message on console.

Ioctl() calls

Description

Before making an *ioctl()* call to a special file (device driver description file in our case), the device must have been opened by the RTLinux task, using the driver's *open()*

function call which may look like fd = open("/dev/pdiso", O_NONBLOCK).

Then to make any *ioctl()* call user has to indicate the file descriptor (int FD) that has been returned by the *open()* function, a command parameter (unsigned char cmd) and, if required, an argument parameter. The call then may look like *err=ioctl(FD, cmd, arg)* where *err* is an integer returned by the function. In the pdiso16 device driver, *arg* is required while reading digital inputs thus, the *ioctl()* function waits for well sized buffer . User must then provide a ***buffer address*** to the function. Used with CHANGE_PORT command, *arg* takes two values : PORT1 or PORT2 constants.

This section explains the specifications of *cmd* and *arg* parameters and the returned values of the *ioctl()* function.

Command parameter (cmd)

This unsigned char parameter is used to indicate to the driver which port(s) (CHANGE_PORT, BOTH_PORTS) you want to read or to write on the pdiso16 board. You can also indicate that you want to read digital inputs of your board (DIGITAL_INPUT).

cmd=CHANGE_PORT : *read()/write()* functions applied on relay port given in arg field (byte-wise bit manipulation)

cmd=BOTH_PORTS : *read()/write()* functions applied on both relay ports (word-wise bit manipulation)

cmd=DIGITAL_INPUT : read differential digital inputs, result stored at address indicated by the arg parameter.

CHANGE_PORT, BOTH_PORTS and DIGITAL_INPUT are unsigned char (u8) values declared on *com.h* :

```
/*com.h*/
/*ioctl() cmd constants*/
#define CHANGE_PORT 0xA1
#define BOTH_PORTS 0xA2
#define DIGITAL_INPUT 0xA3
```

Argument parameters (arg)

The *arg* parameter is used, on the one hand, to select a port when the CHANGE_PORT command is used, on the other hand, to get back the value of the digital input channels.

Selecting port constants In *com.h* header file :

```
/*ioctl() arg constants*/
#define PORT1 0
#define PORT2 1
```

Example : want to read port2 relay state ?

```
u8 buffer ;   /* u8 : byte-wise variable ( = unsigned char ) */
int err, n ;
...
err=ioctl(fd, CHANGE_PORT, PORT2) ;
if (err == 0) {
    n = read (fd, &buffer, sizeof(buffer)) ;
    if (n > 0) rtl_printf("port2 0x%x\n", buffer ) ;
}
```

Reading the digital input channels I decided to use an *ioctl()* call to read these I/O ports instead of adding another device. Although it can be done easily... The command used is DIGITAL_INPUT to signal the driver it has to read its digital input channels and to get back the value in the result buffer.

Note that the result buffer must be sized according to the number of port you want to read (byte-wise for an only port reading, word-wise for a dual port reading).

Example : want to read port1 then both ports digital input state ?

```
u8    read_buffer_single_port ;
u16  read_buffer_double_port ;
...
ioctl(fd, CHANGE_PORT, PORT1) ;
if (ioctl(fd, DIGITAL_INPUT, &read_buffer_single_port)==0)
    rtl_printf("Digital input PORT1 0x%x\n", read_buffer_single_port) ;

ioctl(fd, BOTH_PORTS) ;
if (ioctl(fd, DIGITAL_INPUT, &read_buffer_double_port)==0)
    rtl_printf("Digital input PORT1 0x%x\n", read_buffer_double_port) ;
```

Returned value

The *ioctl()* call returns 0 on success or -1 on fail. In case of fail, *errno* values are standardized by the include file *<asm/errno.h>* so that you can know what kind of problem has occurred.

If the driver has the required debug level, you can also use the command *dmesg* to see in details where and why the *ioctl()* call has failed.

Write() call

Description

Writing the pdiso16 digital I/O board means to change the state of the relay channels, ie : to change the connection between pins of the selected relay.

Before making a *write()* call to a special file (device driver description file in our case), the device must have been opened by the RTLinux task, using the driver's *open()* function call which may look like fd = open("/dev/pdiso", O_NONBLOCK).

Then to make any *write()* call user has to indicate the file descriptor (int fd) that has been returned by the *open()* function, a command buffer and, its size. The call then may look like *nbwr=write(fd, &buffer, sizeof(buffer))* where *nbwr* is an integer returned by the function : the number of bytes written. In the pdiso16 device driver, user can give a byte-wise or word-wise buffer according to his choice of a single port reading or a dual one.

Controlling relay channels

When a 1 is written to the output, the common and the NO (Normally Open) pins of the relay, are in contact. User can switch a relay state setting a 1 or 0 to the right place within the byte or word wise buffer. Then the buffer is written down to the board's register by the device driver's *write()* call...

```
#define CHANNEL0 0x01
#define CHANNEL1 0x02
...
#define CHANNEL7 0x80
...
int n ;
u8 buffer = 0x00 ;
...
ioctl(fd, CHANGE_PORT, PORT1) ;
buffer |= CHANNEL0 ;     /*Turn on channel 0 ( Less Significant Bit of the byte)*/
buffer &=  CHANNEL1 ;    /*Turn off channel1*/
...
n = write(fd, &buffer, sizeof(buffer)) ;
```

Of course, direct write is the simplest way to access the board registers :

```
int n ;
u8 buffer = 0xA3 ;  /*channel 7 : on , off, on, off, off, off, on, channel 0 : on*/
ioctl(fd, CHANGE_PORT, PORT1) ;
 n = write(fd, &buffer, sizeof(buffer)) ;
```

If the user wish to control only one channel state, without changing the others, he has to use the *read()* function. This example shows how to open channel 3 relay (turn off) :

```
ioctl(fd, CHANGE_PORT, PORT1) ;
read(fd, &buffer, sizeof(buffer)) ;
buffer &=  CHANNEL3 ;    /*Turn off channel3*/
n = write(fd, &buffer, sizeof(buffer)) ;
```

Read() call

Description

Reading the pdiso16 digital I/O board means to control the status of the relay channels.

As for an *ioctl()* or a *write()* call, before making a *read()* call, the device must have been opened.

Then to make any *read()* call user has to indicate the file descriptor (int fd) that has been returned by the *open()* function, a result buffer and, its size. The call then may look like *nbrd=read(fd, &buffer, sizeof(buffer))* where *nbrd* is the number of bytes read. In the pdiso16 device driver, user can give a byte-wise or word-wise buffer according to his choice of a single port reading or a dual one.

Reading relay channels status

– In single port mode, user must provide a 8 bits buffer (u8 = unsigned char). The status of each channel can be accessed using logical operations :

```
int n ;
u8 buffer ;
 ...
ioctl(fd, CHANGE_PORT, PORT2) ;
n = read(fd, &buffer, sizeof(buffer)) ;
if (n>0) {
    /*Channel 0 is the LSB (Less Significant Bit) of the byte*/
    u8 channel_0 = buffer & 0x01 ;
    u8 channel_1 = buffer & 0x02 ;
    ...
    u8 channel_6 = buffer & 0x40 ;
    u8 channel_7 = buffer & 0x80 ;
    /*Channel 7 is the MSB (Most Significant Bit)*/
    ...
    rtl_printf(“channel_0 relay : %s, ( (channel_0 == 0)? "off" : "on") ) ;
    ...
    rtl_printf(“channel_7 relay : %s, ( (channel_7 == 0)? "off" : "on") ) ;
}
```

– In dual port mode, user must provide a 16 bits buffer (u16 = unsigned short). The device driver can put in it, the result of the reading. Each port can be accessed following this rule :

```
int n ;
u16 buffer ;
 ...
ioctl(fd, BOTH_PORTS) ;
```

```
n = read(fd, &buffer, sizeof(buffer)) ;
if (n>0) {
    u8 port_one = buffer & 0x00FF ;             /*PORT1 : The lowest byte of the wo
    u8 port_two = (buffer & 0xFF00) >> 8 ;      /*PORT2 : The highest one*/
    rtl_printf(“port_one=0x%x, port_two=0x%x\n”, port_one, port_two) ;
}
```

Calls from Linux user space : using RT-FIFOs

RT-FIFO implements message passing through rt-kernel space... This kind of communication makes me think of QNX's, except that it is non blocking (which is a pretty big difference !). Another difference is that rt-fifos are uni-directional. User must exchange data through a pair of rt-fifos for bi-directional communication. My device driver requires 2 rt-fifos...

– In a first time user has to open his 2 rt-fifos :

```
if ((Fd_Rd = open("/dev/rtf0", O_RDONLY)) < 0) {
    fprintf(stderr, " Error opening /dev/rtf0 for reading.\n") ;
    fprintf(stderr, " RT-Linux module not active ? ?\n\n") ;
    exit(-1) ;
}
```

Same thing for the other direction : ...Fd_Wr = open("/dev/rtf1", O_WRONLY)...

– Then he has to construct his message, filling a data structure. It is described in the *com.h* header file :

```
/* Messages sent to rtlinux module */
typedef struct {
    u8 what ;
    u16 value ;
} Message_struct ;
#define MESSAGE_SIZE sizeof(Message_struct)
```

Example : command the relay channels

```
Message_struct Send_Cmd ;
...
Send_Cmd.what = WRITE_RELAY ;
Send_Cmd.value = 0xAA ;
```

– Finally, he sends his message through the rt-fifo using the write() call to the proper rt-fifo file descriptor.

```
if (write(Fd_Wr, &Send_Cmd, MESSAGE_SIZE) < 0) {
    fprintf(stderr, "Can't send a command to RT-task\n") ;
    exit(1) ;
}
```

Send request messages

– The what member is the command sent to the device driver. It can take the following values :

/* command codes for the message types (what tag) */
#define WRITE_RELAY 0x1 /*= read() function */
#define READ_STATE 0x2 /*= write() function */
#define TAKE_DATA 0x3 /*= ioctl(fd, DIGITAL_INPUT, ...) function */
AND the ioctl() commands CHANGE_PORT(PORT1/PORT2 in value tag) and BOTH_PORTS.

– The value member contains the buffer to send to command the relay channels, the result of the reading or the data from digital inputs. Its length is linked with the number of ports required.

Receive messages

To receive a message, the Linux user space process must listen to the RTLinux pdiso16 device driver. User can simply implement it with a *read()* call or use *read()* with a *select()* call on the rt-fifo. Example could be fount in the *app.c* source file. A *Message_ struct* data structure must be used while calling the *read()* function.

Acknowledgment messages The device driver passes on an acknowledgment message to the Linux process when no result is required (write or ioctl commands). Thus, the device driver says that it has received the command ant that it has succeeded; the *Message_ struct*'s *what* member contains : RECV_DONE.

```
/* OK code in com.h*/
#define RECV_DONE 0x10
Error codes are sent to the Linux process the same way.
/* Error codes in com.h */
#define RECV_ERR1 -1
#define RECV_ERR2 -2
#define CMD_ERR -3
#define SEND_ERR -4
```

Result messages The result message's *what* member is always TAKE_DATA and its value member is filled with the result of the request previously passed on to the device driver. If a single port access is done, this 16 bits value only contains its lowest byte valid. I recommend to give a 0 filled value member to the *read()* function. In case of a double port access, the entire value field is OK and the lowest byte corresponds with port1.

Miscellaneous

Debug options

At the compilation of the driver, user can specify which level of debug he wants to be displayed on the kernel log. This is done by uncommenting *#define DEBUG* for debug level 1 or both *#define DEBUG* and *#define DEBUG2* for debug level 2 in the *pdiso16.c* file. In real-time In general, debug level 1 displays actions and probable causes of command faults and debug level 2 add the state of important global and local variables at that time so that you can determine what was wrong.

You can also display all the kernel messages by using *dmesg* command.

Note that debug options slow down the device driver.

Special files

Kernel uses special files to save all the systems parameters. Some of those can be very useful to get informations about the device driver :

/proc/devices : this file indexes all the devices drivers installed on the system with their major number and their type (char or block).

/proc/interrupts : this file indexes all the interrupts that have already appear on the system with their interrupt vector, their frequency and the name of the device driver which owns them.

/proc/ioports : this file indexes all the I/O regions that have been taken by devices drivers. The name of the device driver that owns the region is also displayed.

/proc/ksyms : this file indexes all the kernel entry points with their address and the name of the function. You can display these informations with the *ksyms* command.

/proc/modules : this file registers all the loaded modules with their memory occupation and the number of processes/threads that have opened it. You can display these informations with the *lsmod* command.

/proc/version : this file contains the current running kernel version. It is useful to see if your module version is compatible with the current kernel but you can force the module even if the versions are incompatible with *insmod -f*.

/dev/pdiso : this file is the devices files associated with the board using the pdiso16 device driver. You can see major and minor number of this file with *ls -l* command.

/var/log/messages : these file contains all the messages sent by kernel with printk() calls. You can display these messages with the *dmesg* command.

/etc/devinfo : this file indexes all the different device drivers type that can exist with their major and minor number.