INP ENSEEIHT

Communication avec un Robot Lego Mindstorms

BINOME :
DANELON Céline
LAFFITTE Julien

ENCADRANTS :
Jérôme ERMONT
Frédéric BONIOL

Remerciements

Nous tenons à remercier l'ensemble des personnes qui nous ont aidé dans nos recherches :

Pour le protocole µIP : Adam Dunkels.

Pour le protocole LNP : Jérôme Ermont et Emmanuel Chaput qui ont grandement facilité notre compréhension du code.

Pour les questions concernant les Lego Mindstorms : le forum Freelug et plus particulièrement Khan et Philo.

D'une manière générale toutes les personnes qui ont répondu à nos mails : Ecole des Mines de Nantes, Club de Robotique de l'INSA.

Frédérique Coudret, pour sa disponibilité et ses conseils concernant l'environnement LINUX.

Nos encadrants : Jérôme Ermont et Frédéric Boniol pour leur disponibilité et leurs conseils.

Sommaire

1. INTRODUCTION 5
2. QUE SONT LES ROBOTS LEGO MINDSTORMS ? [LEGO01-02] 5
2.1. HISTORIQUE 5
2.2. ORGANISATION DE LA BRIQUE RCX 5
2.3. COMMUNICATION INFRA ROUGE (IR) SUR LA BRIQUE RCX DE BASE [IR 01] 6
2.3.1. Caractéristiques 6
2.3.2. Le protocole 6
2.3.2.1. Introduction 6
2.3.2.2. Commande 7
2.3.2.3. Paquet 7
2.3.2.4. RS232 [IR 01] 8
2.3.2.5. Transmission 9
2.3.3. Les limites d'une communication IR 9
2.4. LA PROGRAMMATION RCX 9
3. BRICKOS [BRICK01-02] 10
3.1. PRESENTATION 10
3.2. LE NOYAU BRICKOS 10
3.3. GESTIONNAIRE DE PROCESSUS 11
3.4. DESCRIPTEUR DE PROCESSUS 11
3.5. ETATS ET TRANSITIONS D'UN PROCESSUS 11
3.6. L'ORDONNANCEUR BRICKOS 12
3.7. GESTION MEMOIRE 13
3.8. LES SEMAPHORES 13
3.9. BRCIKOS ET LE TEMPS REEL 13
4. LNP : LEGOS NETWORK PROTOCOL [LNP01-03] 14
4.1. PRESENTATION 14
4.2. LA PILE DE COMMUNICATION 14
4.2.1. Pré-requis 14
4.2.2. Du point de vue de l'ordinateur 14
4.2.3. Du point de vue du RCX 15
4.2.4. Schémas récapitulatifs 15
4.2.4.1. Les services LNP 15
4.2.4.2. Organisation générale 16
4.3. LES MESSAGES LNP 16
4.4. CONSTRUCTION DES ADRESSES 17
4.5. ATTRIBUTION DES ADRESSES 18
4.6. LA DETECTION DE COLLISION 18
4.7. LE DEMON LNPD [LNP 01] 18
5. µIP [UIP01] 19
5.1. CARACTERISTIQUES 19
5.2. GESTION MEMOIRE 20
5.3. API 20

5.4. IMPLANTATION DE µIP 20
5.4.1. La boucle de contrôle principale 21
5.4.2. Internet Protocol 21
5.4.2.1. Fragmentation IP 21
5.4.2.2. Broadcast et multicast 22
5.4.3. ICMP : Internet Control Message Protocol 22
5.4.4. TCP : Transmission Control Protocol 22
5.4.5. UDP : User Datagram protocol 22
5.5. SERVICES OFFERTS PAR LNP A µIP 22
5.5.1. Pré-requis 22
5.5.2. Intégration dans le kernel de BrickOS 23
5.5.3. TCP/IP et le démon LNPd 23

6. CONSEILS D'INSTALLATION [HOWTO 01] 24

6.1. CONSTRUCTION DU CROSS COMPILER 24
6.1.1. Binutils 24
6.1.2. Installation du GCC Cross Compiler 24
6.2. BRICKOS 25
6.3. POUR TESTER L'INSTALLATION 26
6.3.1. Télécharger le firmware 26
6.3.2. Lancer le programme de Démo 26
6.4. CHARGEMENT DE µIP 27

7. RESULTATS OBTENUS 27

8. CONCLUSION 28

1. Introduction

Ce projet présente une étude des robots Lego Mindstorms, petits calculateurs très prisés dans les laboratoires de recherche aussi bien pour faire du temps réel que de l'intelligence artificielle. Notre but était d'analyser la capacité de communication des ces robots afin de modéliser un système temps réel de base.

La principale difficulté consistait à comprendre comment cette communication était mise en œuvre et à l'intégrer sur le robot.

Dans un premier temps, nous nous sommes intéressés à BrickOS, système d'exploitation dédié à la gestion de la communication sur les mindstroms et à l'utilisation maximale de leurs capacités calculatoires.

En complément de celui-ci a été développé un protocole spécifique pour la communication infra rouge entre un PC et un ou plusieurs robots, le protocole LNP.

Ensuite, nous nous sommes interrogés sur les possibilités de mise en place d'une pile IP réduite. Pour cela nous avons étudié le protocole µIP, ses principales caractéristiques et son interfaçage avec le protocole LNP.

2. Que sont les robots LEGO Mindstorms ? [LEGO01-02]

2.1. Historique

Les LEGO Mindstorms sont nés il y a une quinzaine d'années de la collaboration entre LEGO et le Massachusetts Institutes of Technology (MIT). Fred Martin, un chercheur du MIT, a participé grandement au développement de la technologie actuellement utilisée dans les microprocesseurs. C'est ensuite la contribution du Professeur Seymour Papert qui a permis l'intégration d'un langage de programmation dans la brique LEGO. En effet, ce dernier est un pionnier dans l'intelligence artificielle et est à l'origine du Logo[1] programming language. En travaillant conjointement avec LEGO, il a donc participé à son intégration à la brique LEGO.

Le maître mot de LEGO est, « A first-time user with basic PC skills can design, program, and build a simple robot within one hour. »

2.2. Organisation de la brique RCX

Le Cœur du kit Mindstorms est la brique RCX programmable. Le RCX comporte 3 entrées capteurs et 3 sorties moteurs, possède 32 Ko de mémoire vive ainsi que de la mémoire morte à laquelle on ne peut avoir accès. Ce « petit cerveau » peut exécuter 1000 commandes par secondes. Il fonctionne avec un microcontrôleur 8 bits de la gamme Hitachi H8/3297 à une vitesse de 16 MHz. De plus, pour communiquer avec un ordinateur un port infra-rouge a été mis en place.

[1] Logo est un langage fonctionnel, interprété, qui permet une utilisation directe sans passer par une phase de compilation.

Communication Infra rouge avec un PC

Figure 1 : Organisation du RCX

2.3. Communication Infra Rouge (IR) sur la brique RCX de base [IR 01]

Dans cette partie n'est abordée que la communication implantée sur le RCX de base, on ne fera référence qu'à une communication série classique, aucune information relative à l'USB ne sera détaillée.

2.3.1. Caractéristiques

La communication entre un ordinateur et le RCX se fait au moyen d'un port IR. Il s'agit d'une communication half duplex, en effet le récepteur IR est saturé pendant que l'émetteur fonctionne. Le RCX utilise une porteuse à 38kHz, cependant dans sa nouvelle version 2.0 le récepteur est accordé sur une modulation à 76kHz, ce qui permet la compatibilité descendante avec des produits Lego plus récents (Spybots, Manas...) et ascendante avec les anciens RCX.

Le débit est de 2400 bps, autrement dit un bit toutes les 417 µs ce qui se rapproche des performances des télécommandes pour téléviseurs classiques.

2.3.2. Le protocole

2.3.2.1. <u>Introduction</u>

La communication est faite en mode maitre-esclave où le RCX est un esclave. Chaque donnée est transportée dans un paquet.

On encode à 2400bps, avec un code NRZ (Non Return To Zero). La trame s'organise avec 1 bit de start, 8 bits de données, 1 bit de parité et un bit de stop ce qui est caractéristique d'une communication série de type RS232, comme nous allons le voir par la suite.

Start	Data	Odd parity	Stop

Figure 2 : Trame IR

Quatre niveaux de communication sont mis en œuvre :

- Commande
- Paquet
- RS232
- Transmission (modulation/longueur d'onde)

2.3.2.2. Commande

Ce niveau de communication permet d'envoyer des commandes au RCX ou de recevoir des requêtes. Les commandes sont de quatre types :

- Immediate commands (y compris les requêtes d'information)
- Download commands
- Remote Control Commands

2.3.2.3. Paquet

Comme mentionné précédemment tout type de données est encapsulé dans un paquet constitué :

- D'une en-tête : 55 FF 00
- Des bits de données, chaque octets est suivi de son complément ainsi pour transmettre '55 F7' on envoie '55 AA F7 08'
- Un checksum qui correspond à la somme de tous les octets, lui aussi suivi de son complément

Le format général d'un paquet est le suivant :

0x55 0xff 0x00 D1 ~D1 D2 ~D2 ... Dn ~Dn C ~C

où D1... Dn représente le corps du message, et C = D1 + D2 + ... Dn

Exemple

Un message est une commande immédiate de type "F7" suivie par le numéro du message et encapsulé dans un paquet ce qui se traduit comme suit :

55 ff 00 f7 08 12 ed 09 f6

est un paquet envoyant le message '0x12' au RCX.

- Le RCX n'exécute jamais deux fois le même opcode sur une ligne, par conséquent, chaque commande a deux occurrences. Une avec le troisième bit positionné à 1 et l'autre avec ce même bit à 0. Pour envoyer une commande deux fois il faut donc alterner ce bit.

Exemple

L'envoie de données peut être une requête (ordinateur vers RCX) ou une réponse (RCX vers ordinateur). La réponse se traduit pas le complément à 1 du opcode de la requête et réciproquement.

Op 0x10 Alive Request / Op 0xef Alive Response
Op 0x18 Alive Request / Op 0xe7 Alive Response
0x10 et 0xef sont complémentaires
0x18 et 0xe7 sont complémentaires
0x10 et 0x18 différent seulment par le bit 0x08
0xef and 0xe7 différent seulment par le bit 0x08

Remarque :

Le schéma utilisé pour la transmission résulte en un nombre égal de bits à 0 et à 1, ce qui permet à un récepteur de corriger une erreur qui proviendrait d'un signal de lumière constant en soustrayant la valeur moyenne du signal. Par ailleurs, on remarquera que l'entête est aussi constituée d'un nombre équivalent de 0 et de 1, ce qui permet de réveiller le récepteur avant que les données utiles soient envoyées.

2.3.2.4. RS232 [IR 01]

Voici une liste des différents échanges relatifs à une communication série.

PC	IR-tower	Name	Description	Notes
1	1	CD	Carrier Detect	not connected
2	3	RD	Receive Data	RCX==>PC
3	2	TD	Transmit Data	PC==>RCX
4	4	DTR	Data Terminal Ready	connected, but not used
5	5	SG	Signal Ground	
6	6	DSR	Data Set Ready	not connected
7	8	RTS	Ready To Send	must be high to receive or send data
8	7	CTS	Clear To Send	used to detect if tower is connected to PC, because it's connected in the IR-tower to RTS
9	9	RI	Ring Indicator	not connected

2.3.2.5. Transmission

Un ‘0’ est codé par un pulse de 417µs à une fréquence de 38kHz, pour un ‘1’ la fréquence est nulle.

2.3.3. Les limites d’une communication IR

L’utilisation de l’IR pour communiquer des données au RCX n’est pas sans problème, en effet, la perte de paquets est chose courante. Tout d’abord ce système est très directif, autrement dit la Tour USB et le capteur du RCX doivent être en vue directe à une distance inférieure à 1m (les rayons infra rouge ne peuvent pénétrer des objets opaques). De plus, il ne doit pas être utilisé dans une pièce avec une lumière trop vive qui viendrait alors perturber les échanges.

De plus, des pertes trop nombreuses peuvent devenir handicapantes lorsque l’on veut établir un système temps réel.

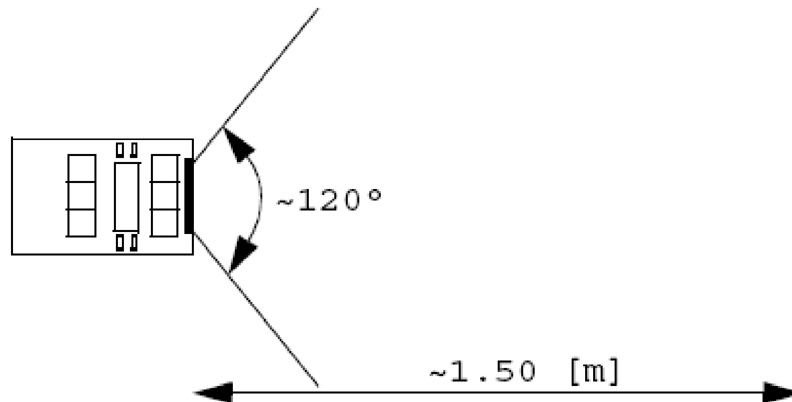
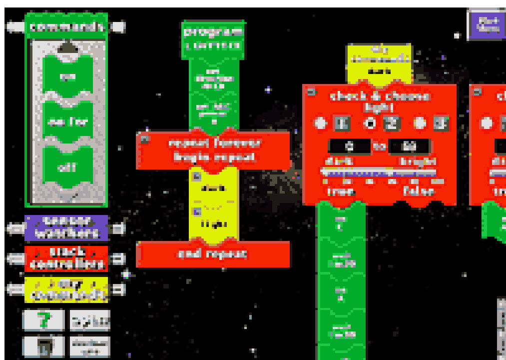


Figure 3 : Champ de vision Infra rouge d’un module RCX

2.4. La programmation RCX



Au départ, tout a été optimisé pour rendre la programmation la plus simple possible. L’environnement de programmation est un environnement graphique qui permet à partir d’une bibliothèque de composants de faire des glisser, déposer. Ainsi, il n’y a aucune ligne de code à écrire.

Des blocks de code qui représentent des commandes sont affichés sous forme d’icônes. On peut ainsi choisir la commande désirée parmi celles disponibles dans la bibliothèque. Quelques clicks suffisent donc pour charger un programme et l’exécuter.

De plus, chaque RCX peut stocker cinq programmes différents permettant à un même robot d’exécuter plusieurs actions différentes.

Bien que très simple ce langage de programmation reste assez limité et donc sont apparus de nouveaux langages tels que NQC (Not Quite C), PbForth (reprise de l'ancien

langage Forth) et un système d'exploitation BrickOS (anciennement LegOS). C'est à ce dernier que nous allons nous intéresser.

3. BrickOS [BRICK01-02]

3.1. Présentation

BrickOS (anciennement LegOS) est un système d'exploitation embarqué open source écrit par Markus Noga qui vient remplacer le firmware d'origine de la brique RCX.

Il offre un environnement de programmation en C/C++ permettant ainsi de surpasser le logiciel de programmation graphique fourni avec la brique. En effet, il donne un accès total à la mémoire du RCX et fait bénéficier de la puissance du C : « Think 32 k not 32 variables ». Autrement dit, on est plus limité à l'utilisation de 32 variables mais on peut définir un ensemble de données qui se partagent les 32 Ko de mémoire. Il offre également une bien meilleure exploitation de la liaison Infra-Rouge entre PC et RCX.
Des utilitaires[2] pour charger l'OS sur le RCX et les programmes compilés y sont associés.

C'est un noyau monolithique : ses sources sont compilées dans la même image binaire qui est ensuite téléchargé sur le RCX. Il n'intègre pas de système de fichier puisqu'il ne possède aucun périphérique de stockage de masse, mais il propose néanmoins un certain nombre de fonctionnalités :

- Gestion du multitâche avec préemption,
- Implantation des sémaphores (POSIX 1003.b compliant),
- Ordonnanceur à système de priorité multiple,
- Gestion dynamique de la mémoire.

Le noyau brickOS est placé dans la partie supérieure de la RAM du RCX tandis que les programmes utilisateurs sont implantés dans sa partie inférieure.

3.2. Le noyau brickOS

BrickOS utilise un système de scrutation pour communiquer avec ses sous ensembles : la fonction systeme_handler scrute les différents ensembles du RCX en appelant tour à tour leur handler respectif. Cette fonction est appelée via un vecteur d'interruption invoqué toutes les millisecondes par le timer 16 bits du RCX.

A chaque appel, plusieurs opérations sont réalisées :

- Incrémentation de system timer,
- Appel du pilote des moteurs puis de celui du son,
- Examination de LNP,
- Appel du pilote des boutons,
- Mise à jour de l'indicateur de l'état des batteries du RCX,
- Mise à jour de l'affichage LCD.

La dernière opération effectuée est la vérification du changement de contexte. Si le compteur de temps partagé du processus courant est dépassé, alors la fonction tm_switcher est appelée : c'est elle qui réalise le changement de contexte avec l'aide de l'ordonnanceur.

[2] Comme l'utilitaire firmdl3.

3.3. Gestionnaire de processus

Le gestionnaire de processus implémente un système multitâche préemptif, c'est-à-dire que le temps de calcul processeur est alloué périodiquement à chaque processus et pour une durée de temps prédéfinie.

Par défaut, cette durée est de 20 ms. Cependant le quantum de temps minimum est de 6 ms tandis que le maximum est fixé à 255 ms.

Lors de sa création, chaque processus se voit également alloué une priorité qu'il conserve jusqu'à sa mort. 20 niveaux de priorité ont été définis, le plus bas étant 0 et le plus haut étant 20.
Le pointeur priority_head permet d'accéder aux différents niveaux de priorité en pointant sur le niveau de priorité le plus élevé. Chaque descripteur de processus est inclus dans une liste doublement chaînée et comporte un pointeur priority.

Tout processus comporte un descripteur et l'ensemble des descripteurs est organisé en files dont notamment une par niveau de priorité.
Avant toute exécution de programme, la structure contient uniquement deux descripteurs de processus : un de priorité 0 et l'autre de priorité 20. Le premier est utilisé pour mettre le RCX en veille, c'est-à-dire que lorsqu'il n'y a aucun programme à exécuter, un processus d'attente est exécuté avec la priorité la plus basse.
Le second descripteur correspond au processus utilisé pour la réception de données via le port de communication IR.

3.4. Descripteur de processus

Chaque processus brickOS est créé en appelant la fonction execi qui prend en paramètre le nom d'une procédure système ou utilisateur. De plus, La fonction doit retourner une valeur entière et avoir comme paramètre le nombre d'arguments suivi de ceux-ci (char *argv[]).
Lors de sa création, execi ajoute son descripteur à la structure générale, l'opération étant protégée par un mécanisme de sémaphore afin d'assurer l'exclusion mutuelle entre les différents processus.

Il est également décrit par une structure de données pdata_t contenant les informations suivantes :

- Pointeur de pile,
- Pointeur de début de la structure de pile,
- Etat du processus,
- Priorité du processus,
- Pointeur sur le prochain processus de la file,
- Pointeur sur le processus précédent.

3.5. Etats et transitions d'un processus

Comme pour un processus Unix, les processus brickOS se caractérisent par un certain nombre d'états définissant leur statut. Ces états sont au nombre de 5 :

- Détruit (*dead*) : le processus est terminé et sa pile désallouée,
- Zombie : le processus s'est terminé mais sa pile n'est pas libérée,
- Bloqué (*waiting*) : le processus est bloqué en attente d'un évènement,
- Prêt (*sleeping*) : le processus est prêt à être exécuté,
- Actif (*running*) : le processus est en exécution.

Ces 5 états permettent de construire un automate du cycle de vie d'un processus :

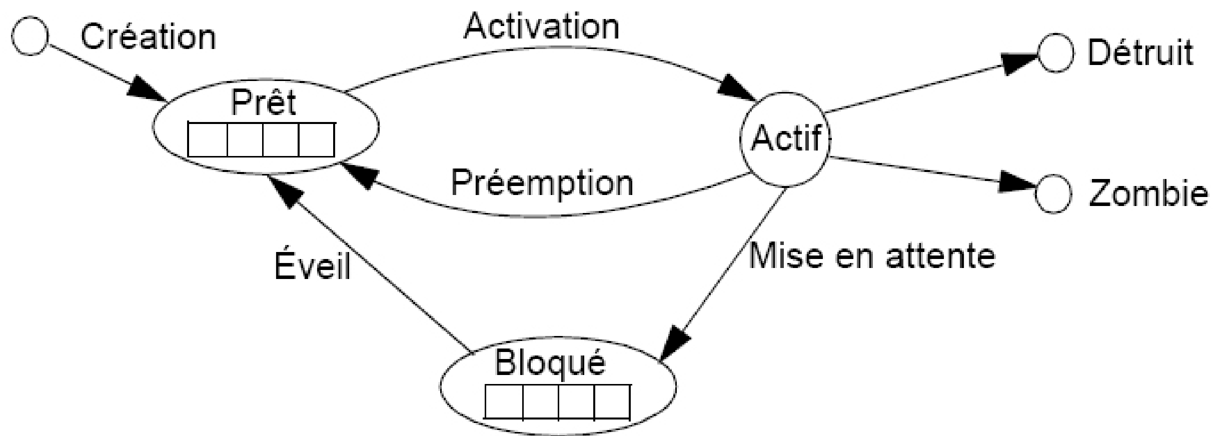


Figure 4 : Cycle de vie d'un processus

Cette information concernant l'état du processus est utilisée par l'ordonnanceur de brickOS pour la distribution des temps de calcul RCX mais également pour réaliser la libération des piles mémoire des processus terminés.

Chaque processus brickOS peut également stopper volontairement son exécution en appelant l'une des fonctions suivantes :

- **Yield() :** le processus restitue la ressource processeur même si il se trouve dans son quantum d'émission,
- **Msleep(int time) :** le processus attend l'expiration d'un délais (en ms) passé en paramètre de la fonction,
- **Sleep(int time) :** le processus attend l'expiration d'un délais (en s) passé en paramètre de la fonction,
- **Wait_event(evt) :** le processus se met en attente d'un évènement passé en paramètre de la fonction. Elle joue le rôle d'un réveil pour le processus qui a au préalable été mis en attente.
- **Sem_wait(x) :** primitive bloquante.

3.6. L'ordonnanceur brickOS

L'ordonnanceur de brickOS permet d'allouer les ressources du RCX périodiquement aux différents processus en cours d'exécution et pendant une durée limitée (Cf. *2.3 Gestionnaire de processus*). Pour cela, il utilise un mécanisme de round robin et examine les listes de processus dans l'ordre de priorité décroissant : si le processus suivant est dans l'état prêt, alors il est élu et passe dans l'état actif.

Lorsqu'une liste de priorité est terminée, l'ordonnanceur passe à la suivante.

Il y a ici combinaison entre des ressources partagées et protégées par un mécanisme de sémaphore et l'ordonnancement de processus par priorité : cela peut introduire un phénomène d'inversion de priorité. Il peut en effet arriver qu'un processus de haute priorité soit bloqué en attente des ressources du RCX par un processus de moindre priorité qui consomme son temps de parole.

3.7. Gestion mémoire

BrickOS n'utilise pas de mécanisme de gestion de la mémoire avancé tel que la pagination ou la segmentation mémoire. Il utilise un schéma d'allocation linéaire de la mémoire ou chaque bloc comprend un en-tête de 4 octets et n octets de données :

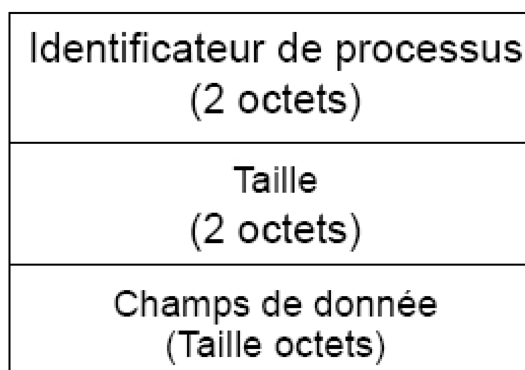


Figure 5 : Bloc mémoire de base

Dans chaque en-tête, 2 octets sont réservés pour l'identifiant du processus tandis que les 2 derniers sont utilisés pour spécifier la taille des données qui suivent.

La mémoire est répartie entre le noyau et les programmes utilisateurs.

3.8. Les sémaphores

Les sémaphores implantés dans brickOS répondent au standard POSIX 1003.b et permettent la synchronisation entre processus concurrents. Les opérations de base possibles sont les suivantes :

- Décrémenter le compteur de manière atomique et bloquer éventuellement le processus appelant,
- Incrémenter le compteur de manière atomique.

Le détail et l'utilité des fonctions liées aux sémaphores dans brickOS sont disponibles dans le document [BRICK02].

Dans les programmes utilisateurs (clients ou pour le RCX) la gestion des sémaphores devra être réalisée soigneusement de manière à éviter les situations de famine pour les processus de basse priorité. Dans le cas contraire, on ne pourra pas garantir que tout processus pourra accéder aux ressources dans un temps fini.

3.9. BrcikOS et le temps réel

L'analyse de brickOS nous permet de voir qu'il met en œuvre un système de scrutation pour communiquer avec les sous-ensembles, l'allocation périodique d'un temps de calcul processeur, la mise en œuvre de priorité, le round robin qui sont autant de mécanisme indispensable à la mise en œuvre du temps réel.

4. LNP : LegOS Network Protocol [LNP01-03]

4.1. Présentation

Dans le cadre de l'utilisation de la communication Infra Rouge (IR), BrickOS a développé le protocole LNP concurrent de celui de Lego : il est utilisé pour les communications entre un ordinateur et un ou plusieurs RCX.

LNP est inclus dans le noyau de BrickOS, mais il est cependant nécessaire de télécharger le package LNP pour l'utiliser sur une station de travail (le plus couramment Unix). Dans ce cas, un démon *LNPd* est créé sur la machine communicante et nécessite l'utilisation d'une bibliothèque *liblnp*.

4.2. La pile de communication

Pour illustrer le fonctionnement du protocole LNP, nous allons suivre les différentes opérations effectuées sur un paquet jusqu'à l'écriture des données dans la mémoire du RCX.

4.2.1. Pré-requis

- Le RCX doit être allumé et BrickOS doit avoir été chargé en mémoire.
- Le démon LNPd est lancé sur la machine UNIX.
- La Tour USB est connectée.

Avant d'être transmises au RCX par l'intermédiaire de la connexion IR, les données doivent être traitées par le démon LNPd. Pour ce faire, un addressing handler ou un integrity handler doit être appelé (on rappelle que la seule différence entre ces deux handlers est que l'adressing handler envoie des paquets LNP auquel on a ajouté une adresse source et destination). Le choix de la routine se fait en fonction du type de connexion que l'on veut offrir : les messages d'intégrité sont utilisés dans le cas d'un broadcast tandis que les messages adressés permettent d'établir du point à point.

De manière générale on se soucie peu de ce choix, en effet, notre communication n'implique qu'un seul RCX et un seul ordinateur.

Nous avons travaillé avec le handler mis en place par le fichier lnptest.c qui est un adressing handler.

4.2.2. Du point de vue de l'ordinateur

On doit faire appel à une fonction du type :

lnp_addressing_write(unsigned char *data, unsigned char length, int dest_addr, int port)[3]:

Fonction qui permet d'envoyer des données sur le réseau. Les paramètres data et length permettent respectivement de préciser les données à envoyer et leur longueur. Les paramètres dest_addr et port permettent respectivement de spécifier l'adresse de destination du paquet et le port sur lequel envoyer les données.

Elle passe ensuite le paquet à la fonction ***lnp_logical_write(unsigned char * data, int length***) qui se charge de la communication avec le socket pour transmettre les données à la connexion IR.

[3] Le nom des fonction peut changer suivant la version de LNP utilisé, nous avons travaillé avec la version 0.9

4.2.3. Du point de vue du RCX

La réception des messages sur le RCX se fait au moyen d'interruptions. L'ensemble des fonctions relatives à ce traitement est contenu dans le **lnp-logical.h**. Une machine à état est construite pour recevoir et envoyer les octets, elle reconnaît et décode les trames LNP et pour chacun teste leur validité par un calcul du checksum.

Les principales fonctions mises en jeu sont **set_integrity_handler()**, respectivement **set_adressing_handler()** : à chaque arrivée de messages elles associent le bon handler.

Le RCX, comme l'ordinateur, peuvent être uniquement dans deux états : tx_state == TX_ACTIVE qui correspond à l'envoie de messages et le tx_state != TX_ACTIVE pour la réception.

Lors de la réception, les octets sont ensuite passés à la fonction ***lnp_integrity_bytes()***. A ce niveau de la machine à états la validité de la trame est analysée. Si elle est correcte elle est ensuite passée à la fonction ***get_packet()*** qui vérifie l'exactitude de l'identifiant de l'hôte et du numéro de port, si il y a des erreurs la trame est ignorée.

Le handler analyse ensuite le paquet et copie éventuellement les informations dans la mémoire du RCX.

Remarque

L'envoi de données du RCX à l'ordinateur est similaire, en effet les mêmes fonctions sont présentes sur le PC et le RCX.

4.2.4. Schémas récapitulatifs

4.2.4.1. Les services LNP

Les services présentés précédemment s'organisent en deux couches correspondant aux deux couches les plus basses du modèle OSI :

- Le niveau liaison de données se divise en deux entre integrity et adressing,
- la couche physique est représentée par le niveau logical.

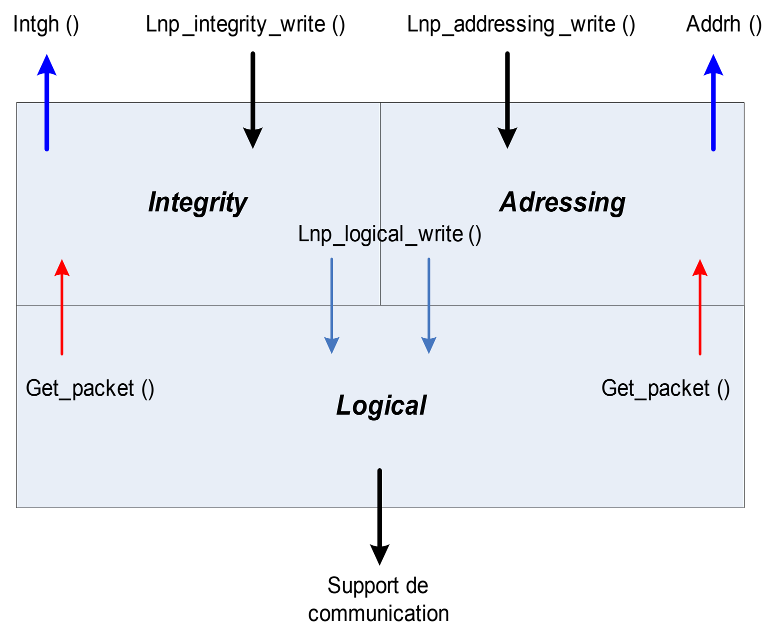


Figure 6 : Les services LNP

Lors d'une demande d'écriture, suivant le handler choisi, on fait appel à la fonction **lnp_xxx_write()** qui elle-même appelle **lnp_logical_write()**. Les données sont ensuite dirigées vers les sockets de **lnplib**.

Lors d'une réception de données, la couche logical invoque la fonction **get_packet()**, qui elle-même fait appel à **intgh()** ou **addrh()** selon le handler.

4.2.4.2. Organisation générale

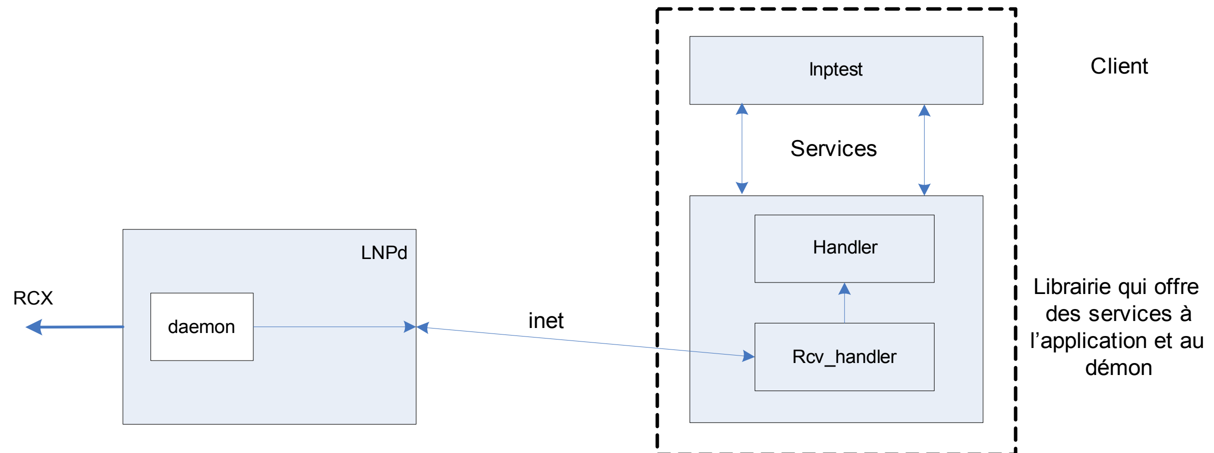


Figure 7 : Fonctionnement LNP

Sur l'ordinateur, le démon LNPd doit être lancé. Il émet et reçoit des messages provenant du RCX par la tour infrarouge. Il communique via des sockets inet avec la librairie lnplib et accède ainsi aux services fournis par LNP.
Les données du PC sont transmises vers le démon à travers un socket. La librairie liblnp qui sera liée à l'application envoie sur un socket.

4.3. *Les messages LNP*

Il existe 2 types de messages générés par LNP :

- ***Les messages d'intégrité :*** ils ne contiennent aucunes informations d'adressage et peuvent être considérés comme l'équivalent d'un mécanisme de broadcast.

F0	LEN	IDATA	CHK

F0 : identifies an integrity packet
LEN : length of IDATA section, 0 to 255
IDATA : payload data
CHK : checksum

Figure 8 : Message LNP d'intégrité

Le champ longueur de données étant sur un octet, les données sont limitées à une taille de 255 octets et le paquet a donc une taille maximale de 258 octets.

- ***Les messages d'adressage :*** ils sont équivalents à un message d'intégrité contenant à la fois des données et des informations d'adressage dans le champ *IDATA*.

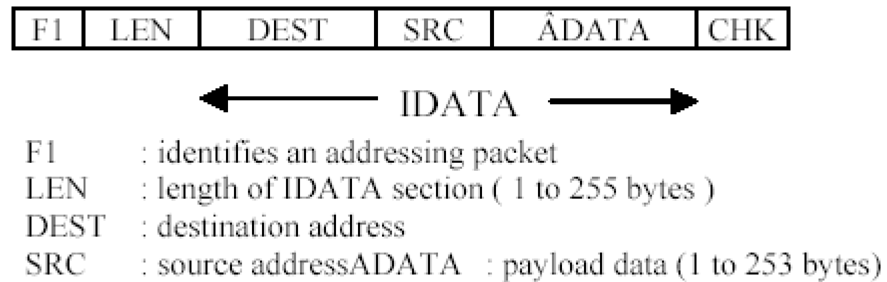


Figure 9 : Message LNP d'adressage

Ce sont des paquets similaires aux paquets UDP c'est à dire qu'il n'y a aucunes garanties sur leur arrivée. En revanche, si les paquets arrivent ils ne contiendront aucune erreur. Les paquets sont toujours envoyés à un hôte spécifique sur un port spécifique sur lequel un handler doit écouter : dans le cas contraire ou si une erreur se produit, alors le paquet est rejeté. Les champs *DEST* et *SRC* correspondent respectivement à l'adresse du destinataire et de la source.

Dans l'état actuel, le protocole LNP n'est donc pas fiabilisé. Il existe cependant un troisième type de paquets qui permet l'acquittement de données. L'émetteur attend un acquittement du récepteur avant de poursuivre sa transmission.

Un système de time-out est mis en place et qui force la retransmission d'une donnée. Le nombre de retransmissions est limité à 5 par défaut avant la destruction du paquet.

Cependant ce type d'échange n'est pas standardisé dans le protocole LNP et n'a pour l'instant été implanté que dans le programme de téléchargement des données vers le RCX.

Un protocole tel que LNP de type UDP constitue le meilleur choix lorsque l'on privilégie les performances par rapport à l'intégrité, par exemple dans le cas d'une communication audio où l'on supporte de perdre quelques bits.

4.4. Construction des adresses

Les adresses LNP source et destination sont contenues dans le champ de données du message d'adressage (Cf. *3.1 Les messages LNP*). Chaque champ est codé sur un octet, qui se divise lui-même entre une adresse hôte et un numéro de port associé.

La macro CONF_LNP_HOSTMASK définie dans le fichier /lnp/sys/lnp.h ou /dll-src/config.h permet de spécifier le nombre de bits qui sont respectivement utilisés pour l'adresse et le port de l'hôte. L'utilisateur peut donc fixer le format des champs selon le nombre de RCX qu'il souhaite faire communiquer et le nombre de processus destinés à être implantés.

Par défaut, CONF_LNP_HOSTMASK a pour valeur 0xF0 : il y a donc 4 bits réservés pour les adresses hôtes et 4 bits réservés pour leurs ports, soit 16 hôtes comprenant chacun 16 ports.

Pour définir un adressage spécifique, il suffit de positionner la macro avec la valeur hexadécimale correspondante, par exemple 0x80 pour 2 machines comprenant chacune 128 ports.

4.5. Attribution des adresses

L'adresse de l'hôte peut être définie grâce à la macro CONF_LNP_HOSTADDR : elle est donc spécifiée de manière statique lorsque l'on télécharge un programme sur le RCX ou lorsque l'on exécute un programme client sur le PC.

L'adresse de destination des messages est quand à elle passée en paramètre de la fonction lnp_adressing_write() : de cette manière, on définit l'adresse de destination de chaque paquet envoyé.

Remarque :

On ne spécifie l'adresse de destination que dans le cas des messages d'adressage puisque les messages d'intégrité sont diffusés à l'ensemble des éléments connectés.

4.6. La détection de collision

L'algorithme de détection de collision implanté dans le protocole LNP permet de vérifier que les bits que l'on vient de transmettre ont bien été reçus par le destinataire.

Si une collision est détectée, la fonction txend_handler() du fichier lnp-logical.c est appelée de manière à stopper la transmission. Il est en effet inutile de continuer l'émission puisque LNP n'intègre pas de mécanisme de correction d'erreur : les erreurs dues au lien IR ne peuvent être corrigées.

Par ailleurs, afin d'éviter une situation de blocage lors de l'émission des différents hôtes, LNP met en œuvre un mécanisme de CSMA / CD. Lorsque le RCX reçoit un octet spécifique, il ne peut plus émettre pendant une durée définie dans la macro LNP_BYTE_SAFE du fichier include/lnp/sys/lnp-logical.h.
Le RCX scrute alors le support pour vérifier qu'aucune autre émission n'est pas déjà en cours : si le support est libre, alors il peut émettre un nouvel octet. Lorsqu'il a terminé la transmission d'une trame complète, l'hôte perd à nouveau le droit d'émettre pour une période spécifiée dans la macro LNP_WAIT_TXOK. Cette temporisation de l'émission permet aux éventuels autres RCX du réseau d'envoyer leurs données.
Si une collision se produit, chaque RCX stoppe sa transmission en cours pendant une durée aléatoire contenue dans la variable allow_tx. Les ré-émissions pourront ainsi se faire progressivement sans nouvelle collision.

Ce mécanisme permet d'éviter les situations d'interblocage entre les différents hôtes, ce qui est primordial si on veut faire du temps réel.

4.7. Le démon LNPd [LNP 01]

Le daemon LNPd, un démon linux, permet de communiquer plus facilement avec le RCX, lors de l'utilisation de BrickOS. Il n'est pas obligatoirement nécessaire, il n'est utile que

dans le cas où l'on souhaite établir une communication entre le code sur le RCX et celui sur l'ordinateur.

Il ne fait pas partie intégrante de BrickOS et doit donc être téléchargé séparément. La seule chose à faire pour l'utiliser est de le lier à l'application.

Ce démon comme nous le verrons par la suite est indispensable pour l'intégration de µIP.

5. µIP [uIP01]

5.1. Caractéristiques

µIP constitue l'implémentation la plus minimaliste de la pile IP définie par la RFC1122. Elle est destinée à être embarquée sur des microcontrôleurs tels que le RCX ayant une faible quantité de mémoire embarquée ainsi qu'une faible puissance de calcul. En effet, la taille du code est de l'ordre de quelques kilo-octets et l'utilisation de la RAM peut-être configurée de façon à n'occuper que quelques centaines d'octets.

De plus, l'utilisation de cette pile facilite l'interfaçage avec LNP puisque l'intégration de lnplib a été prévue. Dans le cas de IP classique il aurait fallu développer de nouvelles fonctions pour appeler les sockets du noyau.

Cette pile µIP ne peut gérer qu'une seule interface réseau et se concentre sur les protocoles IP, ICMP et TCP : le protocole UDP n'est implanté que de manière rudimentaire comme nous le verrons au paragraphe 5.4.5.

Le tableau suivant dresse un récapitulatif des différentes caractéristiques présentes dans la pile µIP :

Feature	uIP
IP and TCP checksums	x
IP fragment reassembly	x
IP options	
Multiple interfaces	
UDP	
Multiple TCP connections	x
TCP options	x
Variable TCP MSS	x
RTT estimation	x
TCP flow control	x
Sliding TCP window	
TCP congestion control	Not needed
Out-of-sequence TCP data	
TCP urgent data	x
Data buffered for rexmit	

Figure 10 : Caractéristiques de µIP

Plus généralement, l'ensemble des fonctionnalités requises pour une communication point à point entre 2 hôtes ont été conservées. En effet, un algorithme de calcul de checksum est mis en œuvre, la gestion de plusieurs ports est supportée permettant ainsi

l'exécution de plusieurs applications simultanées. En revanche il n'y a pas de mécanisme de fenêtre glissante, le contrôle de congestion n'étant pas nécessaire puisque l'on se place dans le cadre d'une communication point à point.

5.2. Gestion mémoire

La pile µIP n'utilise pas une gestion de la mémoire dynamique : elle dispose d'un tampon global pour stocker les paquets ainsi qu'une table statique afin de conserver les états de connexions.

Le tampon pourra contenir un seul paquet de taille maximale : lorsqu'un paquet est reçu et qu'il contient des données, l'application de niveau supérieur doit être avertie. Par ailleurs, les données utiles devront être copiées par l'application afin qu'elles ne soient pas perdues lors de l'arrivée du paquet suivant. Si ce dernier arrive alors que les données sont en cours de traitement, il est stocké dans un second tampon soit par l'équipement réseau soit par son driver.

Le tampon global est à la fois utilisé pour les paquets entrants et les en-têtes des paquets sortants : il est utilisé pour stocker les en-têtes des paquets sortants avant transmission.

En cas de retransmission, les données ne sont pas stockées dans le tampon, c'est l'application qui se chargera de les reproduire à la demande de µIP.

5.3. API

Les API socket BSD employées dans les systèmes d'exploitation Unix ne conviennent pas pour µIP parce qu'elles nécessitent des ressources mémoire trop importantes. Le mécanisme implanté dans µIP utilise un système d'évènements dans lequel une application est invoquée en réponse à un certain type d'évènement.

Pour ce faire une application (fonction C) est mise en place au dessus de µIP. La pile l'appelle lorsque des données sont reçues, quand des données ont été transmises avec succès, quand une connexion a été établie ou encore pour demander une retransmission.

Les temps de réponses doivent être faibles puisque l'application doit être capable de traiter les paquets entrants et les demandes de connexion, en même temps que la pile IP reçoit les paquets.

Comme expliqué précédemment, pour une meilleure gestion de la mémoire, µIP fait appel à l'application pour régénérer les paquets à retransmettre. Un flag est alors envoyé pour demander une retransmission : l'application le vérifie et produit la donnée adéquate.

Cette technique ne complexifie pas l'application puisque la retransmission s'apparente à l'envoi d'un paquet classique : la seule difficulté se situe au niveau de µIP qui doit savoir quand la demander.

5.4. Implantation de µIP

Contrairement à la pile IP classique, l'implantation des protocoles de µIP est étroitement liée dans le but de réduire l'utilisation mémoire.

5.4.1. La boucle de contrôle principale

Une boucle de contrôle principale a été définie : elle consiste en deux opérations réalisées périodiquement :

- Vérification de l'arrivée d'un paquet depuis le réseau,
- Vérification de la validité des timers.

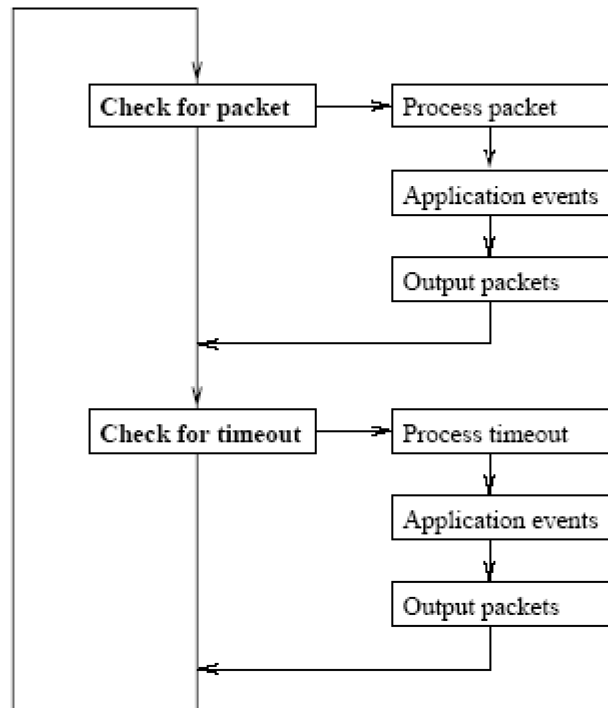


Figure 11 : Boucle de contrôle principale

Lorsqu'un paquet arrive, la routine de la pile TCP/IP est invoquée. Le paquet est traité par la pile ou l'application concernée qui peut alors générer un ou plusieurs paquets de réponse traités par les couches basses.

Lorsqu'un timer expire, il fait appel aux mécanismes TCP. Les couches basses se chargent de retransmettre le paquet perdu.

5.4.2. Internet Protocol

La seule différence avec le protocole IP classique est que l'on ne tient pas compte des options.

5.4.2.1. Fragmentation IP

µIP reprend le principe utilisé dans IP, à savoir l'utilisation d'un buffer pour stocker tous les fragments reçus. Toutefois, un seul paquet à la fois pourra être réassemblé, ce qui ne semble pas poser de problème puisque ce mécanisme est rarement mis en place.

5.4.2.2. Broadcast et multicast

Pour l'instant aucune implémentation n'est encore développée.

5.4.3. ICMP : Internet Control Message Protocol

Par souci de simplification, seuls les messages echo request et echo reply ont été implémentés. Cela est fait de manière simple : pour répondre à un echo request on échange l'adresse source et destination dans l'en-tête du paquet de réponse et on recalcule le checksum.

5.4.4. TCP : Transmission Control Protocol

L'implémentation de TCP est pilotée par l'arrivée des paquets et l'expiration de timer par l'intermédiaire de la boucle de contrôle principale. On s'adresse ensuite à l'application pour un traitement adéquat selon que l'on a reçu un paquet de données ou un acquittement.
Chaque fonctionnalité de TCP est donc mise en place par l'intermédiaire d'une fonction spécifique. Par exemple pour recevoir des données on utilisera :

- la fonction uip_newdata() pour savoir lorsque des données ont été envoyées
- la fonction uip_datalen() pour récupérer la longueur des données

Pour une liste exhaustive des fonctions se reporter au chapitre 1.4 de [uIP02]

5.4.5. UDP : User Datagram protocol

Le protocole UDP implanté dans µIP n'est pas encore complètement achevé. En effet, la réception ou l'envoie de paquets en broadcast ou multicast n'est pas supportée, seule des applications essentielles sont développées comme l'envoie de requêtes DNS.

On peut aussi définir le nombre de connexions UDP concurrentes et si le checksum doit être calculé. On précisera également le nom de la fonction C qui doit être appelée quand un datagramme UDP est reçu.

On pourra se reporter au chapitre 5.15 de [uIP02] pour plus d'informations sur la configuration d'UDP.

Remarque :

Il est dommage que seule une implantation rudimentaire de UDP soit présente dans µIP. En effet, pour des communications temps réel c'est le protocole le plus adapté puisqu'il n'y a aucune retransmission et donc aucune perte de temps.

5.5. Services offerts par LNP à µIP

5.5.1. Pré-requis

Tout d'abord, il faut charger la pile µIP sur le RCX. Ce thème sera abordé dans la partie *5 Procédures d'installation*.

5.5.2. *Intégration dans le kernel de BrickOS*

L'idée est d'utiliser la couche d'intégrité comme moyen de transport des paquets IP ce qui permet de bénéficier de la détection de collision tout en ayant un overhead minimal de 3 octets correspondants à F0, la longueur et le checksum.

De plus, le fait d'utiliser une plie TCP/IP, même réduite, sur le RCX et une passerelle LNP-TCP/IP sur l'ordinateur permet également d'utiliser un langage de programmation qui supporte les sockets.

Par ailleurs en utilisant LNP on continue à faire des économies de mémoire.

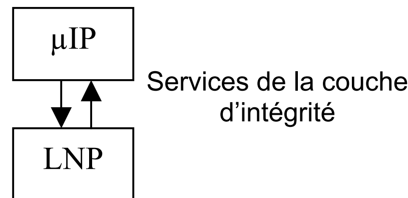


Figure 12 : Interfaçage entre LNP et µIP

5.5.3. *TCP/IP et le démon LNPd*

Le démon LNPd est utilisé pour interagir avec l'application cliente en tant que passerelle TCP/IP-LNP.

Sur le RCX, la pile µIP se charge de la transmission des paquets en s'appuyant sur les messages d'intégrité.

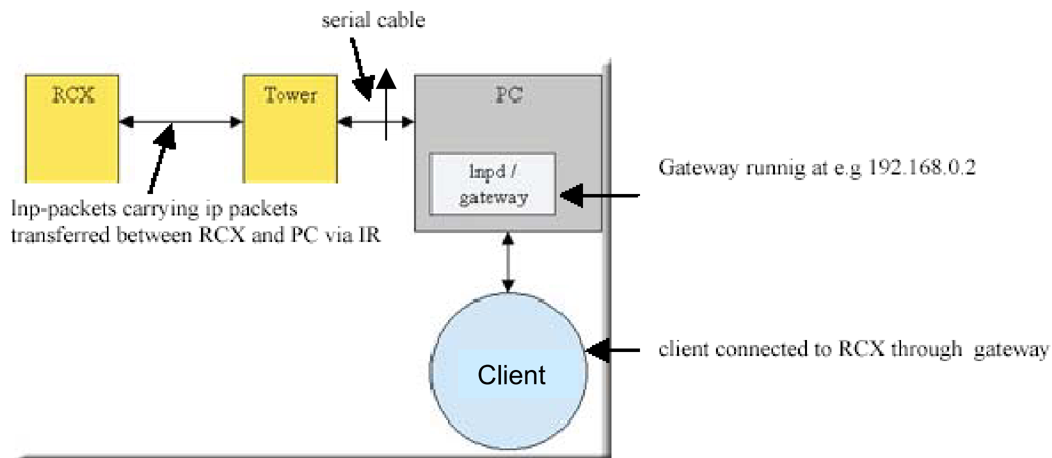


Figure 13 : µIP et LNPd

En effet, il est plus facile d'utiliser les messages d'intégrité ; dans le cas contraire il est nécessaire de développer une table statique contenant une association entre le numéro de port LNP et l'adresse de chaque RCX. Elle serait stockée sur le PC et remise à jour périodiquement par un mécanisme de polling. On laissera donc à l'application le soin de déterminer si les messages la concernent.

De plus le plus souvent, un seul RCX est considéré, donc l'utilisation du broadcast n'est pas gênante.

6. Conseils d'installation [HOWTO 01]

Cette partie vise à décrire la configuration et l'installation de BrickOS sur un noyau 2.6.x[4].

6.1. Construction du cross compiler

6.1.1. Binutils

Les GNU binutils sont une collection d'outils binaires. Parmi ceux-ci il y a le Gnu Linker (ld) et le GNU assembler (as). La version à installer est la 2.15[5], les sources sont téléchargeables sur ftp.gnu.org/gnu/binutils.

<u>Configuration et installation</u>

Télécharger la dernière version des binutils et la sauvegarder dans /usr/local/src.
Ouvrir un shell, se placer dans /usr/local/src et extraire les sources :
% tar –xvf binutils-2.15.tar

Se placer dans le répertoire binutils
% cd binutils-2.15

On peut passer maintenant à la configuration pour le micro contrôleur HITACHI H8300
% ./configure – target=h8300-hms –prefix=/usr/local

(si une erreur est levée penser à vérifier que la librairie libc6-dev est correctement installée)

On commence alors la configuration
% make

On passe en root pour installer les binutils
% make install

La phase suivante est l'installation du cross compiler.

6.1.2. Installation du GCC Cross Compiler

Il faut choisir la dernière version du GNU Compiler Collection (GCC) qui pour le moment est la 3.4.2. Les sources sont téléchargeables sur ftp://ftp.gnu.org/gcc.

---

[4] Il est impératif d'avoir un noyau 2.6 ou supérieur pour que le port USB de la tour soit supporté.
[5] Il est conseillé de bien vérifier les numéros de versions afin de ne pas avoir de problème d'installation et d'imcompatibilté entre les différents outils.

Configuration et installation

Télécharger la dernière version de GCC et la sauvegarder dans /usr/local/src.
Télécharger la dernière version de newlib et la sauvegarder dans /usr/local/src.

Modifier son path
% export PATH=/usr/local/bin :$PATH

Extraire les sources de GCC dans /usr/local/src
%tar –xvf gcc-3.4.2.tar

Extraire les sources de newlib dans /usr/local/src
%tar –xvf nexlib-1.12.0.tar

Copier les librairies dans le répertoire source de GCC
% cp –r newlib-1.12.0/nexlib gcc-3.4.2
% cp –r newlib-1.12.0/libgloss gcc-3.4.2

Créer un répertoire d'installation pour le GCC
% mkdir build-gcc

Se positionner dans ce répertoire
% cd build-gcc

On configure maintenant GCC en tant qu'un cross compiler pour le code C et C++
% ../gcc-3.4.2/configure —target=h8300-hms –prefix=/usr/local –enable-languages=c,c++ --with-gnu-as –with-gnu-ld –with-newlib

Compiler (cela peut prendre beaucoup de temps)
% make

Installer
% make install

6.2. BrickOS

La dernière version de BrickOS est la 0.2.6.10.6, elle se télécharge sur http://sourceforge.net.
Le seul problème possible est que la tour USB n'est pas encore supportée pour cela il faut donc appliquer un patch, seul le mode série est inclus.

Configuration et installation

Télécharger la dernière version de BrickOS et la sauvegarder dans /usr/local/src.
Télécharger le patch pour permettre le support de GCC 3.4 et de la tour USB et la sauvegarder dans /usr/local/src.

Extraire les sources dans /usr/local/src
% tar –xvf brickos-0.2.10.6.tar

Se placer dans le répertoire BrickOS
% cd brickos-0.2.10.6

Appliquer le patch
% patch –p1 <../ brickos-0.2.10.6-gc-3.4.2-usb.patch

Lancer le script de configuration
% ./configure

Si le gcc n'est pas trouvé, il faut éditer le script de configuration et ajouter le bon path dans la variable TOOL_PATH (ici /usr/local)

Compiler (cela peut prendre beaucoup de temps)
% make

Installer
% make install

6.3. Pour tester l'installation

6.3.1. Télécharger le firmware

La première étape consiste à télécharger le firmware sur le RCX. Avec la tour USB le port utilisé doit ressembler à /dev/usb/legousbtower0. Le simple fait de connecter la tour provoque une mise à jour automatique du noyau qui charge le module adéquat. Il est possible de vérifier le bon déroulement de la procédure dans var/log/messages :

```
Sep 27 10:21:12 labor5 kernel: drivers/usb/misc/legousbtower.c: LEGO USB
Tower #0 now attached to major 180 minor 160
Sep 27 10:21:12 labor5 kernel: drivers/usb/misc/legousbtower.c: LEGO USB
Tower firmware version is 1.0 build 134
Sep 27 10:21:12 labor5 kernel: usbcore: registered new driver legousbtower
Sep 27 10:21:12 labor5 kernel: drivers/usb/misc/legousbtower.c: LEGO USB
Tower Driver v0.95
Sep 27 10:21:14 labor5 udev[13133]: configured rule in
'/etc/udev/rules.d/udev.rules' at line 111 applied, 'legousbtower0' becomes
'usb/%k'
Sep 27 10:21:14 labor5 udev[13133]: creating device node
'/dev/usb/legousbtower0'
```

Pour télécharger le firmware sur le RCX on utilise le programme firmdl3 de brickOS. Il se trouve dans /usr/local/bin après l'installation et est accessible via le PATH. Le fichier du firmware est /usr/local/lib/brickos/brickOS.srec. Avant de commencer le téléchargement le RCX doit être allumé et placé en vue directe de la tour.

% firm3dl --tty /dev/usb/legousbtower0 /usr/local/lib/brickos/brickOS.srec

Le tty doit contenir le nom du bon périphérique[6].

6.3.2. Lancer le programme de Démo

BrickOS contient plusieurs programmes de démonstration dans : /usr/local/share/doc/brickos/examples/demo

[6] Afin de ne pas avoir à préciser le périphérique il suffit de l'inclure dans un variable d'environnement : export RCXTTY=/dev/usb/legousbtower0

Se placer dans /usr/local/share/doc/brickos/examples/demo
% cd /usr/local/share/doc/brickos/examples/demo

Le makefile contient toutes les données nécessaires
% make

Sont alors créés plusieurs fichiers compilés d'extension .lx

On peut alors charger sur le RCX le programme hello-world :
% dll -tty /dev/usb/legotower0 /usr/local/share/doc/brickos/examples/demo/helloworld.lx

Pour l'exécution il suffit d'appuyer sur le bouton start du RCX.

6.4. Chargement de µIP

Le chargement de µIP est semblable à celui d'un des programmes de démonstration, il suffit de générer un fichier binaire .lx et de le transférer sur le RCX.

Du point de vue de l'ordinateur, il faut coder un client qui appelle les services de µIP.

Nous nous sommes inspirés du code du ficher **lnptest.c** disponible avec les sources de LNP pour le réaliser. Toutefois la compilation présente encore des problèmes.

Dans le cadre d'un ping l'application sur le PC fait appel à la fonction **echo_request()**.

Les fonctions de µIP sont donc vues comme une librairie pour l'application cliente.

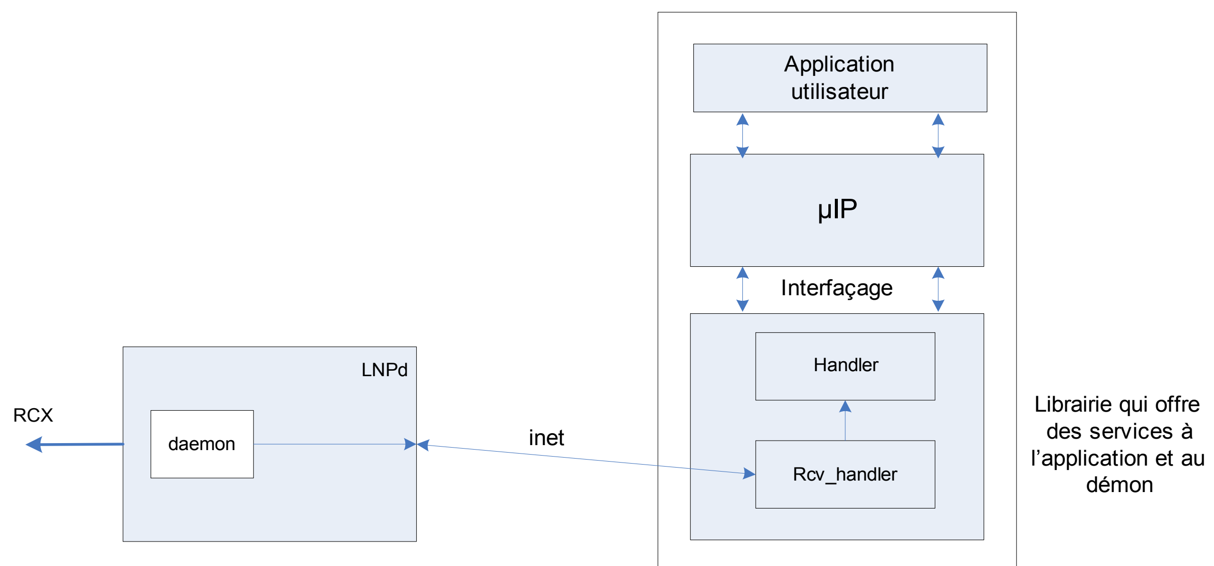


Figure 14 : Intégration de µIP avec LNP

7. Résultats obtenus

Nous avons installé sans difficulté les différents éléments permettant la communication entre le PC et le RCX : BrickOS, LNP et µIP. Toutefois, un problème de communication subsiste sur l'ordinateur et nous n'avons donc pas pu visualiser concrètement un échange complet entre le RCX et notre machine client.

En effet, nous avons dû modifier le code du protocole LNP afin qu'il puisse supporter la tour Lego USB ; nous avons ainsi réussi à transmettre des données en half duplex depuis le RCX vers le PC et inversement.

Cependant les échanges ne se font pas correctement et les données ne remontent pas jusqu'au client lorsque nous transmettons des données du RCX vers l'ordinateur. Dans le sens inverse, des collisions ont lieu.

Au cours de nos recherches nous avons pu identifier que ces problèmes provenaient de la communication entre sockets : les écritures / lectures entre les sockets du démon LNPd et celles de la librairie lnplib sont visiblement erronées ce qui entraîne un comportement anormal. Il nous a manqué quelques jours pour les résoudre complètement.
Le schéma suivant présente où se situe cette erreur :

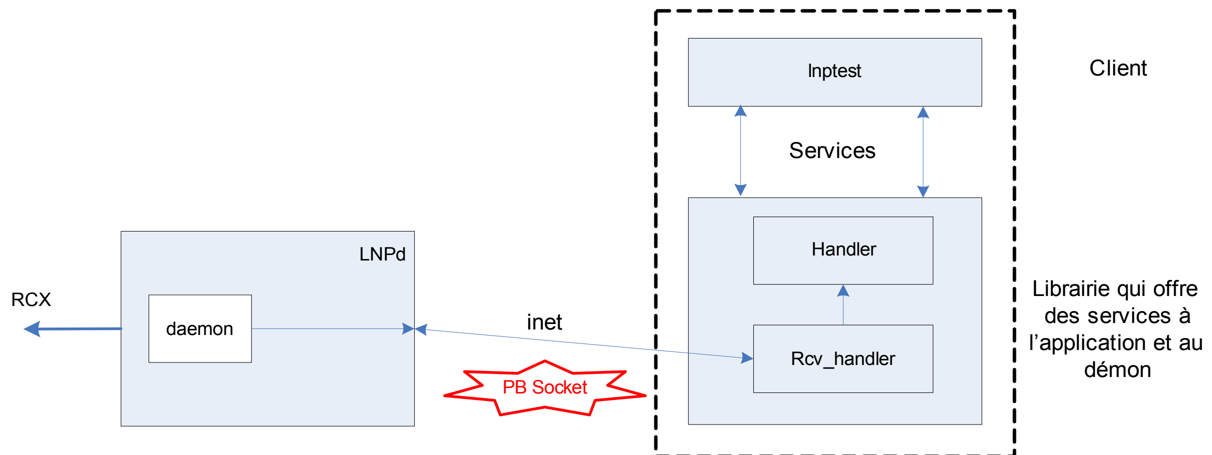


Figure 15 : Problème de communication entre sockets

Une fois résolus, µIP étant chargée sur le RCX, il sera alors possible d'exécuter la pile TCP / IP simplifiée et donc de vérifier le bon fonctionnement d'un protocole de niveau supérieur.

8. Conclusion

L'implantation des moyens de communication, BrickOS et LNP, sur le RCX se fait sans aucun problème. Il suffit de compiler les sources binaires sur le PC et de charger le firmware sur le RCX à l'aide de l'utilitaire firmdl3. De plus l'interfaçage avec une couche de niveau trois, comme µIP, qui fait appel à la librairie lnplib, se fait tout aussi simplement. Par conséquent, comme nous l'a montré l'analyse de brickOS, on disposerait les fonctionnalités nécessaires à une modélisation temps réel sur les robots Lego Mindstorms. L'utilisation de ces petits calculateurs permettrait ainsi la réalisation de maquettes temps réel de base.

Les difficultés rencontrées concernent essentiellement les drivers du matériel, en particulier le support de la tour USB, et la communication entre sockets inet.
Ainsi, pour les travaux à venir il serait intéressant de suivre l'évolution du projet brickOS dont la version 1.0 devrait être prochainement disponible, incluant un support natif de la tour Lego USB. De la même manière, le développement du protocole LNP reste actif et une nouvelle version intégrant un calcul du checksum au champ de données sera publiée dans les mois à venir.

Par ailleurs, comme le suggère Olaf Christ [LNP01], il pourrait être intéressant de réfléchir à l'implantation d'un autre protocole de niveau 2, tel que SLIP[7] ou PPP et de le comparer avec LNP. Un de leurs avantages majeur serait d'être multi plate-formes.

Enfin, pour pallier les limites de la liaison IR, on pourrait envisager l'utilisation de moyens de communication sans fils plus récents, WIFI et Bluetooth par exemple, en étudiant leur mise en œuvre, leur coût...

[7] Serial Link Internet Protocol RFC 1055 : Protocole de dialogue avec un réseau auquel on accède par simple liaison série

Table des Figures

Figure 1 : Organisation du RCX 6
Figure 2 : Trame IR 7
Figure 3 : Champ de vision Infra rouge d'un module RCX 9
Figure 4 : Cycle de vie d'un processus 12
Figure 5 : Bloc mémoire de base 13
Figure 6 : Les services LNP 15
Figure 7 : Fonctionnement LNP 16
Figure 8 : Message LNP d'intégrité 16
Figure 9 : Message LNP d'adressage 17
Figure 10 : Caractéristiques de µIP 19
Figure 11 : Boucle de contrôle principale 21
Figure 12 : Interfaçage entre LNP et µIP 23
Figure 13 : µIP et LNPd 23
Figure 14 : Intégration de µIP avec LNP 27
Figure 15 : Problème de communication entre sockets 28

Bibliographie

[BRICK01] http://brickos.sourceforge.net/about.htm#What%20is%20brickOS
[BRICK02] E. Basilico, "Exécutif LegOS" UNIGE / Université de Genève, centre universitaire d'informatique

[HOWTO 01] Matthias Ehmann, 2004
http://did.mat.uni-bayreuth.de/~matthias/veranstaltungen/ws2004/mindstorms/doc/brickos-howto.html

[IR 01] Stef Mientki, may 2001
http://oase.uci.kun.nl/~mientki/Lego_Knex/Lego_electronica/IR_tower/IR_tower.htm

[LEGO01] http://www.robotbooks.com/Lego-Mindstorms.htm
[LEGO02] http://www.automatesintelligents.com/labo/2002/archiveslegos.html

[LNP01] Olaf Christ, "TCP/IP enabled legOS" Student research project March 3, 2002
[LNP02] http://www.cs.brown.edu/courses/cs148/brickOS/HOWTO/lnp.html
[LNP03] Mike Ash, http://legos.sourceforge.net/HOWTO/x405.html

[uIP01] Adam Dunkels, "Full TCP/IP for 8-Bit Architectures" Swedish Institute of Computer Science, 2003
[uIP02] Adam Dunkels, "The uIP TCP/IP stack" Swedish Institute of Computer Science, 2003