HES-SO / Ecole d'ingénieurs de Genève
Laboratoire d'informatique industrielle

Mise en œuvre de la *carte MESH*

Yann Sagon

Historique

Yann Sagon	02 avril 08	modifications yottaOS
Yann Sagon	15 avril 08	ajout de la partie transmission de donnée
F. Rusco Y. Sagon	27 janvier 09	ajout description serveur et réseau général

 30 janvier 2009

http://eig.unige.ch/lii

Table des matières

Chapitre 1

Introduction

1.1 Carte MESH

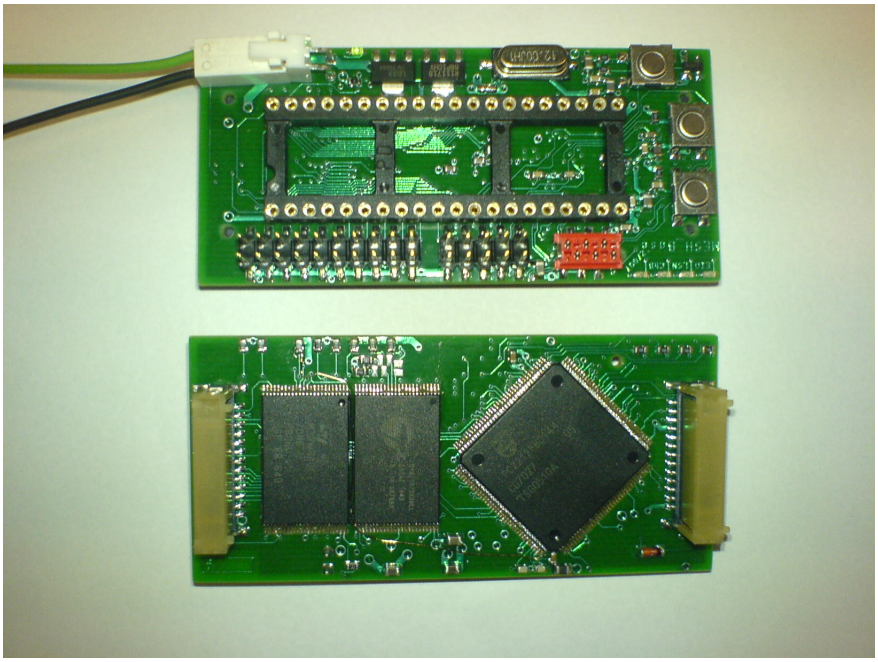

FIGURE 1.1 – Version LPC2212 de la *carte MESH*

La *carte MESH* (figure 1.1) est une petite carte développée au LSN[2] pour les besoins du projet RASMAS/MESH. Deux modèles existent, le plus performant dispose d'un processeur ARM de la famille LPC22XX pouvant accéder à de la mémoire flash et statique supplémentaire présente sur la carte et dispose également de la possibilité de connecter une mémoire 8 bits sur son port d'extension. La mémoire statique est le modèle CY62167DV30 qui a une taille de deux méga octets, tandis que la mémoire flash est le modèle M29W640FB qui apporte un espace de stockage de huit méga octets. Le plus petit modèle dispose quand à lui d'un processeur ARM de la famille LPC21XX et aucun ajout de mémoire additionnelle n'est possible. Ce dernier modèle ne dispose que de `16Kb` de mémoire statique et de `128Kb` de mémoire flash.

1.2 YottaOS

Dans le cadre du projet *RASMAS/MESH* un petit OS préemptif nommé *YottaOS* à été crée et mis à disposition par l'institut *MIS*. Les explications fournies dans ce document sont basées sur cet OS. Outre la documentation de *YottaOS*[4], le code source est également bien documenté. Il est par ailleurs fourni avec des exemples. *YottaOS* ne sera donc pas détaillé ici, mais uniquement son utilisation avec la *carte MESH*.

1.3 Communication avec la carte MESH

1.3.1 JTAG

La communication entre un PC et la *carte MESH* est effectuée au travers du port *JTAG* 20pins de la *carte MESH* avec un adaptateur *Amontec JTAGKey-tiny*. Le logiciel utilisé pour effectuer la liaison entre l'adaptateur *Amontec JTAGKey-tiny* et le PC est *OpenOCD*[11].

1.3.2 RS232

Outre la communication utilisant le port JTAG, il est également possible de communiquer entre un PC et la *carte MESH* à travers le port série 6pin de la *carte MESH*. Attention, il est nécessaire d'utiliser un adaptateur de tension afin d'être compatible avec un port série de PC. Le logiciel utilisé pour communiquer avec la cible par ce biais est fourni par *NXP*. Il s'agit de *LPC2000 Flash utility*[9].

1.4 Chaîne de compilation

La chaîne de compilation choisie est la chaîne classique *GNU* GCC, AS, LD et autres Binutils. Cette chaîne de compilation à l'avantage d'être largement utilisée et de ce fait il est plus aisé d'obtenir du support depuis de nombreux sites communautaires. Le code source de la chaîne est également disponible et modifiable selon les besoins.

1.5 Environnement de développement

L'environnement de développement proposé est le programme *Eclipse* auquel le greffon *CDT* est adjoint. Ce dernier permet de prendre en charge le langage *C* et *C++*.

1.6 Vue d'ensemble

La figure 1.2 montre une vue d'ensemble du processus de communication entre un PC et la *carte MESH*.

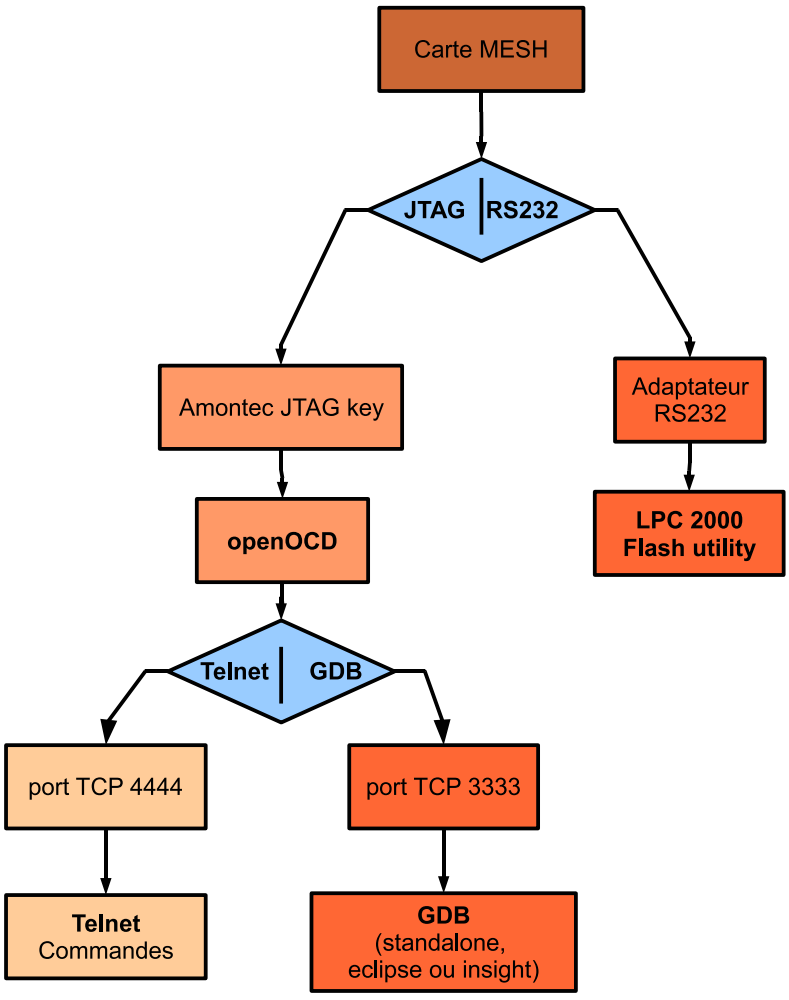

FIGURE 1.2 – Processus de communication PC-MESH

Chapitre 2

Mise en oeuvre rapide

Une documentation rédigée au format microsoft help est disponible à l'adresse suivante :
`https://wiki.lii.eig.ch/index.php/Projet_Rasmas#Documentation`

Chapitre 3

Présentation et installation des outils logiciels nécessaires

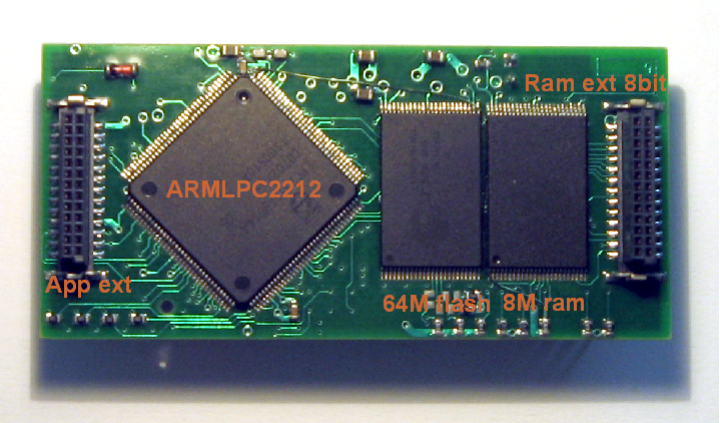

FIGURE 3.1 – Dos de la *carte MESH*

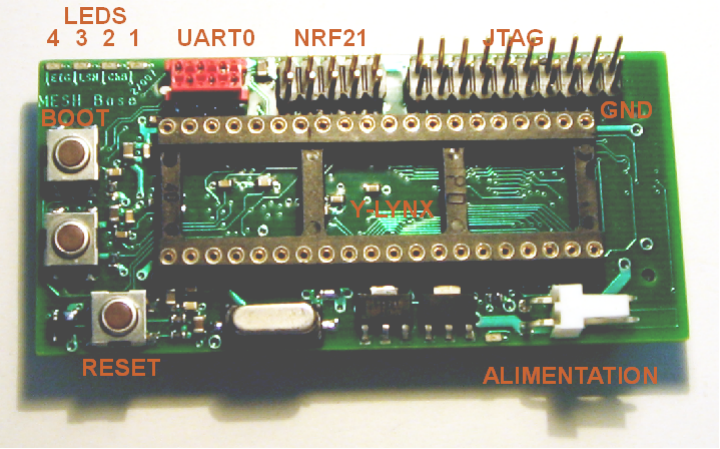

FIGURE 3.2 – Face de la *carte MESH*

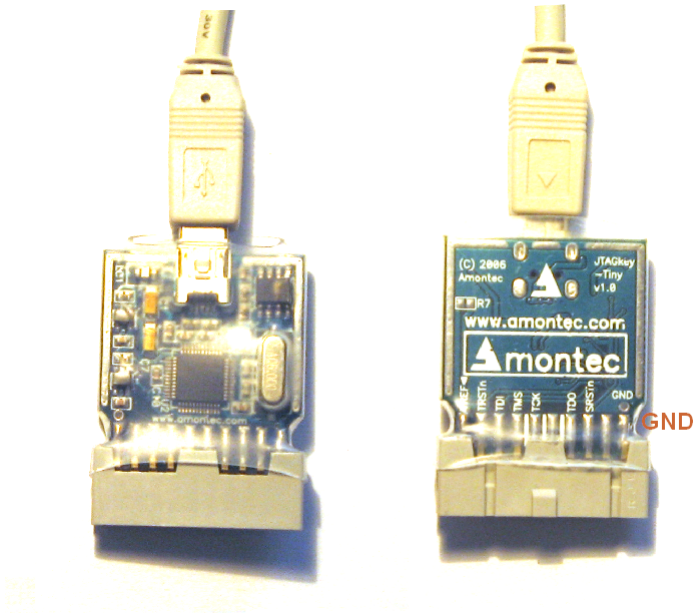

FIGURE 3.3 – Adaptateur *Amontec JTAGKey-tiny*

3.1 LPC2000 Flash utility

La société Philips (NXP) fournit gratuitement le programme *LPC2000 Flash utility* qui permet de flasher la mémoire de la *carte MESH* en utilisant son premier port série (`UART0` (fig. 3.2).

Le programme *LPC2000 Flash utility* peut être téléchargé sur le site de NXP[9]. Son installation ne pose pas de problèmes et n'est donc pas détaillée dans ce document.

3.2 Adaptateur Amontec JTAGKey-tiny

Le pilote de l'adaptateur *Amontec JTAGKey-tiny* (fig. 3.3) est librement téléchargeable depuis le site internet de Amontec[1]. A noter qu'il n'y a pas de version spécifique pour le modèle *tiny* de l'adaptateur, il suffit donc de prendre la version prévue pour le modèle standard de l'adaptateur. Commencer par installer le pilote, puis connecter l'adaptateur. Lorsque l'adaptateur est détecté par *Microsoft Windows*, se référer aux figures 3.4, 3.5 et 3.6. Ceci va installer le canal A de l'adaptateur. Refaire la même manipulation pour le canal B.

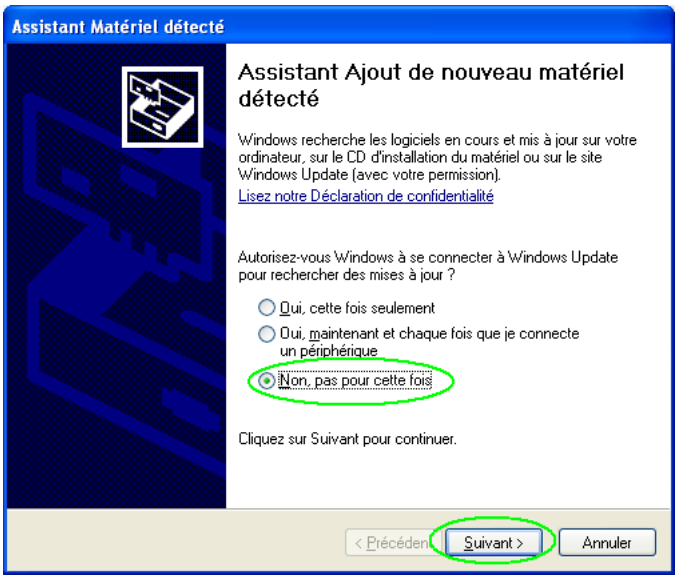

FIGURE 3.4 – Détection du périphérique

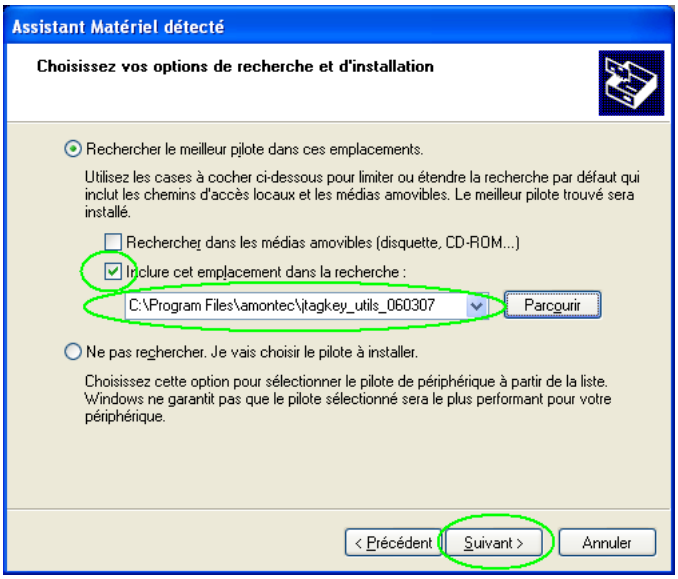

FIGURE 3.5 – Emplacement du pilote

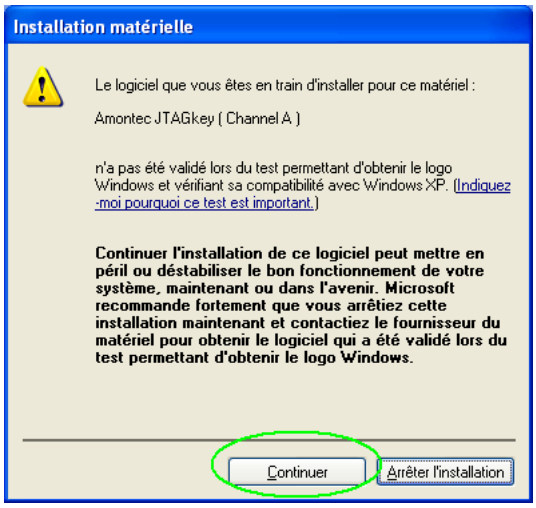

FIGURE 3.6 – Validation du choix

3.3 OpenOCD

3.3.1 Présentation

OpenOCD[11] est un logiciel qui sert à déverminer un programme sur une cible matériel, à programmer cette dernière ainsi qu'à tester la scrutation des frontières (boundary scan). Il sert d'interface entre la cible matériel et le programme de déverminage (GDB). *OpenOCD* permet de télécharger du code sur une cible matériel en mémoire statique ou en mémoire flash.

3.3.2 Installation

OpenOCD est disponible pour la plateforme Microsoft Windows sur le site de *Yagarto*[6]. La version actuelle est la ~~`re141-rc01`~~ `re204-rc01`. Lancer l'installation et se référer à la figure 3.7

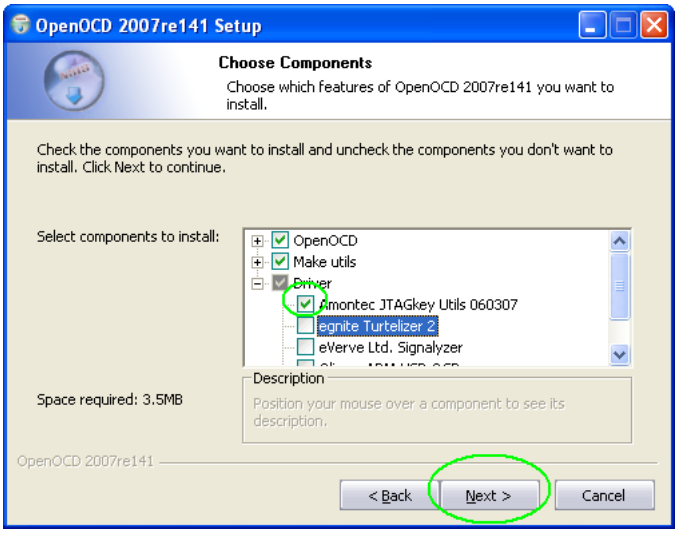

FIGURE 3.7 – Choix des pilotes de *OpenOCD*

3.3.3 Configuration de OpenOCD

La configuration de *OpenOCD* est dépendante de la cible. La configuration est effectuée dans un fichier externe. La configuration permettant de communiquer avec la *carte MESH*est décrite dans le listing A.1 en annexe. La configuration nécessaire au flashage de la mémoire de la *carte MESH*est décrite dans le listing A.2 en annexe. Ce dernier fichier nécessite un script lui indiquant comment flasher la *carte MESH* qui est disponible en annexe (listing A.3). Adapter si nécessaire le listing A.2 afin de spécifier l'emplacement de ce script.

3.3.4 Création d'un raccourci Microsoft Windows

Le plus simple pour exécuter *OpenOCD* est de créer un raccourci *Microsoft Windows*. Voir la figure 3.8 pour les paramètres importants. Adapter ces paramètres le cas échéant.

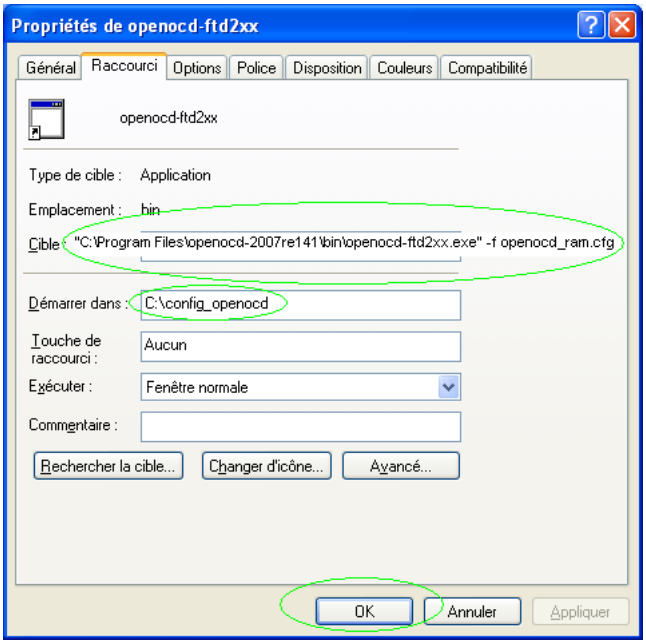

FIGURE 3.8 – Création d'un raccourci pour *OpenOCD*

3.3.5 Lancement de OpenOCD depuis Microsoft Windows

Lors du lancement de *OpenOCD*, le pare-feu (fig. 3.8) de *Microsoft Windows* peut éventuellement se déclencher. Ceci est normal car *OpenOCD* a besoin de deux sockets réseau en local sur les ports `3333` et `4444`. Choisir débloquer pour que *OpenOCD* fonctionne normalement.

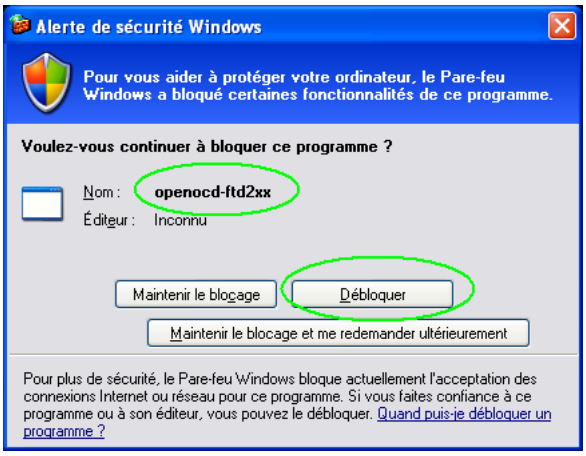

FIGURE 3.9 – Pare-feu de *Microsoft Windows*

3.3.6 Lancement de OpenOCD depuis Eclipse

*OpenOCD*peut être lancé directement depuis *Eclipse*. Choisir *External Tools* dans le menu *Run* de *Eclipse*. La figure 3.10 montre les points importants.

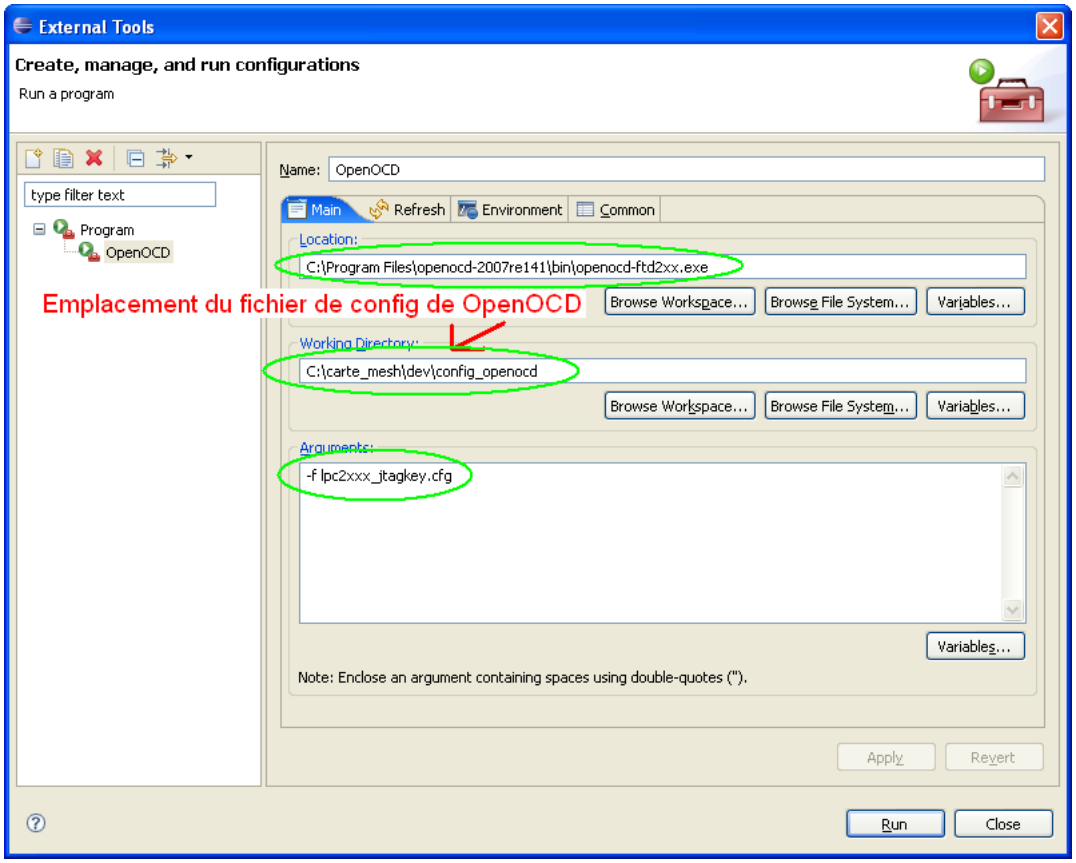

FIGURE 3.10 – Lancement de *OpenOCD* depuis *Eclipse*

3.4 YAGARTO GNU ARM toolchain

L'installation de *YAGARTO GNU ARM toolchain* fournit la chaîne de compilation nécessaire ainsi que le dévermineur *GDB* et une interface à ce dernier, *Insight*. Si la chaîne de compilation ne sera utilisée que avec cette cible, il est possible de ne pas installer la partie *big endian* de *YAGARTO GNU ARM toolchain* (fig. 3.11).

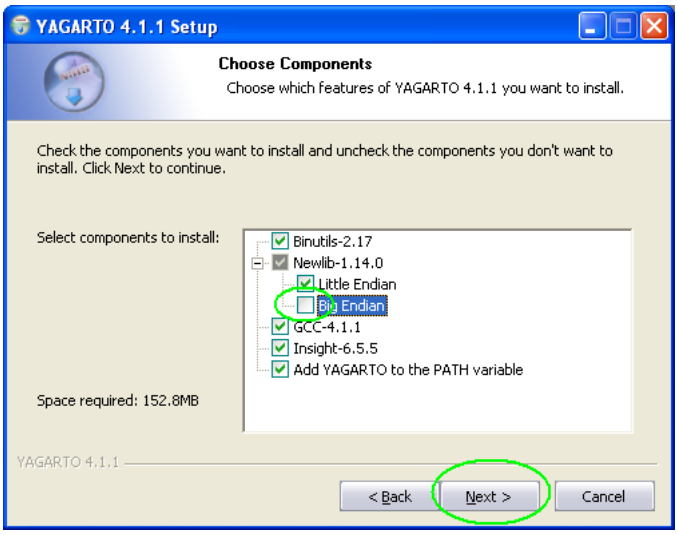

FIGURE 3.11 – Sélection des composants de *YAGARTO GNU ARM toolchain*

Attention, la chaîne de compilation fournie sur le site de *Yagarto* n'est pas compatible avec les chemins au format unix. Une version compatible avec les chemins unix est disponible sur le site de *GNU ARM toolchain*[8].

3.5 Java

L'IDE utilisé (sec. 3.6) nécessite la présence d'un runtime Java[13]. Son installation ne pose pas de problème et n'est pas détaillée. Pour référence, la version 6 de Java est installée lors de la rédaction du document.

3.6 Environnement de développement intégré

Un IDE (Integrated Development Environment) est fourni sur le site de *Yagarto*[5]. Il s'agit en fait de l'environemment *Eclipse* préconfiguré avec le greffon *Zillin CDT* qui est spécialement adapté au débuggage de logiciels en langage C ou C++ et adapté à l'embarqué. Son installation ne pose pas de problème et n'est pas détaillée. Un point est néanmoins important. Il faut s'assurer de ne pas utiliser d'espace dans le chemin constituant l'espace de travail (workspace) d'*Eclipse*.

Chapitre 4

Programmation

4.1 Eclipse

4.1.1 Lancement

Eclipse a besoin d'un espace de travail dédié et demande de le spécifier lors du lancement (fig. 4.1). Attention à spécifier un emplacement ne contenant pas d'espaces ou de caractères spéciaux.

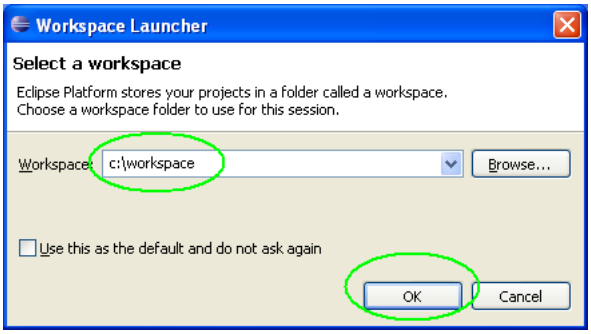

FIGURE 4.1 – Sélection de l'espace de travail

4.1.2 Création d'un projet

Pour créer un nouveau projet, aller dans le menu *File* et cliquer sur *new* puis sur *project*. Voir ensuite les figures 4.2, 4.3 et 4.4.

4.1.3 Compilation

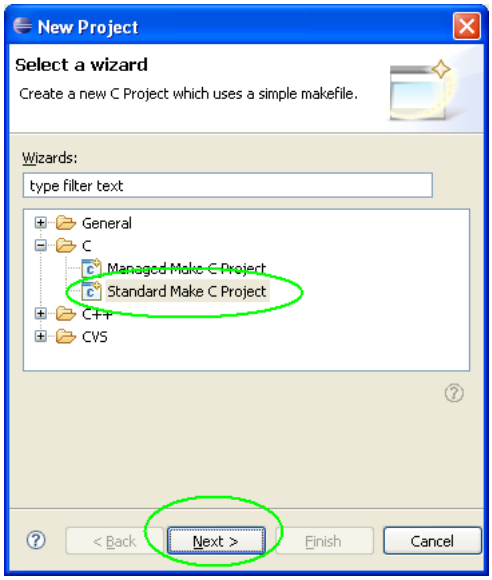

FIGURE 4.2 – Création d'un nouveau projet

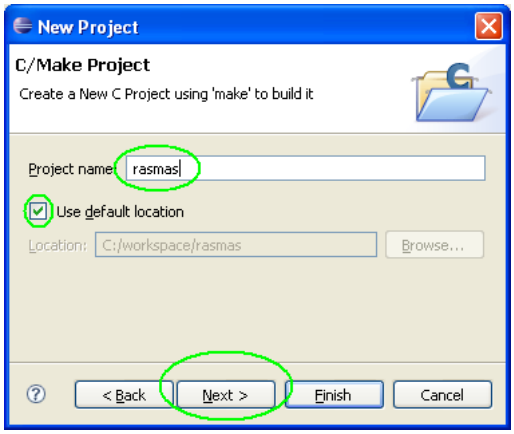

FIGURE 4.3 – Choix de l'emplacement et du nom du projet

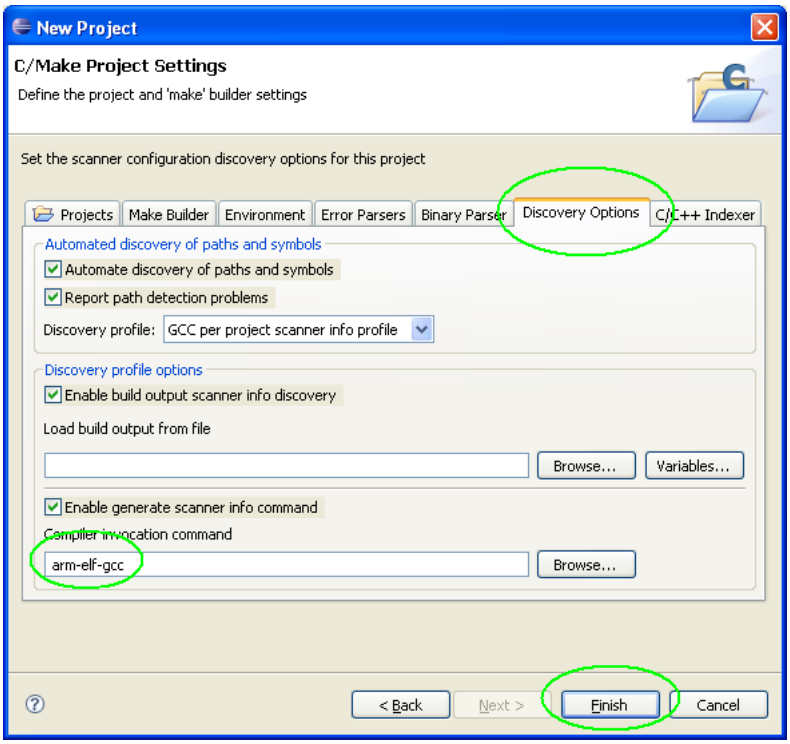

FIGURE 4.4 – Choix du compilateur

Une fois le projet crée, *Eclipse* propose de passer en perspective *C/C++* (fig. 4.5, accepter ce changement.

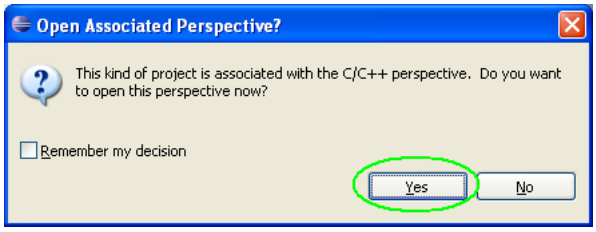

FIGURE 4.5 – Changement de perspective

En cliquant avec le bouton droit sur le projet, il est possible d'importer des fichiers existant dans le projet (fig. 4.6 et 4.7).

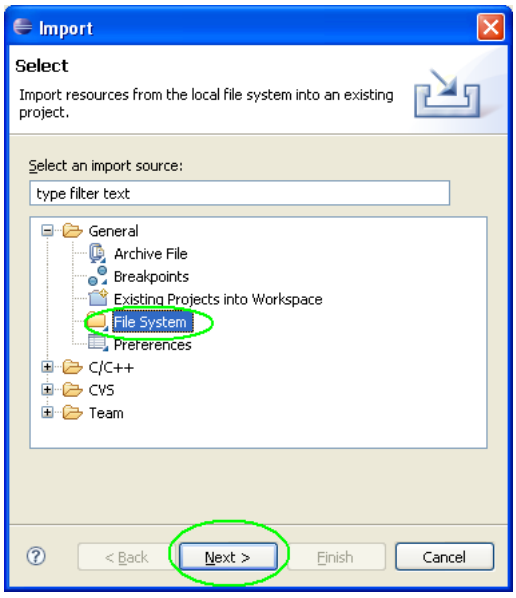

FIGURE 4.6 – Type d'importation

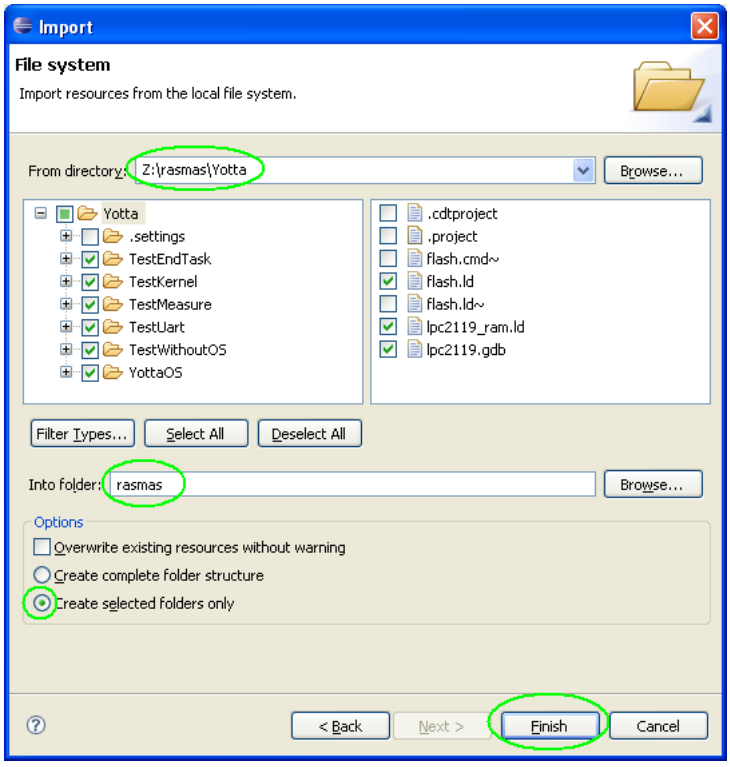

FIGURE 4.7 – Choix des fichiers

4.1.4 Création d'un profil de débuggage

Afin de débugger, le plus simple est d'avoir un exécutable ARM prévu pour être mis en mémoire et non en flash. Autrement, il est nécessaire de flasher l'exécutable préalablement sur la cible, ce qui n'est pas adapté à du code en développement. De plus, seul deux points d'arrêt matériel sont

à disposition. Il n'est en effet pas possible d'utiliser les points d'arrêt software lors de l'exécution en mémoire flash. Il faut commencer par créer un nouveau profile de débuggage en cliquant sur *Debug..* dans le menu *Run*. Se baser sur la figure 4.8 pour créer le profile. Ensuite Indiquer à *Eclipse* quel débugger utiliser (fig. 4.9) puis entrer les commandes nécessaires à l'initialisation de la cible ainsi qu'au téléchargement du code en mémoire (fig. 4.10). Pour plus de détails concernant ce dernier point, se référer à la section A.5 des annexes.

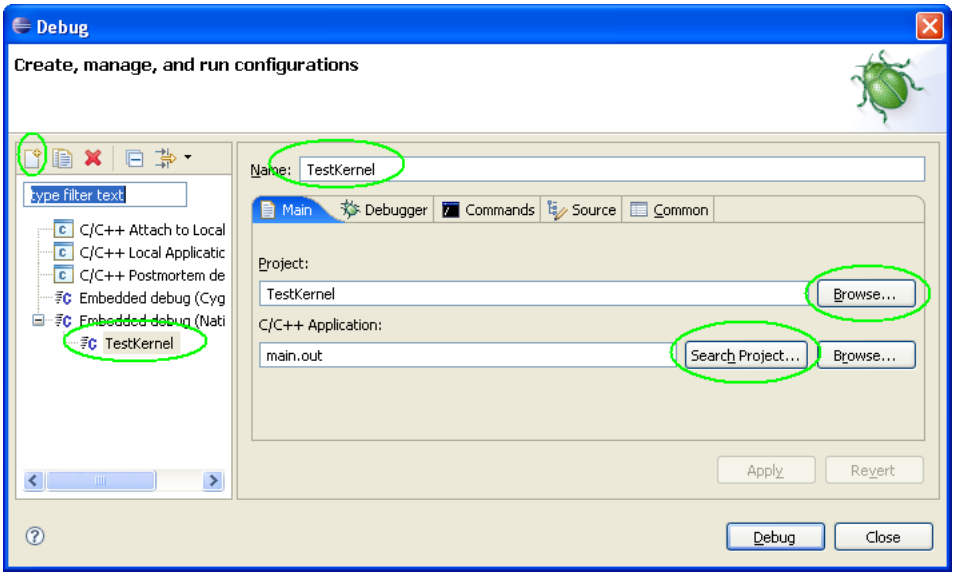

FIGURE 4.8 – Création d'un profile de débuggage

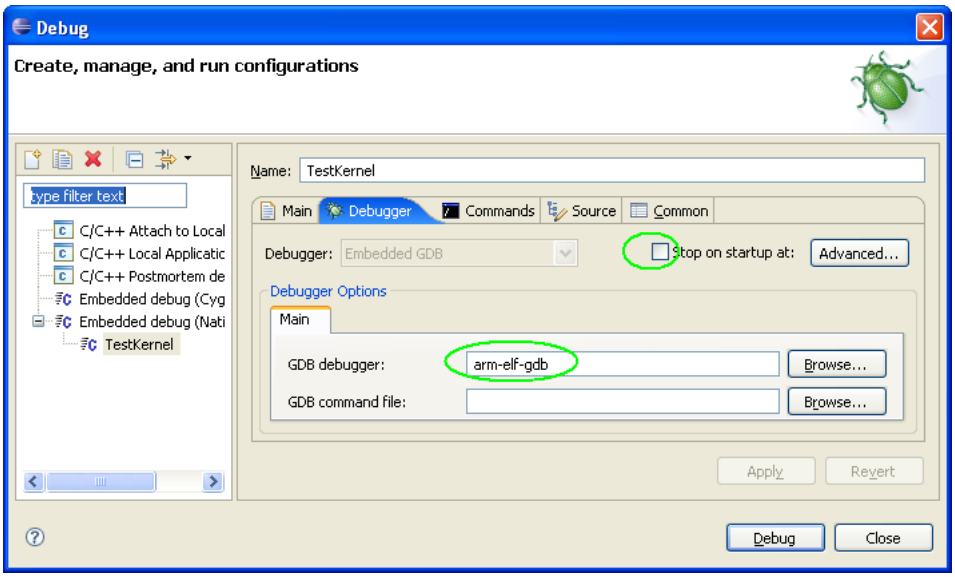

FIGURE 4.9 – Choix du débugger

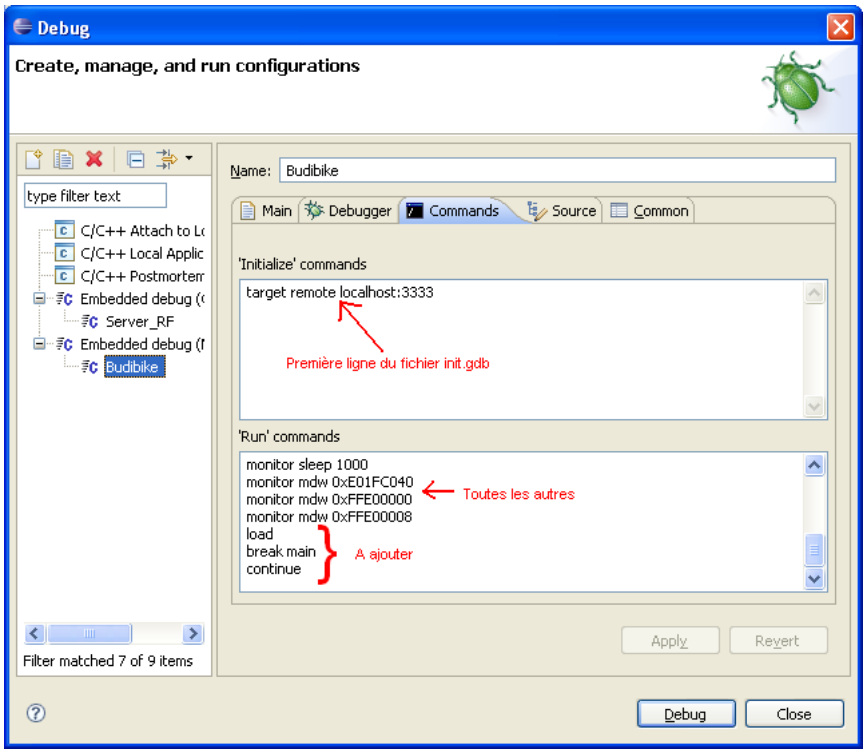

FIGURE 4.10 – Commandes d'initialisation de la cible

4.1.5 Envoie de messages de déverminage depuis la cible MESH

Lors d'une session de déverminage, il est possible d'envoyer des messages à travers le canal DCC du processeur ARM, donc par le bias du port JTAG. L'avantage est de laisser libre les deux ports séries pour un autre usage. Pour mettre en oeuvre ce moyen de communication, il faut

4.1.6 Lancement d'une session de débuggage

Afin d'effectuer un débuggage en mémoire sur la cible, il faut préalablement exécuter *OpenOCD* (sec. 3.3.5)puis sélectionner le profil de débuggage précédement crée (section 4.1.4). Cliquer sur *Debug* afin de commencer la session de débug. *Eclipse* va commencer par télécharger le code sur la cible puis va s'arrêter sur la première instruction composant la fonction *main*. La figure 4.11 montre un aperçu des possibilitées de debug offertes par *Eclipse* en conjonction avec *GDB*.

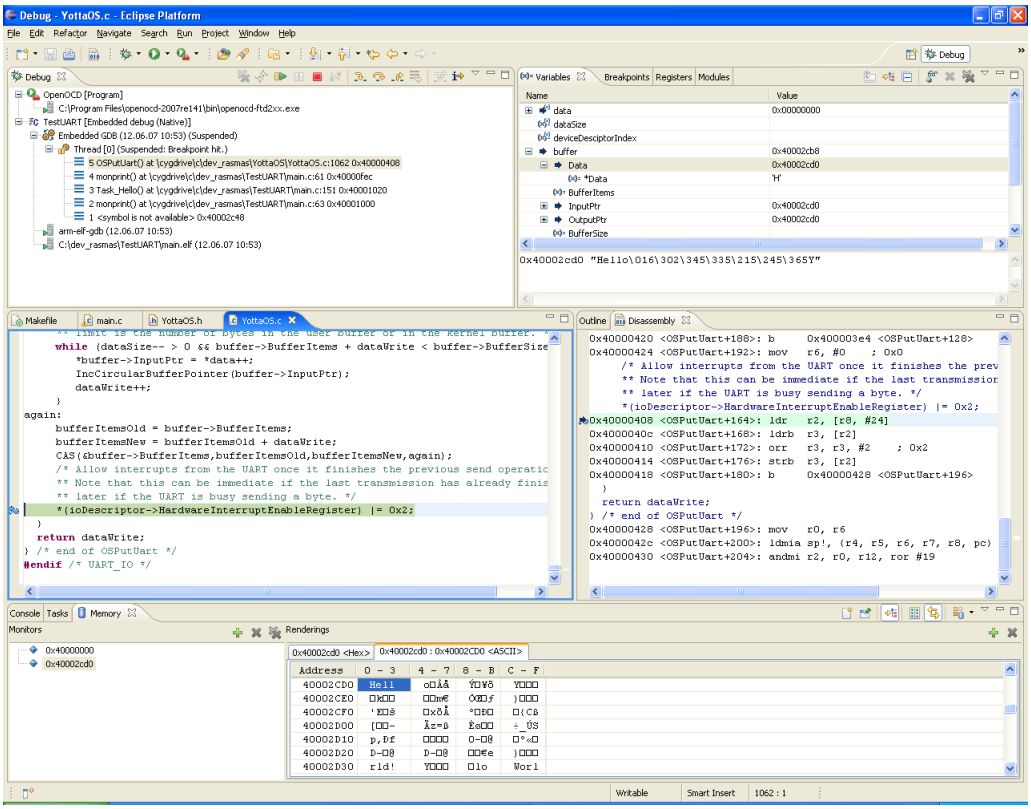

FIGURE 4.11 – Session de débuggage avec *Eclipse*

4.1.7 Codes d'erreurs lors de la session de débuggage

Si une erreure a lieu, c'est à dire qu'un des handler autre que interrupt et service est atteint, les leds de la cible mesh clignotent a intervalle régulier.

```
1 = undefined instruction (one blinks ........ long pause)
2 = prefetch abort        (two blinks ........ long pause)
3 = data abort            (three blinks ...... long pause)
4 = swi handler           (four blinks ....... long pause)
```

4.1.8 Drapeaux optionels

`_EXTERNAL_MEMORY_` ASM Si défini, active la mémoire externe de la cible MESH `_DEBUG_` C Si défini, ne met pas le CPU en mode veille.

Chapitre 5

Intégration avec *Esterel*

5.1 Préambule

Esterel[7] est un langage de programmation réactif synchrone, particulièrement adapté à la programmation matériel. Un programme *Esterel* peut être converti en langage VHDL ou C afin d'être synthétisé sur une FPGA ou intégré dans un programme existant. L'environnement de développement de choix est *Esterel Studio*. Ce dernier permet de créer un programme *Esterel* de façon textuel et/ou graphique (sync charts). Une première approche de développement du protocole de transmission de la carte mesh non utilisé en définitive est modélisée en langage *Esterel*. Cette documentation n'explique pas le fonctionnement d'*Esterel*. Le lecteur peut se référer au livre *Programmation synchrone de systèmes réactifs avec* Esterel *et les Syncharts*[15] qui représente une base très complète concernant le développement avec *Esterel*.

5.2 Création d'un programme *Esterel*

5.2.1 Création d'un automate

La figure 5.1 représente deux automates simples dans l'environnement de développement faisant usage de la notion de parallélisme offerte par *Esterel*. La figure 5.2 montre quant à elle la déclaration des signaux d'entrées/sorties qui interagissent avec le programme.

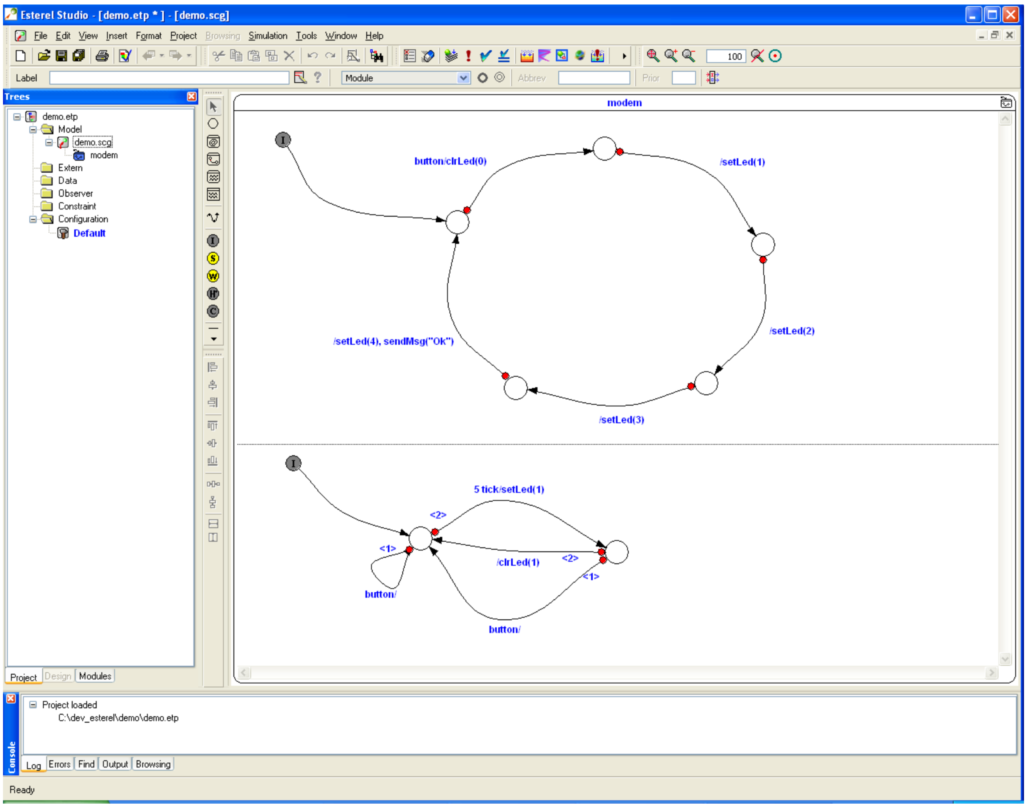

FIGURE 5.1 – Ecran principal d'*Esterel*

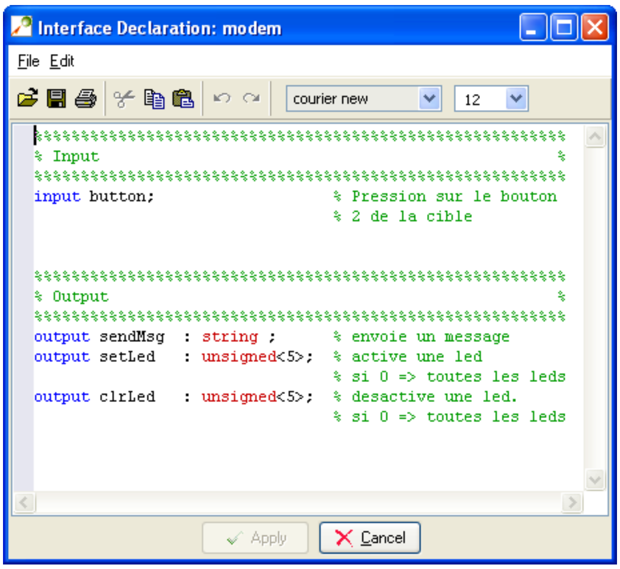

FIGURE 5.2 – Déclaration de l'interface *Esterel*

5.2.2 Génération du code *C*

Une fois le programme simulé et testé, il est nécessaire de générer le code équivalent en langage C. Ceci est réalisé en sélectionnant *Generate code* dans le menu *Project* de *Esterel Studio*. Le code

généré par *Esterel* se trouve dans le répertoire indiqué ci-dessous :

`chemin_de_base_du_projet\Default\code\`

Il est constitué de fichiers suivants :

`nom_du_module.c`
`nom_du_module_strl.h`

Le fichier *.c* contient l'automate esterel, tandis que le fichier *.h* contient le prototype des fonctions formant l'API de l'automate *Esterel*.

5.3 Inclusion des fichiers d'entête dans le programme principal

L'inclusion du fichier d'entête dans le programme *C* ou *C++* à destination de la *carte MESH* est effectuée grâce aux lignes suivantes, ceci pour un automate *Esterel* nommé *demo*.

```
/* Include pour Esterel */
#include "demo_strl.h"
#include "demo.h" //si existant
```

5.4 Implémentation des fonctions de sorties de l'automate

Il est nécessaire d'implémenter le code des prototypes des fonctions de sorties présents dans le fichier d'entêtes `demo_esterel.h`. Le prototype de ces fonctions est automatiquement généré par *Esterel* suivant l'interface (voir figure 5.2) du module utilisé. En l'occurrence, dans le cas du module *demo*, il y a trois signaux valués en sortie et un signal non valué en entrée. Le listing suivant montre un exemple d'implémentation de ces trois fonctions.

```
void demo_O_sendMsg(string str){
   monprint(str, 2);
}

void demo_O_setLed(unsigned_int_type val){
   setLed(val);
}

void demo_O_clrLed(unsigned_int_type val){
   clrLed(val);
}
```

5.5 Intégration de l'automate *Esterel* dans *YottaOS*

5.5.1 Modifications dans le fichier *C* principal

La fonction de réaction de l'automate doit être sollicitée à interval régulier. Dans l'exemple, elle est appelée chaque milliseconde dans une tâche *YottaOS*. Il est nécessaire de s'assurer que le traitement de chacune des transitions composant l'automate est réalisable durant ce temps. Le cas échéant, le comportement du programme sera indéterminé. Le listing suivant montre une tâche *YottaOS* qui appelle la fonction de réaction de l'automate, et prend en compte l'appuie du bouton 2 de la *carte MESH*.

```
void Task_Demo(void){
   if(buttonState){ // le bouton a ete appuye
      demo_I_button(); // notification a l'automate
      buttonState = FALSE; // quitancement du bouton
   }
```

```
   demo(); //reaction de l'automate
      OSEndTask();
}
```

5.5.2 Modification à apporter au fichier `Makefile`

Il faut ajouter le fichier `demo.c` dans la liste des objets à compiler au fichier `Makefile`. Dans celui qui est fourni en exemple, il s'agit de la ligne suivante :

```
SRC =  functions.c \
 $(YOTTA)YottaOS.c \
 $(YOTTA)common.c \
 $(TARGET).c \
 $(ESTEREL)demo.c
```

Chapitre 6

Utilisation de la mémoire supplémentaire

6.1 Préambule

La *carte MESH* est déclinée en deux versions (sec. 1.1). Ce chapitre donne la marche à suivre afin d'activer la mémoire statique et la mémoire flash présente sur la *carte MESH* disposant du processeur LPC22xx. Ce dernier dispose d'un contrôleur de mémoire pouvant gérer jusqu'à quatre bancs de mémoires.

Le contrôleur de mémoire est paramétrable grâce au registre `PINSEL2` qui spécifie comment les mémoires sont connectés physiquement au processeur ainsi qu'aux registres `BCFG0-3` qui spécifient les paramètres de temps d'accès et largeur de bus de donnée de chaque banc de mémoire.

6.1.1 Assignation des broches

L'assignation des broches est commune à tout les bancs de mémoires. Elle est effectuée grâce au registre `PINSEL2 - 0xE002C014`. Il suffit d'écrire à l'adresse de ce registre la valeur `0x0F814114`. Pour comprendre pourquoi, se référer à la description du registre `PINSEL2` dans le manuel utilisateur du LPC22xx.

6.2 Temps d'accès

A finir :

$$30MHz => 33\mu s/clk$$
$$60MHz => 16\mu s/clk$$
$$t_w = 45ns => 2clk$$

no bank	memoire	adresse de base
0	flash	`0x8000 0000`
1	-	-
2	static ram	`0x8200 0000`
3	ext `8bit` ram	`0x8300 0000`

TABLE 6.1 – Bancs de mémoires

6.3 Mémoire statique

6.3.1 Caractéristique

La mémoire statique présente sur la *carte MESH* est le modèle CY62167DV30[3] de *Cypress*. Il s'agit d'une mémoire de 16Mbit (1M x 16) de haute performance. Elle dispose d'un mode d'économie d'énergie qui s'active lorsque les lignes d'adresses ne sont pas modifiées durant un certain laps de temps.

6.3.2 Configuration

La configuration associée à la mémoire statique présente sur la *carte MESH* se fait en modifiant le registre `BCFG2 - 0xFFE00008` avec la valeur `0x10001420`. Pour comprendre pourquoi, se référer à la description du registre `BCFG2` dans le manuel utilisateur du LPC22xx.

6.4 Edition de liens

Il est nécessaire de d'indiquer au *linker* l'adresse des nouvelles zones mémoires. Ceci s'effectue dans un des fichiers de script de ld (lst. A.6 ou A.4 en annexe). Toutes les sections mis à part `.init` vont dans la mémoire supplémentaire. La section `.init` contient les vecteurs d'interruptions qui doivent se trouver à l'adresse de base de la mémoire interne ou à l'adresse `0x0` (dépend de la valeur du registre `MEMMAP`) ainsi que la partie nécessaire à l'initialisation du processeur ARM.

```
MEMORY
{
  ram     : org = 0x40000000, len = 16k
  /* ajout memoire externe */
  ram\_ext : org = 0x82000000, len = 2m
}
```

6.5 Téléchargement du code

Il est nécessaire d'activer la mémoire supplémentaire avant de télécharger le code sur la *carte MESH*. Pour cela il faut exécuter les deux lignes suivantes depuis *GDB*.

```
monitor mww 0xFFE00008 0x10001420
monitor mww 0xE002C014 0x0F814114
```

Le plus simple est de les ajouter comme paramètres à *Eclipse* (fig. 4.10 sec. 4.1.4).

6.6 Mémoire flash

6.6.1 Caractéristiques

La mémoire flash présente sur la *carte MESH* est le modèle `M29W640FT70N6E` de *ST*. Il s'agit d'une mémoire flash de 64Mbit ayant une vitesse de 70ns. La mémoire dispose de 135 blocs d'une taille de 8KB et de 64KB (fig. 6.1). Les blocs peuvent être protégés et déprotégés par groupe de quatre afin de prévenir un effacement accidentel. La mémoire flash dispose également d'un bloc de donnée étendu d'une taille de 256B qui peut-être protégé de manière non réversible. La mémoire flash peut-être configurée en mode x8 ou x16 pour l'accès aux données. Sur la *carte MESH* elle est configurée en mode x16 (broche `BYTE` haute).

Fonction	BCFG0 (mémoire flash)	BCFG2 (mémoire statique externe)	BCFG3 (mémoire 8bit externe)
IDCY	0	3	0
WST1	2	3	2
RBLE [a]	1	1	1
WST2	1	2	1
WPERR	0	0	0
WP	0	0	0
BM	0	0	0
MW	1	1	0

TABLE 6.2 – Configuration des bancs de mémoire

a. Attention, si pas à 1, la ligne write enable n'est pas activée lors d'une écriture

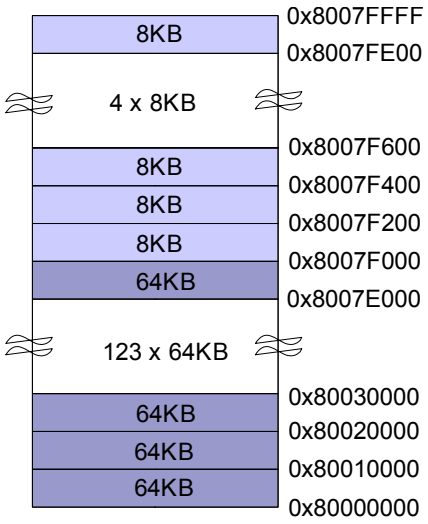

FIGURE 6.1 – Organisation de la mémoire flash

6.6.2 Configuration

L'activation de la mémoire flash est effectué à l'aide de deux registres : `BCFG0` et `PINSEL2`. Le tableau ci-dessous détaille le contenu du registre `BCFG0`

6.6.3 Accès à la mémoire flash

La lecture d'une donnée dans la mémoire flash est effectuée suivant le même procédé que pour une mémoire statique traditionnelle. L'écriture d'une donnée est par contre très différente. La mémoire flash dispose d'un interpréteur de commande. Ce dernier analyse chaque écriture effectuée sur le bus de la mémoire. Une commande consiste en l'écriture de plusieurs octets à des adresses spécifiques. Le tableau 6 de la documentation de la mémoire flash décrit les différentes commandes possibles. Le code suivant permet de lire l'identificateur de la mémoire flash :

```
0x555 = 0xAA // première écriture
0x2AA = 0x55 // deuxième écriture
0x555 = 0x90 // troisième écriture
id = ADRESSE_DE_BASE // (0x8000 0000)
```

Il faut noter que les adresses doivent être décalée de un à gauche (`0x555 ■ 1`) car le bus d'adresse de la mémoire flash est décalé de 1 entre le processeur et la mémoire flash (fig. 6.2

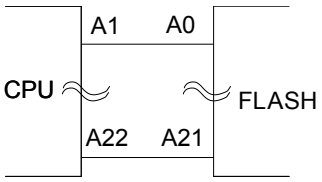

FIGURE 6.2 – Bus d'adresse entre la mémoire flash et le cpu

Des primitives pour utiliser la mémoire flash externe d'une carte Olimex[10] sont adaptées à la mémoire flash présente sur la *carte MESH*.

Chapitre 7

Contrôle et programmation de la carte mesh

7.1 Préambule

La carte mesh peut être programmée soit par le biais de son premier port série, soit par son connecteur JTAG. La programmation par le port JTAG est beaucoup plus rapide mais peut être impossible dans certains cas. En effet le port JTAG de la carte peut-être désactivé de façon logicielle, la seule méthode de programmation encore possible est par le port série.

7.2 Communication RS232

Il est possible de programmer la mémoire statique ou FLASH en utilisant le programme *LPC2000 Flash utility*[9] fourni par NXP (ex Philips). Afin de pouvoir connecter la *carte MESH* au port série d'un PC, il faut démarer la *carte MESH* dans le mode *programmation sérielle*. Pour ce faire, appuyer sur le bouton *Boot série* (fig. 7.2) puis appuyer sur le bouton *Reset*. Relâcher alors le bouton *Reset* puis à son tour le bouton *Boot série*. Exécuter le programme et entrer les paramètres adéquats (fig. 7.1) et démarer également la *carte MESH* en mode *programmation sérielle*. Si la communication est effective, le fait de cliquer sur le bouton *Read device ID* va afficher diverses informations sur la cible. Il est alors possible de manipuler la mémoire statique ou flash. A noter que le programme accepte uniquement un fichier de programamtion au format *Intel hex*.

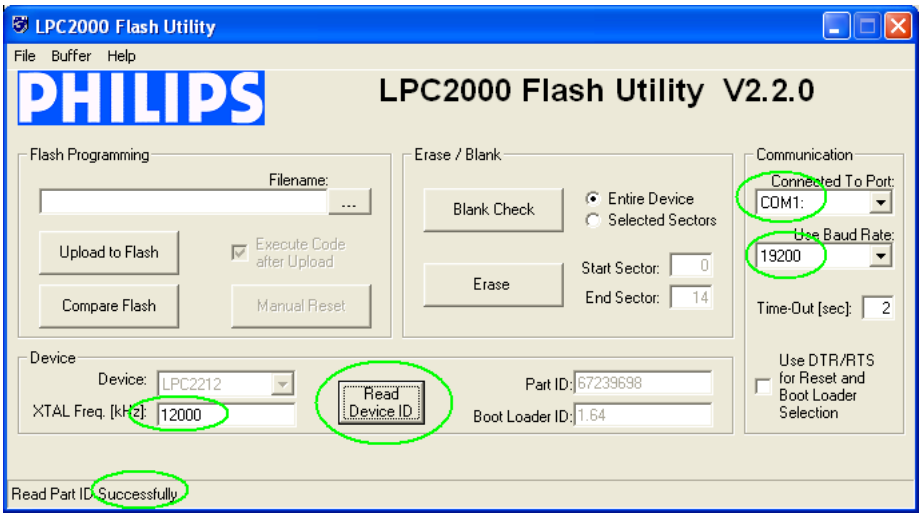

FIGURE 7.1 – Paramètres de *LPC2000 Flash utility*

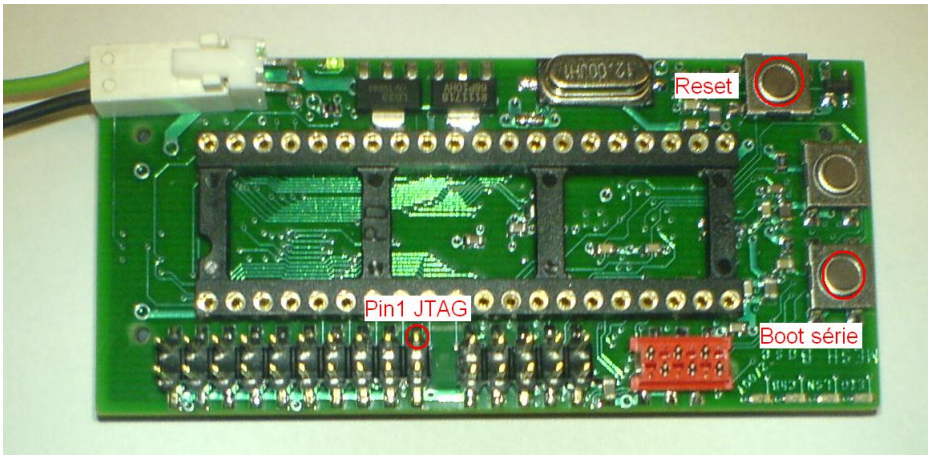

FIGURE 7.2 – Détail de la *carte MESH*

7.3 Communication JTAG

OpenOCD

La communication avec la *carte MESH* à travers le port JTAG permet de contrôler totalement le processeur. Il est ainsi possible de programmer la mémoire statique et la mémoire flash de la *carte MESH* ainsi que d'effectuer du déverminage. Commencer par connecter l'adaptateur *Amontec JTAGKey-tiny* en faisant attention au sens, il n'y a pas de détrompeur (pin1 du connecteur sur pin1 de la carte (fig. 7.3 et 7.2).

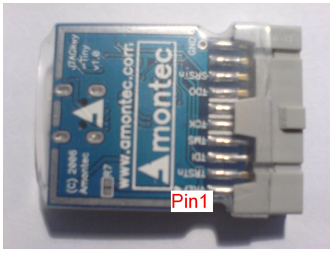

FIGURE 7.3 – Arrière de l'adaptateur JTAG

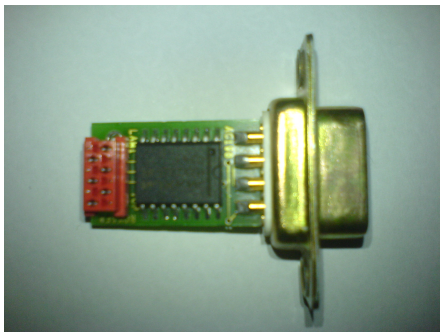

FIGURE 7.4 – Adaptateur port série

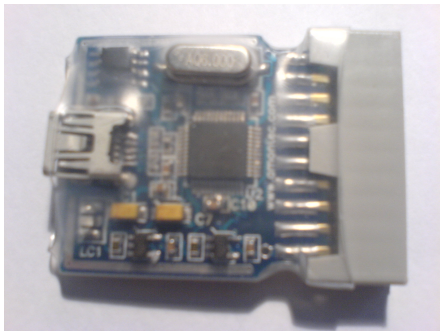

FIGURE 7.5 – Face avant de l'adaptateur JTAG

Une fois la connexion physique effectuée, exécuter *OpenOCD* en double cliquant sur l'icône du raccourci crée précédemment (fig. 3.8). Si la connexion est effective, la fenetre de *OpenOCD*doit ressembler à celle de la figure 7.6.

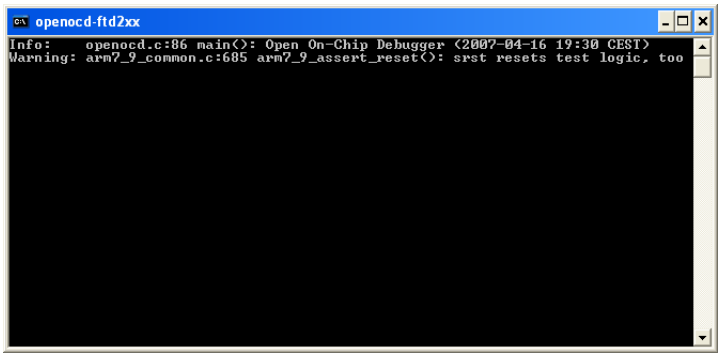

FIGURE 7.6 – Connexion avec *OpenOCD*

Le message de warning est normal et n'empêche pas le fonctionnement du système. Il signifie que les broches `trst` et `srst` sont activées en même temps, ce qui est normal pour une cible telle que la *carte MESH*. Cette limitation empêche d'arrêter la cible à l'adresse `0x0`. Le déverminage est possible à partir de l'adresse `main`.

Interaction

Afin d'intéragir avec *OpenOCD*, il faut ouvrir une connection telnet sur le port 4444. Aller dans le menu démarer de *Microsoft Windows* et cliquer sur *Exécuter*. Saisir `telnet` et cliquer sur *ok* puis procéder comme illustré sur la figure 7.7. Saisir *help* pour connaître les commandes disponnibles.

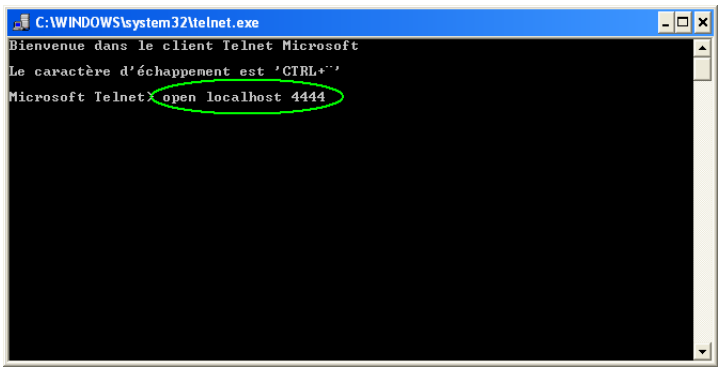

FIGURE 7.7 – Connexion telnet sur *OpenOCD*

Chapitre 8

Module radio

8.1 Préambule

Il est prévu d'utiliser le module radio *Y-Lynx TRM8053-025*[12] dans le cadre du projet *RASMAS/MESH*. La *carte MESH* prévoit à cet effet un connecteur 40 pin pour y loger le module radio.

8.2 Communication avec la *carte MESH*

Le module *Y-Lynx TRM8053-025* dispose de deux modes de communications avec l'environnement hôte (*carte MESH*). Il y a le mode SPI et le mode RS232. Dans le cadre du projet *RASMAS/MESH* le mode RS232 est sélectionné, car très simple de mise en oeuvre. Il est possible de transférer un maximum de 100 octets de payload par trame au module radio

8.3 Fonctionnement du module radio

Le module *Y-Lynx TRM8053-025* est un module radio fonctionnant sur la bande de fréquence comprise entre 868MHz et 870Mhz. Il dispose de 50 canaux de fréquences prédéfinis mais néanmoins paramétrables. Le canal zéro est un canal réservé à l'émission en broadcast d'un signal de synchronisation (*beacon*) entre les modules. La communication entre modules peut-être effectuée de trois manières différentes.

FHSS - Saut de fréquence (Frequency Hopping Spread Spectrum) Les transmissions dans ce mode sont réalisées en effectuant des sauts de fréquence afin d'obtenir une meilleure connectivité, dans le cas où une des fréquences de transmission serait parasitée. Dans ce mode, le module effectue par défaut 50 sauts de fréquences par seconde. Les fréquences utilisées sont programmables. Un des module radio doit être configuré en tant que serveur (émission de *beacon*) afin de les synchroniser entre eux et tout les modules doivent utiliser les mêmes paramètres.

TDMA - Multiplexage temporel (Time Division Multiple Access) Le mode opératoire est principalement le même que pour le mode de saut de fréquence, néanmoins, le *saut* de fréquence a lieu sur la même fréquence. Afin d'utiliser le mode TDMA, il faut configurer le module radio pour n'utiliser qu'une seule fréquence.

FDMA - Multiplexage fréquenciel (Frequency Division Multiple Access) Chaque client communique avec une fréquence définie et seul le module défini en tant que serveur (émission de *beacon*) effectue un saut fréquentiel. Ce mode est adapté à un réseau en étoile.

Le tableau 8.1 présente une comparaison des différents modes avec leurs paramètres.

Mode	Nb canaux	Durée canal	Nb fréquence
FHSS	50 (0x32)	2 (0x2)	50
TDMA	Nb clients	Dépends vitesse	1
FDMA	Nb clients	Dépends vitesse	Nb clients

TABLE 8.1 – Résumé des différents modes de fonctionnement du module radio

8.4 Paramètres importants

8.4.1 channel duration

Il faut s'assurer que le plus grand paquet que nous prévoyons d'envoyer puisse être transmis par le module radio dans un temps inférieur au paramètre channel duration. Ce dernier spécifie le temps passé par le module radio dans un canal. L'émission d'un beacon prend environ 20ms, il s'agit donc du temps minimal possible. Il est important de spécifier un temps les plus petit possible, car il n'est possible d'envoyer qu'un paquet par canal. Le chanel duration peut être estimé selon la formule 8.1

$$t_{init\_channel} + t_{listen\_time} + t_{ack} + t_{deasert} + overhead = t_{channel\_duration} \tag{8.1}$$

t_init_channel vaut environ 3.6ms t_listen_time peut être estimé pour un paquet de taille maximale (100 octets) transmis à un vitesse rf de 76.1 kbit/s selon la formule 8.2

$$t_{listen\_time} = \frac{100 * 8}{76.1 * 10^3} = 10.51ms \tag{8.2}$$

t_deasert correspond au temps nécessaire au module radio pour transmettre les données reçues à l'hôte. Dans notre cas, cette liaison est effectuée en RS232 à 115'200 kbit/s. Il faut compter 10bit par octet transmis (bit de stop et de start). Ce temps peut être estimé pour un paquet de taille maximale (100 octets) selon la formule 8.3.

$$t_deasert = \frac{100 * 10}{115.2 * 10^3} = 8.69ms \tag{8.3}$$

La valeur de t_channel_duration peut donc être estimée selon la formle 8.4

$$t_{channel\_duration} = 22.8 + t_{ack} + overheadms \tag{8.4}$$

Il est nécessaire de tenir compte d'un overhead du notament au fait que les données transmises par le module radio sont encapsulées. Les valeurs de channel duration doivent être choisies par multiple de 10. Une valeur acceptable déterminée empiriquement pour le channel duration est de 40 ms. Il est possible de constater que le channel_duration est trop petit si la pin "hop" du module radio reste à l'état haut pendant une durée plus importante que le channel duration sélectionné. La broche "hop" est monitoré par la carte mesh (timer 1 cap1.0) afin de déterminer si le temps de "hop" est trop important.

8.5 Configuration du module radio

Le module *Y-Lynx* est configurable de manière simple. Il suffit de mettre la broche *configuration* à l'état bas et d'envoyer la commande souhaitée (en l'occurrence par l'UART1 de la *carte MESH*). Cette dernière est composée de un octet qui spécifie la longueur de la commande ainsi que de un ou plusieurs octets représentant la commande et ses éventuels paramètres. Le module va répondre (acquittement) en renvoyant la taille de la réponse, le code de la commande ainsi qu'un éventuel paramètre. Si une erreur devait survenir, le module radio envoie la taille de la commande, le code `0x03` ainsi qu'un code numérique spécifiant l'erreur. Le tableau des erreures est disponible dans la documentation du module radio.

Exemple de dialogue entre la *carte MESH*et le module radio :

direction	1eroctet [a]	2^{e}octet [b]	3^{e}octet [c]	commentaire
Host->Radio	`0x01`	`0x55`	-	Interrogation du *mode beacon* du module radio
Radio->Host	`0x02`	`0x55`	`0x00`	Réponse du module radio, en l'occurrence le *mode beacon* est ici en mode client

TABLE 8.2 – Bancs de mémoires

a. Taille de la commande sans compter cet octet
b. Commande
c. Paramètre éventuel

Chapitre 9

YottaOS

9.1 Tâches

9.2 Piles de service et IRQ

YottaOS utilise deux piles : une pile de service et une pile destinée au traitement des interruptions. La taille de la pile de service est définie dans le fichier ld utilisé par l'assembleur (ld). Il est nécessaire de s'assurer d'avoir une pile de service d'une taille suffisante, faute de quoi un dépassement de pile se produira. La pile de service est initialisée avec un motif afin de détecter facilement sa taille minimale nécessaire. La pile réservée au traitement des interruptions est également initialisée avec un motif, mais sa taille ne doit pas être modifiée. En effet, le traitement d'une interruption utilise toujours la même taille de pile. L'examen du fichier exécutable au format .elf fournit l'adresse de la fin de la pile. Le code suivant montre la commande à exécuter ainsi que le résultat. Noter que la pile est descendante. L'examen de l'adresse `_stack_svc_start` après l'exécution d'un programme permet de savoir si un dépassement de pile à eu lieu. Si aucun dépassement n'a eu lieu, cette adresse doit contenir le motif `0xDEADBEEF`.

```
$ arm-elf-objdump.exe -x main.elf | grep _stack
82010f20 g       .bss   00000000 _stack_svc_end
8200ef20 g       .bss   00000000 _stack_svc_start
8200ef00 g       .bss   00000000 _stack_irq_start
8200ef20 g       .bss   00000000 _stack_irq_end
```

9.3 Prise en charge des interruptions

YottaOS permet la prise en charge des interruptions de type IRQ uniquement. Les interruptions de type FIRQ ne sont pour l'instant pas utilisées. Lors de l'occurrence d'une interruption, la procédure de traitement des interruptions, *IRQHandler* définie dans le fichier `yottaOS_a.S`. Lors de l'occurrence d'une interruption, son numéro unique est présent dans le registre *VICVectAddr*. *YottaOS* gère les interruptions à travers une table (emphTabDevice) dont chaque entrée est un pointeur sur une structure devant contenir au minimum l'adresse du handler associé associé à l'interruption (ISR). Cette structure peut contenir optionellement des paramètres qui seront passés en paramètre au handler de l'interruption. Le listing suivant montre un exemple d'une telle structure. Cette dernière dispose d'un paramètre afin de définir pour quel interruption (EINT0 à EINT3) elle est utilisée.

```
typedef struct EINT_DESCRIPTOR {
2       void (*InterruptHandler)(struct EINT_DESCRIPTOR *);
        UINT8 eintnumber;
} EINT_DESCRIPTOR;
```

Interruption	Numéro
OS_IO_TIMER0	0
OS_IO_TIMER1	1
OS_IO_UART0	2
OS_IO_UART1	3
OS_IO_PWM0	4
OS_IO_I2C	5
OS_IO_SPI0	6
OS_IO_SPI1	7
OS_IO_PLL	8
OS_IO_RTC	9
OS_IO_EINT0	10
OS_IO_EINT1	11
OS_IO_EINT2	12
OS_IO_EINT3	13
OS_IO_ADC	14
OS_IO_CAN1Tx	15
OS_IO_CAN2Tx	16
OS_IO_CAN1Rx	17
OS_IO_CAN2Rx	18

TABLE 9.1 – Mappage des interruptions

Listing 9.1 – Initialisation d'une interruption externe

```
void InitializeEINT(UINT8 port){
   EINT_DESCRIPTOR *ioDescriptor;
   ioDescriptor = malloc(sizeof(EINT_DESCRIPTOR)); // Allocate memory for the
       struct
   ioDescriptor->InterruptHandler = EintInterruptHandler; // Put the ISR addr. in
       the struct
   switch (port){
      case OS_IO_EINT0:
         ioDescriptor->eintnumber = 0;
         OSSetIODescriptor(OS_IO_EINT0, ioDescriptor); // Assign the struct to the
             table of interrupt
                                                       // vectors This is a yottaos
                                                           function.
         PINSEL1 |= 0x1 << 0;                    // Activate the pin P0.16 as EINT0
         VICVectAddr = 0x0;              // Clear interrupts
         VICVectCntl9 = 0x20 | 14;       // Enable interrupt with EINT0 source
         VICVectAddr9 = OS_IO_EINT0;     // Set return address to the EINT0
             Tabdevice index
         setEINTMode(port, LEVEL, HIGH);
         EXTINT = 0x1 << 0;
         VICIntEnable |= 0x1 << 14;      // Enable EINT0 interrupts
         break;
      ...
```

Le tableau 9.1 montre le mapping des interruptions entre une source d'interruption et son numéro tel qu'il est prévu par *YottaOS*. L'assignation est effectuée grâce à la procédure `OSSetIODescriptor` (list. 9.1).

9.4 Temps

YottaOS étant un OS temps réel, la notion de temps est importante.

Vitesse [bps]	Temps pour 1 octet [ms]	Temps pour Payload max [a][ms]
9′600	1.04	104
19′200	0.52	52
38′400	0.26	26
57′600	0.17	17
115′200	0.086	8.6

TABLE 9.2 – Temps de transmission RS232

a. 100 octets

9.4.1 Transfert série

L'envoie et la réception de caractères par le port série est très lente (en comparaison avec la fréquence de fonctionnement du processeur ARM). Chaque transmission d'un caractère nécessite 10 bits. En effet, une trame RS232 est composée dans notre cas de 1 bit de démarrage, 8 bits de données et 1 bit de d'arrêt. En sachant que le codage NRZ employé est bivalent, nous pouvons considérer que un baud équivaut à un bit. Le tableau 9.2 présente les temps physiques (électriques) nécessaires à la transmission de caractères sur le port RS232.

9.5 Modifications apportées à YottaOS

De nombreuses modifications ont été apportées en cours de route à YottaOS, soit en collaboration avec Bertrand Hurst, soit de notre propre initiative. Exemple : Ajout de fifos pour recevoir un paquet entier du module radio (premier caractère indique la taille) Modification des fifos afin de pouvoir transférer des paquets SLIP Prise en compte et configuration du memory accelerator (MAM) Prise en charge et configuration de la mémoire supplémentaire

Chapitre 10

MESH serveur

10.1 Préambule

Le serveur MESH est l'implémentation logicielle de L'élément serveur 14.2.3. Le but du serveur est :

- gestion des noeuds du réseau mesh ;
- récolte et stockage de données ;
- être un point d'accès unique au réseau mesh pour les utilisateurs.

10.2 Interfaces principales

On peut diviser les interfaces du serveur en deux groupes principaux :

- côté paserelle :
  - un socket udp en écoute qui permet de recevoir des messages asynchrones de type alarme (sec. 14.3.1) et également permettre la découverte des nouveaux noeuds.
  - un socket tcp client pour interagir avec les noeuds du réseau mesh. Ce canal est utilisé pour effectuer des acquisitions de données et des changements de paramètres suite à la requête faite par des clients ou le serveur lui même.
- côté client :
  - un socket tcp en écoute en attente de connexion de client. Ce socket représente le seul point d'accès possible sur le réseau mesh. Un socket tcp dédié est attribué lors de la connexion de chaque client.
- stockage de données : Le serveur dispose d'un espace de stockage permettant de mémoriser des données en provenance de noeuds mesh. Ses données sont par la suite accessibles aux utilisateurs.

10.3 Structures de données

Le serveur emploie différentes structures de données dans le but de gérer les interfaces, les communications et les données. La structure `serverItem` contient un pointeur vers un processus qui va consommer les commandes déposées dans la `dataBufferList` par des processus tiers.

```
/* file : meshsvc2.0_server.h */

struct serverItem{
        struct process*          process;
        struct dataBufferList*   dataBufferList;
        char                     name[MAX_LEN_ENTITY_NAME+1];
}serverItem;
```

La structure `udpComm` contient les descripteurs des sockets UDP et les tampons utilisés par le thread qui gère la communication udp.

```
/* meshsvc2.0_udp.h */

struct udpComm{
   struct process*              process;
   uint8_t                      recvBuffer[MAX_LEN_BUFFER_UDP];
   uint16_t                     dumpBufferLen;
   uint8_t                      accuBuffer[MAX_LEN_INTERNAL_BUFFER_TCP];
   int16_t                      accuBufferLen;
   struct sockaddr_in           socketFD;
   int                          socketID;
   size_t                       sizeSocketFD;
   struct sockaddr              senderSocketFD;
   int                          senderSocketID;
   size_t                       sizeSenderSocketFD;
}udpComm;
```

La structure `alarmSvc` contient la référence aux communications UDP ainsi que l'emplacement destiné à la réception des informations.

```
/* file : meshsvc2.0_alarmSvc.h */

struct alarmSvc{
   struct process*          process;
   struct dataBufferList*   outDataBufferList;
   struct udpComm*          udp;
}alarmSvc;
```

La structure `tcpComm` contient les descripteurs des sockets nécessaires à la communication TCP ainsi que les tampons utilisés pour la réception et l'envoie de données. Cette structure est utilisée aussi bien du côté client que du côté passerelle.

```
/* file : meshsvc2.0_tcp.h */

struct tcpComm{
   struct process*              process;
   uint8_t                      sendBuffer[MAX_LEN_INTERNAL_BUFFER_TCP];
   uint16_t                     sendBufferLen;
   uint8_t                      recvBuffer[MAX_LEN_INTERNAL_BUFFER_TCP];
   uint8_t                      accuBuffer[2*MAX_LEN_INTERNAL_BUFFER_TCP];
   uint16_t                     accuBufferLen;
   uint16_t                     dumpBufferLen;
   struct dataBufferList*       dataBufferListToSend;
   uint8_t*                     dataBufferToSend;
   uint32_t                     dataBufferLenToSend;
   struct dataBufferList*       dataBufferListToRecv;
   struct sockaddr_in           socketFD;
   int                          socketID;
   size_t                       sizeSocketFD;
   struct sockaddr_in*          socketAcceptedFD;
   int*                         socketAcceptedID;
   size_t*                      sizeSocketAcceptedFD;
}tcpComm;
```

La structure `serviceItem` est utilisée pour la gestion d'un processus qui a la charge d'accepter les nouvelles connections en provenance des clients.

```
/* file : meshsvc2.0_service.h */

struct serviceItem{
   struct process*          process;
   struct tcpComm*          tcp;
}serviceItem;
```

La structure `cardItem` contient la description d'un noeud du réseau mesh. Soit le nom et la liste des applications disponibles.

```
/* file : meshsvc2.0_card.h */

struct cardItem{
    char                    name[MAX_LEN_ENTITY_NAME+1];
    struct applicationItem** appList;
}cardItem;
```

La structure `applicationItem` contient la description d'une application présente sur un noeud, ainsi qu'un pointeur sur la structure `tcpComm` qui permet de communiquer entre le serveur et le noeud lui-même. L'accès exclusif à l'application est garanti par un mutex contenu dans la structure.

```
/* file : meshsvc2.0_application.h */

struct applicationItem{
    struct tcpComm*         tcp;
    pthread_mutex_t         mtx;
    char                    name[MAX_LEN_ENTITY_NAME+1];
}applicationItem;
```

La structure `userInterfaceItem` contient les informations nécessaires à la réception et à l'envoie des données suite à la requête envoyées par un client.

```
/* meshsvc2.0_userInterface.h */

struct userInterfaceItem{
    struct process*         process;
    struct dataBufferList*  inDataBufferList;
    struct dataBufferList*  outDataBufferList;
    struct tcpComm*         tcp;
    pthread_mutex_t         mtx;
    struct process*         processForwardData;
}userInterfaceItem;
```

La structure `dataBufferList` contient la liste des éléments des tampons utilisés dans le serveur, ainsi que le mutex permettant un accès exclusif aux données.

```
/* file : meshsvc2.0_dataBuffer.h */

struct dataBufferListItem{
    uint8_t                     data[MAX_LEN_DATA_BUFFER];
    uint32_t                    len;
} dataBufferListItem;

struct dataBufferList{
    struct dataBufferListItem*  dataBufferListItem[MAX_LEN_DATA_BUFFER_LIST];
    uint32_t                    dataBufferListIndexInsert;
    uint32_t                    dataBufferListIndexRead;
    uint8_t                     flagListEmpty;
    uint8_t                     flagListFull;
    pthread_mutex_t             mtx;
}dataBufferList;
```

La structure `process` est utilisée pour la gestion des threads qui ĺ composent le serveur.

```
/* file : meshsvc2.0_process.h */

struct process{
     pthread_t                  pthreadId;
     uint32_t                   nextState;
     pthread_mutex_t            mtxNS;
     uint32_t                   currentState;
     pthread_mutex_t            mtxCS;
     uint32_t                   flags;
     pthread_mutex_t            mtxProc;
```

```
}process;
```

La structure `timerScheduler` met à disposition un timer contenant un date prochaine échéance, une valeur de recharge ainsi que son étât.

```
/* file meshsvc2.0_timer.h */

struct timerScheduler{
   uint16_t                    state;
   time_t                      nextShootTimerValue; //absolute 2008-11-05 12-30-30
   time_t                      rechargeTimerValue;  //relative 0-0-0 0-30-0 :
       repeat in 30 minutes
   pthread_mutex_t             mtx;
}timerScheduler;
```

La structure `scheduledOperationItem` contient les descripteurs des tampons et des fichiers qui seront contiennent les données à traiter ainsi que les données reçues lors d'une échance du timer.

```
/* file : meshsvc2.0_scheduledOperation.h */

struct scheduledOperationItem{
   struct process*          process;
   struct meshFile*         inFile;
   struct meshFile*         outFile;
   struct dataBufferList*   inDataBufferList;
   struct dataBufferList*   outDataBufferList;
   struct timerScheduler*   timer;
   struct applicationItem*  ap;
   struct process*          processStore;
   char                     name[MAX_LEN_ENTITY_NAME+1];
}scheduledOperationItem;
```

La structure `meshFile` permet l'accès mutuel à un fichier par l'utilisation de mutex.

```
/* meshsvc2.0_file.h */

struct meshFile{
   char                filename[MAX_LEN_FILENAME+1];
   FILE*               fp;
   pthread_mutex_t     mtx;
}meshFile;
```

10.4 Threads

La figure 10.1 représente la division en thread du serveur mesh. chaque bulle représente un thread. L'existence des threads est dynamique, ceci suivant les événements et la nécessité des communications. Les processus crées statiquement sont les threads `server`, `alarm/udp` et `service/tcp`.

10.4.1 Démarrage du serveur

Le processus principal créé les structures de données et réserve l'espace mémoire nécessaire à son fonctionnement. Lors de l'initialisation des structures de données, les threads statiques (`server`, `alarm/udp` et `service/tcp`) sont crées. Les processus udp et alarm gère la réception des messages d'alerte. Ces derniers sont mémorisés par le processus `alarm`. Le processus `server` lit les données reçues par le processus `alarm` et décode les instructions.

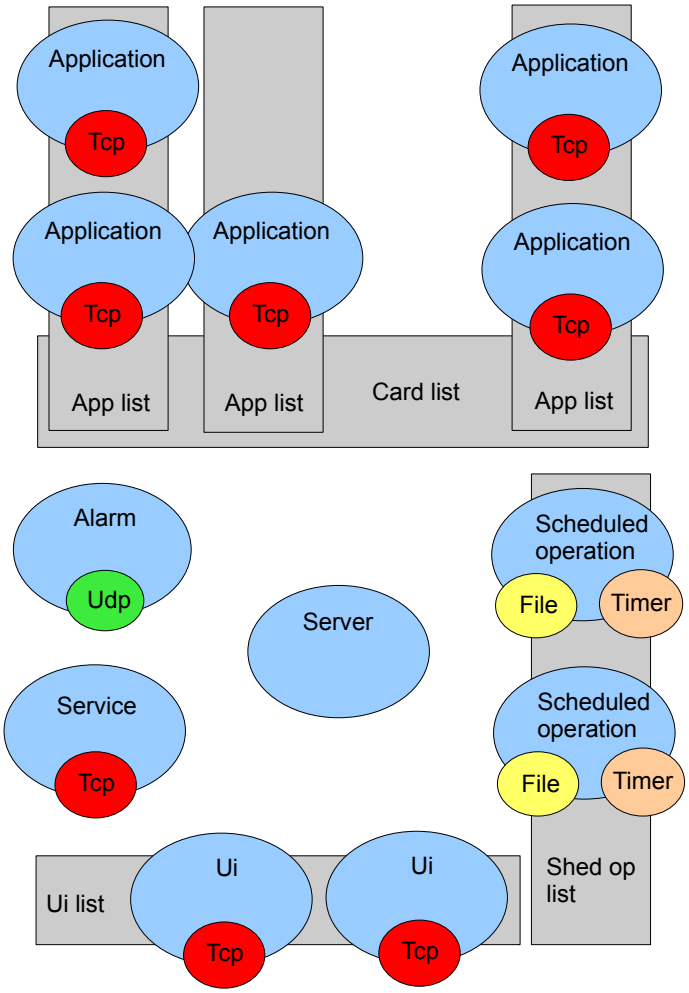

FIGURE 10.1 – Schema bloc des threads

10.4.2 Nouveau noeud sur le réseau mesh

Lors de la mise sous tension d'une carte mesh, cette dernière doit s'annoncer auprès du serveur. Ce dernier indique à la carte qu'il l'a prise en compte en lui envoyant un message de keep-alive de manière périodique. La figure 10.2 détail les échanges de messages qui ont lieu dans ce cas. Se référer à la section 10.6 pour plus de détail sur le contenu de chaque message.

Algorithm 1 Annonce d'un nouveau noeud

```
loop
  if nouveau_paquet_udp!=NULL then
    lecture par le processus alarm
    enregistrement dans buffer
    processus server décode donnéé présente dans buffer
    initialise un élément de card list
    initialise la première application (contrôle) de app list relative à ce noeud
    exécute le processus de cette application
  end if
end loop
loop
  application (contrôle) demande la liste des applications presentes sur la cart mesh
  application (contrôle) initialise les donnees relative à chaque nouvelle appplication
  application (contrôle) envoie set timeout au noeud
  loop
    attend PING du noeud
    application (contrôle) set timeout to card
  end loop
end loop
```

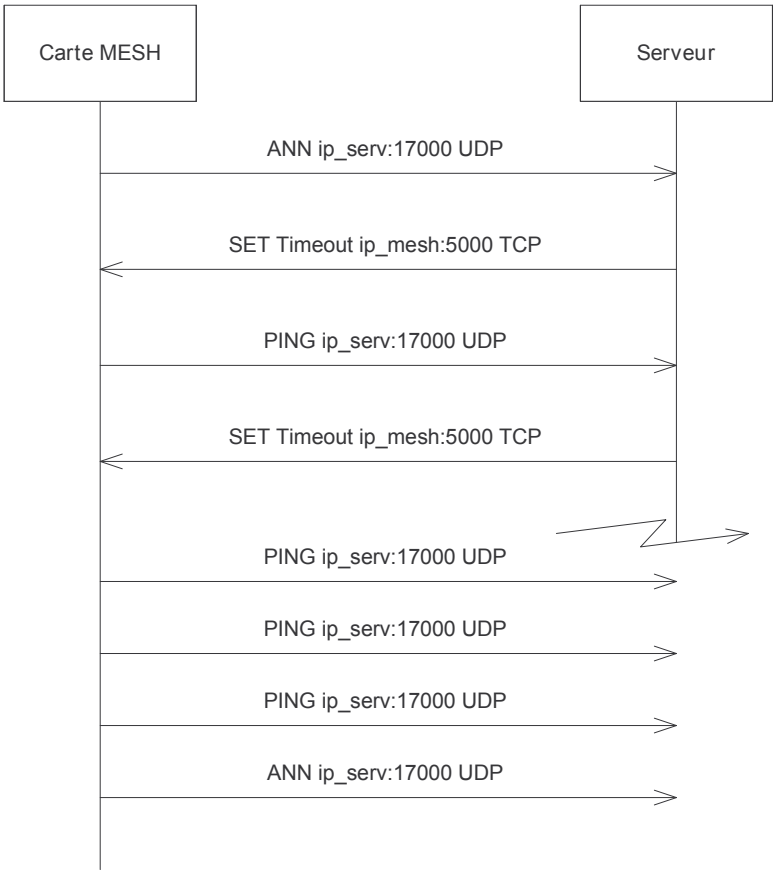

FIGURE 10.2 – Echange des messages initiaux lors de l'ajout d'une carte mesh

10.4.3 Connexion d'un utilisateur

Le processus `service` gère le processus tcp qui est en attente d'une nouvelle connection. Lors d'une nouvelle connexion, un nouveau processus `ui` ainsi qu'un socket dédié à la connexion sont crées. L'utilisateur envoie une requête à destination d'une processus application. Ce dernier se charge de contacter la carte afin de lui transmettre la requête et se met en attente de la réponse. Les données reçues sont lues par le processus `ui` et transmises au client demandeur. Le processus `ui` et le processus `application` concernés sont terminés.

Algorithm 2 Connexion d'un utilisateur

```
loop
  if nouveau_client_tcp !=NULL then
    creation d'un processus ui
    attente de la requete du client
    creation du processus application
    requete lue par le processus application
    envoie de la requête au noeud sélectionné
    reception de la reponse par le processus application
    transmission de la reponse par le processus ui au client
    fin du processus ui et application crées précédemment
  end if
end loop
```

10.4.4 Operation planifiée

Le processus `scheduled operation` vérifie l'horloge et à l'échéance spécifié par le timer, crée un processus application. Le processus `application` envoie une requête à destination d'un noeud du réseau mesh. Le processus `application` attend la réponse et la transmet au processus `scheduled operation` lors de son arrivée. Ce dernier écrit les données reçues dans un espace de stockage. Le processus `application` concerné est terminé. La prochaine échéance est spécifiée dans le timer.

Algorithm 3 Opération planifiée

```
loop
  if timer echeance <= horloge then
    creation du processus application
    requete transmise au processus application par le processus scheduled_operation
    envoie de la requête au noeud sélectionné
    reception de la reponse par le processus application
    transmission de la reponse au processus scheduled_operation
    ecriture de la reponse dans un espace de stockage
    fin du processus application crée précédemment
    mise a jour du timer
  end if
end loop
```

10.5 Application de test

Le serveur est livré avec des applications de test, qui permettent de simuler les différents protocols de communications. Ces applications servent à valider le protocole de communication soit entre le serveur et la carte mesh, soit entre le serveur et une application utilisateur.

10.6 Protocole de transmission

10.6.1 Préambule

Les définitions des noms symbolique des bits utilisés se trouvent dans le fichier d'entete meshsvc2.0.h et sont reproduits ci-dessous pour information.

```
#define VALUE_TIMEOUT  (1 << 0)
#define VALUE_APPLICATION_LIST  (1 << 1)
#define VALUE_LAST   (1 << 0)
```

FIGURE 10.3 – Definition des valeurs pour l'application CTRL

```
#define BEG_SYMBOL   0x2A  //*
#define END_SYMBOL   0x23  //#
```

FIGURE 10.4 – Definition des balises de début et fin

```
#define VALUE_SRV  0x01 // message a destination du processus serveur
srvAddress // identifiant du serveur
```

FIGURE 10.5 – Definition de diverses valeurs du protocole MESH

```
#define VALUE_ANN  (1 << 0)
#define VALUE_PNG  (1 << 1)
#define VALUE_SET  (1 << 2)
#define VALUE_GET  (1 << 3)
#define VALUE_AAP  (1 << 4)
```

FIGURE 10.6 – Defintion des valeurs de l'application ALARM

```
#define VALUE_CMD  (1 << 0)
#define VALUE_RPL  (1 << 1)
#define VALUE_DAT  (1 << 2)
#define VALUE_END  (1 << 3)
#define VALUE_ACK  (1 << 4)
#define VALUE_NACK (1 << 5)
#define VALUE_ERR  (1 << 6)
#define VALUE_ALM  (1 << 7)
```

FIGURE 10.7 – Definition des valeurs de l'application CMD

```
#define VALUE_LAST   (1 << 0)
```

FIGURE 10.8 – Definition des valeurs de l'application ACC

Les paquets transmis sur le réseau sont composés selon la figure 10.9

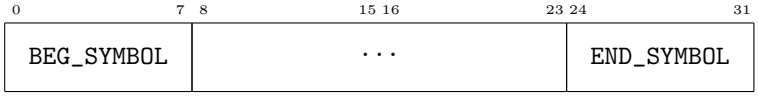

FIGURE 10.9 – format de base d'un paquet sur le réseau mesh

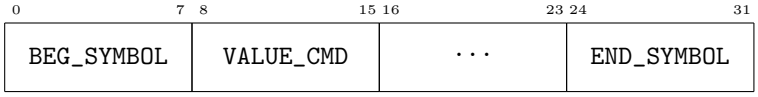

FIGURE 10.10 – format de base d'une commande sur le réseau mesh

La figure 10.11 représente une réponse type qui est composée de trois éléments :
– un élément de début de réponse
– un ou plusieurs éléments contenant des données
– un élément de fin de réponse

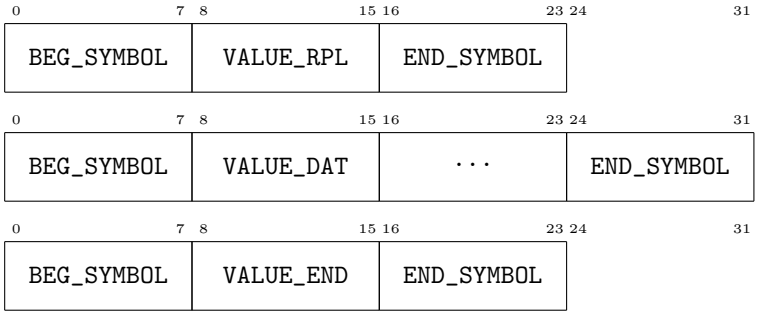

FIGURE 10.11 – format de base d'une réponse en provenance d'une carte mesh

Lors d'une commande envoyée par le serveur à destination de la carte et ne nécessitant pas de données en retour, cette dernière acquitte simplement la commande par un paquet décrit par la figure 10.12 ou par la figure 10.13 dans le cas ou la commande ne peut pas être honorée.

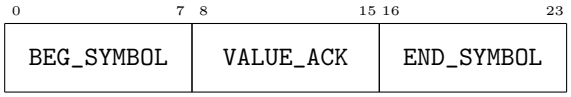

FIGURE 10.12 – Acquitement d'une commande par une carte mesh

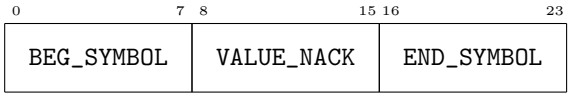

FIGURE 10.13 – Non acquitement d'une commande par une carte mesh

10.6.2 Format des paquets détaillé pour les applications mesh

Mesh Alarm et Control

Le serveur attend un paquet d'annonce envoyé par une carte mesh nouvellement disponnible sur le réseau. Le paquet est composé selon la figure 10.14. L'annonce contient une adresse ip et un port tcp qui sont en fait le moyen de contacter l'application contrôle présents sur la carte.

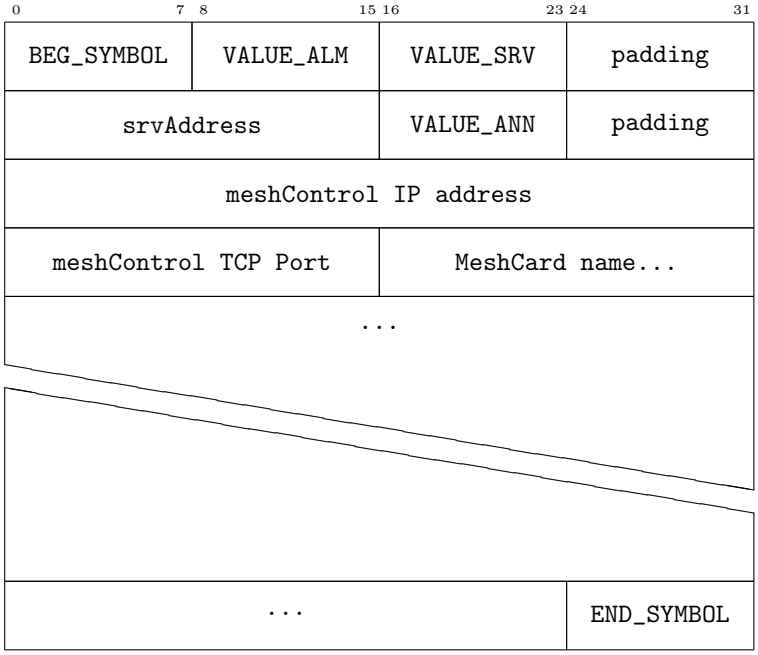

FIGURE 10.14 – Annonce d'une carte mesh non prise en charge par le serveur

La figure 10.15 représente le paquet que le serveur reçoit périodiquement de la carte mesh, une fois que cette dernière est connue du serveur. Le but de ce message est de notifier le serveur que la carte est fonctionnelle.

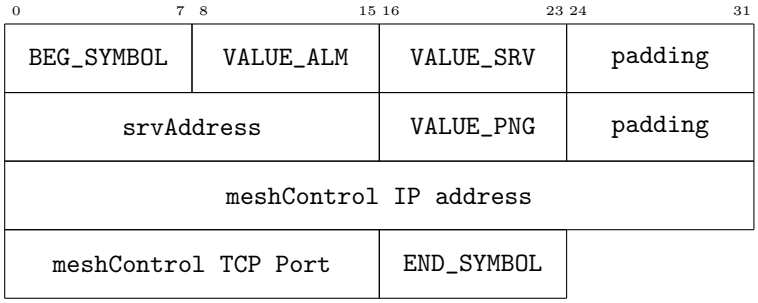

FIGURE 10.15 – Message ping d'une carte mesh à destination du serveur

Le paquet décrit par la figure 10.16 est envoyé par le serveur à chaque réception d'un message d'annonce ou de ping afin de quittancer la carte.

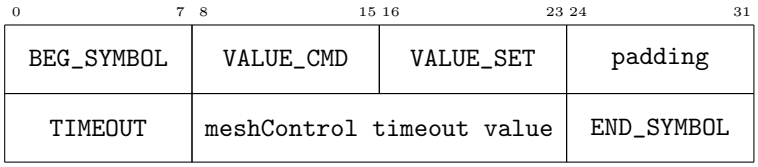

FIGURE 10.16 – Envoie d'un message de timeout du serveur à une carte mesh

Une fois un nouveau noeud accepté par le serveur, ce dernier demande au noeud la liste des applications à dispostion. Le paquet 10.17 décrit cette requête.

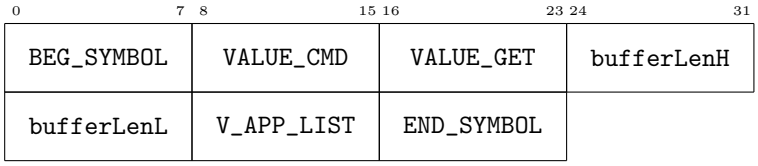

FIGURE 10.17 – meshServer set timeout packet format

La réponse à la demande des applications 10.17 est effectuée par la carte d'après le format spécifié par la figure 10.18. Chaque paquet contient une application et ses paramètres de connexion.

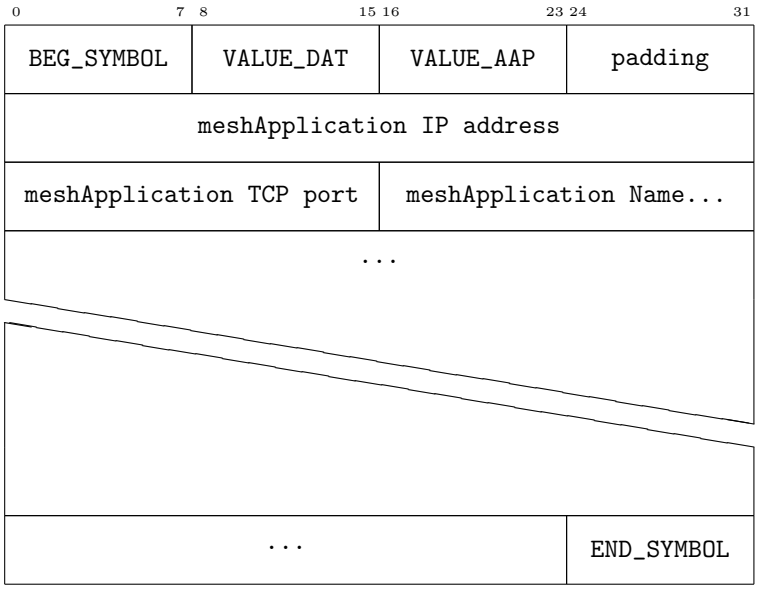

FIGURE 10.18 – meshControl-aap-packet-format

Exemple d'application (mesh accéléromètre)

Les figures 10.19 et 10.20 représentent le format utilisé par le serveur pour intérroger une application présente sur une carte mesh ainsi que les données renvoyées par cette dernière.

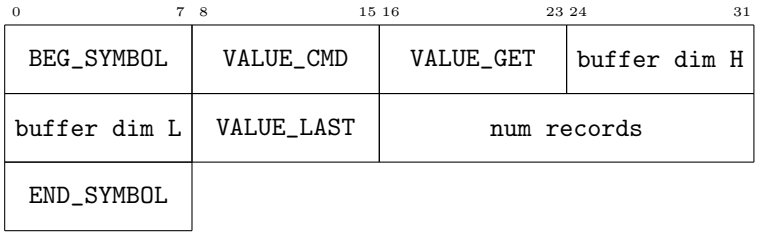

FIGURE 10.19 – meshServer set timeout packet format

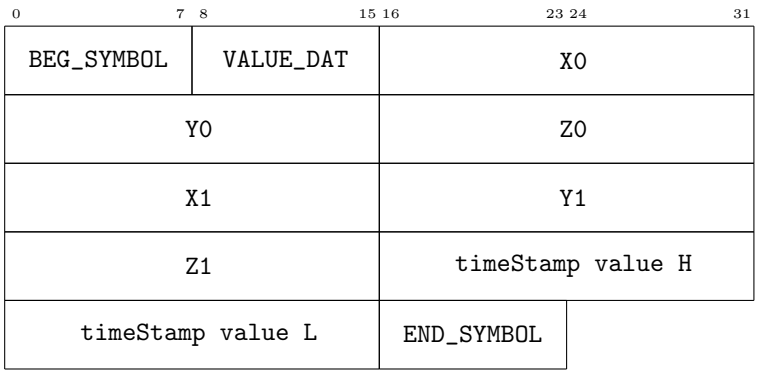

FIGURE 10.20 – meshServer set timeout packet format

10.7 Structure des fichiers

L'application serveur est structurée de la manière suivante :

```
meshServer2.0
|-- Makefile                                règles de compilation
|-- build                                   repertoire temporaire utilisé lors de
|                                           la compilation
|-- data                                    espace de stockage des données reçues
|-- etc                                     configuration
|-- include
|   |-- meshsvc2.0.h                        regroupement des fichiers d'include
|   |-- meshsvc2.0_alarmSvc.h               struct alarm
|   |-- meshsvc2.0_application.h            struct application
|   |-- meshsvc2.0_card.h                   struct carte
|   |-- meshsvc2.0_dataBuffer.h             struct tampons
|   |-- meshsvc2.0_file.h                   struct descripteurs de fichiers
|   |-- meshsvc2.0_process.h                struct gestion des processus
|   |-- meshsvc2.0_scheduledOperation.h     struct operations planifiée
|   |-- meshsvc2.0_server.h                 struct server
|   |-- meshsvc2.0_service.h                struct serveur tcp pour ui
|   |-- meshsvc2.0_tcp.h                    struct tx rx pour tcp
|   |-- meshsvc2.0_timer.h                  struct timer pour operation planifiée
|   |-- meshsvc2.0_udp.h                    struct tx rx pour udp
|   `-- meshsvc2.0_userInterface.h          struct interface utilisateur
|-- log
|-- src
|   |-- meshsvc2.0.c                        initialisation
|   |-- meshsvc2.0_alarmSvc.c               function et processus alarm
|   |-- meshsvc2.0_application.c            function et processus application
|   |-- meshsvc2.0_card.c
|   |-- meshsvc2.0_dataBuffer.c
|   |-- meshsvc2.0_file.c                   function et processus file
|   |-- meshsvc2.0_process.c
|   |-- meshsvc2.0_scheduledOperation.c     function et processus operations
|   |                                       planifiée
|   |-- meshsvc2.0_server.c                 function et processus server
|   |-- meshsvc2.0_service.c                function et processus serv tcp pour ui
|   |-- meshsvc2.0_tcp.c                    function et processus tx rx pour tcp
|   |-- meshsvc2.0_timer.c
|   |-- meshsvc2.0_udp.c                    function et processus tx rx pour udp
|   `-- meshsvc2.0_userInterface.c          function et processus interface client
`-- target
    `-- meshServer
```

FIGURE 10.21 – Structure de fichier du serveur

Les applications de test sont structurés de la manière suivante :

```
testMeshServer2.0
|-- compile_meshAccelerometer.sh          aide a la compilation
|-- compile_meshAlarm.sh                  aide a la compilation
|-- compile_meshControl.sh                aide a la compilation
|-- compile_meshLievre.sh                 aide a la compilation
|-- compile_meshUi.sh                     aide a la compilation
|-- compile_meshUiRealACCED.sh            aide a la compilation
|-- meshAccelerometer.c                   simule l'application ACCED sur mesh
|-- meshAlarm.c                           simule l'application ALARMD sur mesh
|-- meshControl.c                         simule l'application CONTROLD (1) sur
|                                         carte mesh
|-- meshLievre.c                          simule l'application LIEVRED sur mesh
|-- meshPing.c                            simule l'application CONTROLD (2) sur
|                                         la carte mesh
|-- meshUi.c                              simule une application utilisateur Ui
|                                         sur un ordinateur distant
`-- meshUiRealACCED.c                     simule une application utilisateur
                                          pour interagir avec l'application ACCED
                                          sur la carte mesh
```

FIGURE 10.22 – Structure de fichiers de l'application de test du serveur

Chapitre 11

Passerelle mesh <=> Ethernet

11.1 Préambule

Le réseau mesh doit pouvoir être capable de communiquer avec un réseau de type ethernet, ceci afin qu'il soit accessible depuis un poste de contrôle relié à internet par exemple. Les cartes mesh communiquent entre elles par ondes radio et si elles ont besoin de communiquer avec l'extérieur, doivent passer par un noeud du réseau mesh disposant d'une double interface : radio et sériel. En effet, les cartes mesh ne disposent pas d'interface réseau, il est donc nécessaire d'encapsuler le traffic TCP/IP en provenance des différents noeuds mesh à l'aide du protocole slip.

11.2 Préparation d'une passerelle

11.2.1 NSLU2

Afin de disposer d'une passerelle mesh <=> Ethernet facile à configurer, il est décidé d'utiliser un boitier NSLU2 de Linksys. Ce boitier n'est initialement pas destiné a cet usage, mais a plusieurs avantages : peu onéreux (env. 100 CHF) très petit dispose d'une interface ethernet et de deux ports usb Une distribution linux est instalable dessus, mettant à disposition tout les outils nécessaires pour effectuer le travail de passerelle.

11.2.2 Installation de OpenWRT

Présentation

La distribution linux la plus complète disponible pour le boitier NSLU2 est OpenWRT, en version 8.09. Il s'agit d'une distribution linux basée sur un kernel 2.6.x et spécialement conçue pour du matériel embarqué. La distribution peut être adaptée aux besoins.

Téléchargement

Openwrt est fourni sous forme binaire ou sources. La version binaire ne dispose pas du support du protocole slip, il est donc nécessaire de télécharger les sources de la distribution. Le plus pratique est de les télécharger sur le serveur svn de openwrt.

```
svn co https://svn.openwrt.org/openwrt/branches/8.09
```

Prérequis

Le plus simple pour pouvoir compiler les sources est de disposer d'un pc utilisant linux. quelques outils sont nécessaires : bison, gawk, flex, build-essential, zlib1g-dev Une partie des fichiers sources nécessaires sera téléchargé durant la compilation par ftp, s'assurer d'avoir accès au protocol ftp

depuis le pc (pc dans la classe 10.194.18x.xxx de l'eig entre autre). La chaine de compilation croisée sera crée lors de la compilation de openwrt, ce qui veut dire que la première compilation prend beaucoup de temps.

Personnalisation

Une fois le téléchargement et les prérequis satisfaits, exécuter l'interface de personnalisation de la configuration globale et ou du noyau. Taper sur la barre d'espace pour activer ou un un service. Si une étoile est présente à côté du nom, ce service sera inclus dans l'image binaire qui est crée lors de la compilation tandis que si un m est présent, ce service sera compilé en tant que package qu'il est possible d'installer par la suite.

Image et outils

L'interface de personnalisation se lance ainsi : `make menuconfig`

Noyau

Il peut être nécessaire de personnaliser le noyau pour ajouter divers éléments. L'interface de personnalisation s'exécute grâce à la commande suivante : `make kernel_menuconfig`

Compilation

Une fois les étapes préalables effectuées, la compilation à proprement parler est effectuée grâce à la commande `make`. Si aucune erreur n'a lieu, l'image binaire résultant de la compilation est placée dans le répertoire `bin`, tandis que les différents packages pouvant être installés par la suite sont présents dans le répertoire `bin/packages`

Flash

Un utilitaire, `upslug2`, est prévue pour flasher dans le nslu2 l'image binaire précédemment génerée. Son installation est effectuée ainsi sur ubuntu : `sudo apt-get install upslug2` Il faut préférablement connecter le nslu2 directement au pc qui va flasher l'image, l'utilitaire utilisant la couche 2 osi pour effectuer le transfert. Ensuite, il est nécessaire de démarrer le nslu2 en mode upgrade `http://www.nslu2-linux.org/wiki/HowTo/UseTheResetButtonToEnterUpgradeMode` Le flash de l'image s'effectue ainsi : `sudo upslug2 -d eth0 -i bin/openwrt-nslu2-squashfs.bin` Si aucune erreur ne s'est produite, le nslu2 est redémarré en quelques minutes. Prendre garde au fait que la led status/ready ne s'allume pas, ce qui peut donner l'impression que le boîtier ne fonctionne pas.

11.2.3 Utilisation de OpenWRT

Configuration du réseau

La configuration du réseau est définie dans le fichier `/etc/config/network` Configuration pour le réseau de l'école :

```
config interface lan
       option ifname   eth0
       option proto    static
       option ipaddr   '10.194.185.147'
       option netmask  '255.255.252.0'
       option gateway  '10.194.184.1'
       option dns      '129.194.4.6'
       option hostname nslu2
```

Ajout d'une route statique

```
config route mesh
   option interface sl0
   option target 192.168.3.0
   option netmask 255.255.255.0
   option gateway 192.168.2.1
```

Création de l'interface sl0

```
slattach -p slip -s 115200 ttyUSB0
```

script : création d'un script de démarrage de l'interface sl0

```
#!/bin/sh /etc/rc.common
# Copyright (C) 2006 OpenWrt.org

START=50
start() {
        [ -d /dev/ttyUSB0 ] && slattach -p slip -s 115200 ttyUSB0
}

stop() {
killall slattach
}
```

Persmission

```
chmod a+x /etc/init.d/slip
```

Activation

```
/etc/init.d/slip enable
```

Activation de ip forwarding Afin que le boitier nslu2 serve de relais, il est nécessaire d'activer la fonctionnalité de ip forwarding.

```
echo "1" > /proc/sys/net/ipv4/ip_forward
```

Il faut également soit configurer soit désactiver le firewall. Par manque de temps, le firewall est totalement désactivé :

```
/etc/init.d/firewall disable
```

Chapitre 12

transmission de données

12.1 Pile IP

Afin de standardiser les transmissions de données et de faciliter l'intercommunication entre différentes plateformes (PC et carte MESH par exemple), les transmissions se font selon le protocole IP. Une pile ip a donc été choisie et adaptée afin de pouvoir fonctionner avec les autres couches de la carte mesh. La pile ip micro ip est utilisée. Cette dernière est disponible librement sous forme de code source sous réserve d'acceptation de la licence de Adam Dunkels, ce qui ne pose aucun problème pour notre usage. La pile ip supporte les couches de transport TCP et UDP.

12.1.1 Serial Line IP

Afin de permettre la communication entre le réseau "internet" et le réseau mesh, une des cartes mesh sert de passerelle et dispose de deux moyens de communications. Le premier étant le module radio afin de communiquer avec le réseau mesh et le second étant un port série utilisant le protocole SLIP, qui permet l'encapsulation d'une trame IP sur un port série. Le protocole SLIP est un protocole très simple défini par une RFC. Aucun contrôle d'erreur n'est effectué, ce travail étant de la responsabilité de la couche supérieure (IP). De même, aucune notion d'adresse ip n'est présente. Il est donc nécessaire de configurer l'interface SLIP en lui spécifiant une adresse locale et une adresse distante. Il s'agit d'une interface "point à point".

La figure 12.1 montre un datagram SLIP.

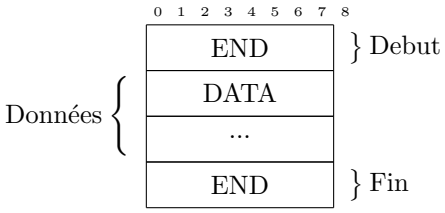

FIGURE 12.1 – Datagram SLIP

Si les données à transmettre contiennent le caractères END ou ESC, ce dernier est échappé par le caractère ESC et suivi soit du caractère ESC_END soit du caractère ESC_ESC.

La configuration de SLIP en utilisant l'environnement Linux est aisée car tout le nécessaire est déjà fourni. Il faut commencer par créer l'interface réseau virtuelle qui sera liée à un port série déterminé.

```
slattach -s 115200 -p slip /dev/ttyUSB1
```

Il reste ensuite à configurer la nouvelle interface réseau sl0 qui a est crée par la manipulation précédente. Cette configuration se fait par les même mécanismes que pour une interface réseau traditionnelle. Il reste encore à spécifier le MTU de la connexion. Ce dernier doit être d'une taille de 60 octet au minimum mais d'une taille de 255 octet au maximum.

```
ifconfig sl0 192.168.0.1 pointopoint 192.168.0.2 mtu 100
```

Afin qu'il soit possible de joindre une autre adresse ip du réseau mesh que celle de la passerelle, il est nécessaire d'ajouter une route

```
route add -net 192.168.0.0/24 sl0
```

12.1.2 Analyse d'une trame

Puisque le protocole SLIP utilise le mécanisme standard de la couche ip, il est possible d'effectuer la capture de trames ip à l'aide d'un logiciel tel que wireshark.

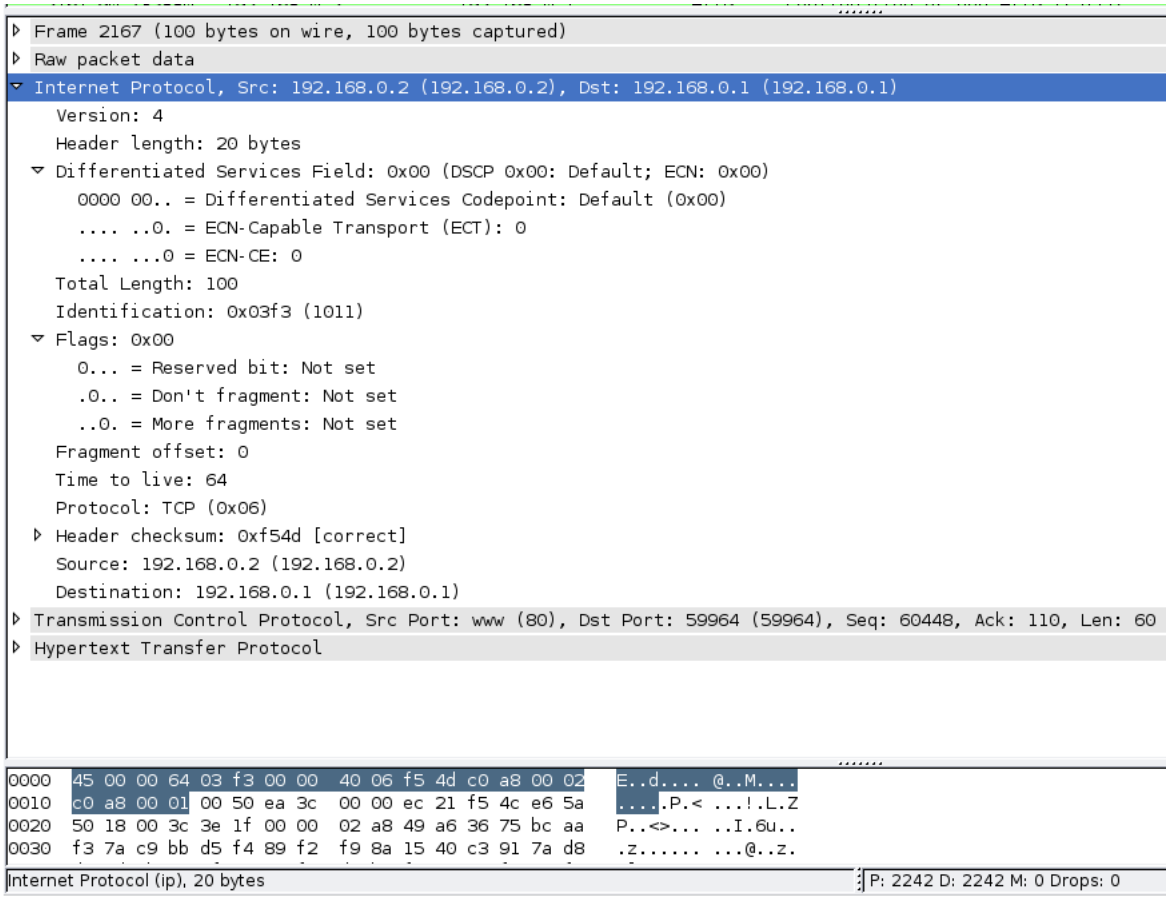

FIGURE 12.2 – Détail IP de la trame avec wireshark

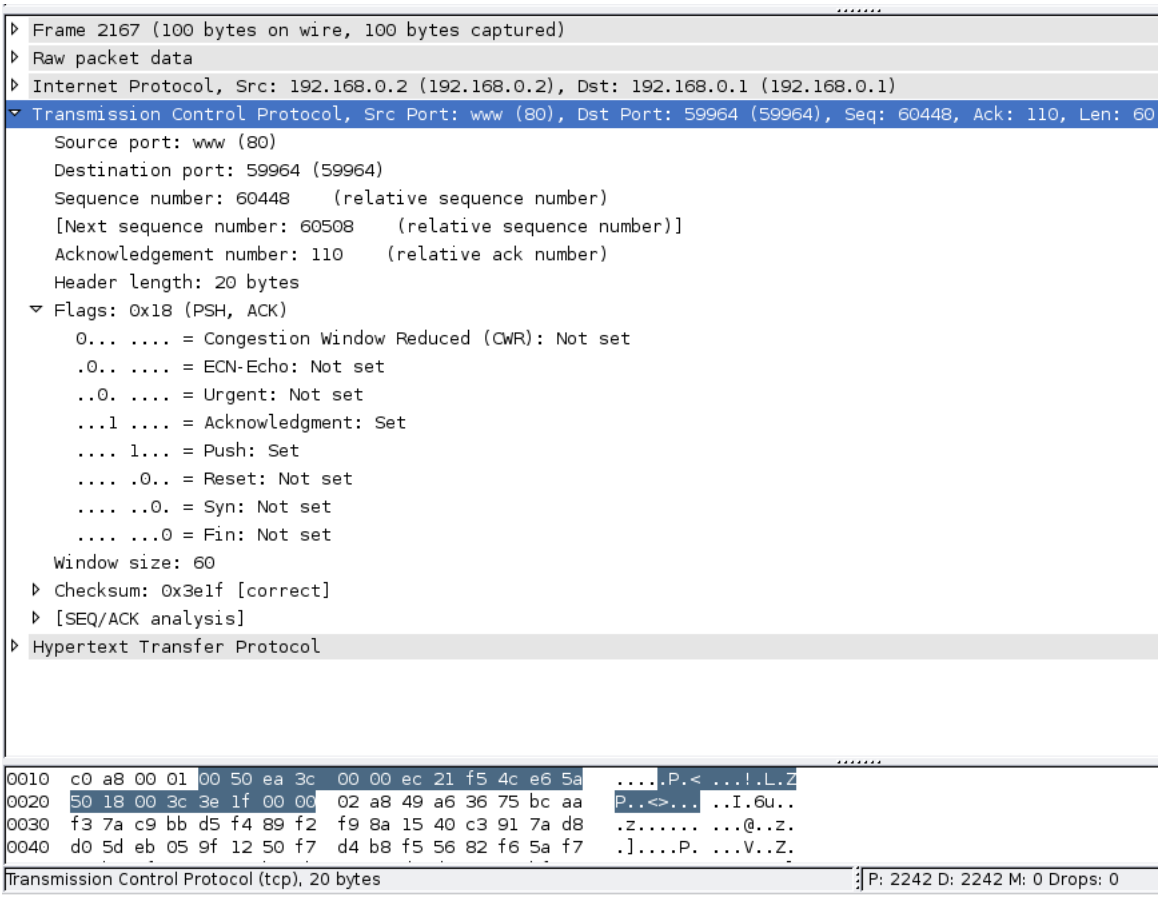

FIGURE 12.3 – Détail TCP de la trame avec wireshark

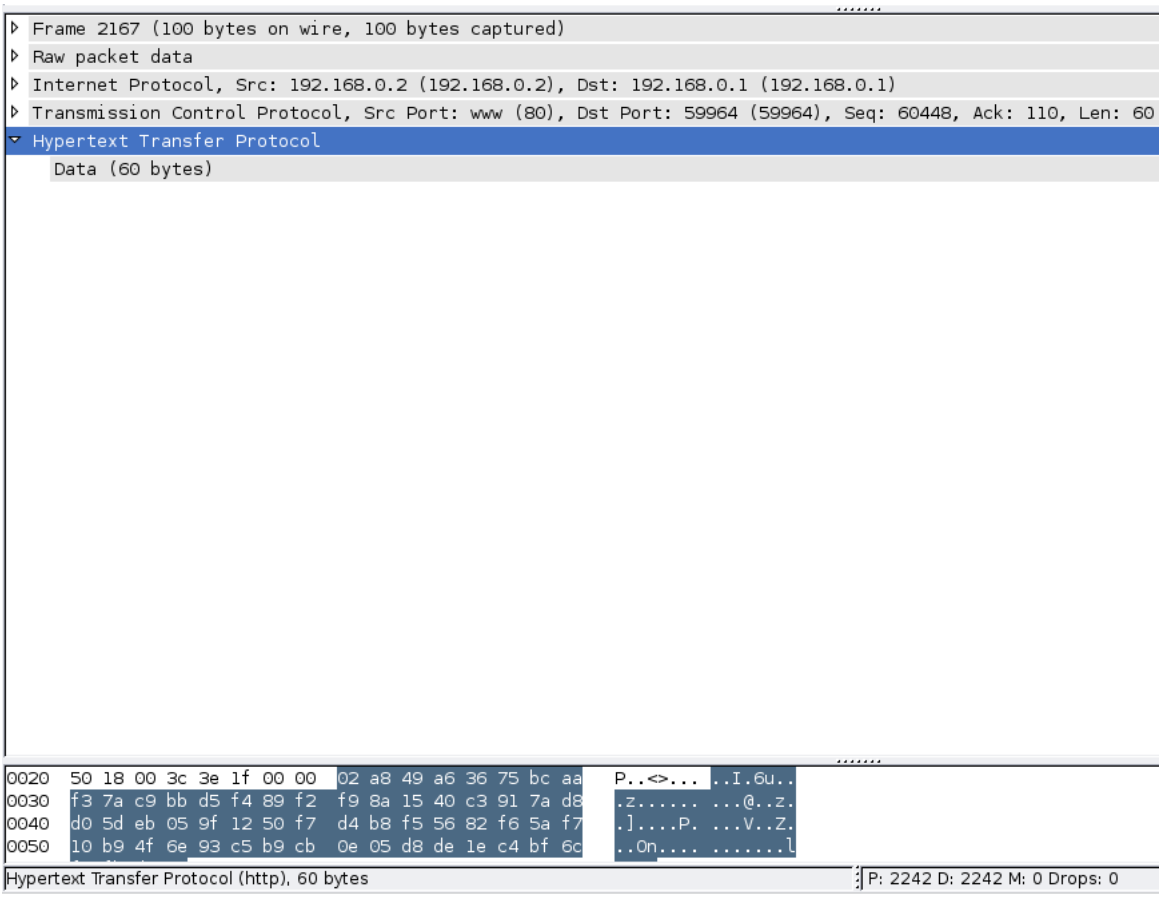

FIGURE 12.4 – detail payload de la trame avec wireshark

12.1.3 Performance

Vitesse théorique maximale : 115'200 baud/seconde Comme nous utilisons un codage bivalent, nous pouvons affirmer que 115'200 baud/seconde équivaut à 115'200 bit/seconde. Le transfert d'un caractère nécessite 8bit pour le caractère lui même plus 1 bit de start et 1 bit de stop. Ce qui fait qu'un vitesse de 11'520 octet/seconde est possible. Le transfert de donnée en utilisant un mtu de 100 et le protocole TCP permet un transfert effectif de maximum 60 octet par trame. En effet, la couche ip fait 20 octet, de même que la couche tcp. $\frac{11520*60}{100} = 6912 octets/s$

En pratique, cette vitesse n'est pas atteinte, elle se situe aux alentours de 1,5kbit/s. La principale raison est qu'il est nécessaire d'attendre la réception d'une trame de quitancement de la part de l'ordinateur client. De par la construction de la pile ip, seule un paquet est traité entre chaque réception d'un paquet de quitancement. La plus part des piles ip modernes envoient un paquet de quitancement de manière retardée, ceci jusqu'à 200ms.

Les pages servies par le serveur web sont stockées dans le répertoire

`apps/webserver/httpd-fs`

L'execution de l'application

`makefsdata`

se charge de convertir les fichiers devant être servis par le serveur web en fichier en un fichier nommé

`httpd-fsdata.c`

qui est linké à l'exécutable uip.

12.1.4 Adapation

clock-arch.h

copier le fichier clock-arch.h dans le projet et adapter la valeur `CLOCK_CONF_SECOND`. Il s'agit de la granularité du timer. Nombre de tick du timer nécessaire pour faire 1 seconde.

uip-conf.h

Configuration spécifique à un projet.

```
typedef UINT8 u8_t;
typedef UINT16 u16_t;
#define UIP_CONF_BUFFER_SIZE     100
```

uipopt.h

Configuration de uip (dépendamment des valeurs fournies dans uip-conf.h). Nous n'utilisons pas de couche ethernet, il est donc possible d'indiquer à uip que la longueur du champ nécessaire à ethernet est de 0.

```
#define UIP_LLH_LEN     0
```

clock-arch.c

```
clock_time_t
clock_time(void)
{
  return getTime();
}
```

12.2 Ajout d'une application uip

dans `config.h` ajouter une constante contenant le numéro de port de l'application et spécifier si l'application doit être activée :

```
#define UIP_HTTPD_PORT 80

#define UIP_HTTPD 1
```

Ou

```
#undef UIP_HTTPD

#ifdef UIP_HTTPD
if(is_app_enabled(&global_application_uip_enabled, "HTTPD")) {
httpd_init();

if (uip_app_register(httpd_appcall, REG_TCP, HTONS(UIP_HTTPD_PORT), REG_PASSIVE, "HTTPD", TRUE))
while (1)
;
}
}
#endif
```

Chapitre 13

Taches YottaOS nécessaires

13.1 Préambule

Plusieurs tâches sont nécessaires au bon fonctionnement de base des noeuds mesh. L'interdépendance des tâches et leur utilité sont expliqués dans ce chapitre.

13.2 Transmission série

Les tâches nécessaires à un noeud de type passerelle sont :

Algorithm 4 Receive_Slip_Data

```
loop
  if freeItem fifo uip_input_paquet then
    if paquet = osGetUart then
      enqueue paquet ⇒ freeItem
    end if
    libère espace libre
  end if
end loop
```

Algorithm 5 Send_Data_Slip 100 100

```
loop
  if paquet en cours == null then
    prend nouveau paquet slip_fifo_paquet
  end if
  if paquet en cours != null then
    envoie paquet[index] sur uart
    if tout envoyé then
      libère slip_fifo_paquet
      reset index
    end if
  end if
end loop
```

13.3 Transmission radio

Les tâches nécessaires à un noeud de type radio sont.

Algorithm 6 Send_Data_RF 100 100

```
loop
  if synchonized then
    if rf not busy then
      if rf not ack then
        if pas de paquet precedent then
          paquet ⇐ dequeue rf_fifo_paquet
          if not paquet then
            return  return
          end if
          if rndWait < getTime then
            rndWait ⇐ (RF_CHANNEL_COUNT * RF_CHANNEL_DURATION * RND) +
            getTime
            envoie paquet uart rf
          end if
        end if
      else
        if paquet != null then
          release paquet rf_fifo_paquet
        end if
        if paquet ⇐ dequeuerf_fifo_paquet then
          envoie paquet uart rf
        end if
      end if
    end if
  else
    if noeud n'est pas la passerelle then
      print  desynchronisé
    end if
  end if
end loop
```

13.4 Les tâches nécessaire aux deux type de noeuds

Algorithm 7 Receive_Rf_Data 100 100

```
to_be_refurbed ⇐ TRUE
loop
  if fifo_paquet = getNodeFifo uip_input_paquet == NULL then
    fifo plein..
    return return
  end if
  if tmp_paquet = getUart rf then
    if tmp_paquet.data == 0x03 then
      print decodeError
    else if tmp_paquet.data == 0xD2 then
      print ackRF
    else if tmp_paquet.data == 0xD3 then
      print donnee
      fifo_paquet ⇐ tmp_paquet
      enqueue fifo_paquet
    end if
  else
    refurb fifo_paquet uip_input_paquet
  end if
end loop
```

Algorithm 8 Synchronized 1000 1000

```
firstlow ⇐ FALSE
loop
  if not RFSyncPin() then
    if firstlow then
      desynchronized ⇐ TRUE
    end if
    firstlow ⇐ TRUE
  else
    desynchronized ⇐ FALSE
    firstlow ⇐ FALSE
  end if
end loop
```

Algorithm 9 Receive_Data

```
loop
  if paquet fifo uip_input_paquet then
    buffer_uip ⇐ paquet
    libère paquet fifo uip_input_paquet
    if paquet ne doit pas etre forwardé then
      res ⇐ traitementPaquet
      if res > 0 then
        traitement reponse
      end if
    else
      forward paquet
    end if
  else if doit effectuer traitement periodique then
    traitement periodique tcp
    if UDP activé then
      traitement periodique udp
    end if
  end if
end loop
```

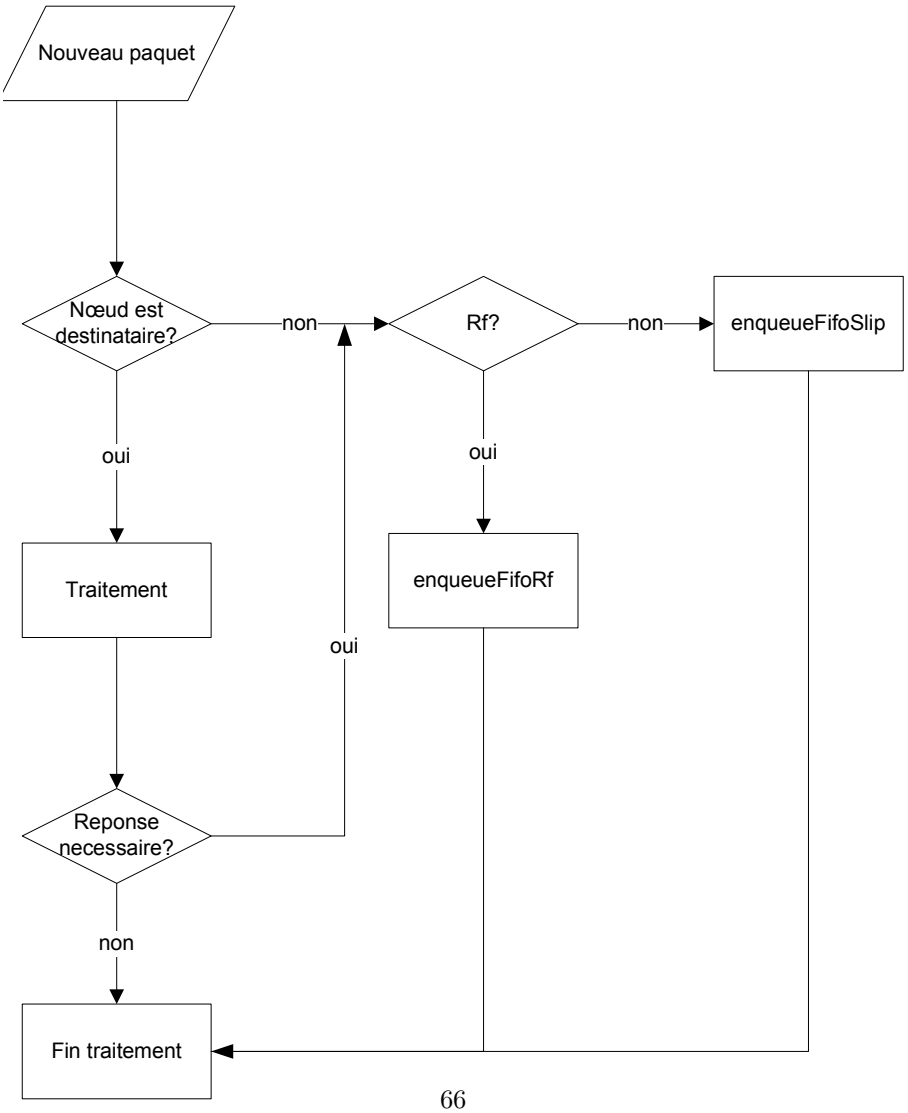

FIGURE 13.1 – Diagramme de flux de reception et traitement d'un paque ip

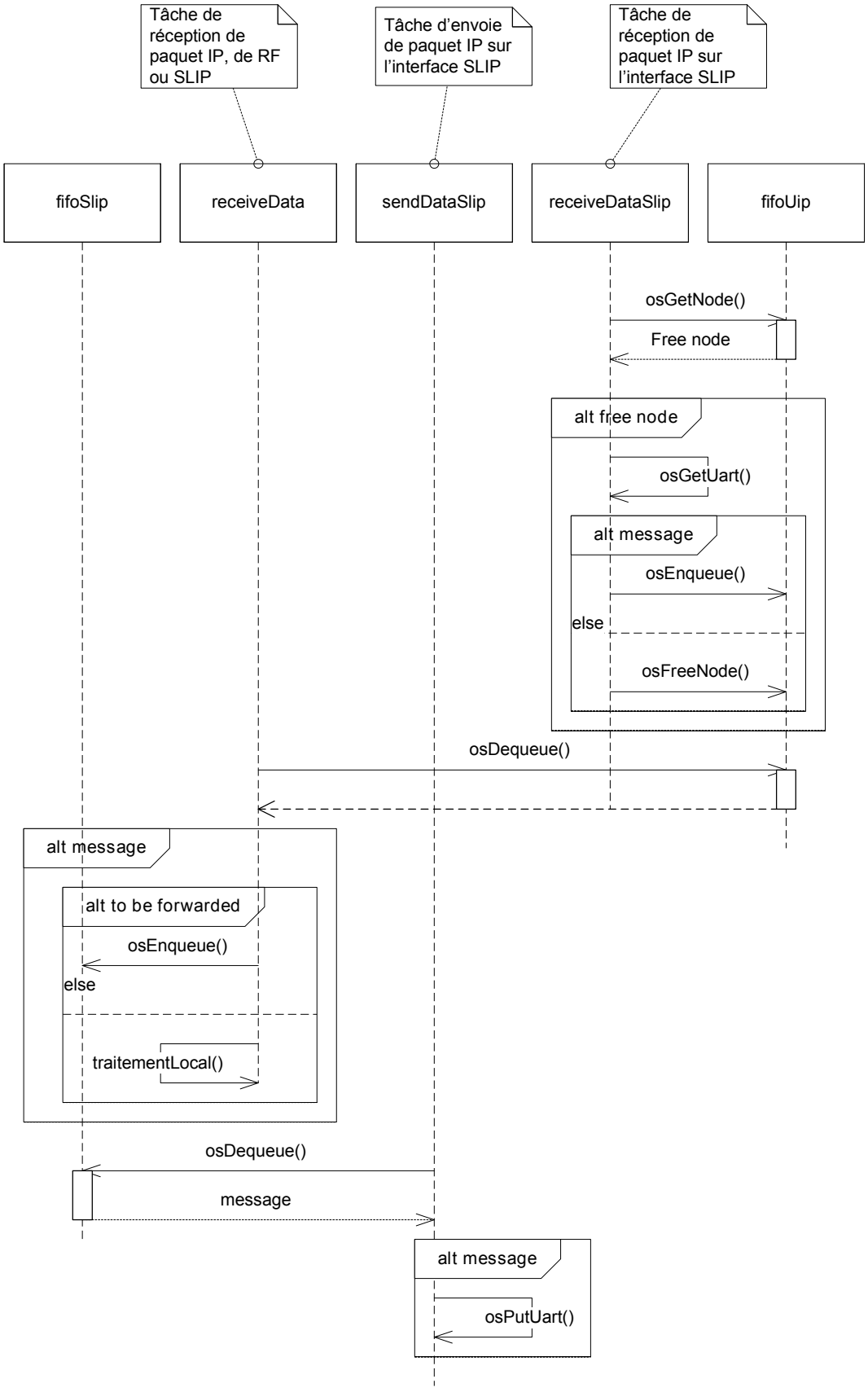

FIGURE 13.2 – Diagrame de séquence de reception et traitement d'un paque ip sur interface SLIP

13.5 Tâches d'acquisitions

13.6 Tâches diverses

Chapitre 14

Reseau mesh

14.1 Préambule

Les applications utilisées par le réseau mesh peuvent être classifiées en trois catégories : Alarme/Transmission de données périodiques Transmission de données à la demande et contrôle. La figure 14.1 représente une vue d'ensemble du système d'acquisition de données.

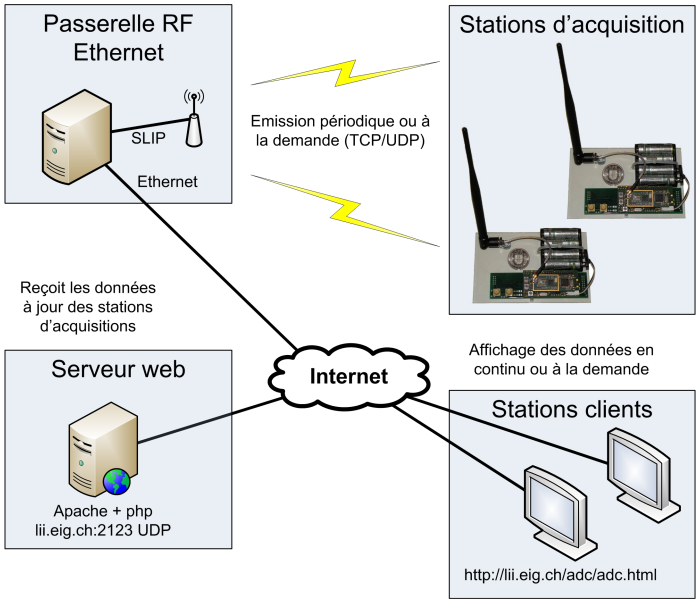

FIGURE 14.1 – Schema bloc general

La figure 14.2 explique plus en détail la topologie du réseau mesh, y compris les connections entre les différents éléments.

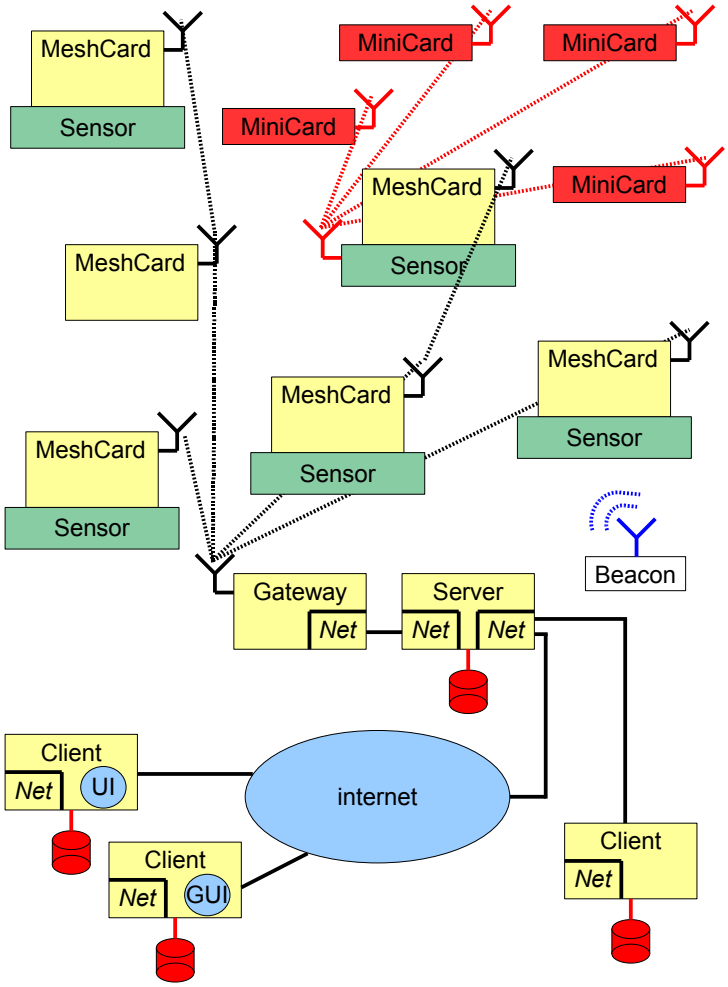

FIGURE 14.2 – Schema bloc general détaillé

14.2 Eléments à disposition

14.2.1 Carte mesh

La figure 14.3 montre les différentes configurations utilisées dans le système mesh. Trois cartes sont présentées : une carte de base communiquante sans fil, une carte de même type incluant un capteur ou un actuateur et une cart comportant également un deuxième module radio permettant de communiquer entre différents types de réseau.

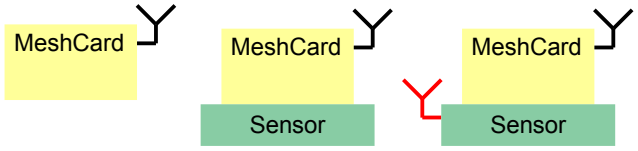

FIGURE 14.3 – Element mesh card

14.2.2 Carte passerelle

La figure 14.4 permet d'interfacer les noeuds sans fil avec le réseau filaire. La carte passerelle effectue un routage entre deux classes d'adresses : la classe spécifique au réseau filaire et le réseau sans fil.

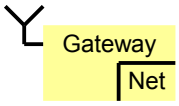

FIGURE 14.4 – Element mesh passerelle

14.2.3 Element serveur

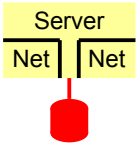

FIGURE 14.5 – Element mesh serveur

L'élément représenté par la figure 14.5 est constitué d'un ensemble matériel et logiciel qui interagit avec les noeuds, par échange de message. Il permet également aux différents clients de communiquer avec les noeuds du réseau mesh par son intermédiaire. Le serveur est capable de récupérer des données de manière autonome et de les stocker pour un traitement ultérieur. Plus de détails dans le chapitre 10.

subsectionElement mesh interface utilisateur

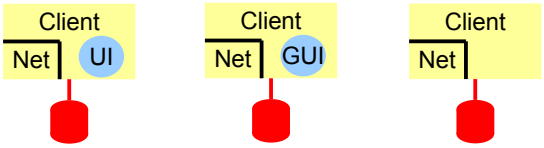

FIGURE 14.6 – Element mesh interface utilisateur

La structure modulaire des éléments mesh permettent de disposer de plusieurs types d'interface utilisateur (UI) selon les besoins propres à chaque application. Exemples d'interfaces utilisateur :

- ligne de commande ;
- interface web dynamique ;
- logiciel applicatif sur différentes plateforme.

14.3 Emission de données sans requête

Lors de la mise sous tension, chaque noeud du réseau mesh va émettre périodiquement un message UDP à destination d'un éventuel serveur 14.7. Ce type de communication peut être utilisé par une application qui nécessite de transmettre de manière automatique des données traitées une application récepteur.

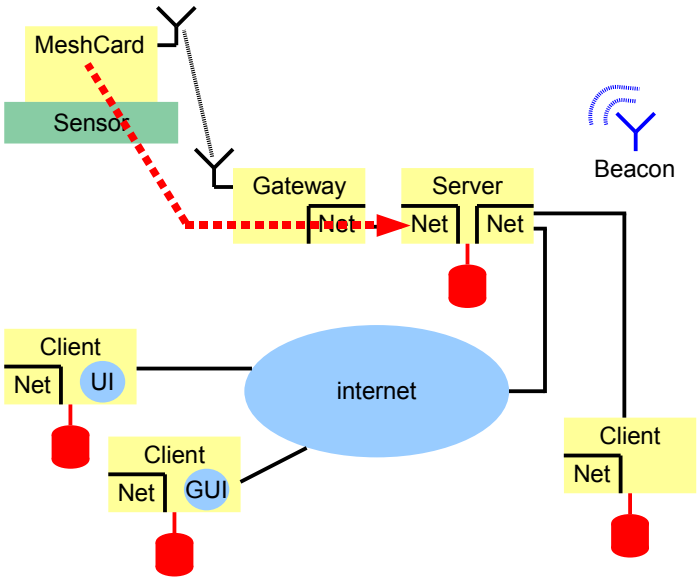

FIGURE 14.7 – Application mesh alarm

14.3.1 Alarme

Ce message contient l'adresse ip du noeud émetteur et le numéro de port de son application de contrôle. Dès qu'un serveur reçoit ce message, il va tenter d'établir une connexion TCP sur son application de contrôle.

14.3.2 Mise à jour de vu metres

Utilisation des adc comme source de données. Création d'une application qt pour faire passerelle entre messages udp et multiples clients tcp Création d'une application web pour mettre à jour les vumetres suivant les données recues par l'application qt.

14.4 Transmission de données à la demande

Ce type d'application met à disposition un socket TCP de type serveur qui attend les requêtes en provenance des clients, ceci par l'intermédiaire de l'élément serveur 14.2.3. Ce type d'application peut être couplé avec une tâche YottaOS 9.1 qui se charge de préparer les données à transmettre.

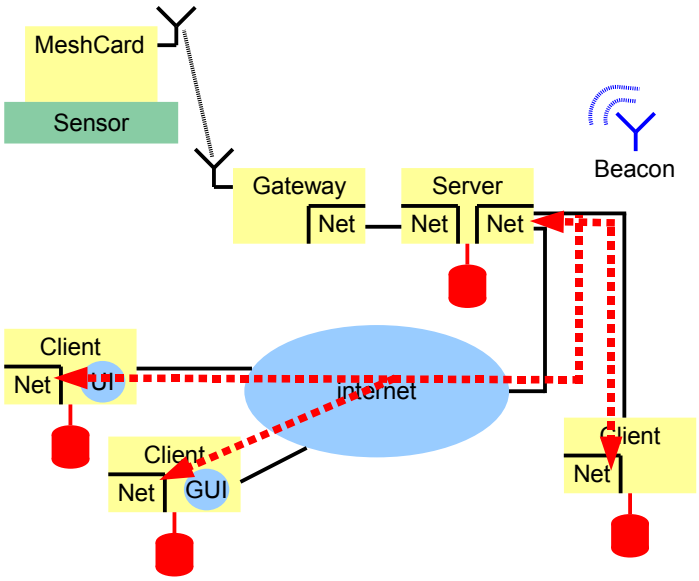

FIGURE 14.8 – Application mesh à la demande

14.4.1 Application de contrôle

Un exemple d'application utilisant ce type de communication est l'application de contrôle qui permet d'interroger la carte afin de connaître ses caractéristiques et de contrôler son fonctionnement. Une fois que le serveur à réussi à se connecter à l'application de contrôle, c'est à lui d'envoyer périodiquement un message au noeud afin que ce dernier n'envoie plus de messages d'alarme. Le serveur connecté à l'application de contrôle peut interroger le noeud afin de connaître les applications et leurs ports à disposition.

Chapitre 15

Applications MESH

15.1 Liste

Le tableau 15.1 liste les différentes applications implémentées et à implémenter.

Nom	Numéro	Affichage graphique	Alarme	...
ECG	1	Osqoop	Oui	
Velo	2	Osqoop		
Accéléromètre	3			Rapide
Chatière	4			
EMG	5	Osqoop	Oui	Temps réel
Température	6	Osqoop		
GPS	7	Google map		
Table de routage	8	QRoutingTableTreeWidget		

TABLE 15.1 – Liste des applications

15.2 Liaison entre UI et carte

Les commandes servant à échanger des données entre l'application GUI et les applications de la carte MESH sont définies à la sous-section B.3.2.

15.3 ECG

L'application ECG récupère les signaux d'un électrocardiogramme. Ces signaux passent à travers un convertisseur analogique-numérique avant d'être récupérés par la tâche d'acquisition. Les mesures ont une taille de 10 bit. Pour éviter de perdre trop de place en mémorisant chaque mesure sur 16 bit, trois mesures sont compressées ensemble afin d'avoir une donnée faisant 30 bit. Ainsi, lorsque cette donnée est écrite en flash, il n'y a que 2 bit de "perte" ; mais ces 2 bit pourraient être utilisés pour stocker d'autres informations.

15.4 Velo

L'application vélo utilise une carte additionnelle permettant de récupérer les informations de quatre vélos simultanément. Ces informations sont simplement un signal "1-0" suivant si le détecteur fixé au cadre du vélo a détecté un passage de roue. L'application vélo compte le nombre de passage de roue pour chaque vélo et incrémente un compteur, par vélo, de 32 bit.

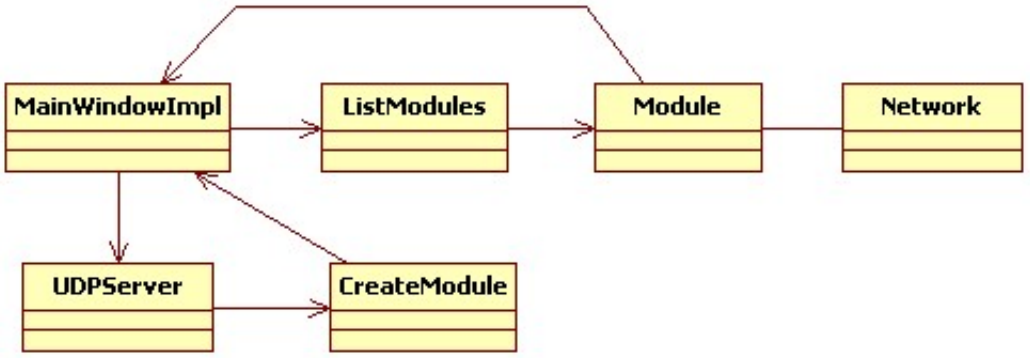

FIGURE 15.1 – Graph UML des interconnexions des classes

Une tâche écrivant dans la flash stocke l'état de l'horloge système ainsi que la valeur actuelle du vélo. Il ainsi possible de connaître l'interval de temps entre deux enregistrements. Ces enregistrements peuvent ensuite être lus et envoyés au GUI pour traitement.

Le GUI permet également de modifier la valeur du frein électromagnétique des vélos.

15.5 GUI

L'application Qt pour le projet se trouve sur un ordinateur (Windows, GNU/Linux, MacOS). Cette application permet de gérer les modules et récupérer les données se trouvant sur ces modules.

L'application est composée de plusieurs classes dont voici la liste :

- `MainWindowImpl` : gère la fenêtre principale ;
- `ListModules` : liste et gère les modules ;
- `Module` : contient toutes les informations d'un module physique ;
- `Network` : gère le réseau d'un module ;
- `UDPServer` : serveur UDP ;
- `CreateModule` : créé un objet module d'après les informations de UDPServer.

La figure 15.1 présente les interconnexions entre les classes.

La classe `MainWindowImpl` gère la fenêtre principale en proposant des fonctions d'altération de cette fenêtre. Les classes souhaitant modifier l'affichage doivent passer par la classe `MainWindowImpl` ; c'est pourquoi la classe `Module` a un lien sur `MainWindowImpl`. Cette dernière contient également la liste des modules, gérée par la classe `ListModules`. Cette liste ne peut être accédée que depuis `MainWindowImpl` c'est pourquoi la classe `CreateModule` a également un lien sur `MainWindowImpl`.

La liste des modules est gérée par `ListModules` qui contient les fonctions permettant d'interagir avec les modules, telle que l'envoi de message. Pour envoyer accéder à un module, il est obligatoire de passer par `ListModules` et lui indiquant l'identifiant du module désiré. Cette dernière se charge ensuite de rediriger la demande vers le module désiré (géré par la classe `Module`).

La classe `Module` contient toutes les informations d'un module physique. La partie réseau d'un module est gérée par la classe `Network`. Cette dernière gère l'envoi des messages ainsi que la réception des messages entrant. Les informations devant être affichées sont envoyées à la classe `MainWindowImpl`.

La classe `UDPServer` gère un serveur UDP. Ce dernier est utilisé pour permettre aux modules de s'identifier auprès du GUI. Le serveur UDP créé un thread `CreateModule` qui créera un objet de la classe `Module` et l'ajoutera à la liste (en passant par `MainWindowImpl`). Lors de la création de l'objet module, la connexion TCP est initialisée.

Une possibilité d'amélioration serait d'ajouter une classe par application permettant de gérer les messages reçus dans la classe `Network`.

15.6 uIP-MESH

L'application MESH pour uIP sert d'interface de communication entre les tâches de *YottaOS*et le GUI. Cette application reçoit les données du GUI et les envoie aux différentes tâches puis reçoit les données des tâches et les envoie au GUI.

Des FIFOs sont utilisés pour communiquer entre l'application de uIP et les tâches. Ces FIFOs supportent les problèmes d'accès concurrents des tâches.

Chapitre 16

Adressage et applications

16.1 Preambule

Les noeuds du réseau mesh doivent être accessible par une adresse ip individuel.

16.2 Id mesh

Les cartes sont flashées avec le même code binaire, ceci pour raison de simplicité. Ceci implique qu'il faille pouvoir différencier à l'exécution chaque carte mesh. Le block xxx de la mémoire flash externe est réservé et contient un identificateur propre à chaque carte mesh. La lecture de cet identificateur permet de différencier le code exécuté sur chaque carte.

16.3 configuration

Voila un exemple complet de configuration d'une carte mesh (réseau et application)

```
case (UINT32) 0x00000005: {
                        uip_ipaddr(&rf_ip, 192, 168, 3, 6);
                        uip_ipaddr(&slip_ip, 192, 168, 2, 6);
                        set_ip_config(&global_config_ip, "CARTE5", rf_ip, slip_ip,
                            5, CLIENT, TRUE);
                        add_app(&global_application_uip_enabled, "ADC_UDPC", 2125);
                        add_app(&global_application_uip_enabled, "HTTPD", 80);
                        break;
                }
```

Détail : La ligne suivante configure l'adresse ip associée au module radio y-lynx

```
uip_ipaddr(&rf_ip, 192, 168, 3, 6);
```

La ligne suivante configure l'adresse ip associée au module radio y-lynx

```
uip_ipaddr(&slip_ip, 192, 168, 2, 6);
```

La ligne suivante configure spécifie le nom de la carte mesh ainsi que son type (SERVER ou CLIENT) utilisé lors de la configuration du module radio. Le dernier champ boolén spécifie si la carte mesh est une passerelle slip (TRUE) ou non (FALSE).

```
set_ip_config(&global_config_ip, "CARTE5", rf_ip, slip_ip, 5, CLIENT, TRUE);
```

Il faut s'assurer qu'il n'y ait un qu'un seul noeud mesh qui soit défini en tant que SERVER dans le réseau.

16.4 Réseau RF

Les adresses rf des cartes mesh sont compris entre 192.168.3.1 et 192.168.3.254

16.5 Réseau

16.6 Applications

16.6.1 Activation application

Les applications uip et mesh doivent être activées globalement dans le fichier config.h Desactivation httpd

```
#undef UIP_HTTPD
```

Activation httpd

```
#define UIP_HTTPD 1
```

Les applications micro ip (préfixées par `UIP`) doivent être activées ou non, ceci de façon individuel dans le fichier `mesh_fuction_ip_config.c`

```
add_app(&global_application_uip_enabled, "ADC_UDPC", 2125);
```

Chapitre 17

synchrnoisation temporelle

17.1 Préambule

Il est nécessaire dans un réseau de type mesh de disposer d'une base de temps commune, afin de pouvoir dater correctement les données capturées. Il n'est actuellement pas possible de transmettre d'information temporelle dans le signal de synchronisation du signal radio des modules y-lynx, il faut donc procéder autrement.

17.2 Real time clock (RTC)

Les cartes mesh disposent d'un RTC par l'intermédiaire du processeur arm. Ce dernier a une précision d'une seconde, ce qui est suffisant pour une base de temps commune. Une fois le rtc mis à jour, il est possible d'utiliser un timer 32 bit du processeur arm, configuré avec une granularité de 1ms.

17.3 Network time protocol

La mise à jour du RTC est effectuée par le protocole NTP.

Chapitre 18

Optimisations

18.1 Accès à la mémoire flash interne

La famille des processeurs LPC2xxx dispose d'un contrôleur de mémoire flash amélioré qui peut être activé ou non. La configuration de ce contrôleur est ajoutée dans le fichier `yottaOS_a.S`. Le fonctionnement de ce contrôleur n'a d'effet que pour l'accès à la mémoire flash interne. En effet, cette mémoire est spéciale dans le sens où elle est partagée en deux parties égales, chacune accessible en concurrence. Le contrôleur est ainsi capable de charger des données un peu à la manière d'un système en pipeline. Il faut garder à l'esprit qu'il ne s'agit pas d'une acceleration constante et que dans certain cas, aucune accélération ne sera obtenue. Pour la mesure du pire temps d'exécution d'un tâche, il ne faut pas oublier de désactiver le contrôleur. Il peut être réactivé par la suite à fin d'économie de temps de calcul et donc d'énergie.

18.2 Taille de la pile de service

La pile de service de *YottaOS* est partagée entre toutes les tâches. Il faut s'assurer qu'elle soit de taille suffisante sous peine de corrompre les données adjacentes. La pile est initialisée avec un motif particulier (*0xDEADBEEF*) au démarrage de *YottaOS*. Après avoir exécuté le programme, il est ainsi possible de *voir* jusqu'où la pile a été utilisée en analysant l'espace mémoire de la pile.

Chapitre 19

Problèmes

19.1 Eclipse

19.1.1 Chemin Cygwin <=> *Microsoft Windows*

dans le cas ou cygwin est installé, il faut ajouter un mapping du chemin windows vers linux. Window->Preferences->C/C++->Debug->Common Source Lookup Pathand add a mapping from C : to /cygdrive/c . (fig. 19.1).

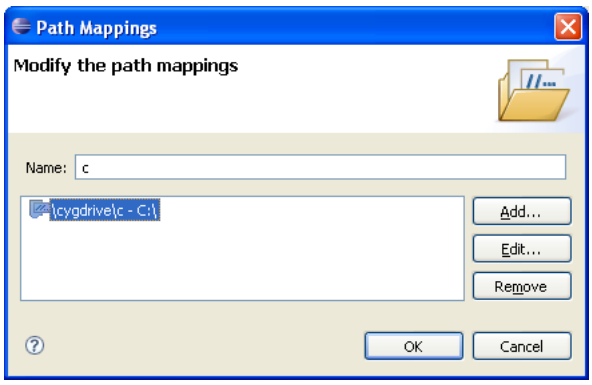

FIGURE 19.1 – Ajout d'un mapping Cygwin <=> Windows

19.1.2 Suspend and Resume

La version actuelle de *CDT* ne supporte pas le fait de faire un suspend et un resume dans une session de debuggage. Cette fonctionalité sera supportée dans la version 4.0 de *CDT*.

19.2 Esterel

Esterel a besoin de l'exécutable *link.exe* afin de créer un exécutable nécessaire à la simulation. Un conflit potentiel existe avec les binaires fournis par la chaîne de compilation. En effet le programme *link.exe* fourni par ce bias est prévu pour effectuer un raccourcis windows et *Esterel* tente de l'utiliser à la place de celui fourni avec *Visual Studio*. Etant donné que le programme *link.exe* fourni par la chaîne de compilation n'est pas utilisé, le fait de le renommer ou de l'effacer règle le problème.

19.3 Désactivation JTAG

Lors de l'exécution d'un programme comportant des tâches sous *YottaOS*, il n'est plus possible de communiquer avec la cible à travers le port JTAG. La cause de ce comportement est que lorsque aucune tâche n'est en activité, seule la tâche oisive est en activité. Cette dernière met le processeur en mode économie d'énergie, ce qui a pour effet de désactiver le port JTAG. Seul une interruption peut réveiller le processeur, en l'occurrence l'horloge. YottaOS permet désormais de désactiver l'économie d'énergie. Il faut mettre la directive `_DEBUG_` comme paramètre à GCC afin de désactiver la gestion de l'énergie.

19.4 Module radio

19.5 Problèmes

La documentation du module *Y-Lynx TRM8053-025* comporte plusieurs erreurs ou imprécisions qui empêchent le bon fonctionnement du module avec la *carte MESH*.

- La documentation fournie correspond au module Y-LYNX TRM8053-500. La bonne version est en fait disponible sur demande.
- Les broches RX et TX sont inversés dans la documentation. Nous avons donc modifié la carte MESH en conséquence.
- La vitesse par défaut du module en mode RS-232 est de 57'600bps dans la documentation. Il faut en fait communiquer avec la carte à 9'600bps. Il est possible d'augmenter cette vitesse jusqu'à 115'200bps par programmation.
- La communication en mode RS-232 est désactivée. Il faut mettre la broche *CS* (chip select) du bus SPI au niveau `1` en tout temps, ceci dès la mise sous tension du module, faute de quoi le module passe en mode SPI et ne peut plus revenir en mode RS-232. Une résistance de pull-up est ajoutée à la *carte MESH*.
- Lors de l'envoie d'une donnée à l'aide de la commande *send_data* et dans le cas d'une erreur, le module répond avec un code d'erreur de quatre bytes (non spécifié dans la documentation). Le quatrième octet indique en fait qu'une erreur est commise dans *data_mode_tx* et ou *data_mode_rx*. La valeur du quatrième octet spécifie de quel champ il s'agit, grâce à la position du bit d'erreur. Exemple : `0x04` signifie erreur sur l'adresse de destination.
- Utilisation de la carte en mode TDMA. Le fait de limiter le nombre de canaux de fréquences empêche de sélectionner un canal dont l'indice est plus grand que le nombre de canal utilisés. Exemple : en cas d'utilisation d'un seul canal, seul le canal 1 peut être utilisé. Le canal 0 sert à envoyer le *beacon* et donc aucune autre donnée ne peut y être envoyée. Il faut donc au minimum utiliser deux canaux.
- Nous devons toujours déterminer si nous pourrons utiliser le beacon dans notre application. Initialement, il était convenu que non. D'après la documentation, il ne semble pas possible de se passer de beacon. En effet, il est précisé qu'un module client qui ne reçoit pas de beacon entre en mode veille. Néanmoins, si le beacon à une portée suffisante, il pourrait être envisageable de l'utiliser. Ainsi nous pourrions diminuer de façon importante le nombre de collisions en utilisant un canal de fréquence par module par exemple.
- Les modules ont été fournis sans antenne. Il semble obligatoire d'en mettre une faute de quoi le transistor de sortie peut-être endommagé. Nous tentons de souder un fil de 8cm sur le module. Malheureusement, un des deux modules nous a été livré endommagé et la piste où il faut souder l'antenne est arrachée. Nous devons donc ouvrir le module afin de souder l'antenne.
- La commande self-test n'existe plus. Le module effectue désormais un self-test à la mise sous tension et en cas d'erreur, envoie en boucle la valeur du code d'erreur sur l'UART.
- La communication est aléatoire (un octet reçu de temps en temps). Le fait de modifier le *network_id* sur un client non synchronisé empêche toute émission future d'un message. Le workaround est de passer le module en mode server, de modifier le *network_id* et de passer de

nouveau en mode client. Nous avons testé cette manipulation, elle ne semble pas suffisante. Nous n'avons pas modifié le *network_id* et la communication est fonctionnelle.

– La vitesse maximale de transmission RF est de 76.1 kbit/s avec les versions de module que nous avons à disposition (0.33)
– La transmission de plusieurs octets consécutifs de la *carte MESH* au module radio peut être long (tab. 9.2). *YottaOS* est un système d'exploitation multitâche préemptif, ce qui signifie qu'une tâche peut être avortée durant son exécution et reprise par la suite. Si la tâche qui envoie une commande au module radio est interrompue, il se peut donc que le module radio considère l'envoie de la commande comme terminé et dans ce cas, incomplète. Il est possible de définir le temps maximum entre deux octets reçus par le module radio. Il faut néanmoins garder ce temps le plus petit possible sous peine d'avoir un grand temps d'attente entre l'envoie consécutif de deux commandes.

Bibliographie

[1] Drivers jtagkey-tiny. Internet. http://www.amontec.com/jtagkey.shtml#drivers. 10

[2] Lsn. 5

[3] Cypress. Cy62167dv30 cmos static ram. Internet, sep 2006. http://download.cypress.com.edgesuite.net/design_resources/datasheets/contents/cy62167dv30_8.pdf. 28

[4] Claude Evéquoz and Bertrand Hurst. Yottaos : un noyau temps réel à basse consommation. Internet, 06 2007. 6

[5] Michael Fischer. Intergrated development environment. Internet. http://www.yagarto.de/download/yagarto/yagarto-ide-20061002-setup.exe. 15

[6] Michael Fischer. Yagarto - yet antother gnu arm toolchain. Internet. http://www.yagarto.de/. 12

[7] Inria. Esterel : a synchronous reactive programming language. Internet. http://www-sop.inria.fr/esterel.org/. 23

[8] ARM Ltd. Gnu arm toolchain for cygwin, linux and macos. internet, 08 2006. http://www.gnuarm.com/. 15

[9] NXP. Lpc 2000 flash utility v2.2.3. Internet, 05 2004. http://www.nxp.com/products/microcontrollers/support/software_download/lpc2000/index.html. 6, 10, 31

[10] Olimex. Lpc2214 header prototype development board with external flash and sram. Internet, 12 2004. http://www.olimex.com/dev/lpc-h2214.html. 30

[11] Dominic Rath. Open on-chip debugger. Internet. http://openocd.berlios.de/web/. 6, 12

[12] Y-Lynx Sarl. Ylx-trm8053-025. Internet, April 2007. http://www.y-lynx.ch/socket_modems_TRM8053-500.php. 35

[13] SUN. Java 6.1. Internet. http://javadl.sun.com/webapps/download/AutoDL?BundleId=11193. 15

[14] Various. Make wikipedia. Internet, 2004. http://fr.wikipedia.org/wiki/Make. 95

[15] Luigi Zaffalon. *Programmation synchrone de systèmes réactifs avec Esterel et les SyncCharts.* Presses Polytechniques et universitaires romandes, 2005. 23

Table des figures

Annexe A

Annexe

A.1 Problèmes résolus

A.1.1 YottaOS

Le teste d'un programme un peu plus complexe que les exemples fournis à fait ressortir un problème. L'exécution du programme se terminait aléatoirement. Après une investigation des différentes possibilités de problème, il s'est avéré que la mémoire alloué à la pile est insuffisante. Afin de détecter plus aisément un dépassement de pile, cette dernière est remplie avec un pattern spécifique (`DEADBEEF` pour la pile de service et `CAFEBABE` pour la pile `IRQ`). Ceci est effectué par un code additionnel dans le fichier `yottaOS_a.S`.

L'utilisation de la procédure *osResetUART* depuis un handler d'interruption n'est pas possible. En effet la procédure *osResetUART* afin d'effectuer une affectation atomique désactivait et réactivait les interruptions, ceci sans tester l'état initial des interruptions. Il a également été nécessaire d'ajouter un test pour voir si le buffer était vide. Si la procédure était appelée alors que le buffer était vide, un block était libéré alors qu'il n'y avait rien à libérer.

A.2 OpenOCD

A.2.1 RAM

Listing A.1 – Fichier de configuration de *OpenOCD* en RAM

```
#daemon configuration
telnet_port 4444
gdb_port 3333

#interface
interface ft2232
ft2232_device_desc "Amontec JTAGkey A"
ft2232_layout jtagkey
ft2232_vid_pid 0x0403 0xcff8
jtag_speed 50 # vitesse = clock / jtag_speed + 1

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe

jtag_nsrst_delay 333
jtag_ntrst_delay 333
```

```
#target configuration
daemon_startup reset

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
target arm7tdmi little run_and_halt 0 arm7tdmi-s_r4
run_and_halt_time 0 30

working_area 0 0x40000000 0x40000 nobackup

# For more information about the configuration files, take a look at:
# http://openfacts.berlios.de/index-en.phtml?title=Open+On-Chip+Debugger
```

A.2.2 FLASH

Principal

Listing A.2 – Fichier de configuration de *OpenOCD*en FLASH

```
#daemon configuration
telnet_port 4444
gdb_port 3333

#interface
interface ft2232
ft2232_device_desc "Amontec JTAGkey A"
ft2232_layout jtagkey
ft2232_vid_pid 0x0403 0xcff8
jtag_speed 10

#use combined on interfaces or targets that can't set TRST/SRST separately
reset_config trst_and_srst srst_pulls_trst

#jtag scan chain
#format L IRC IRCM IDCODE (Length, IR Capture, IR Capture Mask, IDCODE)
jtag_device 4 0x1 0xf 0xe
jtag_nsrst_delay 333
jtag_ntrst_delay 333

#target configuration
daemon_startup reset

#target <type> <startup mode>
#target arm7tdmi <reset mode> <chainpos> <endianness> <variant>
#pour flasher une application
target arm7tdmi little run_and_init 0 arm7tdmi-s_r4

run_and_halt_time 0 30


#flash configuration
target_script 0 reset c:\temp\temp.ocd #fichier de configuration pour flasher la
    carte
working_area 0 0x40000000 0x40000 nobackup

# flash bank lpc2000 <base> <size> 0 0 <variant> <target#> <clock> ['calc_checksum
    ']
# mthomas: LPC2138 @ 12MHz 0x7D000 from 500*1024 (not 512!)
# ysagon : LPC2212 @ 12MHz 0x1F400 from 125*1025
# <variant>, which may be 'lpc2000_v1' (all but 213x and 214x)
flash bank lpc2000 0x0 0x20000 0 0 lpc2000_v1 0 12000 calc_checksum
```

Programmation de la flash

Listing A.3 – Script de configuration pour flasher la *carte MESH*

```
##############################################################################
#                                                                            #
```

```
# Fichier de configuration pour flasher la carte MESH                    #
#                                                                        #
##########################################################################

wait_halt
armv4_5 core_state arm

#en cas de flash lock. Ne pas laisser en permanence!!!!
#mww 0xffffff64 0x5a000004
#mww 0xffffff64 0x5a002004

flash write 0 main.bin 0x0
mww 0xfffffd08 0xa5000401
reset
shutdown
```

A.3 Linker

A.3.1 Exécution en FLASH

Listing A.4 – Fichier de configuration pour le linker LD version flash

```
/* Copyright (c) 2006 MIS Institute of the HEIG affiliated to the University of
    Applied
** Sciences of Western Switzerland. All rights reserved.
** IN NO EVENT SHALL THE MIS INSTITUTE NOR THE HEIG NOR THE UNIVERSITY OF APPLIED
** SCIENCES OF WESTERN SWITZERLAND BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
    SPECIAL,
** INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND
     ITS
** DOCUMENTATION, EVEN IF THE MIS INSTITUTE OR THE HEIG OR THE UNIVERSITY OF
    APPLIED
** SCIENCES OF WESTERN SWITZERLAND HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
    DAMAGE.
** THE MIS INSTITUTE, THE HEIG AND THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN
    SWIT-
** ZERLAND SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
    THE
** IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE
    SOFT-
** WARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE MIS INSTITUTE NOR THE
    HEIG
** AND NOR THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN SWITZERLAND HAVE NO
    OBLIGATION
** TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
*/

ENTRY(ResetHandler)
SEARCH_DIR(.)

SVC_STACK_SIZE_H = 0x1000;
SVC_STACK_SIZE_L = 0x1000;
IRQ_STACK_SIZE = 0xc;


MEMORY
{
  flash : org = 0x00000000, len = 256K
  ram_ext : org = 0x82000000, len = 2M
}

SECTIONS
{
  .init :
  {
    *(.vectors);
    . = ALIGN(4);
```

```
    } > flash

    .code :
    {
      *(.text);
      . = ALIGN(4);
      *(.rodata);
      . = ALIGN(4);
      *(.rodata*);
      . = ALIGN(4);
      *(.glue_7t);
      . = ALIGN(4);
      *(.glue_7);
      . = ALIGN(4);
      _etext = .;
    } > flash

    .data :
    {
      _data = .;
      *(.data)
      . = ALIGN(4);
      _edata = .;
    } > ram_ext AT >flash

    .bss :
    {
      _bss_start = .;
      *(.bss)
      *(COMMON)
      . = ALIGN(4);
      _bss_end = .;
      _stack_svc_start = .;
      . += SVC_STACK_SIZE_H;
      . = ALIGN(4);
      _stack_svc_middle = .;
      . += SVC_STACK_SIZE_L;
      . = ALIGN(4);
      _stack_svc_end = .;
      _stack_irq_start = .;
      . += IRQ_STACK_SIZE;
      . = ALIGN(4);
      _stack_irq_end = .;
      end = .;
      _heap = .;
    } > ram_ext
    _eheap = ORIGIN(ram_ext) + LENGTH(ram_ext) - 4;
}
```

Listing A.5 – Fichier de configuration pour le linker LD version flash

```
/* Copyright (c) 2006 MIS Institute of the HEIG affiliated to the University of
    Applied
** Sciences of Western Switzerland. All rights reserved.
** IN NO EVENT SHALL THE MIS INSTITUTE NOR THE HEIG NOR THE UNIVERSITY OF APPLIED
** SCIENCES OF WESTERN SWITZERLAND BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
    SPECIAL,
** INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND
     ITS
** DOCUMENTATION, EVEN IF THE MIS INSTITUTE OR THE HEIG OR THE UNIVERSITY OF
    APPLIED
** SCIENCES OF WESTERN SWITZERLAND HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
    DAMAGE.
** THE MIS INSTITUTE, THE HEIG AND THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN
    SWIT-
** ZERLAND SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
    THE
```

```
** IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFT-
** WARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE MIS INSTITUTE NOR THE HEIG
** AND NOR THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN SWITZERLAND HAVE NO OBLIGATION
** TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
*/
ENTRY(ResetHandler)
SEARCH_DIR(.)

SVC_STACK_SIZE_H = 0x1000;
SVC_STACK_SIZE_L = 0x1000;
IRQ_STACK_SIZE = 0xc;


MEMORY
{
  flash : org = 0x00000000, len = 256K
  ram   : org = 0x40000000, len = 16k
}

SECTIONS
{
  .init :
  {
    *(.vectors);
    . = ALIGN(4);
  } > flash

  .code :
  {
    *(.text);
    . = ALIGN(4);
    *(.rodata);
    . = ALIGN(4);
    *(.rodata*);
    . = ALIGN(4);
    *(.glue_7t);
    . = ALIGN(4);
    *(.glue_7);
    . = ALIGN(4);
    _etext = .;
  } > flash

  .data :
  {
    _data = .;
    *(.data)
    . = ALIGN(4);
    _edata = .;
  } > ram AT >flash

  .bss :
  {
    _bss_start = .;
    *(.bss)
    *(COMMON)
    . = ALIGN(4);
    _bss_end = .;
    _stack_svc_start = .;
    . += SVC_STACK_SIZE_H;
    . = ALIGN(4);
    _stack_svc_middle = .;
    . += SVC_STACK_SIZE_L;
    . = ALIGN(4);
    _stack_svc_end = .;
```

```
      _stack_irq_start = .;
      . += IRQ_STACK_SIZE;
      . = ALIGN(4);
      _stack_irq_end = .;
      end = .;
      _heap = .;
    } > ram
    _eheap = ORIGIN(ram) + LENGTH(ram) - 4;
}
```

A.3.2 Exécution en RAM

Listing A.6 – Fichier de configuration pour le linker LD version ram externe

```
/* Copyright (c) 2006 MIS Institute of the HEIG affiliated to the University of Applied
** Sciences of Western Switzerland. All rights reserved.
** IN NO EVENT SHALL THE MIS INSTITUTE NOR THE HEIG NOR THE UNIVERSITY OF APPLIED
** SCIENCES OF WESTERN SWITZERLAND BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL,
** INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND ITS
** DOCUMENTATION, EVEN IF THE MIS INSTITUTE OR THE HEIG OR THE UNIVERSITY OF APPLIED
** SCIENCES OF WESTERN SWITZERLAND HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
** THE MIS INSTITUTE, THE HEIG AND THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN SWIT-
** ZERLAND SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
** IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE SOFT-
** WARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE MIS INSTITUTE NOR THE HEIG
** AND NOR THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN SWITZERLAND HAVE NO OBLIGATION
** TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
*/

ENTRY(ResetHandler)
SEARCH_DIR(.)

SVC_STACK_SIZE_H = 0x1000;
SVC_STACK_SIZE_L = 0x1000;
IRQ_STACK_SIZE = 0xc;


MEMORY
{
  ram : org = 0x40000000, len = 16k
  ram_ext : org = 0x82000000, len = 2M
}

SECTIONS
{
  .init :
  {
    *(.vectors);
    . = ALIGN(4);
  } > ram

  .code :
  {
    *(.text);
    . = ALIGN(4);
    *(.rodata);
    . = ALIGN(4);
    *(.rodata*);
```

```
        . = ALIGN(4);
        *(.glue_7t);
        . = ALIGN(4);
        *(.glue_7);
        . = ALIGN(4);
        _etext = .;
      } > ram_ext

      .data :
      {
        _data = .;
        *(.data)
        . = ALIGN(4);
        _edata = .;
      } > ram_ext

      .bss :
      {
        _bss_start = .;
        *(.bss)
        *(COMMON)
        . = ALIGN(4);
        _bss_end = .;
        _stack_svc_start = .;
        . += SVC_STACK_SIZE_H;
        . = ALIGN(4);
        _stack_svc_middle = .;
        . += SVC_STACK_SIZE_L;
        . = ALIGN(4);
        _stack_svc_end = .;
        _stack_irq_start = .;
        . += IRQ_STACK_SIZE;
        . = ALIGN(4);
        _stack_irq_end = .;
        end = .;
        _heap = .;
      } > ram_ext
      _eheap = ORIGIN(ram_ext) + LENGTH(ram_ext) - 4;
}
```

Listing A.7 – Fichier de configuration pour le linker LD version ram interne

```
/* Copyright (c) 2006 MIS Institute of the HEIG affiliated to the University of
    Applied
** Sciences of Western Switzerland. All rights reserved.
** IN NO EVENT SHALL THE MIS INSTITUTE NOR THE HEIG NOR THE UNIVERSITY OF APPLIED
** SCIENCES OF WESTERN SWITZERLAND BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT,
    SPECIAL,
** INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE AND
     ITS
** DOCUMENTATION, EVEN IF THE MIS INSTITUTE OR THE HEIG OR THE UNIVERSITY OF
    APPLIED
** SCIENCES OF WESTERN SWITZERLAND HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
    DAMAGE.
** THE MIS INSTITUTE, THE HEIG AND THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN
    SWIT-
** ZERLAND SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
    THE
** IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE
    SOFT-
** WARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND THE MIS INSTITUTE NOR THE
    HEIG
** AND NOR THE UNIVERSITY OF APPLIED SCIENCES OF WESTERN SWITZERLAND HAVE NO
    OBLIGATION
** TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.
*/

ENTRY(ResetHandler)
```

```
SEARCH_DIR(.)

SVC_STACK_SIZE_H = 0x1000;
SVC_STACK_SIZE_L = 0x1000;
IRQ_STACK_SIZE = 0xc;


MEMORY
{
  ram : org = 0x40000000, len = 16k
}

SECTIONS
{
  .init :
  {
    *(.vectors);
    . = ALIGN(4);
  } > ram

  .code :
  {
    *(.text);
    . = ALIGN(4);
    *(.rodata);
    . = ALIGN(4);
    *(.rodata*);
    . = ALIGN(4);
    *(.glue_7t);
    . = ALIGN(4);
    *(.glue_7);
    . = ALIGN(4);
    _etext = .;
  } > ram

  .data :
  {
    _data = .;
    *(.data)
    . = ALIGN(4);
    _edata = .;
  } > ram

  .bss :
  {
    _bss_start = .;
    *(.bss)
    *(COMMON)
    . = ALIGN(4);
    _bss_end = .;
    _stack_svc_start = .;
    . += SVC_STACK_SIZE_H;
    . = ALIGN(4);
    _stack_svc_middle = .;
    . += SVC_STACK_SIZE_L;
    . = ALIGN(4);
    _stack_svc_end = .;
    _stack_irq_start = .;
    . += IRQ_STACK_SIZE;
    . = ALIGN(4);
    _stack_irq_end = .;
    end = .;
    _heap = .;
  } > ram
  _eheap  = ORIGIN(ram) + LENGTH(ram) - 4;
}
```

A.4 Makefile

Afin de simplifer les différentes opérations de compilation, édition de liens, programmation etc, l'utilitaire `make` ([14]) est utilisé, conjointement avec un fichier de script nommé `Makefile`. Ce document n'a pas pour but d'expliquer en détail la syntaxe de `Makefile` mais d'en indiquer les principaux composants afin que l'utilisateur puisse adapter le fichier Makefile à ses besoins. Un fichier `Makefile` est constitué de plusieurs types d'instructions : commentaire : `# Source directory` variable : `SRCDIR = C :/workspace/` cible : `all : begin gccversion sizebefore build sizeafter finished end` règles :

```
{# Compile: create object files from C source files. ARM/Thumb
$(COBJ) : %.o : %.c
@echo
@echo $(MSG_COMPILING) $<
$(CC) -c $(THUMB) $(ALL_CFLAGS) $(CONLYFLAGS) $< -o $@
```

Les cibles sont les paramètres passés à l'utilitaire make. Eclipse permet de définir les cibles souhaitées. Toutes les cibles ne sont pas utilisées. En effet, beaucoup de cibles sont des cibles internes et ne doivent pas être appelées directement.

FIGURE A.1 – Cibles du projet testcomm

Listing A.8 – Makefile pour un projet utilisant YottaOS

```
# Hey Emacs, this is a -*- makefile -*-
#
# WinARM template makefile
# by Martin Thomas, Kaiserslautern, Germany
# <eversmith@heizung-thomas.de>
#
# Modified by Yann Sagon, Switzerland
# <yann.sagon@hesge.ch>
#
# based on the WinAVR makefile written by Eric B. Weddington, Jörg Wunsch, et al.
# Released to the Public Domain
# Please read the make user manual!
#
#
# On command line:
#
# make all = Make software.
#
# make clean = Clean out built project files.
#
# make program = Download the hex file to the device, using lpc21isp
#
# (TODO: make filename.s = Just compile filename.c into the assembler code only)
#
# To rebuild project do "make clean" then "make all".
#
# Changelog:
# - 17. Feb. 2005  - added thumb-interwork support (mth)
# - 28. Apr. 2005  - added C++ support (mth)
# - 29. Arp. 2005  - changed handling for lst-Filename (mth)
# - 08. Jun. 2007  - adapted for the MESH/RASMAS programm by ysagon
#

# MCU name and submodel
```

```
MCU = arm7tdmi-s
SUBMDL = lpc_2xxx
#THUMB    = -mthumb
THUMB =
#THUMB_IW = -mthumb-interwork
THUMB_IW =

# Yotta place
YOTTA = ../YottaOS/

# Esterel project place
ESTEREL = /cygdrive/c/esterel/modem/Default/Code/

## Create ROM-Image (final)
#RUN_MODE=ROM_RUN
## Create RAM-Image (debugging)
RUN_MODE=ROM_RUN


# Output format. (can be srec, ihex, binary)
FORMAT = binary


# Target file name (without extension).
TARGET = main


# List C source files here. (C dependencies are automatically generated.)
# use file-extension c for "c-only"-files
SRC =  modem_data.c functions.c $(YOTTA)YottaOS.c $(YOTTA)common.c $(TARGET).c \
                $(ESTEREL)modem.c

# List C source files here which must be compiled in ARM-Mode.
# use file-extension c for "c-only"-files
SRCARM =

# List C++ source files here.
# use file-extension cpp for C++-files (use extension .cpp)
CPPSRC =

# List C++ source files here which must be compiled in ARM-Mode.
# use file-extension cpp for C++-files (use extension .cpp)
#CPPSRCARM = $(TARGET).cpp
CPPSRCARM =

# List Assembler source files here.
# Make them always end in a capital .S.  Files ending in a lowercase .s
# will not be considered source files but generated files (assembler
# output from the compiler), and will be deleted upon "make clean"!
# Even though the DOS/Win* filesystem matches both .s and .S the same,
# it will preserve the spelling of the filenames, and gcc itself does
# care about how the name is spelled on its command-line.
ASRC =

# List Assembler source files here which must be assembled in ARM-Mode..
ASRCARM = $(YOTTA)YottaOS_a.S

# Optimization level, can be [0, 1, 2, 3, s].
# 0 = turn off optimization. s = optimize for size.
# (Note: 3 is not always the best optimization level. See avr-libc FAQ.)
OPT = 0
#OPT = 0

# Debugging format.
# Native formats for AVR-GCC's -g are stabs [default], or dwarf-2.
# AVR (extended) COFF requires stabs, plus an avr-objcopy run.
#DEBUG = stabs
```

```
DEBUG = dwarf-2

# List any extra directories to look for include files here.
#     Each directory must be seperated by a space.
#EXTRAINCDIRS = ./include
EXTRAINCDIRS = $(YOTTA) $(ESTEREL) ./

# Compiler flag to set the C Standard level.
# c89   - "ANSI" C
# gnu89 - c89 plus GCC extensions
# c99   - ISO C99 standard (not yet fully implemented)
# gnu99 - c99 plus GCC extensions
CSTANDARD = -std=gnu99

# Place -D or -U options for C here
CDEFS =  -D$(RUN_MODE) -DUART_IO

# Place -I options here
CINCS =

# Place -D or -U options for ASM here
ADEFS =  -D$(RUN_MODE) -D_GNU_ASSEMBLER_


# Compiler flags.
#  -g*:          generate debugging information
#  -O*:          optimization level
#  -f...:        tuning, see GCC manual and avr-libc documentation
#  -Wall...:     warning level
#  -Wa,...:      tell GCC to pass this to the assembler.
#    -adhlns...: create assembler listing
#
# Flags for C and C++ (arm-elf-gcc/arm-elf-g++)
CFLAGS = -g$(DEBUG)
CFLAGS += $(CDEFS) $(CINCS)
CFLAGS += -O$(OPT)
CFLAGS += -Wall -Wcast-align -Wcast-qual -Wimplicit
CFLAGS += -Wpointer-arith -Wswitch
CFLAGS += -Wredundant-decls -Wreturn-type -Wshadow -Wunused
CFLAGS += -Wa,-adhlns=$(subst $(suffix $<),.lst,$<)
CFLAGS += $(patsubst %,-I%,$(EXTRAINCDIRS))

# flags only for C
CONLYFLAGS  = -Wstrict-prototypes -Wmissing-declarations
CONLYFLAGS += -Wmissing-prototypes -Wnested-externs
CONLYFLAGS += $(CSTANDARD)

# flags only for C++ (arm-elf-g++)
# CPPFLAGS = -fno-rtti -fno-exceptions
CPPFLAGS =

# Assembler flags.
#  -Wa,...:   tell GCC to pass this to the assembler.
#  -ahlms:    create listing
#  -gstabs:   have the assembler create line number information; note that
#             for use in COFF files, additional information about filenames
#             and function names needs to be present in the assembler source
#             files -- see avr-libc docs [FIXME: not yet described there]
##ASFLAGS = -Wa,-adhlns=$(<:.S=.lst),-gstabs
ASFLAGS = $(ADEFS) -Wa,-adhlns=$(<:.S=.lst),-g$(DEBUG)

#Additional libraries.

#Support for newlibc-lpc (file: libnewlibc-lpc.a)
#NEWLIBLPC = -lnewlib-lpc

MATH_LIB = -lm
```

```
CPLUSPLUS_LIB = -lstdc++

# Linker flags.
#  -Wl,...:     tell GCC to pass this to linker.
#    -Map:      create map file
#    --cref:    add cross reference to  map file
LDFLAGS = -nostartfiles -Wl,-Map=$(TARGET).map,--cref
LDFLAGS += -lc
LDFLAGS += $(NEWLIBLPC) $(MATH_LIB)
LDFLAGS += -lc -lgcc
LDFLAGS += $(CPLUSPLUS_LIB)

# Set Linker-Script Depending On Selected Memory
ifeq ($(RUN_MODE),RAM_RUN)
LDFLAGS +=-T$(YOTTA)$(SUBMDL)_ram.ld
else
LDFLAGS +=-T$(YOTTA)$(SUBMDL)_flash.ld
endif

# ---------------------------------------------------------------------------

# Define directories, if needed.
## DIRARM = c:/WinARM/
## DIRARMBIN = $(DIRAVR)/bin/
## DIRAVRUTILS = $(DIRAVR)/utils/bin/

# Define programs and commands.
SHELL = sh
CC = arm-elf-gcc
CPP = arm-elf-g++
OBJCOPY = arm-elf-objcopy
OBJDUMP = arm-elf-objdump
SIZE = arm-elf-size
NM = arm-elf-nm
REMOVE = rm -f
COPY = cp
# specify OpenOCD executable (pp is for the wiggler, ftd2xx is for the USB debugger
    )
#OPENOCD = openocd-pp.exe
OPENOCD = openocd-ftd2xx.exe

# Define Messages
# English
MSG_ERRORS_NONE = Errors: none
MSG_BEGIN = -------- begin --------
MSG_END = --------  end  --------
MSG_SIZE_BEFORE = Size before:
MSG_SIZE_AFTER = Size after:
MSG_FLASH = Creating load file for Flash:
MSG_EXTENDED_LISTING = Creating Extended Listing:
MSG_SYMBOL_TABLE = Creating Symbol Table:
MSG_LINKING = Linking:
MSG_COMPILING = Compiling C:
MSG_COMPILING_ARM = "Compiling C (ARM-only):"
MSG_COMPILINGCPP = Compiling C++:
MSG_COMPILINGCPP_ARM = "Compiling C++ (ARM-only):"
MSG_ASSEMBLING = Assembling:
MSG_ASSEMBLING_ARM = "Assembling (ARM-only):"
MSG_CLEANING = Cleaning project:


# Define all object files.
COBJ      = $(SRC:.c=.o)
AOBJ      = $(ASRC:.S=.o)
COBJARM   = $(SRCARM:.c=.o)
```

```
AOBJARM   = $(ASRCARM:.S=.o)
CPPOBJ    = $(CPPSRC:.cpp=.o)
CPPOBJARM = $(CPPSRCARM:.cpp=.o)

# Define all listing files.
LST = $(ASRC:.S=.lst) $(ASRCARM:.S=.lst) $(SRC:.c=.lst) $(SRCARM:.c=.lst)
LST += $(CPPSRC:.cpp=.lst) $(CPPSRCARM:.cpp=.lst)

# Compiler flags to generate dependency files.
### GENDEPFLAGS = -Wp,-M,-MP,-MT,$(*F).o,-MF,.dep/$(@F).d
GENDEPFLAGS = -MD -MP -MF .dep/$(@F).d

# Combine all necessary flags and optional flags.
# Add target processor to flags.
ALL_CFLAGS = -mcpu=$(MCU) $(THUMB_IW) -I. $(CFLAGS) $(GENDEPFLAGS)
ALL_ASFLAGS = -mcpu=$(MCU) $(THUMB_IW) -I. -x assembler-with-cpp $(ASFLAGS)


# Default target.
all: begin gccversion sizebefore build sizeafter finished end

build: elf hex lss sym

elf: $(TARGET).elf
bin: $(TARGET).bin
hex: $(TARGET).hex
lss: $(TARGET).lss
sym: $(TARGET).sym

# Eye candy.
begin:
        @echo
        @echo $(MSG_BEGIN)

finished:
        @echo $(MSG_ERRORS_NONE)

end:
        @echo $(MSG_END)
        @echo


# Display size of file.
HEXSIZE = $(SIZE) --target=$(FORMAT) $(TARGET).hex
ELFSIZE = $(SIZE) -A $(TARGET).elf
sizebefore:
        @if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_BEFORE); $(ELFSIZE);
            echo; fi

sizeafter:
        @if [ -f $(TARGET).elf ]; then echo; echo $(MSG_SIZE_AFTER); $(ELFSIZE);
            echo; fi


# Display compiler version information.
gccversion :
        @$(CC) --version


# Program the device.

# specify OpenOCD configuration file (pick the one for your device)
OPENOCD_CFG = 'C:\config_openocd\openocd_flash.cfg'
OPENOCD_CFG = 'C:\carte_mesh\dev\config_openocd\openocd_flash.cfg'

program: $(TARGET).bin
        @echo "Flash Programming with OpenOCD..."
```

```
        $(OPENOCD) -f $(OPENOCD_CFG)
        @echo "Flash Programming Finished."

303
# Create final output files (.hex, .eep) from ELF output file.
# TODO: handling the .eeprom-section should be redundant
%.hex: %.elf
        @echo
308     @echo $(MSG_FLASH) $@
        $(OBJCOPY) -O $(FORMAT) $< $@


# Create final output files (.hex, .eep) from ELF output file.
313 # TODO: handling the .eeprom-section should be redundant
%.bin: %.elf
        @echo
        @echo $(MSG_FLASH) $@
        $(OBJCOPY) -O $(FORMAT) $< $@
318

# Create extended listing file from ELF output file.
# testing: option -C
%.lss: %.elf
323     @echo
        @echo $(MSG_EXTENDED_LISTING) $@
        $(OBJDUMP) -h -S -C $< > $@


328 # Create a symbol table from ELF output file.
%.sym: %.elf
        @echo
        @echo $(MSG_SYMBOL_TABLE) $@
        $(NM) -n $< > $@
333

# Link: create ELF output file from object files.
.SECONDARY : $(TARGET).elf
.PRECIOUS : $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(CPPOBJ) $(CPPOBJARM)
338 %.elf:  $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(CPPOBJ) $(CPPOBJARM)
        @echo
        @echo $(MSG_LINKING) $@
        $(CC) $(THUMB) $(ALL_CFLAGS) $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(CPPOBJ
            ) $(CPPOBJARM) --output $@ $(LDFLAGS)
#       $(CPP) $(THUMB) $(ALL_CFLAGS) $(AOBJARM) $(AOBJ) $(COBJARM) $(COBJ) $(
    CPPOBJ) $(CPPOBJARM) --output $@ $(LDFLAGS)
343
# Compile: create object files from C source files. ARM/Thumb
$(COBJ) : %.o : %.c
        @echo
        @echo $(MSG_COMPILING) $<
348     $(CC) -c $(THUMB) $(ALL_CFLAGS) $(CONLYFLAGS) $< -o $@

# Compile: create object files from C source files. ARM-only
$(COBJARM) : %.o : %.c
        @echo
353     @echo $(MSG_COMPILING_ARM) $<
        $(CC) -c $(ALL_CFLAGS) $(CONLYFLAGS) $< -o $@

# Compile: create object files from C++ source files. ARM/Thumb
$(CPPOBJ) : %.o : %.cpp
358     @echo
        @echo $(MSG_COMPILINGCPP) $<
        $(CPP) -c $(THUMB) $(ALL_CFLAGS) $(CPPFLAGS) $< -o $@

# Compile: create object files from C++ source files. ARM-only
363 $(CPPOBJARM) : %.o : %.cpp
        @echo
```

```
        @echo $(MSG_COMPILINGCPP_ARM) $<
        $(CPP) -c $(ALL_CFLAGS) $(CPPFLAGS) $< -o $@

368
    # Compile: create assembler files from C source files. ARM/Thumb
    ## does not work - TODO - hints welcome
    ##$(COBJ) : %.s : %.c
    ##      $(CC) $(THUMB) -S $(ALL_CFLAGS) $< -o $@
373

    # Assemble: create object files from assembler source files. ARM/Thumb
    $(AOBJ) : %.o : %.S
        @echo
378     @echo $(MSG_ASSEMBLING) $<
        $(CC) -c $(THUMB) $(ALL_ASFLAGS) $< -o $@


    # Assemble: create object files from assembler source files. ARM-only
383 $(AOBJARM) : %.o : %.S
        @echo
        @echo $(MSG_ASSEMBLING_ARM) $<
        $(CC) -c $(ALL_ASFLAGS) $< -o $@

388
    # Target: clean project.
    clean: begin clean_list finished end


393 clean_list :
        @echo
        @echo $(MSG_CLEANING)
        $(REMOVE) $(TARGET).hex
        $(REMOVE) $(TARGET).bin
398     $(REMOVE) $(TARGET).obj
        $(REMOVE) $(TARGET).elf
        $(REMOVE) $(TARGET).map
        $(REMOVE) $(TARGET).obj
        $(REMOVE) $(TARGET).a90
403     $(REMOVE) $(TARGET).sym
        $(REMOVE) $(TARGET).lnk
        $(REMOVE) $(TARGET).lss
        $(REMOVE) $(COBJ)
        $(REMOVE) $(CPPOBJ)
408     $(REMOVE) $(AOBJ)
        $(REMOVE) $(COBJARM)
        $(REMOVE) $(CPPOBJARM)
        $(REMOVE) $(AOBJARM)
        $(REMOVE) $(LST)
413     $(REMOVE) $(SRC:.c=.s)
        $(REMOVE) $(SRC:.c=.d)
        $(REMOVE) $(SRCARM:.c=.s)
        $(REMOVE) $(SRCARM:.c=.d)
        $(REMOVE) $(CPPSRC:.cpp=.s)
418     $(REMOVE) $(CPPSRC:.cpp=.d)
        $(REMOVE) $(CPPSRCARM:.cpp=.s)
        $(REMOVE) $(CPPSRCARM:.cpp=.d)
        $(REMOVE) .dep/*

423
    # Include the dependency files.
    -include $(shell mkdir .dep 2>/dev/null) $(wildcard .dep/*)


428 # Listing of phony targets.
    .PHONY : all begin finish end sizebefore sizeafter gccversion \
    build elf hex lss sym clean clean_list program
```

A.5 GDB

Listing A.9 – Fichier d'initialisation lpc21xx pour GDB

```
#
# Project:       RASMAS/MESH
# File:          init.gdb
# Date:          10 mar. 2007
# Author:        Yann Sagon <yann.sagon@hesge.ch>
# Description:   Fichier d'initialiation de la cible MESH pour GDB
# Usage:         depuis GDB: source init.gdb
# Modifications: 11 jul. 2007 Ajout des commentaires
#

# connection a openocd
target remote localhost:3333

# reset de la cible
monitor reset

# attente de une seconde
monitor sleep 1000

# attente active jusqu'a ce que la cible soit prete
monitor poll

# soft reset de la cible puis halt
monitor soft_reset_halt

# activation des breakpoints soft
monitor arm7_9 sw_bkpts enable
monitor sleep 1000

# Activation de la memoire externe (LPC22xx uniquement)
# Registre BCFG0 (flash)
monitor mww 0xFFE00000 0x10001463 # 0x8000 0000
# pire cas: 0x1000FBEF
# Registre BCFG2 (ram ext 16bit)
monitor mww 0xFFE00008 0x10001463 # 0x8200 0000
# Registre PINSEL2
monitor mww 0xE002C014 0x0F814114

# ecriture dans le registre MEMMAP afin mapper les vecteurs d'interuptions
# 0x1 - Vecteurs mappes en flash interne (0x0)
# 0x2 - Vecteurs mappes en ram interne   (0x40000000)
# 0x3 - Vecteurs mappes en ram externe   (0x82000000)
monitor mww 0xE01FC040 0x2
# deuxieme ecriture, meme valeur, pour etre sur ok
monitor mww 0xE01FC040 0x2
monitor sleep 1000
# lecture de MEMMAP pour verifications (doit retourner la valeur specifiee ci-
    dessus)
monitor mdw 0xE01FC040
monitor mdw 0xFFE00000
monitor mdw 0xFFE00008
```

Annexe B

Elements obsolets

B.1 Préambule

Les éléments ci-dessous sont laissés dans la documentation à titre de référence même si ils ne sont plus tous d'actualité.

B.2 Interconnexions entre différents appareils

La figure B.1 indique comment interconnecter les différents appareils. Légende de l'image :
- GUI : application Qt permettant de communiquer avec les cartes MESH ;
- PC XP : un ordinateur fonctionnant avec Windows XP ;
- PC Linux : un ordinateur fonctionnant avec Kubuntu 7.10 ;
- G MESH : gateway MESH ;
- C MESH : carte MESH ;
- Tunnel SSH : tunnel SSH entre les deux ordinateurs ;
- SLIP : connexion par protocole SLIP ;
- RF : connexion par fréquence radio.

Le tunnel SSH est un tunnel ouvert entre PC XP et PC Linux pour permettre de dialoguer avec les cartes MESH sans installer le procotole SLIP sur les différents ordinateurs fonctionnant avec Windows XP.

B.3 Protocoles de transmissions

B.3.1 Paquet TCP/IP

Le MTU d'un paquet IP est fixé à 100 octets à cause du module YLinx. De ces 100 octets, il faut enlever 20 octets pour l'overhead du protocole IP et 20 octets pour l'overhead du protocole TCP. Il ne reste donc plus que 60 octets à disposition pour transmettre des données.

Le protocole comprend, dans l'ordre :
- id-app : 1 octet : numéro de l'application destinataire/émettrice ;
- length : 1 octet : longueur du champ data ;
- fragmentation : 2 octets : numéro de fragmentation du paquet ;
- timestamp : 4 octets : temps auquel a été envoyé le paquet ;
- data : 52 octets : données.

Le champ length ne contient que la longueur du champ data, car l'overhead est fixe (8 octets). Il n'y a donc pas besoin de le compter à chaque fois. Le champ fragmentation permet d'indiquer si le paquet est un paquet fragmenté (>= 1) ou non (0). Ce champ a été ajouté pour éviter de passer par la fragmentation qu'effectue le protocole IP. Il est de cette manière plus simple de savoir si un paquet a été fractionné et de le traiter. Il faut savoir avant d'envoyer le premier paquet si

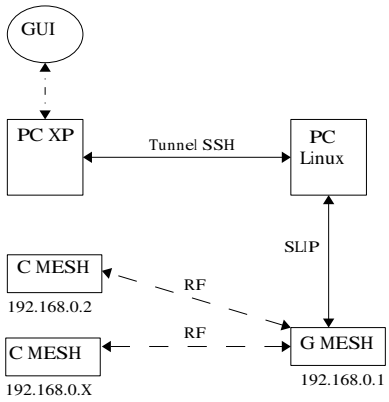

FIGURE B.1 – Interconnexions entre les différents appareils

les données seront fractionnées. En cas de fractionnement, le premier paquet devra contenir la valeur 1 dans le champ fragmentation. Cette valeur sera ensuite incrémentée pour chaque nouveau paquet fragmenté. Si les données ne seront pas fractionnées, la valeur du champ fragmentation doit contenir la valeur 0. Le champ id-app contient le numéro de l'application destinataire si le paquet est envoyé de l'application de l'ordinateur ou le numéro de l'application émettrice si le paquet est envoyé de l'application du module. Ce champ permet de savoir si un paquet a été fractionné et de le traiter. Il faut savoir avant d'envoyer le premier paquet si les données seront fractionnées. En cas de fractionnement, le premier paquet devra contenir la valeur 1 dans le champ fragmentation. Cette valeur sera ensuite incrémentée pour chaque nouveau paquet fragmenté. Si les données ne seront pas fractionnées, la valeur du champ fragmentation doit contenir la valeur 0. Le champ id-app contient le numéro de l'application destinataire si le paquet est envoyé de l'application de l'ordinateur ou le numéro de l'application émetrice si le paquet est envoyé de l'application du module. Le champ timestamp contient le temps du module (d'après la fonction getTime() se trouvant dans common/functions.h) ou un nombre incrémente à chaque envoi si le paquet est émis de l'application sur l'ordinateur. Le champ data contient les données efficace émises. Sa taille de 52 octets peut paramètre petite, mais en tenant compte du champ fragmentation la taille des données contenues dans un "paquet" (au sens large) devient largement suffisante.

La structure du protocole est définie comme ci-dessous en C.

```
struct mesh_msg
{
        UINT8  id_app;
        UINT8  length;
        UINT16 fragmentation;
        UINT32 timestamp;
        UINT8  data[MESH_PROTOCOL_FIELD_DATA];
};
```

Plusieurs constantes existent pour définir la taille des différents champs. Ces constantes se trouvent dans le fichier `uip/apps/mesh/meshdeftype.h`.

B.3.2 Commandes

Avant propos

Les commandes envoyées de l'application de l'ordinateur vers l'application du module suivent également un protocole.

ECG

Deux commandes sont disponibles :
- TEST : commande de test ;
- SEND : demande d'envoi des enregistrements mémorisés ;

La commande "TEST" renvoie le texte "REPONSE ECG TEST" si la tâche est capable de répondre.

La commande "SEND$NOOO$" renvoie les enregistrements mémorisés. Le champ N (entier non signé sur 8 bit) est utilisé pour préciser le nombre d'enregistrements à envoyer. Le champ OOO (entier non signé sur 24 bit) est utilisé pour préciser un offset. L'entier du champ N ne permet de demander que les 256 derniers enregistrements. L'offset est utilisé si d'autres enregistrements sont nécessaire.

Vélo

Trois commandes sont disponibles :
- TEST : commande de test ;
- SEND : demande d'envoi des enregistrements mémorisés ;
- CMDB : pilote le frein électromagnétique.

La commande ”TEST” renvoie le texte ”REPONSE VELO TEST” si la tâche est capable de répondre.

La commande ”SEND*NOOO*” renvoie les enregistrements mémorisés. Le champ N (entier non signé sur 8 bit) est utilisé pour préciser le nombre d'enregistrements à envoyer. Le champ OOO (entier non signé sur 24 bit) est utilisé pour préciser un offset. L'entier du champ N ne permet de demander que les 256 derniers enregistrements. L'offset est utilisé si d'autres enregistrements sont nécessaire.

La commande ”CMDB*PMN*” pilote le frein électromagnétique du vélo. Le champ P (entier non signé de 8 bit) indique le pourcentage de freinage ; sa valeur doit être dans l'interval `0x00` et `0x64`. Le champ M (entier non signé de 8 bit) indique la tension maximale du frein ; sa valeur doit être dans l'interval `0x00` et `0x0A`. Le champ N indique le numéro du vélo ; sa valeur doit être dans l'interval `0x00` et `0x03`.