REIJASSE Eric
QI Wenbo
KHARBOUSH Karim

Filière Electronique 3ème année
Option Systèmes Embarqués

Rapport du projet avancé SE de troisième année

Internet des objets : Réseau de capteurs basé sur Raspberry Pi et le protocole Zigbee

Encadrant : Patrice KADIONIK

Année 2016-2017

Sommaire

Introduction

Le but de notre projet était de réaliser un réseau de capteurs, c'est-à-dire un ensemble de capteurs interconnectés par un réseau de communication. Les capteurs utilisés étaient des capteurs de température et d'humidité, et les diverses données étaient traitées à l'aide de cartes Raspberry Pi et Arduino connectées via un réseau Zigbee.

Après une description du cahier des charges, dans ce rapport sont présentées les différentes étapes de réalisation du projet : d'abord l'installation d'un système d'exploitation Linux embarqué sur les cartes Raspberry Pi et en utilisant l'outil Buildroot, puis la configuration de la liaison UART, l'ajout de package et la génération de l'image sur carte microSD, et après cela une manière simplifiée de recompiler les packets sur la carte à l'aide de la compilation croisée. Ensuite sont décrits la configuration et l'utilisation du réseau Zigbee, l'application que nous avons développé sur Raspberry Pi pour récolter les données des capteurs, la programmation d'une carte Arduino pour acquérir les données d'un autre capteur, et enfin la configuration finale des Raspberry Pi pour démarrer automatiquement les programmes d'acquisition et de transmission.

1- Présentation du cahier des charges

Le projet consiste à réaliser des prototypes de IoT (Internet of Things) en utilisant des cartes Raspberry Pi, de divers capteurs et des modules XBee. Un capteur de température DS1624 mesure la température ambiante, et envoie les données lues à une carte Raspberry Pi 3 via la liaison I2C. La carte Raspberry Pi 3 va ensuite collecter et traiter les données reçues, et les envoyer à travers un module XBee sur le réseau ZigBee. Un autre capteur de température et d'humidité DHT11 est lu par une carte Arduino, et un deuxième module XBee s'occupe d'envoyer ces informations sur le même réseau ZigBee. Finalement, une carte Raspberry Pi B+ joue le rôle de coordinateur, elle synchronise la transmission de données du réseau ZigBee, collecte les données envoyées, et les envoie à un PC via la liaison série. Le rôle du PC peut éventuellement être remplacé par la carte Raspberry Pi B+, puisqu'elle est dotée d'un système Linux embarqué qui permet de réaliser toutes les tâches du PC. Le schéma ci-dessous illustre notre système:

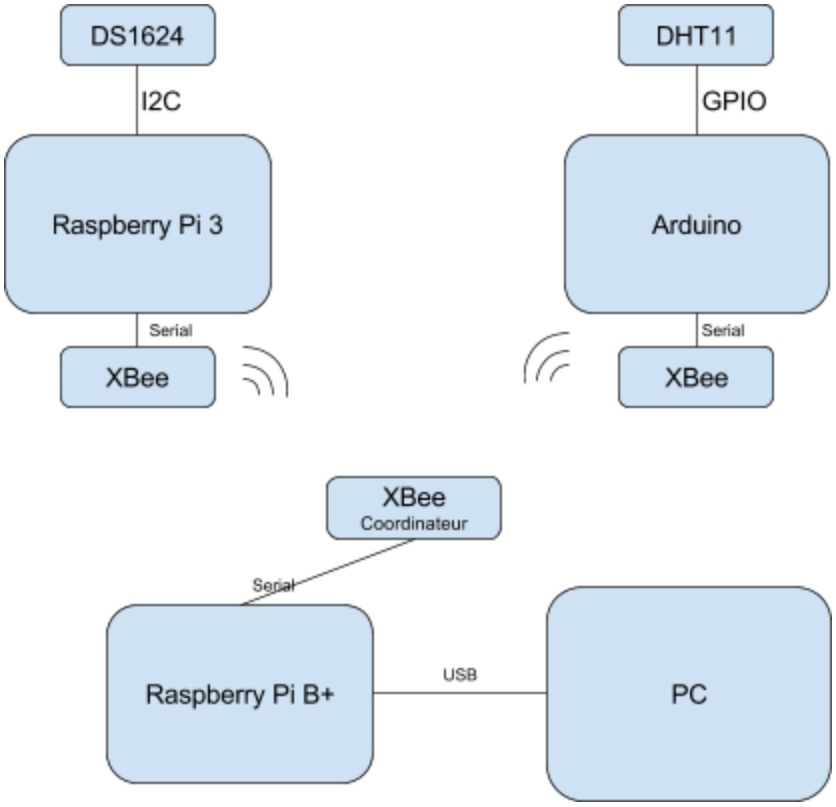

Figure 1 - Schéma global du système de réseau de capteurs

Durant ce projet, nous avons mis en oeuvre une distribution embarquée de Linux sur des cartes Raspberry Pi, ainsi que les programmes en C qui permettent la communication avec les périphériques. Nous avons également configuré les modules XBee pour établir notre réseau ZigBee.

2- La carte Raspberry Pi

Dans ce projet ont été utilisées deux cartes Raspberry Pi : le modèle de dernière génération étant la Raspberry Pi 3 Model B, et la Raspberry Pi B+ qui est une version améliorée de la Raspberry Pi de première génération. La Raspberry Pi peut être décrite comme un ordinateur de la taille d'une carte électronique embarquée. La Raspberry Pi est un projet de la Raspberry Foundation, dont l'objectif est de faciliter l'apprentissage de l'électronique, de la microinformatique et la programmation. Pour ce faire la Raspberry Pi est également accessible à bas coût : le dernier modèle (Pi 3) peut en effet être acheté à un prix de 33$.

Au niveau des caractéristiques techniques, pour la Raspberry Pi 3 Model B on retrouve un processeur ARMv8 quatre coeurs basse consommation cadencés à 1.2GHz, 1 Giga-octet de mémoire vive RAM (Random Access Memory), quatre ports USB et 40 broches entrée-sortie, un slot micro-SD, du Wifi 802.11n, le Bluetooth 4.1, et même un port Ethernet. Pour la vidéo et

l'audio, la Raspberry Pi dispose également d'un port Jack audio 3.5mm, d'une sortie HDMI, d'une interface d'affichage (DSI), d'une interface caméra (CSI), et d'un processeur graphique VideoCore IV 3D.

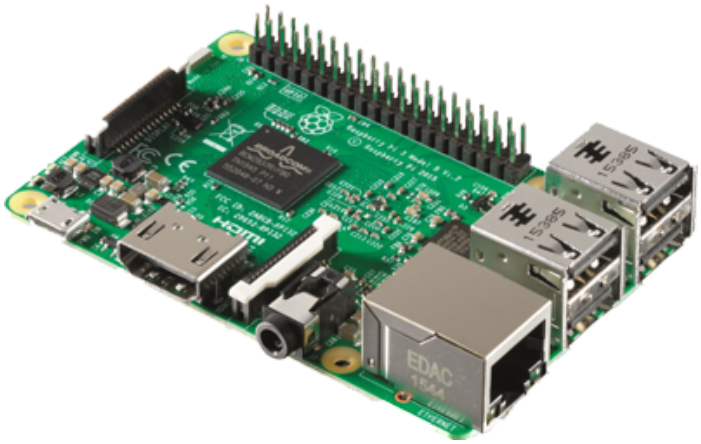

Figure 2 - La carte Raspberry Pi

La carte Raspberry Pi est ainsi un véritable ordinateur offrant de nombreuses possibilités pour le développement et la réalisation de projets en électronique et informatique. Grâce au slot micro-SD intégré, on peut installer un système d'exploitation complet comme Linux sur une carte micro-SD, ce qui permet de faciliter l'utilisation des ressources matérielles de la carte et une implémentation intégrée des fonctionnalités de base.

3- Linux pour l'embarqué

Embedded Linux est l'utilisation du noyau Linux (kernel) et des différents composants libres dans un système embarqué. Depuis 2000, Linux est devenu de plus en plus populaire sur les systèmes embarqués. Aujourd'hui, on trouve du Linux embarqué sur beaucoup des appareils électroniques comme les smartphones, tablettes, routeurs et objets connectés. Un exemple majeur de Linux embarqué est l'OS mobile Android qui est développé par Google.

L'avantage essentiel de Linux et du logiciel libre en systèmes embarqués est la possibilité de réutiliser (re-use) des programmes et des briquent qui existent. Cet avantage permet de développer des produits basés sur des briques existantes. Il existe aussi d'autres avantages comme : le faible coût, le contrôle total sur la partie software du système et la haute qualité des différentes briques et des différents programmes.

La figure suivante représente l'architecture d'un système Linux Embarqué. Les briques logiciels de ce système sont :

- Une chaîne de compilation croisée : un compilateur qui est sur le PC de développement mais qui compile le code et génère l'exécutable pour la carte cible
- Bootloader : responsable de l'initialisation, du chargement et de l'exécution du noyau Linux.

- Noyau Linux : responsable de la gestion de la mémoire et des processus, et contient les pilotes des périphériques, la pile réseau et fournit des services à l'espace utilisateur.
- Bibliothèque standard du C : l'interface entre le noyau Linux et l'espace utilisateur.
- Applications et Librairies.

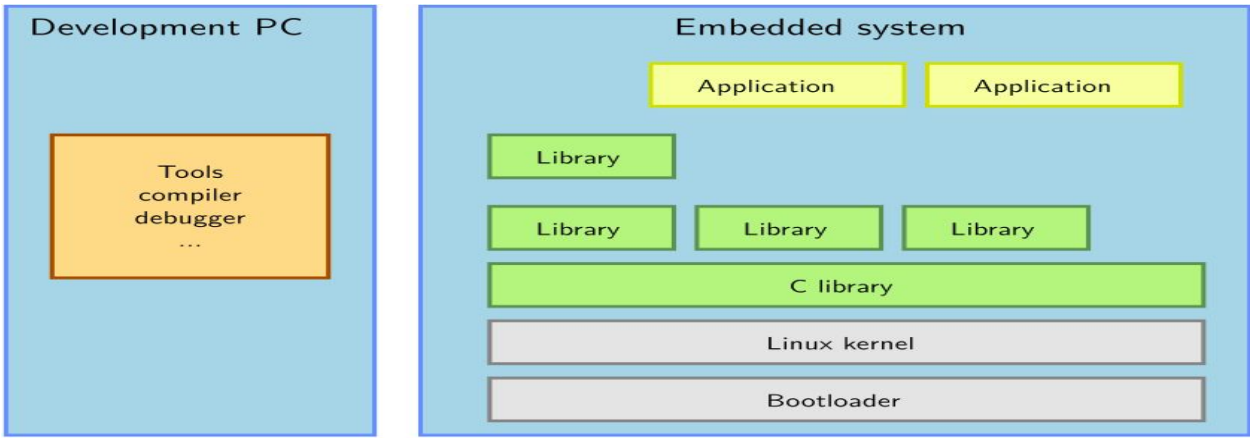

Figure 3 - Architecture d'un système Linux embarqué

4- L'outil Buildroot pour construire un Linux embarqué

Dans notre projet, nous avons utilisé l'outil Buildroot. Ce dernier est un outil qui simplifie et automatise le processus de construction d'un système Linux embarqué. Pour cela, Buildroot est capable de générer une chaîne de compilation croisée, un root filesystem (RFS), l'image du noyau Linux et le bootloader pour la carte cible. La figure suivante montre le schéma de fonctionnement de Buildroot :

Figure 4 - Schéma de fonctionnement de l'outil Buildroot

Les données d'entrée sont les différentes sources des composants qui constituent la distribution comme le noyau Linux et BusyBox ou les différentes applications développées par le concepteur. Les sorties du système sont le Bootloader, le noyau et le root filesystem.
Buildroot suit les différentes étapes suivantes pour générer les sorties :

1. la création dans *output/host* de la chaîne de compilation croisée pour l'architecture cible.

2. la création dans *output/target* du squelette de la distribution à partir du squelette initial fourni par Buildroot.
3. la compilation des différents composants et applications.
4. la création de l'image binaire du système dans *output/images*.

Pour construire le système, il faut d'abord installer Buildroot depuis le site internet (https://buildroot.org/download.html) . Ensuite, pour configurer Buildroot pour la carte cible Raspberry Pi 3, il suffit d'écrire cette commande :

make raspberrypi3_defconfig

Puis on écrit la commande *make* pour construire le RFS et on recopie l'image sur la carte SD avec la commande suivante :

sudo dd if=output/images/sdcard.img of=/dev/sdX

5- La liaison UART entre la Raspberry Pi et le PC

Dans le cadre de notre projet la carte Raspberry Pi était branchée en liaison série avec le PC. Or, la liaison entre le PC et la Raspberry s'établissant traditionnellement en USB et la Raspberry ne disposant pas de port RS-232, il a fallu utiliser un adaptateur USB/liaison série, comportant d'un côté le port USB à brancher au PC, et de l'autre côté trois fils (orange, jaune et noir) à relier adéquatement au broches de la Raspberry. Ci-dessous un schéma représentant les broches de la carte :

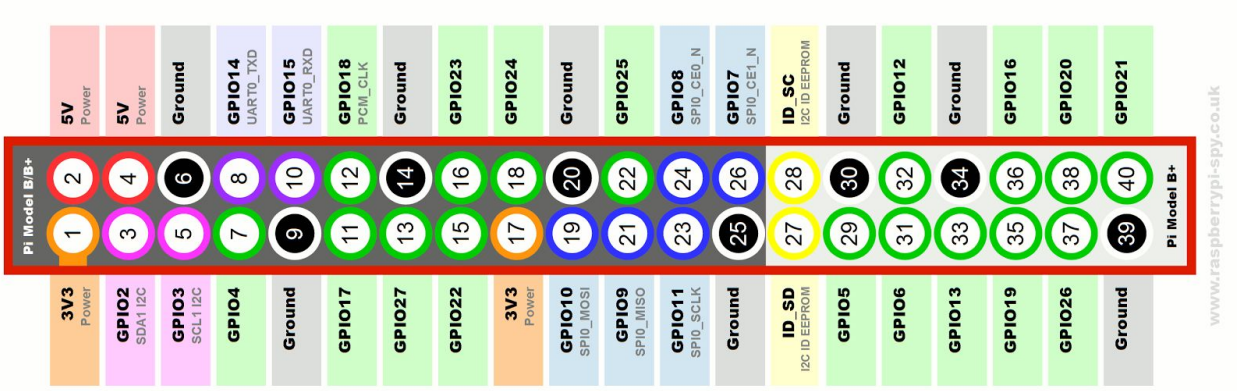

Figure 5 - Les broches de la Raspberry Pi 3 - *source : raspberrypi-spy.co.uk*

Ainsi le fil noir était connecté à la masse (broche 6), le fil jaune à la broche de transmission Tx (broche 8), et le fil orange à la broche de réception Rx (broche 10).

Pour que la carte Raspberry puisse communiquer avec l'ordinateur par l'outil minicom, il faut activer la liaison UART. Pour cela, dans le cas d'une carte Raspberry Pi B+, il suffit d'ajouter au fichier *config.txt* la ligne:

enable_uart=1
Dans le cas d'une carte Raspberry Pi 3, la ligne à ajouter est:
dtoverlay=pi3-miniuart-bt

L'application minicom peut ensuite être ouverte à l'aide de la commande :
minicom -b 115200 -o -D /dev/ttyUSB0
avec :

- l'option -b précisant le vitesse de transmission sur la liaison série, ici 115200 baud/s
- l'option -D précisant le périphérique utilisé, ici */dev/ttyUSB0*

Pour éviter à devoir spécifier les paramètres à chaque fois, on peut aussi configurer minicom avec *sudo minicom -s*. Ainsi une fenêtre de dialogue pour minicom s'ouvre et on sélectionne l'option "configuration du port série". On définit ensuite le nom du fichier pour le port série (touche A), ici /dev/ttyUSB0, et on détermine le débit et le format des données (touche E), ici on conserve les paramètres par défauts avec un débit de 115200 bits/s et la configuration 8N1 pour les données (données sur 8 bits, N pour bits impairs, et 1 bit de stop).
On appuie ensuite sur les touches CTRL-A + Z puis sur Q pour sortir de l'application et conserver les paramètres.

6- Les ajouts de package

L'ajout de fonctionnalités à la carte Raspberry Pi (en l'occurrence ici les fonctionnalités pour l'acquisition de la température et l'utilisation de Zigbee) passe par l'ajout de package au Buildroot. Pour ce faire il faut d'abord modifier le fichier *Config.in* situé à la racine de Buildroot. On lui ajoute une entrée "BR2_EXTERNAL" qui contiendra les différents package ajoutés, avec les lignes :

```
config BR2_EXTERNAL
   string
   option env="BR2_EXTERNAL"
```

Puis dans le menu "User-provided options" du fichier, on spécifie les différents package ajoutés avec des instructions du type

```
source "$BR2_EXTERNAL/package2/<nom_repertoire_package>/Config.in"
```

On a choisi ici de placer le tout dans un dossier nommé *package2*.

Outre les modifications dans ce fichier *Config.in* on ajoute également à la racine un fichier *external.mk*. Dans ce fichier on précise l'emplacement de tous les fichiers .mk utilisés pour l'ajout des package à l'aide de l'instruction :

```
include $(sort $(wildcard $(BR2_EXTERNAL)/package2/*/*.mk))
```

Ensuite pour chaque package on ajoute un répertoire *<nom_package>/* dans le dossier */buildroot/package2/* contenant pour commencer deux fichiers :

- Un nouveau fichier *Config.in* spécifique à ce package
- Un fichier *<nom_package>.mk* contenant les instructions de compilation propres au package.

A ces deux fichiers est également ajouté dans le répertoire un sous-dossier contenant deux autres fichiers :

- Le fichier source <nom_package>.c contenant le code source du package associé.
- Un fichier Makefile pour la compilation de ce code source.

Ce dossier est ensuite compressé au format tar.gz, et l'archive en question (*<nom_package>.tar.gz*) est placée dans le même répertoire */buildroot/package2/<nom_package>/* , qui contient donc en tout un fichier *Config.in*, un fichier *.mk*, un dossier et une archive *.tar.gz*.

Lors de la compilation avec Buildroot, le code source est téléchargé dans le dossier *buildroot/dl*, puis copié et compilé sous le répertoire *buildroot/output/build* . Enfin, l'exécutable est généré sous *buildroot/output/target/usr/bin*.

7- Génération et installation d'image sur carte SD

Après avoir ajouté les package adéquats, on peut générer l'image et l'installer sur la carte SD de la Raspberry Pi. Pour ce faire, on configure d'abord Buildroot pour le modèle de la carte sur laquelle on travaille, en l'occurrence ici la Raspberry Pi 3 avec la commande *make raspberrypi3_defconfig*, puis on compile le tout avec *make*. Une fois la compilation terminée (ce qui prend pas mal de temps), on peut observer l'arborescence suivante dans le répertoire *output/images* :

```
output/images/
+-- bcm2710-rpi-3-b.dtb
+-- boot.vfat
+-- kernel-marked/zImage
+-- rootfs.ext4
+-- rpi-firmware/
|   +-- bootcode.bin
|   +-- cmdline.txt
|   +-- config.txt
|   +-- fixup.dat
|   +-- start.elf
|   `-- overlays/
+-- sdcard.img
`-- zImage
```

Figure 6 - Arborescence du répertoire output/images

Il nous reste donc à copier l'image *sdcard.img* sur la carte SD : après avoir inséré la carte SD directement sur l'ordinateur (avec un adaptateur s'il le faut) on transfère l'image avec la commande *sudo dd if=output/images/sdcard.img of=/dev/sdb*, l'option *if* spécifiant l'image à transférer et l'option *of* la partition sur laquelle installer l'image. Enfin, la carte SD est insérée dans la Raspberry Pi, celle-ci est démarrée et on peut voir l'affichage d'un terminal sur le port série.

8- La compilation croisée d'applications

Nous avons trouvé au cours de notre développement que la modification de code source des applications par Buildroot pose parfois des problèmes, et cela présente une efficacité médiocre. Pour cela, nous avons finalement utilisé la méthode de compilation croisée. En effet, pour que l'exécutable soit utilisable sur la carte Raspberry Pi, il faut compiler le code source du programme par le compilateur croisé, qui se trouve dans l'outil Buildroot. Ce compilateur est le compilateur gcc: *arm-buildroot-linux-uclibcgnueabihf-gcc*. On doit ensuite indiquer le chemin de ce compilateur en modifiant *"PATH"* dans le fichier des variables d'environnement *.profile*.
Dans le fichier Makefile du programme, il suffit d'ajouter : *CC=arm-buildroot-linux-uclibcgnueabihf-gcc* pour pouvoir utiliser le compilateur croisé après la modification de *PATH*. L'exécutable généré après la compilation par ce compilateur peut être copié sur la carte SD de la Raspberry Pi et être exécuté directement.
Cette méthode de compilation croisée nous a permis une efficacité de développement et de débuggage bien meilleure qu'en utilisant Buildroot.

9- Le réseau ZigBee

L'essentiel de ce projet de prototypage IoT consiste bien sûr en la connectivité des objets : pour cela, nous avons construit un réseau local ZigBee à travers des modules XBee. Par la suite, nous allons introduire d'abord le protocole ZigBee, et ensuite nous allons présenter la configuration de notre réseau ZigBee.
Le protocole ZigBee est basé sur le standard IEEE 802.15.4 pour les réseaux d'utilisation personnelle (Wireless Personal Area Networks : WPANs). Selon le modèle OSI, il s'appuie sur la norme IEEE 802.15.4 pour les niveaux physique et MAC (Medium ACcess). Il présente une portée de petite à moyenne distance. Les modules XBee standards ont une portée de transmission jusqu'à 100 mètres, quant aux modules XBee que l'on utilise, cette portée peut atteindre 1000 mètres. Ses principaux avantages sont son faible coût, sa faible consommation, sa fiabilité importante et sa simplicité de configuration. Grâce à ses avantages, il peut être utilisé dans les milieux tels que la surveillance de locaux pour la détection de départ de feu, la surveillance de bâtiments contre les intrusions et la Gestion Technique de Bâtiment (GTB).

Au niveau du réseau, de différentes topologies peuvent être mises en oeuvre à travers le protocole ZigBee, tels que point à point, en étoile, ou en maillé. Dans le cas de notre projet, nous avons utilisé une topologie de type étoile.

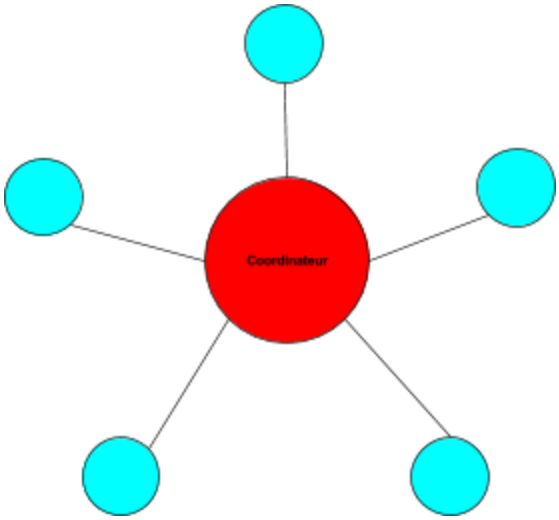

Figure 7 - Topologie en étoile d'un réseau Zigbee

Pour la configuration de ce réseau, il suffit de configurer les modules XBee qui sont à notre disposition. Nous avons utilisé le logiciel XCTU fourni par le fabricant des modules XBee pour la configuration de ceux-ci. Le tableau ci-dessous montre les configurations des différents modules XBee selon le modèle du système présenté dans la partie 2 du rapport:

Module	PAN ID	CH	MY	DH	DL	CE	BD
RPi B+	12	0x0C	0	0	0	1	9600
Arduino	12	0x0C	0	0	0	0	9600
RPi 3	12	0x0C	0	0	0	0	9600

Figure 8 - Configuration des différents modules XBee

Dans cette configuration, le coordinateur (CE=1) s'occupe de la synchronisation de la transmission, et il reçoit toutes les données envoyées sur ce réseau.

RPi configuré comme coordinateur

Comme expliqué précédemment, nous avons utilisé une carte Raspberry Pi B+ comme coordinateur, et une carte Raspberry Pi 3 comme objet connecté. Nous parlons dans un premier temps de la RPi B+ configuré en coordinateur. Nous avons tout d'abord construit une distribution minimale de Linux embarqué grâce à Buildroot avec Busybox installé. Nous avons utilisé la configuration par défaut pour la Raspberry Pi B+, avec l'option

$ make raspberrypi_defconfig

On compile ensuite le noyau et on installe l'image générée sur la RPi B+.

10- L'application Serial2stdout pour transmettre les données sur la liaison série

Le module XBee configuré comme coordinateur est connecté à la carte Raspberry Pi via une liaison série. Nous avons alors créé une application *serial2stdout* qui permet simplement de copier les informations reçues via le port série sur la sortie standard *stdout*.
Lorsqu'on branche le module XBee sur le port série de la RPi, nous avons les messages suivants:

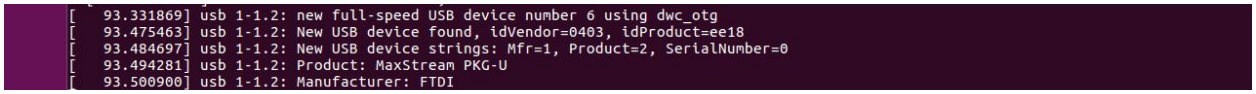

Figure 9 - Communication entre le module XBee et la RPi sur le port série

Nous en constatons que le contrôle du port série du module XBee est réalisé grâce à un module de *FTDI*. Pour pouvoir utiliser ce périphérique, il suffit de lancer le driver de *FTDI*. Ce driver se trouve dans le répertoire */lib/modules/4.4.8/kernel/drivers/usb/serial.* Il s'agit d'un exécutable de kernel sous le nom de *ftdi-sio.ko*. Pour lancer ce driver, il faut entrer la commande *modprobe ftdi-sio.ko*.

Nous avons ensuite identifié le fichier de l'entrée série correspondant au module XBee : il s'agit de */dev/ttyUSB0*. Nous devons alors copier le contenu de ce fichier au fichier */dev/stdout*. On commence par ouvrir le fichier */dev/ttyUSB0* de manière non bloquante pour supprimer l'attribut local du port, puis on bascule le descripteur en mode bloquant pour la suite des opérations. Nous devons aussi configurer les paramètres de la communication série. Sachant que la communication est de parité nulle, avec 8 bits de données, le bit d'arrêt est à 1, et la vitesse est 9600 bits/s, les paramètres à configurer sont :

- l'association des options *PARENB*, *PAREODD* du membre *c_cflag* de la structure *termios* pour la parité. Ici on choisit *PARENB*
- les options *CS5*, *CS6*, *CS7* ou *CS8* du champ *c_cflag* pour le nombre de bits de données. Ici on prend l'option *CS8*
- l'option *CSTOPB* du membre *c_cflag* pour le nombre de bits d'arrêt. Ici on la définit à 1
- la vitesse est aussi contenue dans le champs *c_cflag*. Ici sa valeur est *B9600*.

Une fois que la configuration est finie, on attaque la copie de données. On ouvre le fichier */dev/ttyUSB0*, et dans une boucle infinie, on lit et stocke les données lues depuis ce fichier dans un buffer, et on écrit le contenu de ce buffer dans le fichier */dev/stdout*. Ainsi, les données obtenues sur le port série de la RPi sont copiées sur la sortie standard *stdout*.

Pour utiliser cette application, il faut compiler le programme en utilisant soit Buildroot, soit la compilation croisée. On recopie et lance ensuite l'exécutable sur la RPi avec la commande *./serial2stdout*.
Nous avons finalement testé cette application sur la RPi B+, elle était parfaitement fonctionnelle.

RPi comme objet connecté

Dans un deuxième temps, décrivons la carte Raspberry Pi 3 qui joue le rôle de l'objet connecté. Elle collecte les informations depuis un capteur de température DS1624, et les envoie au réseau ZigBee. Nous avons suivi une démarche similaire à celle pour configurer la Raspberry Pi B+, nous avons tout d'abord construit une distribution de Linux embarqué avec Buildroot. Cette fois-ci, nous prenons la configuration par défaut de la RPi 3:
$ make raspberrypi3_defconfig
Et nous développons ensuite l'application nécessaire pour réaliser la fonction d'objet connecté.

11- Le capteur DS1624 et le protocole I2C

Par souci de temps, nous n'avons pas pu réaliser un PCB personnalisé qui réunit le capteur DS1624 et le module XBee. Nous avons alors repris la carte que l'on avait utilisée en TP Java, sur laquelle on trouve le même capteur.

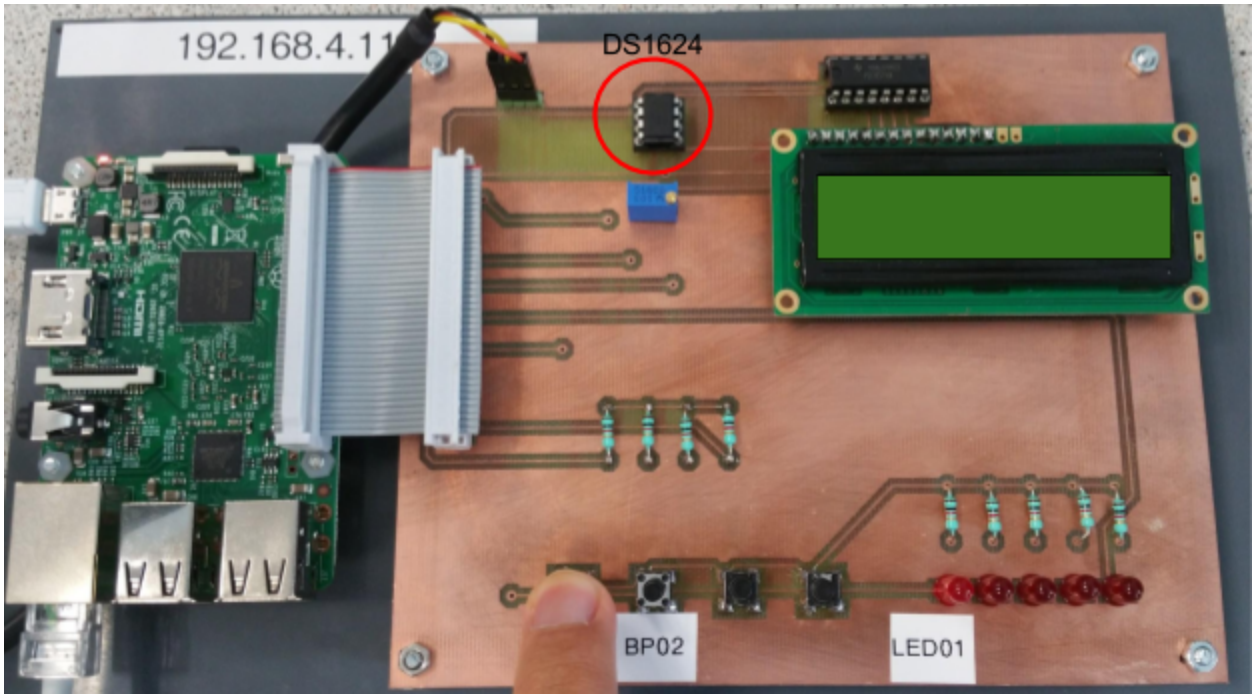

Figure 10 - La carte réunissant le capteur DS1624 et le module XBee connectée à la RPi

La figure ci-dessus montre la carte utilisée. On trouve sur la partie gauche une carte Raspberry Pi 3, et sur la partie droite de divers composants, dont le capteur DS1624. Les deux parties sont reliées avec une nappe.
Le capteur DS1624 est un capteur de température numérique, il peut communiquer avec la RPi avec le protocole I2C(Inter-Integrated Circuit). L'I2C est conçu par Philips, il s'agit d'un bus série avec multi-maître et multi-esclave. Il est très utilisé pour les communications entre les périphériques et le microcontrôleur à courte distance et faible vitesse (jusqu'à 400kbits/s). Le diagramme temporel de communication I2C en général est de la forme :

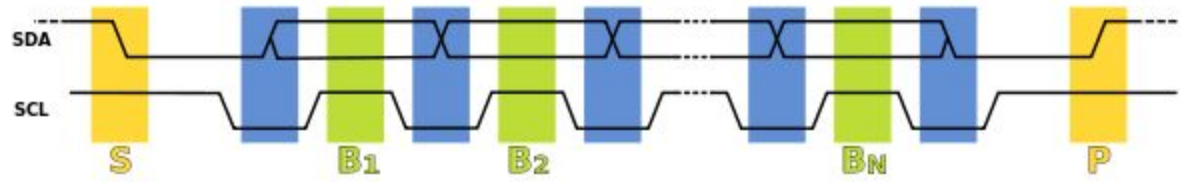

Figure 11 - Diagramme temporel d'une communication I2C

Au repos, la ligne SDA et la ligne SCL sont toutes les deux à niveau haut. Lorsqu'on veut commencer la communication, on tire la tension de SDA vers le bas, et SCL reste haut. Ceci est la condition de START(S sur le diagramme). Ensuite, pendant les périodes bleues, le maître ajuste le niveau de SDA en fonction des données, en gardant SCL au niveau bas. Et pendant les périodes vertes, la valeur de SDA est échantillonnée et capturée quand SCL est à niveau haut. A la fin de transmission, on tire SDA vers le haut lorsque SCL est à niveau haut aussi, ceci est la condition de STOP (P sur le diagramme).
Dans notre cas, le capteur DS1624 possède une adresse d'esclave de 0x48. Les commandes pour la conversion des données sont résumées dans le tableau suivant :

INSTRUCTION	DESCRIPTION	PROTOCOL
TEMPERATURE CONVERSION COMMANDS		
Read Temperature	Reads last converted temperature value from temperature register	AAh
Start Convert T	Initiates temperature conversion (Note 1)	EEh
Stop Convert T	Halts temperature conversion (Note 1)	22h

Figure 12 - Commandes pour la conversion des données

On peut alors lire l'information reçue sur la ligne SDA quand la commande 0xAA est envoyée par le maître. La donnée lue est de 2 octets, son format est montré dans la figure ci-dessous:

	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
MS BYTE	S	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	BIT 7	**BIT 6**	**BIT 5**	**BIT 4**	**BIT 3**	**BIT 2**	**BIT 1**	**BIT 0**
LS BYTE	2^{-1}	2^{-2}	2^{-3}	2^{-4}	0	0	0	0

Figure 13 - Format des données de températures du capteur DS1624

Les 4 derniers bits du LSB sont mis à 0, et le bit 15 de MSB est le bit de signe. Donc pour traduire les données reçues en température en degré Celsius, il suffit d'appliquer le formulaire suivant:

*Temperature = signed_int(MSB) + unsigned_int(LSB >> 4) * 0.0625*

12- La collecte et l'envoi des données du capteur DS1624 sur le module XBee

Nous avons développé une application "DS1624" en C tournant sur la Raspberry Pi 3 en objet connecté, qui a pour rôle d'obtenir l'information du capteur DS1624 et de l'envoyer via le port série au module XBee. Le code de cette application se trouve en annexe. Dans le code, on décrit le protocole I2C ainsi que les différentes configurations et étapes nécessaires pour envoyer les données vers le port série. Le module Xbee est relié via le port série et configuré pour qu'il transmette les données dès qu'il les reçoit. Le déroulement des étapes du code est le suivant :

- Ouverture du port série en lecture écriture via le fichier */dev/ttyUSB0* avec *int fd_tty = open(nom_tty, O_RDWR| O_NOCTTY)*
- Initialisation du capteur DS1624, c'est-à-dire ouverture du port I2C correspondant à ce capteur avec l'instruction *ds1624_init()*

Puis à l'intérieur d'une boucle infinie :

- lecture de la température du capteur sur le port I2C avec les instructions :

```
ds1624_start(fd);
ds1624_stop(fd);
temp = ds1624_read_temp(fd);
```

- Ecriture de cette valeur sur le port série */dev/ttyUSB0* en passant par un buffer avec les instructions *sprintf(buf,"TEMP=%02.1f\n", temp);* et *write(fd_tty, buf, strlen(buf));*
- Attente d'une seconde avant une nouvelle acquisition de température avec *sleep(1);*

Puis enfin à la sortie de la boucle :

- Le port série et le port I2C sont fermés avec *fclose(fd_tty);* et *fclose(fd);* .

13- La collecte et l'envoi des données du capteur DHT11 avec la carte Arduino

Figure 14 - La carte Arduino Uno

Dans notre projet une carte Arduino Uno a été utilisée en tant que End Device à côté de la Raspberry Pi 3, pour collecter des données de température et d'humidité et les envoyer à travers le réseau Zigbee au coordinateur (la Raspberry Pi B+). Pour ce faire un capteur DHT11 et un module XBee étaient connectés à la carte Arduino, le capteur servant à acquérir la température et l'humidité, et le module XBee servant à transmettre les données sur le réseau. La configuration du module XBee est donnée dans le tableau à la partie 9 sur le réseau Zigbee.

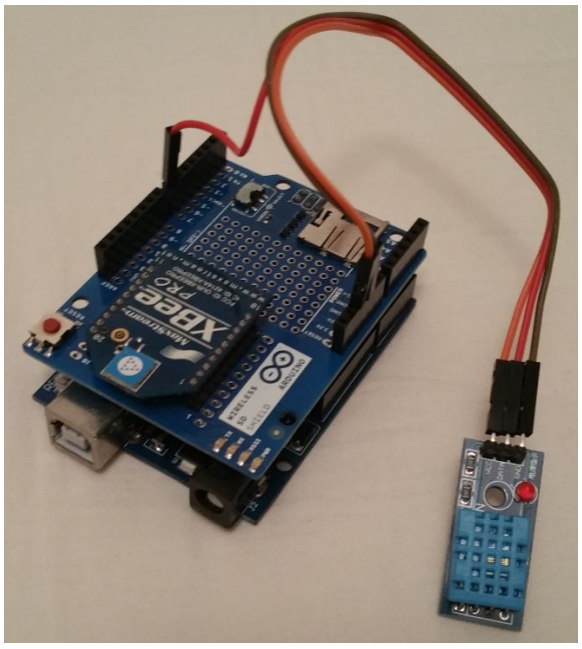

Figure 15 - Le module d'acquisition et transmission basé sur Arduino

Le programme Arduino que nous avons implémenté (dont le code est en annexe) se déroule comme suit :

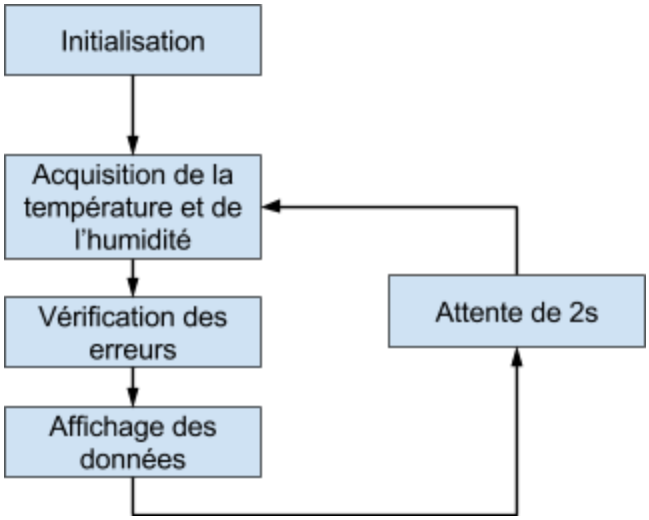

Figure 16 - Schématique du déroulement du programme Arduino

D'abord une étape d'initialisation correspondant à la fonction *setup()* d'Arduino a lieu, où est notamment configurée la vitesse de communication sur la liaison série (à 9600 baud//s) : *Serial.begin(9600).* Puis dans la fonction *loop()* exécutant le coeur du programme en boucle, la température et l'humidité sont lues à l'aide de l'instruction *int chk = DHT.read11(DHT11_PIN)*, DHT étant l'instanciation de la classe *dht* prenant en charge le capteur DHT11. La méthode read11() retourne une valeur permettant de vérifier s'il y a une erreur et de quel type d'erreur il s'agit (erreur d'attente trop longue, erreur de checksum ou erreur inconnue), c'est le rôle de l'étape suivante. Enfin les valeurs d'humidité et de température sont affichées, et le programme attend 2 secondes avant l'acquisition de nouvelles données de température et d'humidité.

14- La configuration finale de la Raspberry Pi

La dernière étape est d'écrire un Script Shell pour qu'on puisse lancer l'application et les drivers dès le démarrage. Dans ce script, on lance d'abord les différents drivers pour le bus I2C et le port série avec les commandes suivants :

modprobe i2c-dev
modprobe i2c-bcm2708
modprobe ftdi-sio

Ensuite, on lance l'application "DS1624" avec la commande :
./usr/bin/DS1624

Il suffit donc ensuite de relier la carte Raspberry PI à un PC et de lancer la commande *minicom* pour que l'application démarre et que la carte envoie les informations collectées. On peut même débrancher la carte de l'ordinateur.

Conclusion

Le bilan du projet a été positif : nous avons en effet abouti à un système fonctionnel où à partir de la carte Raspberry Pi coordinatrice nous étions en mesure à la fois d'afficher les données en provenance de l'autre Raspberry Pi et de la carte Arduino (comme cela a été montré lors de la soutenance).

Le projet nous fut également très instructif car nous avons pu élaborer et mettre en place un système embarqué, ici un réseau de capteurs, de manière complète. Nous avons en effet acquis des compétences en programmation et utilisation de cartes électroniques grâce à la mise à disposition de deux cartes Raspberry Pi et d'une carte Arduino. Nous avons également appris comment mettre en place un environnement embarqué à l'aide de l'installation d'images avec l'outil Buildroot puis comment apporter des modifications mineures à un tel système grâce à la compilation croisée, cette dernière étant plus efficace lorsqu'il s'agit d'ajouter et de débugger une application. Par ailleurs nous avons appris à utiliser le protocole réseau Zigbee (très utilisé dans le secteur de la domotique) et comment le configurer, ainsi que comment programmer et utiliser les modules XBee pour attribuer un rôle sur le réseau à chaque carte (coordinateur, end device) et pour transmettre les données à travers le réseau Zigbee. Enfin, outre l'apprentissage technique ce projet a de plus eu des apports au niveau de la méthodologie et du travail en tant qu'ingénieur : pour mener à bien notre projet il a fallu en effet adopter une démarche rigoureuse dans les procédures de conception et de mise en place de notre système.

Annexe

Script Shell responsable du lancement de l'application DS1624 :

```
#!/bin/sh
#
# Start the network....
#

# Debian ifupdown needs the /run/network lock directory
mkdir -p /run/network

case "$1" in
  start)
    printf "Starting modules: "
 modprobe i2c-dev
 modprobe i2c-bcm2708
 modprobe ftdi-sio
./usr/bin/DS1624
    [ $? = 0 ] && echo "OK" || echo "FAIL"
    ;;
  stop)
    printf "Stopping network: "
    [ $? = 0 ] && echo "OK" || echo "FAIL"
    ;;
  restart|reload)
    "$0" stop
    "$0" start
    ;;
  *)
    echo "Usage: $0 {start|stop|restart}"
    exit 1
esac

exit $?
```

Code de l'application "DS1624"

```
#include <stdio.h>
#include <string.h>
```

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <strings.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>

#define    DS1624_ADDR         0x48
#define    DS1624_READ_TEMP    0xAA
#define    DS1624_START        0xEE
#define    DS1624_STOP         0x22
void ds1624_init();
void ds1624_start();
void ds1624_stop();
double ds1624_read_temp();
void ds1624_close();
int fd_i2c;

void setspeed (struct termios * config, speed_t vitesse)
{
        cfsetispeed(config, vitesse);
        cfsetospeed(config, vitesse);
}


int main(argc, argv)
int argc;
char *argv[];
{
int fd;
double temp;
char *  nom_tty         = "/dev/ttyUSB0";
struct termios configuration;
struct termios sauvegarde;

char buf[256];

int fd_tty = open(nom_tty, O_RDWR| O_NOCTTY);
```

```
	if (fd_tty < 0) {
		perror(nom_tty);
		exit(EXIT_FAILURE);
	}


ds1624_init();

while(1) {
	ds1624_start(fd);
	ds1624_stop(fd);

	temp = ds1624_read_temp(fd);
	sprintf(buf,"TEMP=%02.1f\n", temp);
	if(write(fd_tty, buf, strlen(buf))<0){
		perror("error writing\n");
		exit(EXIT_FAILURE);
	}
	printf("TEMP=%02.1f\n", temp);
	fflush(stdout);

	sleep(1);
}
close(fd);
close(fd_tty);

exit(0);
}


void ds1624_init()
{
char *i2cdev = "/dev/i2c-1";

// Open port for reading and writing
if ((fd_i2c = open(i2cdev, O_RDWR)) < 0) {
	printf("Failed to open i2c port\n");
	exit(1);
}

// Set the port options and set the address of the device we wish to speak to
if (ioctl(fd_i2c, I2C_SLAVE, DS1624_ADDR) < 0) {
	printf("Unable to get bus access to talk to slave\n");
```

```
        exit(1);
}
}


void ds1624_start()
{
char command;

command  = DS1624_START;

if ((write(fd_i2c, &command, 1)) != 1) {
        printf("Error writing DS1624_START command to i2c slave\n");
        exit(1);
}

usleep(10000);
}


void ds1624_stop()
{
char command;

command = DS1624_STOP;

if ((write(fd_i2c, &command, 1)) != 1) {
        printf("Error writing DS1624_STOP command to i2c slave\n");
        exit(1);
}
}


double ds1624_read_temp()
{
char command;
char temp_l, temp_h;
double temp;

command = DS1624_READ_TEMP;
if ((write(fd_i2c, &command, 1)) != 1) {
        printf("Error writing DS1624_READ_TEMP command to i2c slave\n");
        exit(1);
```

```
}

if (read(fd_i2c, &temp_h, 1) != 1) {
                printf("Unable to read from slave\n");
                exit(1);
}

if (read(fd_i2c, &temp_l, 1) != 1) {
                printf("Unable to read from slave\n");
                exit(1);
}

temp_l = temp_l >> 4;
temp = temp_h + ( 0.0625 * temp_l);

return((double) temp);
}

void ds1624_close()
{
close(fd_i2c);
}
```

Code du programme Arduino :

```
#include <dht.h>
dht DHT;
#define DHT11_PIN 7

void setup()
{
  Serial.begin(9600);
  Serial.println("Reading from DHT11");
}

void loop()
{
  int chk = DHT.read11(DHT11_PIN);
  switch (chk)
  {
    case DHTLIB_OK:
    //Serial.print("OK,\t");
```

```
    break;
    case DHTLIB_ERROR_CHECKSUM:
    //Serial.print("Checksum error,\t");
    break;
    case DHTLIB_ERROR_TIMEOUT:
    //Serial.print("Time out error,\t");
    break;
    default:
    //Serial.print("Unknown error,\t");
    break;
  }
  // DISPLAY DATA
  Serial.print("Arduino 01: humidity: ");
  Serial.print(DHT.humidity, 1);
  Serial.print(",\t");
  Serial.print("Temperature: ");
  Serial.println(DHT.temperature, 1);

  delay(2000);
}
```