Rapport de projet de 4ème année

Informatique, Microélectronique, Automatique

Port de RIOT OS pour CC430

Année 2017/2018

Auteur : Baptiste Cartier

Encadrants : Thomas Vantroys / Alexandre Boé / Xavier Redon

Remerciements

Je tiens à remercier toutes les personnes qui ont apporter leur aide dans le cadre de ce projet :

Monsieur BOE, Monsieur REDON et Monsieur VANTROYS, encadrants de ce projet, dont les conseils m'ont permis d'avancer sur ce projet.

Monsieur FLAMEN, responsable du service électronique de Polytech Lille, qui m'a enseigner comment réaliser de bonnes soudures, soudures nécessaires afin de pouvoir programmer les cartes dont j'avais la charge.

Table des matières

1. Introduction

Dans le cadre de la quatrième année de la formation Informatique Microélectronique Automatique (IMA) de Polytech Lille, j'ai pu choisir de travailler sur un projet dont le sujet était à choisir, sujets proposés soit par l'école soit par des acteurs extérieurs. Parmi tous ces choix, j'ai décidé de travailler sur le sujet intitulé « Portage de RIOT-OS sur CC430 pour IOT ».

Polytech Lille possède de nombreuses cartes électroniques basées sur le microcontrôleur CC430, et travaille depuis quelques années avec le système d'exploitation pour Objets Connectés RIOT. RIOT permet le support de beaucoup de microcontrôleurs, mais ne supporte pas encore les capacités de communication du cc430.

L'objectif de ce projet est de permettre à RIOT d'exploiter les capacités de communications du CC430.

Au travers de ce rapport, je vais exposer plus en détails les objectifs de ce projet et mettre en avant le travail que j'ai effectué et ma contribution à RIOT.

2. Présentation du projet

1. Description du projet

Afin de pouvoir profiter pleinement des capacités du cc430 dans le cadre de l'IoT, il a été décidé d'utiliser un système d'exploitation (OS, Operating System) destiné au monde de l'IoT, RIOT OS.

RIOT supporte déjà la famille des MSP430, et un portage pour le cc430 a déjà commencé par la communauté, mais ce portage ne tient pas en compte les capacités radio du cc430.

Afin de profiter pleinement des fonctionnalités de routage de l'OS, il faut donc intégrer ces capacités à RIOT.

2. Objectif

L'objectif du projet est donc de d'ajouter les capacités du cc430 en terme de communications sans fil à RIOT.

Pour ce faire, plusieurs étapes doivent être réaliser :

- Effectuer des recherches sur le fonctionnement du CC430
- Se renseigner sur le fonctionnement de RIOT OS
- Intégrer les sources nécessaires à la communication sans fils dans RIOT
- Faire accepter le portage par la communauté RIOT

Au cours de la réalisation du projet, un autre objectif a été choisi afin de montrer la qualité du portage :

- Mettre en place un réseau d'objets connectés grâce au protocole RPL, protocole supporté par RIOT

3. Travail accompli

1. Prise en main du CC430

a) Mise en place d'une chaine de compilation et de programmation des cartes

La première partie de ce projet a été de mettre en place tous les logiciels nécessaires afin de compiler et de programmer les cartes.

J'ai tout d'abord installé le compilateur dédié aux MSP430, msp430-gcc. Ce compilateur est fourni par Texas Instruments (TI) et est gratuit au téléchargement. Cette version gratuite est complète et supporte tous les microcontrôleurs utilisant des MSP430 et MSP432. Ce compilateur peut être utilisé à partir de ligne de commande ou avec le logiciel Code Composer Studio v6.0 et plus. J'ai décidé de l'utiliser avec la ligne de commande, car il est assez simple à utiliser et cela m'a évité de devoir prendre en main le dit logiciel.

J'ai ensuite installé l'outil permettant de flash les cartes, mspdebug. Afin de programmer les cartes, je dois passer par une carte intermédiaire crée par Ti, le launchpad msp-exp430g2.

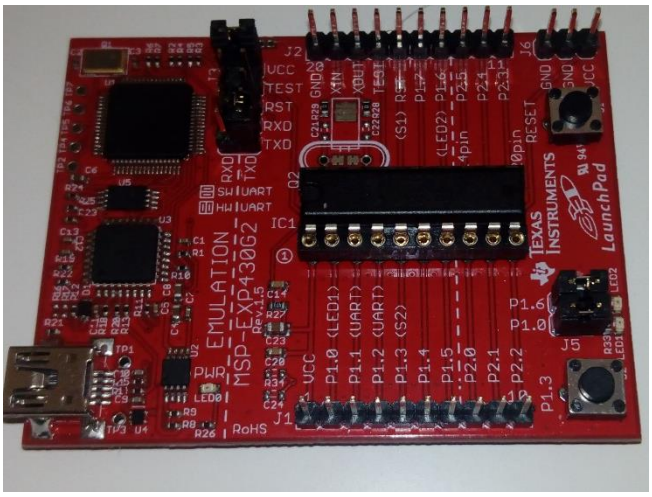

Figure 1 : Launchpad msp-exp430g2

Des fils ont été soudé aux cartes afin de pouvoir les relier au launchpad.

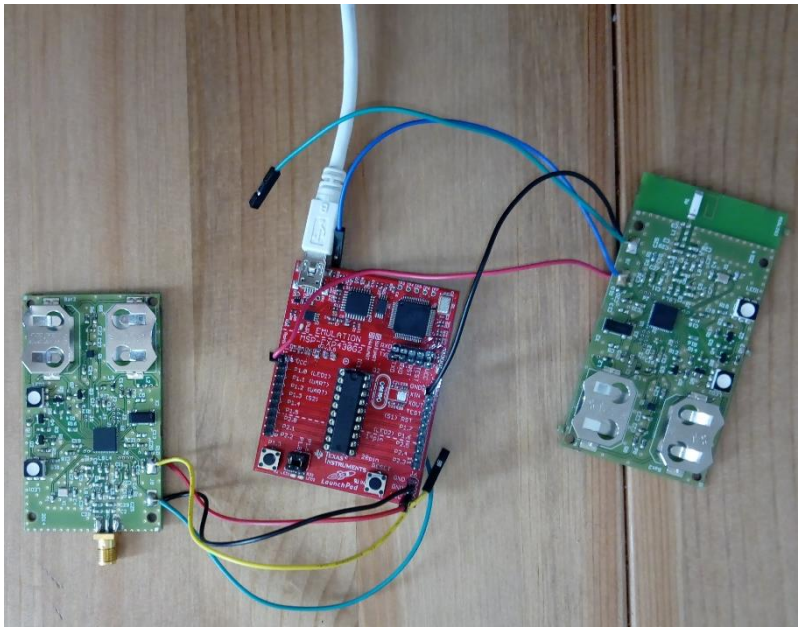

Figure 2 : Montage complet avec 2 cartes et le launchpad

b) Manipulation du cc430 et communication par radio

Les cartes basées sur le cc430 ont déjà été utilisé dans le cadre d'un projet IMA, des sources afin d'utiliser le CC430 existent déjà. Il s'agit du projet IMA5 P11 2015/2016 J'ai donc récupéré ces sources afin de comprendre comment fonctionnait le cc430 et surtout comment interagir avec son module radio intégré.

La première étape a consisté à faire clignoter une LED. Cette étape a été très facile, il m'a simplement fallu utiliser les sources déjà existantes. Comme l'utilisation de la LED et des différents périphériques (ADC, Gestion de la Puissance, …) ne faisaient pas partie du projet et n'avait pas de lien direct avec le module radio, j'ai fait confiance aux sources existant et je les ai utilisés tels quel pour mes tests. En revanche le code déjà existant pour le module radio a été analysé en détail afin de s'assurer que ce code était fonctionnel.

Le cc430 accède à son module par une méthode hybride entre accès direct au registre et communication par bus. Le microcontrôleur émet une commande « strobe » dans le registre dédié à cet effet, RF1AINSTRB, puis insère ou lit les données correspondantes à cette strobe dans les registres RFA1DIN et RFA1DOUT respectivement.

Les principales strobe qui nous intéressent sont les suivantes : SNGLREGRD et SNGLREGWR pour lire et écrire dans les registres du module radio, SNGLRXRD et SNGLTXWR pour lire et écrire un unique octet dans la file d'émission et de réception, RXFIFORD et TXFIFOWR pour lire et écrire plusieurs octets dans la file d'émission et de réception, SRX et STX pour activer le mode réception ou émission.

La transmission des données se fait en remplissant un buffer FIFOTX (First In First Out Transmission). Si lors de la transmission d'un paquet le buffer FIFOTX se retrouve vide avant la fin de l'envoi du paquet, il se produit un buffer underflow, et la transmission est échouée et arrêtée, le paquet étant perdu.

La réception elle s'effectue grâce à un autre buffer FIFORX. Dès qu'un message de préambule est détecté, le buffer se rempli automatiquement. Si les données ne sont pas lues avant le remplissage complet du FIFORX, il y a un buffer overflow, la réception est arrêtée et le paquet perdu.

Le module peut traiter l'émission et la réception de 3 manière différentes : paquet de taille fixe, paquet de taille variable et paquet infini.

En mode paquet de taille fixe, le cc430 n'accepte d'émettre et de recevoir que des paquet d'une certaine taille fixée dans le registre PKTLEN. Ce mode n'est pas très pratique si l'on veut transmettre différentes tailles de paquets.

En mode paquet infini, il est possible de transmettre une quantité choisie de donnée, sans limite de taille, mais cela demande beaucoup de temps processeur en mode envoi, le CPU devant s'assurer de remplir le FIFOTX suffisamment vite pour éviter les buffer underflow, et si un autre processus à la main sur le processeur au mauvais moment la transmission peut échouer. De même, ce moment force le récepteur à passer en écoute active, peu pratique dans un OS.

Enfin, en mode paquet de taille variable, le registre PKTLEN indique seulement la taille maximale tolérée. Dans ce cas, afin de préciser la taille du paquet à émettre, le premier octet transmit doit impérativement être la taille du paquet qui suit. Cette option a été choisi car plus simple à mettre en place et permet d'être plus indépendante vis-à-vis de l'OS. En revanche, cela implique que les paquets transmis ne peuvent dépasser la taille de 255 octets.

Au début du projet, j'avais supposé que les sources que j'avais reçu étais bonnes, supposition renforcée par la suite lorsque j'ai découvert que ces sources provenaient de TI même, mais elles n'étaient fonctionnelles que pour des paquets de la taille du FIFORX et FIFOTX. En effet, ces buffers font une taille de 64 octets, et si ces tailles sont dépassées, il y a overflow. Une interruption est déclenchée quand le buffer RX est plein, ou si le seuil défini dans le registre FIFOTHR est dépassé.

Si l'on souhaite émettre des paquets d'une taille entre 65 et 255 octets, il faut procéder comme suit :

a) Ecrire la taille totale du paquet en tant que premier octet à transmettre
b) Remplir le FIFOTX sans effectué d'overflow
c) Attendre que de la place se libère SANS que le buffer soit intégralement vidé
d) Lire dans le registre TXBYTES le nombre d'octets dans le buffer
e) Compléter l'espace disponible avec de nouvelles données
f) Revenir à l'étape c) jusqu'à ce que toutes les données ont été émises

Pour la réception, il faut agir de tel manière :

a) Attendre l'arrivée d'un paquet, symbolisée par le dépassement du seuil du FIFORX, son remplissage ou la fin du paquet
b) Lire le premier octet pour connaitre la taille totale du paquet
c) Lire combien d'octets sont dans FIFORX dans le registre RXBYTES
d) S'il reste à recevoir plus d'octets que présent dans FIFORX, lire RXBYTES - 1 octets et revenir en a)
e) Sinon lire RXBYTES octets dans FIFORX

Malheureusement, malgré l'implémentation de cet algorithme, les transmissions de paquets de plus de 64 octets ne fonctionnent pas. Je n'ai pas su trouver la cause de ce comportement, je n'ai détecté aucun buffer overflow ou underflow.

2. Prise en main de RIOT OS

a) Analyse de la structure de RIOT OS

Le code source de RIOT peut être récupéré sur GitHub gratuitement, il s'agit d'un logiciel libre couvert par la licence libre LGPL-2.1.

RIOT couvre déjà une partie du portage pour le cc430, surtout la partie la plus importante, le noyau de l'OS.

Les sources de RIOT s'articulent autour d'un système de dossier et de Makefile en cascade complexe mais permettant d'ajouter facilement des sources en plus à l'OS, à condition de savoir où les placer.

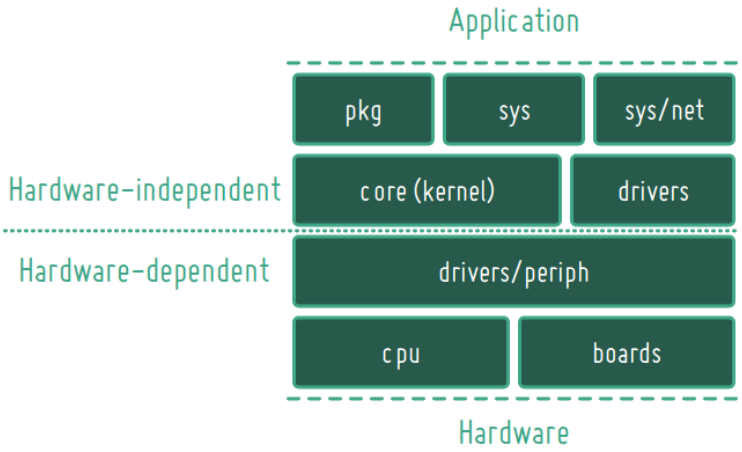

Figure 3 : Organisation générale de RIOT OS

De plus, grâce à ce système de Makefile, il est possible de faire de la compilation modulée et de n'intégrer que les modules qui nous intéresse. Enfin, RIOT dispose d'un mode natif, c'est-à-dire qu'il est possible de compiler et faire tourner RIOT sur un machine Linux classique, permettant de tester des applications avant de passer par le matériel cible.

Parmi les dossiers de l'architecture de RIOT, deux nous intéressent plus particulièrement :

- Le dossier RIOT/cpu/cc430 : le dossier contenant le début de portage du cc430
- Le dossier RIOT/boards : le dossier contenant les différentes cartes supportées par RIOT. La carte sur laquelle je travaillais étant une carte faite en interne par Polytech Lille, est n'est bien évidement pas supporté par RIOT

b) Intégration de la carte à l'architecture de RIOT

La carte n'étant pas supportée par RIOT, avant de pouvoir compiler RIOT, il faut préciser dans le makefile quelle est la carte cible. J'ai donc créé les fichiers et dossiers nécessaires à cette compilation. J'ai pris exemple sur la carte Chronos, carte supportée par RIOT et possédant un cc430.

Dans ce dossier doit se trouver toutes les sources relatives à la carte en elle-même, c'est-à-dire l'utilisation de périphériques en tout genre connectés aux cc430, et aucunes sources relatives au

cc430 lui-même. Comme ce dossier n'existe que pour mes tests et n'a pas vocation à faire partie de RIOT à l'avenir, seule la structure a été conservée, ne voulant pas modifier les sources récupérées précédemment n'ayant pas de rapport direct avec le module radio.

L'architecture créée est la suivante :

```
RIOT/boards/cc430BV
├── include/
├── drivers/
├── Makefile
├── Makefile.features
└── Makefile.include
```

Figure 4 : Architecture basique d'une carte

Dans le dossier include/ se trouve tous les fichiers .h relatifs aux sources de cc430BV et le dossier drivers/ contient toutes les sources pour les drivers des différents périphériques. C'est dans ces dossiers que seront placés toutes les sources non relatives au module radio.

Les différents fichiers de Makefile sont sensiblement les même que ceux de la carte chronos, simplement modifiés pour correspondre à notre carte.

Afin de vérifier que cette architecture convenait à RIOT, j'ai compilé et flashé certains programmes d'exemples simples avec comme cible la carte cc430BV. Afin de faire ceci il suffit de se placer dans le répertoire de l'exemple, et lancer la commande suivante (ici le programme d'exemple hello-world) :

```
/RIOT/examples/hello-world$ make BOARD=cc430BV
```

Figure 5 : commande make pour la carte

Comme le cc430 est déjà supporté, il n'y a pas besoin de lui préciser quel compilateur utilisé, grâce à la chaine de makefile, le bon compilateur est choisi.

3. Développement du portage

a) Première version du portage

Afin de réaliser le portage des capacités radio du cc430, j'ai recherché comment le portage d'un module (radio ou autre) devait être ajouté à RIOT. Pour un module radio, RIOT a mis en place un tutoriel disponible sur son wiki. Malheureusement, ce tutoriel est en construction, et ne porte que sur les modules radio connectés par bus (SPI ou autre) et non sur les modules radios intégrés au microcontrôleur. Il s'avère que l'architecture du port doit avoir la même forme, mais ce n'est pas expliqué et donc je n'ai pas suivi ce tutoriel. Ce point sera traité dans la partie 3. b) . J'ai donc considéré le module radio comme un simple module, et j'ai effectué le port en prenant l'exemple sur l'adc et du timer, c'est-à-dire placer simplement les sources dans le répertoire du microcontrôleur.

Pour faire un portage comme un adc, il faut suivre l'arborescence suivante :

```
RIOT/
└── cpu/
    ├── cc430/
    │   └── RF1A.c
    └── includes/
        └── RF1A.h
```

Figure 6 : Emplacement des fichiers pour la première version du portage

Avec RF1A.h et RF1A.c respectivement le fichier header et le fichier source d'accès au module radio.

Un des intérêts de RIOT est qu'une fois ces fichiers placés au bon endroit, il n'y a pas besoin de modifier les makefiles pour la compilation, les sources sont directement intégrées à la chaine.

Un des problèmes avec une réception d'un message, est qu'on ne sait pas quand ce message va arriver. Afin d'éviter un boucle d'attente active, j'ai décidé d'intégrer la gestion des threads dans mes sources.

Ainsi, si un thread souhaite recevoir un message, il va appeler la fonction *cc430_radio_receive*, dans laquelle se trouve l'appel système *thread_sleep()* permettant à un thread d'indiquer au scheduler de l'OS qu'il doit être retirer de la rotation jusqu'à qu'il soit réveillé par l'opération thread_wakeup.

Ensuite, lors de la réception d'un message, indiqué par l'interruption `RF1AIV_RFIFG4`, indiquant soit la réception d'un paquet, soit le dépassement du seuil du FIFO de réception soit le remplissage intégral du FIFO de réception, on réveille le thread attendant un paquet, qui par la suite lira le paquet reçu. Ceci est effectué grâce au code suivant :

```
#pragma vector=CC1101_VECTOR
__attribute__((interrupt(CC1101_VECTOR)))
void CC1101_ISR(void)
{
    if (RF1AIV == RF1AIV_RFIFG4) {
        thread_wakeup(cc430_radio_pid);
    }
}
```

Figure 7 : code de réveil du thread par l'interruption

Ce code permet la gestion de l'interruption, et de plus permet de limiter le temps passé dans l'interruption.

```
void cc430_radio_receive(void *buffer, uint8_t *length)
{
    cc430_radio_pid = thread_getpid();
    thread_sleep();
    *length = cc430_radio_read_single_reg(RXBYTES);
    cc430_radio_read_burst_reg(RF_RXFIFORD, buffer, *length);
}
```

Figure 8 : Code de réception devant être appelé par un thread

Afin de tester si le port fonctionnait, j'ai créé un programme de test avec deux cartes : une en émission et l'autre en réception. La première carte envoyait deux messages forgés à l'avance, en

alternance avec un délai entre chaque émission. La seconde devait allumer une LED uniquement quand elle recevait l'un des deux paquets. Les deux cartes fonctionnaient avec RIOT en fond, faisant tourner chaque fonction, émission et réception, dans des threads.

Une fois ces tests effectués et le code réécrit pour correspondre au convention de codage de RIOT, j'ai effectué un Pull Request (PR) afin de fusionner mon travail avec RIOT de manière officielle. A ce jour ce PR n'a pas été accepté, et ce probablement car ce port ne correspond pas au port attendu.

b) Deuxième version du portage

Une fois le premier port effectué, afin de valider son efficacité, il a été décidé de mettre en place un réseau dynamique avec le protocole RPL. Au fils des recherches sur comment utilisé un tel réseau avec RIOT, je me suis rendu compte que le port précédemment effectué ne correspondait pas à ce qui était attendu par RIOT.

RIOT étant un OS spécifiquement créé pour l'IoT, il intègre un pile réseau, GNRC, composé de plusieurs couches distinctes. La couche faisant le lien entre la partie physique et les autres couches réseau est la couche nommée netdev.

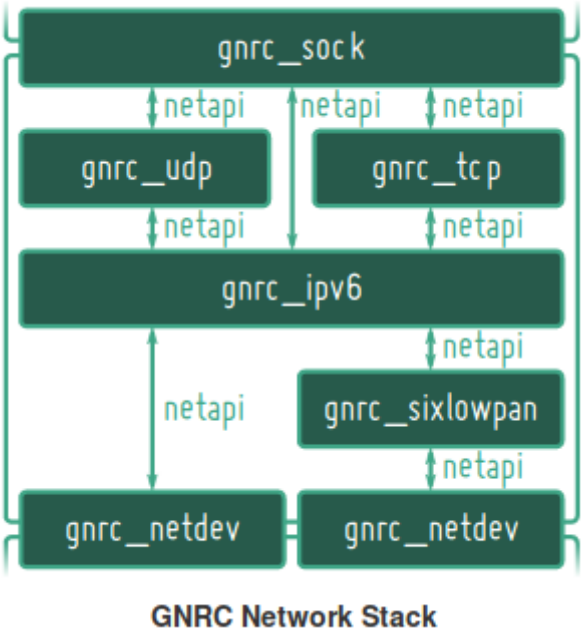

Figure 9 : Pile réseau de GNRC

Le port d'un module radio doit donc être fait au niveau de la couche la plus bas niveau, netdev, afin de rendre compatible le module radio avec le reste de GNRC.

En pratique, la couche netdev est constituée de deux types de fichier :

- Les fichiers correspondant à la couche physique
- Les fichiers faisant le lien entre les fichiers précédent et l'API générique de GNRC

Afin d'effectuer au mieux ce port, j'ai pris exemple sur deux autres ports, celui du driver pour les cc110x, et celui pour le cc2538. Le cc2538 est un microcontrôleur ayant aussi des capacités sans fils intégrées.

Les fichiers de la couche physique correspondent en fait aux sources que j'avais déjà, rien de plus devait être fait.

En revanche, j'ai dû refaire la totalité de l'API afin qu'elle soit adaptée au cc430. Cette API est basée sur les fonctions suivantes :

`void _irq_handler(void)`
Fonction servant simplement à signaler à netdev qu'une interruption matérielle s'est produite afin que netdev la traite hors d'un contexte d'interruption.

`int(*  send )(netdev_t *dev, const iolist_t *iolist)`
Fonction permettant l'envoi d'un paquet contenu dans le paramètre iolist.

`int(*  recv )(netdev_t *dev, void *buf, size_t len, void *info)`
Fonction permettant la réception de paquet. En fonction des différents paramètres, cette fonction doit se comporter de manière différente :

- Si buf == NULL et len == 0, la fonction doit retourner la taille du paquet à recevoir
- Si buf == NULL et len > 0, on doit abandonner le paquet
- Si buf != NULL, la fonction doit écrire le paquet reçu dans buf

`int(*  init )(netdev_t *dev)`
Fonction d'initialisation du module radio, activant le mode réception du module et transmettant à netdev différentes informations sur le module radio.

`void(*  isr )(netdev_t *dev)`
Une fonction servant de handler d'interruption, permettant de définir quelle interruption a été générer et de la traiter dans un thread plutôt que dans une interruption.

`int(*  get )(netdev_t *dev, netopt_t opt, void *value, size_t max_len)`
Fonction retournant une valeur d'une option du module, option choisie par le paramètre opt

`int(*  set )(netdev_t *dev, netopt_t opt, const void *value, size_t value_len)`
Fonction définissant une valeur d'une option du module, option choisie par le paramètre opt

Pour les options de type netopt_t des fonctions _get et _set, j'ai choisi de reprendre les mêmes options que le port pour le cc110x :

- NETOPT_DEVICE_TYPE : importante, permet de préciser aux autres couches quel type de module est utilisé, ici il s'agira d'un module cc110x
- NETOPT_CHANNEL : sur quel channel la radio émet
- NETOPT_ADDRESS : quelle est l'adresse physique du module
- NETOPT_MAX_PACKET_SIZE : quelle est la taille maximale du paquet
- NETOPT_IPV6_IID : l'identificateur IPv6 de l'interface
- NETOPT_ADDR_LEN : taille de l'adresse cible en octet
- NETOPT_SRC_LEN : taille de l'adresse cible en octet
- NETOPT_ADDRESS : Adresse physique du module
- NETOPT_CHANNEL : sur quel channel la radio émet

Certains éléments importants à prendre en compte sont les notions de qualité du lien et de puissance du signal reçue. Afin de mettre en place certains réseaux dynamiques et optimisés, tel RPL, ces deux valeurs sont importantes. Le cc430 possède les capacités de calculer ces deux

valeurs à chaque paquet reçu. Les registres LQI (Link Quality Indicator) et RSSI (Received Signal Strength Indicator) contiennent ces valeurs.

```
if (cc1l0x->pkt_buf.rssi >= 128) {
    cc1l0x_info->rssi = ((int16_t)cc1l0x->pkt_buf.rssi - 256) / 2 - CC1l0X_RSSI_OFFSET;
}
else {
    cc1l0x_info->rssi = (int16_t)cc1l0x->pkt_buf.rssi / 2 - CC1l0X_RSSI_OFFSET;
}

cc1l0x_info->lqi = cc1l0x->pkt_buf.lqi;
```

Figure 10 : Remontée des informations sur le LQI et le RSSI

Le calcul pour le RSSI est décrit dans les documentations du cc430

4. Mise en place d'un réseau sans fils avec le protocole RPL

Afin de montrer que le portage était fonctionnel, il a été décidé de mettre en place un réseau avec le protocole RPL.

a) Explication du protocole RPL

RPL est un protocole de routage pour l'IoT, plus précisément pour les Low Power and Lossy Networks (LLN), ou Réseaux à faible puissance et à pertes.

Le principe de RPL est d'interconnecter les différents objets via la construction d'un graphe acyclique, nommé DODAG (Destination-Oriented Directed Acyclic Graphe). Le but de ce graphe est d'offrir le chemin le plus efficace vers le nœud racine du graphe, la destination. Chaque autre élément (nœud ou feuille) du graphe agit comme un routeur pour les autres éléments.

Chaque nœud possède une caractéristique « rang » permettant de déterminer sa position dans l'arbre. Plus son rang est faible, plus ce nœud est proche de la racine, le rang 0 étant le nœud racine.

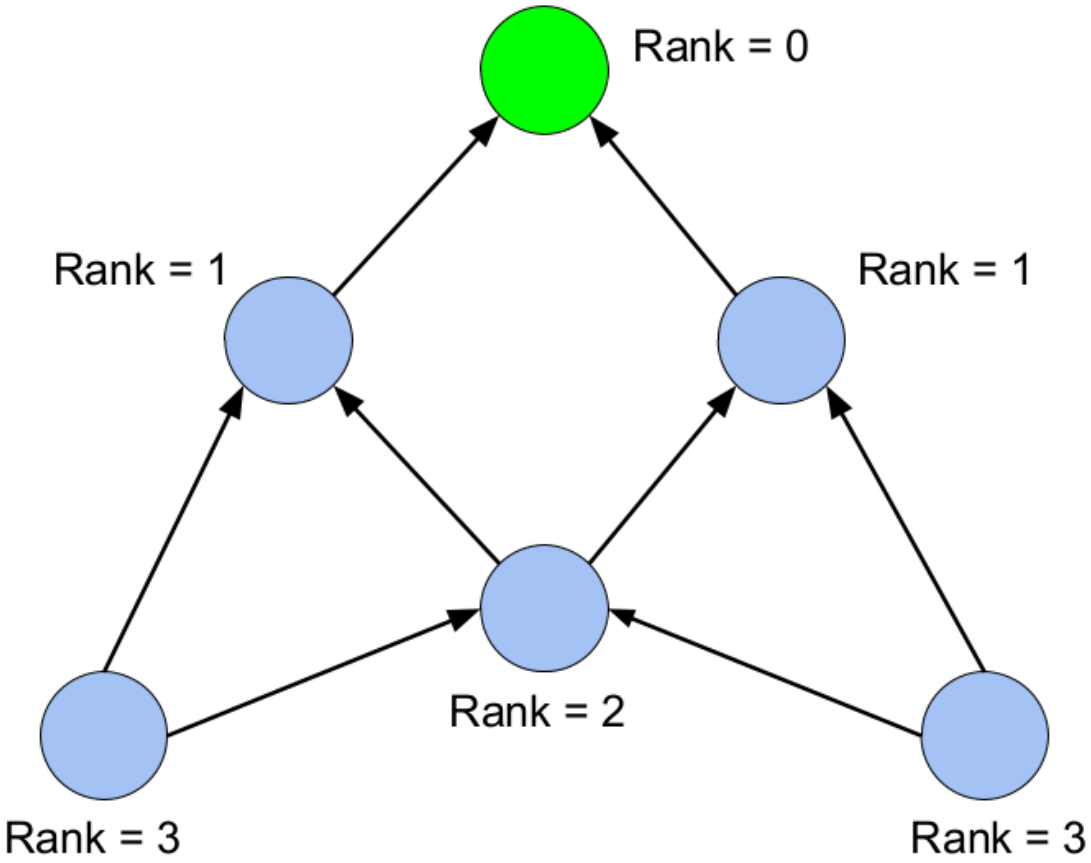

Figure 11 : Exemple de DODAG à 3 niveaux

Le DODAG est construit grâce à plusieurs messages de contrôle de type ICMPv6 :

- DIO (DAG Information Object) : permet de transmettre les informations du DAG à un nœud afin que ce dernier puisse découvrir le réseau et ses paramètres, pour choisir ses parents dans le DODAG et s'intégrer au réseau
- DIS (DAG Information Solicitation) : Demande un DIO à un nœud
- DAO (Destination Advertisement Object) : propage les informations de destination vers le haut du DODAG

Le choix des parents est effectué grâce à une Fonction Objectif (OF, Objective Function). Il est important que les nœuds utilisent la même OF sinon le DODAG pourrait ne pas être optimisé.

Un autre intérêt très important du protocole RPL réside dans le fait que ce DODAG n'est pas statique mais évolue en fonction des évènements, influençant les résultats de l'OF. Par exemple, si le lien entre deux nœuds disparait (disfonctionnement d'un des nœuds, porte métalliques se fermant…), grâce aux messages de contrôle, le DODAG se modifie afin de déterminer la nouvelle configuration.

b) Prise en main de RPL avec RIOT

Afin de comprendre comment mettre en place un réseau RPL avec RIOT, j'ai décidé de suivre le tutoriel dédié à RPL sur le wiki de RIOT. Ce tutoriel a aussi été suivi par les projets IMA5 PFE

2016/2017 P7 et 2017/2018 P15. Grâce à leurs expériences et au tutoriel, la mise en place du réseau RPL a été assez simple.

Il est à noter qu'à ce jour, ce tutoriel n'est pas à jour, certaines commandes ne fonctionnent plus : il faut remplacer la commande fibroute par nib route.

```
Iface  6  HWaddr: c2:f7:95:e7:24:c0
          MTU:1500  HL:64  RTR
          RTR_ADV
          Source address length: 6
          Link type: wired
          inet6 addr: fe80::c0f7:95ff:fee7:24c0  scope: local
          inet6 addr: 2001:db8::1  scope: global  VAL
          inet6 group: ff02::2
          inet6 group: ff02::1
          inet6 group: ff02::1:ffe7:24c0
          inet6 group: ff02::1a
```

Figure 12 : Adresses du nœud racine

Notons l'adresse du nœud racine.

Une fois le DODAG a été créé par la nœud 0 et les nœuds initialisés, on observe bien que les autres nœuds se connectent aux DODAG.

```
rpl
instance table: [X]
parent table:   [X]     [ ]     [ ]

instance [1 | Iface: 6 | mop: 2 | ocp: 0 | mhri: 256 | mri 0]
        dodag [2001:db8::1 | R: 512 | OP: Router | PIO: on | CL: 0s | TR(I=[8,20], k=10, c=9, TC=0s)]
               parent [addr: fe80::c0f7:95ff:fee7:24c0 | rank: 256 | lifetime: 296s]
```

Figure 13 : Table RPL d'un nœud

c) Réseau RPL pour cc430

Une fois la prise en main effectuée, il faut créer un réseau avec RPL avec les vraies cartes.

Afin d'être sûr que le réseau RPL fonctionne correctement, il faut s'assuré qu'il y ai plusieurs rangs : il faut donc couper la liaison entre l'un des nœuds et le nœud racine. Pour ce faire il suffit de placer une plaque métallique entre le nœud en question et la racine, forçant la création d'un DODAG avec au moins un rang 2.

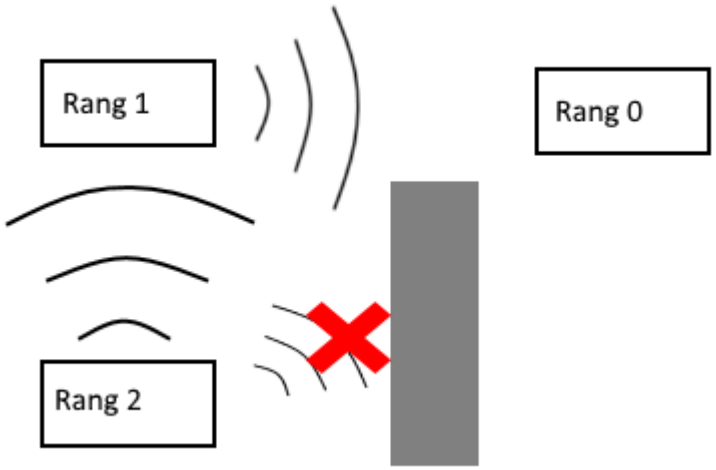

Figure 14 : Protocole de création du réseau RPL

Un problème majeur est survenu lors de cette étape : le tutoriel utilisé par les autres groupes et moi-même possède une empreinte mémoire trop importante.

Cela est effectivement indiqué dans le makefile de ce tutoriel, précisant que la carte chronos n'était pas compatible, et donc le cc430. Cela implique de devoir refaire la création du réseau avec RPL à la main.

```
BOARD_INSUFFICIENT_MEMORY := calliope-mini chronos microbit msb-430 msb-430h \
                             nucleo32-f031 nucleo32-f042 nucleo32-f303 nucleo32-l031 \
                             nucleo-f030 nucleo-f070 nucleo-f072 nucleo-f103 nucleo-f302 \
                             nucleo-f334 nucleo-l053 spark-core stm32f0discovery telosb \
                             wsn430-v1_3b wsn430-v1_4 z1
```

Figure 15 : Extrait du makefile indiquant que le cc430 ne possède pas assez de mémoire

Cela n'était pas un problème pour les autres groupes et lors des tests en mode natif, les systèmes ayant suffisamment de mémoire.

C'est lors des recherches sur l'utilisation de RPL avec une empreinte mémoire plus faible que je me suis rendu compte que la première version du port n'était pas correcte.

Comme ce programme d'exemple n'est pas utilisable, j'ai dû recréer toute la pile gnrc pour essayer de voir si le portage était compatible.

La plupart des exemples et de la documentation afin d'utiliser la pile GNRC étant basée sur l'exemple gnrc_networking et sur l'utilisation du shell de RIOT, malgré mes efforts je n'ai pas su faire fonctionner gnrc.

4. Conclusion

L'objectif de ce projet était de porter les capacités radio du cc430 pour RIOT OS et, au terme de ce projet, la couche netdev de GNRC pour le cc430 est opérationnelle et fonctionne, au moins si utilisée seule comme préconisé. En revanche, son intégration au sein de GNRC reste à démontrer, le cc430 n'ayant pas assez de mémoire pour utiliser les programmes d'exemple. Ce portage reste améliorable (paquets de plus grandes tailles, moins de boucles actives) mais correspond aux attentes de RIOT.

Ce portage constitue une base solide pour des futurs projets, tels que l'intégration complète de ce netdev à GNRC ou encore la création de réseaux RPL.

J'ai trouvé ce sujet très intéressant et dans la lignée de mon projet professionnel, bien qu'assez compliqué à prendre en main, devant comprendre le fonctionnement d'un nouveau microcontrôleur et le fonctionnement de RIOT OS. Les principales difficultés rencontrées proviennent du manque de connaissance sur RIOT et le CC430 en début de projet, ce qui m'a orienté dans de mauvaises directions.

Bien que j'aurais aimé réussir à démontrer l'utilité de mon travail au travers la mise en place d'un réseau RPL, je suis heureux de l'état du projet.

Bibliographie

Documentation sur RIOT OS :

Site officiel de RIOT OS : https://www.riot-os.org/

API de RIOT : http://riot-os.org/api/

Wiki de RIOT : https://github.com/RIOT-OS/RIOT/wiki

Documentation sur le CC430 :

http://www.ti.com/lit/ds/symlink/cc430f5137.pdf

http://www.ti.com/lit/ug/slau259e/slau259e.pdf

Documentation sur RPL :

https://tools.ietf.org/html/rfc6550

Références des projets ayant servi de sources :

RIOT et RPL :

https://projets-ima.plil.fr/mediawiki/index.php/P15_R%C3%A9seau_de_capteurs_temps_r%C3%A9el

https://projets-ima.plil.fr/mediawiki/index.php/P7_R%C3%A9gulation_temps_r%C3%A9el_sur_r%C3%A9seau_sans_fil

CC430 :

https://projets-ima.plil.fr/mediawiki/index.php/P11_Spectateur_augment%C3%A9