Rapport de projet de fin d'études

Informatique, Microélectronique, Automatique

Régulation temps réel sur réseau sans fil

Élèves : OBEISSART Morgan – ROBIC Vincent

Encadrants : Alexandre Boé – Thomas Vantroys

Année scolaire : 2016 – 2017

Sommaire

Introduction

Dans le cadre de notre cinquième et dernière année au sein du département Informatique, Microélectronique et Automatique de Polytech Lille, nous avons choisi de nous pencher sur l'un des sujets de projet proposés par des personnes intérieures à l'école. L'intitulé de ce sujet est « Régulation temps réel sur réseau sans fil » : le but est d'implémenter un protocole de communication avec des notions temps-réel.

Nous avons réalisé ce projet en binôme, avec l'aide de nos encadrants, du personnel de Polytech Lille et de personnes extérieures. Nous souhaitons donc particulièrement remercier M. Boé, M. Vantroys pour leur travail et leur aide en tant que tuteurs de ce projet. Nous remercions M. Redon qui nous a notamment aidé pour la commande du matériel et Mme. Rolland pour son aide sur le dimensionnement de notre carte électronique. Enfin, nous tenons à remercier M. Flamen, responsable du service électronique de Polytech Lille, pour ses connaissances pour la gravure de cette carte électronique ainsi que pour son aide et sa patience qui nous ont permis de résoudre les problèmes rencontrés.

Dans un premier temps, nous commencerons par présenter plus en détail le projet que nous avons choisi. Puis, nous reviendrons sur le travail qui a été effectué. Enfin, nous ferons un bilan de notre projet de fin d'études.

I. Présentation du projet

1. Objectif du projet

L'objectif du projet est d'implémenter un protocole de communication avec des notions temps-réel. L'utilisation de réseaux sans fil pose un grand problème dû à la difficulté d'être sûr de la délivrance des données dans un temps borné. Notre travail consistait donc à nous intéresser à ce problème et à tenter de le résoudre.

2. Description du projet

Ce sujet est un projet très ouvert pour lequel aucun cahier des charges n'avait été définit (cela a d'ailleurs été l'une de nos tâches). C'est la principale raison pour laquelle nous l'avons choisi, car nous pensions qu'il nous apporterait des connaissances multidisciplinaires et qu'il nous apprendrait à gérer un projet. Comme il est très ouvert, nous ne pouvions pas foncer tête baissée sur le codage de la solution. Il a fallu au préalable faire un état de l'art des réseaux temps-réel sans fil. Nous avons choisi notre propre application dont le but est d'illustrer la solution que nous avons adoptée. Il a donc fallu avancer méthodiquement, avec une part importante de gestion de projet.

Pour ce projet, il convenait ainsi de :

- Réaliser des recherches bibliographiques sur les réseaux sans fils temps-réel,
- Choisir les moyens, matériel et logiciel, selon le résultat de ces recherches,
- Définir une application et le cahier des charges,
- Implémenter un protocole de communication pour notre application.

3. Choix techniques : matériel et logiciel

Les différents algorithmes ont été développés sur des cartes de développement STM32F4 discovery. La communication entre ces différentes cartes se font par lien radio. Pour concevoir un réseau temps-réel, il nous a d'abord fallu choisir un système d'exploitation qui fournisse un support temps-réel. Nous

nous sommes ainsi particulièrement intéressés à trois systèmes d'exploitations différents. Voici les caractéristiques que nous leur avons trouvés :

OS	Min RAM	Min ROM	Pilote STM32F4discovery	Support temps réel	Pilote radio
Contiki	< 2kB	< 30 kB	✗	✗	✗
FreeRTOS	< 1kB	~ 5kB	~	✓	✗
Riot OS	~ 1.5kB	~ 5kB	✓	✓	✓

Fig. 1 : Tableau comparatif de différents OS

Comme nous pouvons le constater, Riot OS est le seul système d'exploitation qui fournisse à la fois un support temps réel, le pilote de notre carte de développement ainsi que le pilote de plusieurs puces permettant de réaliser une communication radio. Ainsi, c'est ce système d'exploitation que nous avons utilisé pour nos travaux.

Pour le portage radio, nous avons choisi de concevoir une carte électronique autour d'une des puces pour laquelle Riot propose un driver : la puce Atmel AT86RF231. Nous avons développé cette carte grâce au logiciel Eagle.

Nous avons également utilisé deux servomoteurs pour notre application. Ce point sera abordé dans la prochaine partie.

Nous avons présenté l'objectif de notre projet et en avons fait une courte description. Nous connaissons désormais les choix techniques que nous avons mis en place. Détaillons maintenant le cahier des charges que nous avons défini.

4. Cahier des charges

La définition d'un cahier des charges était l'un de nos principaux objectifs. Nous allons ici le détailler.

Notre objectif est de contrôler les servomoteurs d'un robot via un réseau sans fil temps-réel. Le robot a été réalisé par nos soins. La liaison sans fil est, dans notre cas, une liaison radio. Chaque carte STM32F4discovery, ainsi que le module radio qui y est associé, constituent les nœuds de notre réseau. Le robot, constitué de deux servomoteurs, est piloté par une de ces cartes et est le dernier nœud du réseau. Il avancera toujours en ligne droite, le but ici n'est pas de le piloter de manière plus complexe.

Trois modes de fonctionnement sont à prévoir :

- **Mode normal :** aucune perturbation n'est observée dans le réseau, tout se passe normalement. Le robot avance selon les ordres du donneur d'ordres.
- **Mode dégradé :** dans ce mode, le robot avance à vitesse réduite. Le robot peut rester dans ce mode pendant une durée maximale de 30 secondes.
- **Mode d'arrêt d'urgence :** dans ce mode, le robot s'arrête et attend de recevoir les prochains ordres.

Ainsi, lorsqu'une perturbation a lieu (ligne coupée, parasites sur la ligne, etc) et que le temps de transmission de l'ordre dépasse le délai imparti ou que le robot ne reçoit plus d'ordres, il passe en mode dégradé. Si, pendant la durée du mode dégradé, la transmission est revenue à la normale, le robot repasse en mode normal. En revanche, si la transmission rencontre toujours des problèmes après les 30 secondes du mode dégradé, alors le robot passe en mode d'arrêt d'urgence et s'arrête. Pour revenir du mode dégradé ou mode d'arrêt d'urgence au mode normal, le robot attend de recevoir cinq ordres consécutifs dont le temps de transmission respecte le délai imparti. Ce délai imparti (ou deadline) est fixé selon la durée dans laquelle on souhaite accepter l'ordre. Le temps de transmission d'un ordre dépend bien évidemment de la taille de notre réseau : plus le robot est loin du donneur d'ordres, plus le nombre de nœuds relais est important et donc plus le temps de transmission est grand. Il faudra donc prendre en compte ce paramètre pour le choix de la deadline. A ce stade de définition du cahier des charges, nous ne pouvions pas connaître le temps de transmission d'un paquet

d’un nœud à l’autre et donc nous ne pouvions pas fixer de délai imparti cohérent : nous reviendrons sur ce point dans la partie « Aspect temps-réel ».

Voici un schéma qui permet de résumer le fonctionnement de notre application :

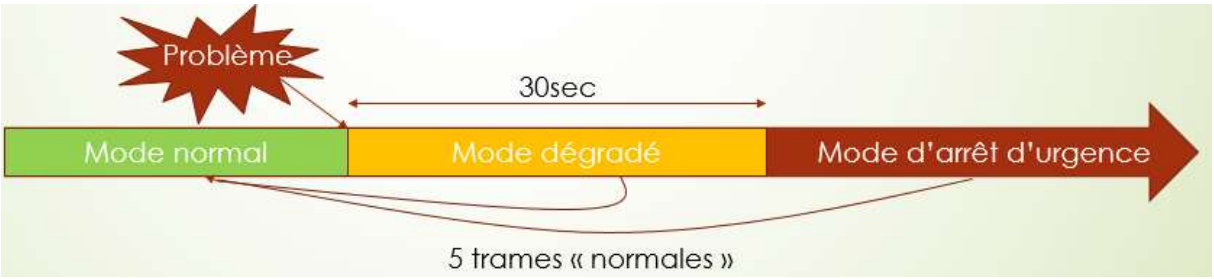


Fig. 2 : Schéma de fonctionnement de notre application

Maintenant que notre cahier des charges est en place, nous allons revenir sur le travail que nous avons accompli pour répondre à celui-ci.

II. Travail accompli

Nous allons ici exposer le travail que nous avons effectué sur ce projet. Nous commencerons d'abord par revenir sur les recherches que nous avons effectuées en début de projet. Puis, nous détaillerons la manière dont nous avons pris en main le système d'exploitation Riot ainsi que la conception et la caractérisation de notre module radio. Ensuite, nous reviendrons sur le premier réseau que nous avons mis en place : il s'agit d'un réseau statique composé de trois nœuds. Nous aborderons également l'aspect temps réel de notre application avant de finir par la manière dont nous avons intégré le protocole de routage RPL.

Il faut noter que nous avons tenté de mettre l'accent sur l'aspect gestion du projet tout au long de notre travail : nous avons régulièrement organisé des réunions avec nos tuteurs, et un compte-rendu a été rédigé pour chacune d'entre elles. Nous avons également utilisé un outil collaboratif (Google Drive) afin de centraliser notre travail et d'avancer plus efficacement et avons tenté de suivre le calendrier prévisionnel fixé.

1. Recherches bibliographiques

Notre sujet étant assez ouvert, la première phase de notre projet consistait à réaliser des recherches bibliographiques sur les réseaux sans fils temps-réel afin de faire un bref état de l'art. Le but de ces recherches est, notamment, de répondre aux questions suivantes :

- Existent-ils des réseaux sans fils temps-réel ?
- Si oui, lesquels ? Quelles sont leurs limites ? Comment fonctionnent-ils ?
- Si non, quelles sont les difficultés connues et que l'on peut rencontrer lorsqu'on souhaite mettre en place un tel réseau ?

Malheureusement, nous n'avons pas eu beaucoup de résultats, que ce soit sur Internet ou encore à la bibliothèque universitaire. Ceci étant simplement dû au fait que peu de travaux ont été effectués sur ce sujet. Néanmoins, nous avons trouvé un article très intéressant qui propose un protocole temps réel pour un réseau linéaire (cf. rubrique « Bibliographie » : Proposition et validation formelle d'un protocole MAC temps réel pour réseaux de capteurs linéaires sans fils). Même s'il s'agit ici d'un protocole MAC et d'un réseau linéaire, cet article nous a permis de comprendre plusieurs concepts et de mieux appréhender la suite du projet. Ces recherches, ainsi que les différentes réunions avec nos tuteurs, nous ont permis de définir l'application donnée ci-dessus. Nos recherches étaient ensuite

essentiellement portées sur le matériel à utiliser pour réaliser cette application (matériel que nous avons déjà défini dans la partie précédente de ce rapport).

2. Prise en main de Riot OS et tests sur STM32F4discovery

Comme nous avons pu le dire, nous avons décidé d'utiliser le système d'exploitation Riot qui fournit un support temps-réel. Ce système nous étant inconnu jusqu'ici, il nous a donc fallu prendre en main ce système, faire des tests sur notre ordinateur puis sur les cartes STM32F4 discovery. Nous allons revenir ici sur les tests que nous avons pu réaliser.

2.1. Prise en main de Riot Os

Nous avons récupéré le code source de Riot depuis Github (code en open source). Pour compiler un projet Riot, il a d'abord fallu installer les chaines de compilation et de débogage ainsi que d'autres paquetages nécessaires. Avant de pouvoir passer aux tests sur la carte de développement, nous avons décidé de tester un des codes exemples proposé par Riot. Ce code, lorsqu'il est compilé et exécuté permet d'afficher dans la console un message de bienvenue : c'est une sorte de « Hello World ». Cet exemple très basique nous a permis de savoir comment compiler un projet Riot.

Dès lors, nous pouvions passer aux tests sur la carte STM32F4discovery.

2.2. Programmes de test utilisant Riot pour STM32F4discovery

Là encore, le premier test que nous avons réalisé était d'allumer une LED de la carte. Pour réaliser ce test simple, il a fallu chercher dans Riot les fonctions qui permettent de gérer, entre autres, ces LEDs. L'avantage d'utiliser un système d'exploitation est de nous faciliter la tâche car des fonctions sont déjà proposées et permettent de faire beaucoup de choses. L'inconvénient est qu'il faut connaître ces fonctions, qu'il faut les trouver, savoir où chercher, et cela représente au début beaucoup de temps.

Une fois ce test passé, nous avons décidé d'inclure un timer dans l'exemple. Le but est de faire clignoter une LED toutes les secondes. Ce test était intéressant à réaliser car nous savions que les timers seraient utiles pour notre application finale. Nous avons eu quelques difficultés à trouver les bonnes fonctions et à comprendre leur bon fonctionnement, mais nous sommes finalement parvenus à réaliser cette fonctionnalité.

Enfin, nous avons tenté de piloter les servomoteurs depuis la carte de développement. Les servomoteurs dont nous disposons sont des servomoteurs à rotation continue. Nous pouvons donc contrôler la vitesse de rotation grâce à un signal PWM. Là encore, Riot propose des fonctions qui permettent de créer ce signal. Une fois ces fonctions trouvées et comprises, nous avons pu contrôler la vitesse de rotation de nos servomoteurs.

Nous avons ainsi réalisé plusieurs tests qui nous ont permis de prendre en main la carte STM32F4discovery sous Riot. Nous savons contrôler nos servomoteurs, et cela sera indispensable pour le bon fonctionnement de notre application. Dès lors, nous pouvions nous pencher sur la partie communication entre deux cartes STM32F4discovery.

2.3. Communication entre deux cartes STM32F4discovery

En attendant que notre carte électronique pour le module radio soit opérationnelle, nous avons décidé de faire un premier test sur notre machine Linux, en « simulant » les nœuds de notre réseau grâce à des interfaces virtuelles connectées entre elles grâce à un bridge. Le but de ce test était de mettre en place une architecture réseau utilisant RPL constituée d'un père et de deux fils. Pour réaliser cette architecture, nous nous sommes appuyés sur un exemple proposé par Riot. Nous sommes parvenus à afficher la table de routage où l'on a bien un parent et deux fils. Lorsqu'on éteint le parent, nous ne pouvions plus communiquer entre les deux fils. Ensuite, nous avons tenté de modifier cette architecture pour avoir un réseau linéaire (soit un père, un fils et un petit-fils). Malheureusement, nous ne pouvions pas faire beaucoup plus de tests intéressant sur ce point sans nos modules radio.

Suite à cela, nous avons décidé de tester une communication entre deux cartes STM32F4discovery grâce aux UARTs qu'intègrent ces dernières. Le but était le suivant : lorsqu'on appuie sur le User Button de la première carte, celle-ci transmet un caractère à la seconde carte qui génère ainsi un signal PWM afin de faire tourner le servomoteur à une certaine vitesse. Si on relâche ce bouton, un autre caractère est envoyé et cela a pour but de stopper le servomoteur. Nous sommes parvenus à réaliser cette application.

Nous avons donc résumé l'ensemble des tests que nous avons pu effectuer. De manière générale, la plupart d'entre eux nous paraissaient simple à priori, mais ils se sont révélés plus compliqués que prévu car il a fallu faire des recherches sur Riot, sachant qu'il n'y a pas beaucoup de support sur ce système étant donné qu'il est assez récent.

Nous allons maintenant passer à une autre partie très importante pour notre projet : la réalisation de la carte électronique pour le portage radio et sa caractérisation.

3. Conception de la carte pour le portage radio

Afin de permettre la liaison sans fils de nos nœuds, nous avons été amenés à utiliser des modules de communication radio. Nous avons décidé de développer ce module autour de la puce AT86RF231, car le système d'exploitation Riot intègre le pilote de celle-ci.

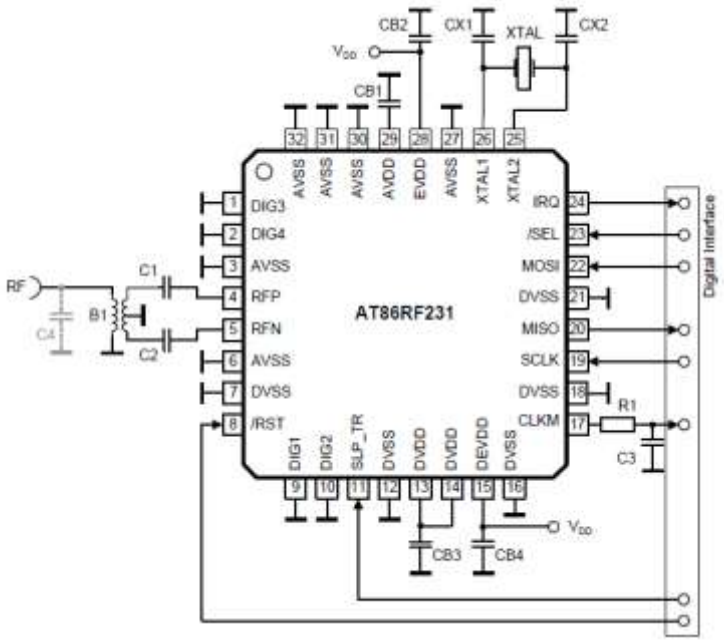


Fig. 3 : Schéma de réalisation du module radio présent sur la datasheet de l'AT86RF231

3.1. Réalisation du schematic sous Eagle

Pour la réalisation de cette carte, nous avons commencé par dessiner, sur Eagle, les différents packages nécessaires. Concernant les entrées/sorties numériques de la carte, nous avons respecté le schéma ci-dessus et avons ajouté une entrée d'alimentation 3,3V et la masse.

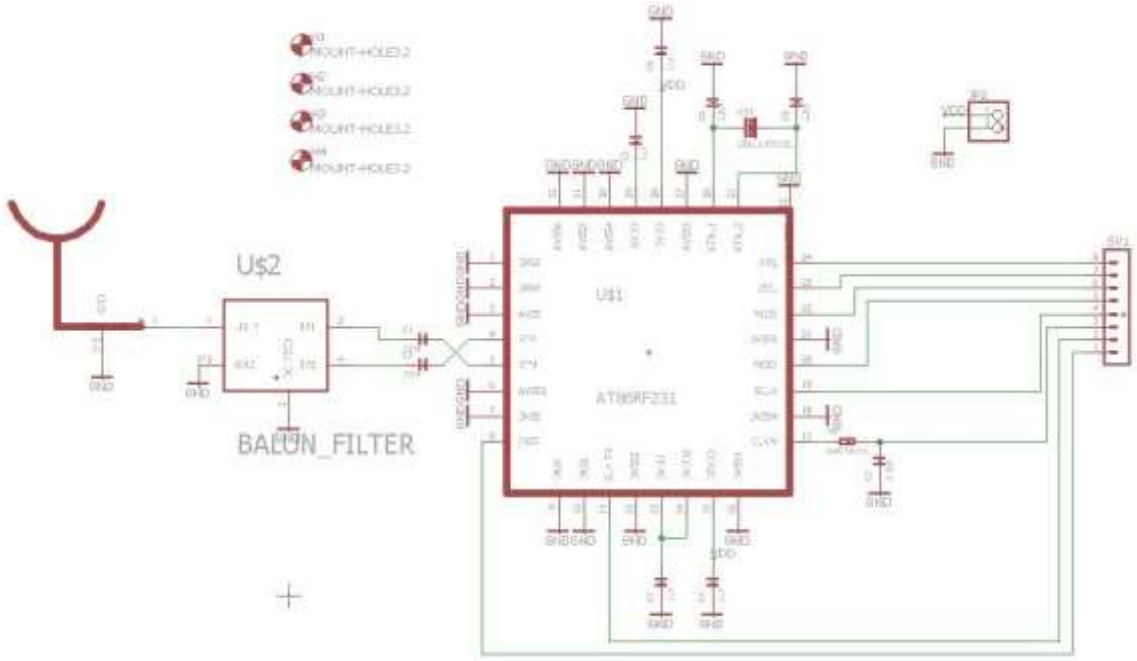


Fig.4 : Schématic du module radio réalisé sur Eagle

3.2. Réalisation du routage

La réalisation du routage était la partie la plus technique pour cette carte et plus particulièrement la transmission haute fréquence. La bande utilisée pour les liaisons de la norme 802.15.4 est à 2,45 GHz ce qui correspond aux hyperfréquences. C'est pourquoi le bloc ci-dessous, constitué d'une antenne, d'un balun et de deux condensateurs, a dû être étudié et dimensionné.

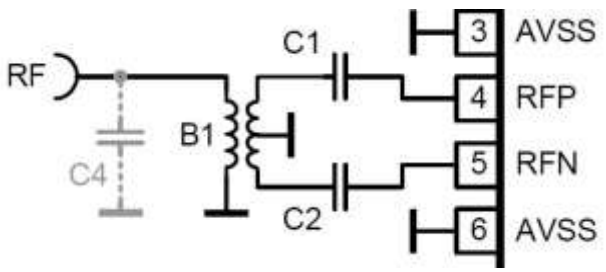


Fig.5 : Bloc à hautes fréquences du module radio

Pour cela rappelons quelques éléments de bases nécessaires à notre étude :

- L'impédance d'entrée Ze d'un quadripôle est l'impédance équivalente qui, mise au borne d'un générateur, donne la même intensité et la même tension.

- L'impédance de sortie Zs d'un quadripôle est l'impédance en série d'un générateur vu par la une résistance Ru en sortie.

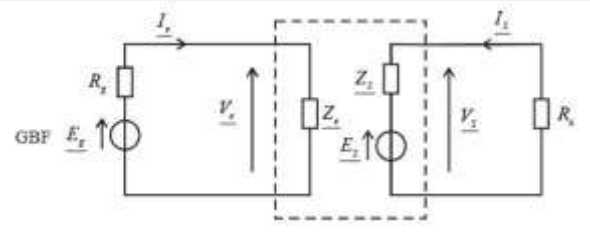


Fig.4 : schéma représentant un impédance d'entrée et de sortie

- « Le principe fondamental de l'adaptation d'impédance est le suivant : en connectant sur une charge de résistance R, une ligne de transmission d'impédance caractéristique R, on retrouvera à l'autre extrémité de la ligne la même résistance R. Autrement dit, la source et la charge de résistance R seront « adaptées » si la ligne qui les relie possède une impédance caractéristique de même valeur. L'adaptation sera conservée quelle que soit la longueur de la ligne. »
(https://fr.wikipedia.org/wiki/Adaptation_d'imp%C3%A9dances)

- L'impédance caractéristique est la résistance vue par le générateur aux premiers instants de la transmission. Elle dépend uniquement des caractéristiques de la ligne.

- Le coefficient de réflexion est un nombre sans dimensions qui indique la quantité d'énergie réfléchie en bout ou en début de ligne. Il est compris entre -1 et +1. Il est égal à 0 si la ligne est adaptée.

- Les feeders sont les lignes qui alimentent les antennes. Ils doivent véhiculer l'énergie jusqu'à l'antenne avec un minimum d'onde réfléchie. Ils doivent donc avoir comme valeur d'impédance caractéristique la valeur de l'impédance de l'antenne.

L'antenne choisie est une antenne CMS qui est utilisée pour l'émission et la réception. Selon la datasheet, son impédance est de 50 Ohms et il en est de même pour l'impédance de sortie du balun. Pour adapter ces deux éléments, il nous fallait une ligne d'impédance caractéristique 50 Ohms. Pour une ligne microstrip, l'impédance caractéristique dépend de ses dimensions et du matériau isolant.

De nombreuse formules sont établies dans la littérature pour calculer cette impédance, nous avons choisi le modèle présent dans logiciel AppCAD.

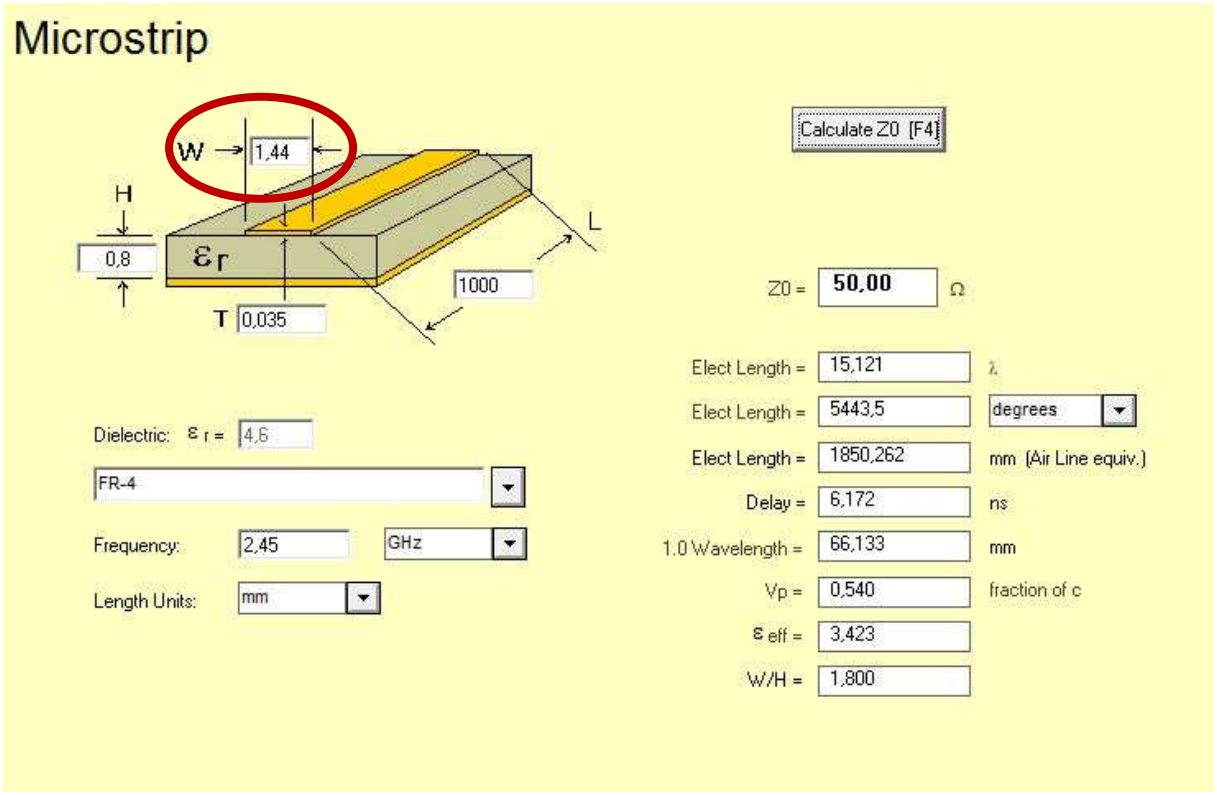


Fig.6 : Résultat du calcul d'impédance caractéristique

A Polytech, il est possible de réaliser des cartes avec des pistes d'épaisseur 35 um avec un diélectrique verre-epoxy FR4 d'épaisseur 0,8 mm. En utilisant ces informations, nous avons abouti à une largeur de piste égale à **1,44 mm**.

Le rôle du balun est de transformer une impédance symétrique en une impédance asymétrique et vice versa. Le balun réalise également une fonction d'adaptation d'impédance entre les ports RFP et RFN et l'antenne. Pour ne pas trop modifier l'impédance différentielle, nous avons décidé de réaliser des pistes de longueur faible, assez large mais surtout des lignes symétriques. Enfin les condensateurs à l'entrées des ports RFP/RFN sont utilisés pour supprimer la composante continue de l'entrée RF provenant de l'antenne.

Afin d'améliorer le transport de l'énergie jusqu'à l'antenne, il est conseillé de rajouter de nombreux vias pour que les plans de masse soient au même potentiel. De plus, l'ajout de vias permet d'atténuer l'effet capacitif des plans de masse situés de part et d'autres du diélectrique. Enfin, pour assurer le bon

fonctionnement de l'oscillateur à quartz, il est préférable de l'isoler du plan de masse en TOP afin d'éviter que l'on fonctionne en mode coplanaire. Nous avons également ajouté des trous afin de permettre le vissage des cartes.

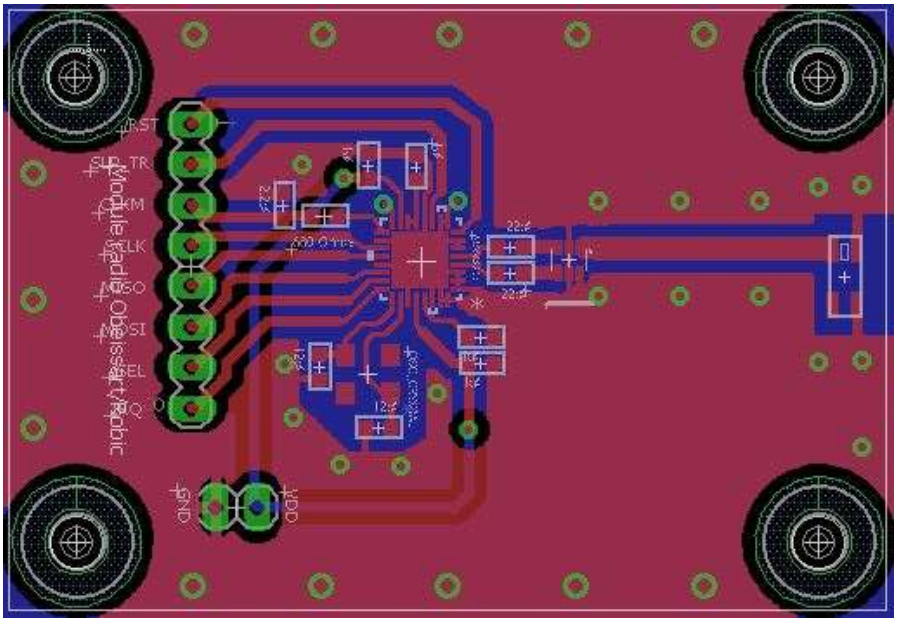


Fig.7 : Routage du PCB du module radio

3.3. Problèmes rencontrés et résolution

Une fois la carte gravée et les composants soudés, nous avons pu tester notre carte. Cependant, la communication SPI entre le module radio et la carte STM32F4 discovery n'était pas fonctionnelle. Nous nous sommes alors rendus compte que l'oscillateur n'oscillait pas, sans trouver la cause de ce dysfonctionnement. Nous avons tenté de remplacer l'oscillateur par un signal équivalent grâce à un GBF et avons observé que la communication était alors fonctionnelle : le problème venait donc bien de l'oscillateur. Après quelques tests, nous nous sommes aperçus que la datasheet fourni par le vendeur n'était pas correcte : le schéma des pins était inversé. De plus, nous avons détecté un court-circuit entre deux pins de la puce RF qui était dû au fait que les pistes étaient proches et que la machine n'était pas assez précise pour le détecter. Une fois ces problèmes réglés, la communication était fonctionnelle et nous parvenions à récupérer plusieurs paramètres liés à la puce (comme l'identifiant du vendeur par exemple).

Nous pouvions passer à la caractérisation de notre module radio.

3.4. Caractérisation du module radio

Par défaut, notre puce AT86RF231 a une fréquence de transmission de données de 250kb/s. A cette fréquence, sa sensibilité de réception est de -101dBm. Nous connaissons également le gain de notre antenne, qui est de -2dBi. Nous avons réalisé un test permettant de connaître, en pratique, la distance maximale avec laquelle 2 nœuds peuvent discuter. Pour cela, nous nous sommes placés dans un couloir (en ligne droite) et avons tenté de placer les deux antennes l'une face à l'autre. Nous avons éloigné petit à petit l'un des nœuds jusqu'à perdre la communication puis avons mesuré la distance. Nous avons programmé la puce avec une puissance d'émission maximale, soit 3dBm. Nous avons obtenu une distance maximale de 30 mètres environ.

En ce qui concerne la consommation du module, nous avons relevé une consommation relativement constante d'environ 12.5 mA.

4. Mise en place d'un réseau statique

Une fois que nos modules radio étaient opérationnels, nous avons pu nous atteler à la réalisation de notre application. Nous avons alors choisi d'avancer étape par étape : avant de mettre en place un réseau dynamique qui intègre le protocole de routage RPL, nous avons décidé de mettre en place un réseau statique constitué de trois nœuds dont un donneur d'ordres, un relais, et le nœud robot.

Pour réaliser ce réseau, il a fallu décrire les routes empruntées par les paquets dans une table FIB (**F**orwarding **I**nformation **B**ase ou table MAC). Il s'agit d'une table qui mappe les adresses MAC vers les ports : elle représente ainsi le chemin que prendra le paquet. Chaque nœud possède sa propre table FIB. De plus, pour avoir une adresse routable, nous avons défini pour chaque nœud une adresse globale du type dead:beef::**HWaddr** (où **HWaddr** représente l'adresse mac courte du nœud). Notre réseau se présentait donc de la façon suivante :

Noeud_1[dead:beef::3402] -----> Noeud_2[dead:beef::3762] -----> Noeud_3[dead:beef::3766]

Il suffisait ensuite de remplir la table FIB de chaque nœud afin que tous les paquets transitent par le nœud intermédiaire (nœud 2). Par exemple, voici la table FIB de nœud 1 :

Noeud_1

```
# > fibroute
# Destination                    Flags           Next Hop                       Flags      Expires         Interface
# dead:beef::3766                0x00000000 H    fe80::3634:5110:3473:3762      0x00000000 NEVER           7
# dead:beef::3762                0x00000000 H    fe80::3634:5110:3473:3762      0x00000000 NEVER           7
```

Fig. 8 : Table FIB du donneur d'ordres

Cette configuration nous permettait donc d'obtenir un réseau statique linéaire permettant au donneur d'ordres (nœud 1) d'envoyer des paquets à la cible (nœud 3) via le nœud 2.

Ensuite, afin de nous rapprocher de notre application, nous avons mis en place une communication UDP entre les nœuds 1 et 3. Pour cela, nous avons créé un socket (ou interface de connexion) UDP entre ces deux nœuds. Le nœud 1 fait office de client qui envoie des messages toutes les secondes alors que le nœud 3 fait office de serveur et ajuste son comportement selon les ordres qu'il reçoit.

Nous pouvions dès lors implémenter partiellement notre application, c'est-à-dire sans notions temps réel pour le moment. Pour ce prototype, nous avons utilisé des timers qui nous permettaient de mettre en place les trois modes de fonctionnement du robot.

Le donneur d'ordre (nœud 1) envoie des paquets UDP contenant les caractères "go" toutes les secondes sur le socket. Le nœud 3 (lié au robot) peut alors se comporter de plusieurs façons :

- S'il reçoit le message « go », il active les moteurs, et réinitialise les timers qui permettent de déclencher les modes dégradé et arrêt d'urgence afin que ceux-ci ne se déclenchent pas.
- S'il ne reçoit pas le message go dans un délai de 5 secondes, le nœud entre en mode dégradé : cela a pour effet de ralentir la vitesse du robot et de déclencher le timer qui permet le passage au mode d'arrêt d'urgence. Si dans les 10 secondes suivantes le robot ne reçoit pas le message, la fonction d'arrêt d'urgence est enclenchée et le robot s'arrête.
- Si le robot est en mode dégradé ou arrêt d'urgence, il repasse en mode normal dès qu'il reçoit le message go.

Cet exemple ne prend pas en compte les contraintes temps réel. Néanmoins, grâce à lui, nous étions parvenus à mettre en place les différents modes de fonctionnement du robot.

Détaillons maintenant la façon dont nous avons intégré les notions de temps réel dans notre travail.

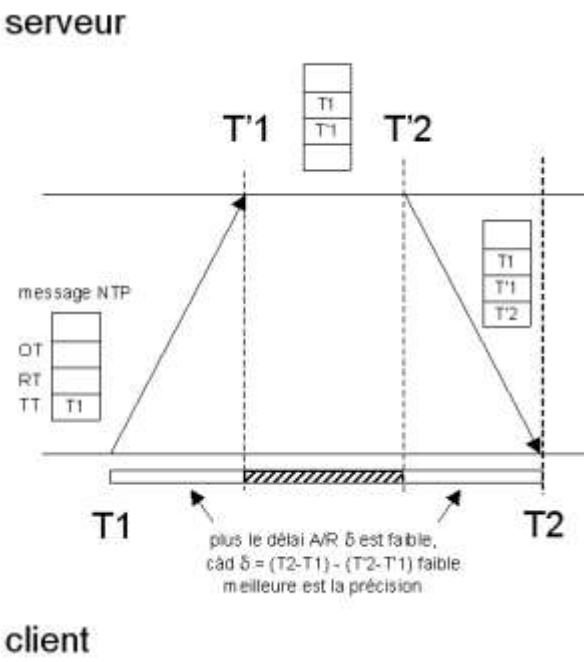
5. Aspect temps-réel

L'aspect temps réel est essentiel dans notre projet. Nous avons décidé de fixer notre contrainte temps réel sur le temps de transmission de l'ordre à partir du donneur d'ordres jusqu'au robot. Pour réaliser cette contrainte, il est nécessaire que le système d'exploitation utilisé fournisse un support temps réel d'où l'intérêt de Riot : ce dernier possède une latence d'interruption très faible (environ 50 cycles d'horloge : la STM32F4 discovery étant cadencée à 168 MHz, cela représente une latence d'environ

300 ns) et implémente un ordonnanceur qui se base sur les priorités. Voyons comment nous avons mis en place la contrainte désirée.

5.1. Mise en place d'un serveur sntp

Tout d'abord, il nous fallait connaître le temps de transmission d'un paquet. Il était donc nécessaire de synchroniser l'horloge du robot avec celle du donneur d'ordre. Pour cela, nous utilisions le protocole SNTP (**S**imple **N**etwork **T**ime **P**rotocol) : ce protocole permet de synchroniser, via un réseau informatique, l'horloge locale de nœuds sur une référence d'heure. Dans Riot OS, cette référence est 1900-01-01 00:00 UTC. Dans notre cas, nous avons choisi le modèle client-serveur : le premier nœud (celui qui donne les ordres) sera le serveur alors que le dernier nœud (celui qui commande les servomoteurs du robot) sera le client. Pour comprendre comment mettre en place ce protocole, nous avons utilisé ce schéma :



A l'instant T1 : lorsque le client émet son message pour interroger le serveur sur l'heure courante, il envoie un message dans lequel il renseigne le champ TT avec l'heure courante T1 indiquée par son horloge locale ;

A l'instant T'1 : lorsque le serveur reçoit le message, il complète aussitôt le champ RT du message avec l'heure courante T'1 indiquée par son horloge locale, et recopie le champ TT dans le champ OT ;

A l'instant T'2 : lorsque le serveur émet son message de réponse, il complète le champ TT du message avec l'heure courante T'2 indiquée par son horloge locale

A l'instant T2 : lorsque le client reçoit le message de réponse, il note aussitôt l'heure T2 de réception indiquée par son horloge locale.

Le client peut alors calculer le délai aller/retour δ de ces 2 messages ainsi que l'écart θ entre son horloge locale et celle du serveur grâce aux formules suivantes :

$$\delta = (T2 - T1) - (T'2 - T'1)$$

$$\theta = \frac{T'1 + T'2}{2} - \frac{T1 + T2}{2}$$

Ainsi, lorsque le donneur d'ordre envoie un message, il intègre dans ce message l'heure à laquelle il l'a envoyé. Lorsque le robot reçoit ce message, il note l'heure et peut connaître le temps de transmission grâce à la formule suivante :

$$Temps\,de\,transmission\ =\ heure\,de\,reception\ -\ heure\,d'emission\ +\ offset\,entre\,les\,deux\,horloges$$

Nous avons ensuite fait quelques tests pour connaître l'influence de la taille du paquet à transmettre et de la distance émetteur-récepteur sur ce temps de transmission (cf. Annexe x et y). Ces tests ont été effectués entre deux nœuds et comme on peut le constater, plus la taille du paquet est grande, plus le temps de transmission est long. En revanche, comme on pouvait s'y attendre, la distance émetteur/récepteur n'a pas d'influence sur celui-ci. Pour fixer une deadline cohérente avec le réseau, il faut prendre en compte le nombre de nœuds. Ainsi, on peut avoir une idée du temps de transmission dans un réseau avec la formule suivante :

$$Temps\,de\,transmission\,entre\,N\,noeuds\ =\ (N-1)\times Temps\,de\,transmission\,entre\,2\,noeuds$$

Nous pouvions dès lors mettre en place notre première application : un réseau statique temps réel composé de trois nœuds.

5.2. Première application : réseau statique temps-réel

Nous avons intégré au réseau statique que nous avions mis en place la contrainte temps réel ci-dessus. Ainsi, l'application se déroulait de la façon suivante :

- Le robot comment par synchroniser son horloge avec le donneur d'ordres.
- Le donneur d'ordres envoie un message au robot toutes les secondes, ce message passant par le nœud relais avant d'atteindre sa destination.
- A la réception de l'ordre, le robot calcule son temps de transmission. Plusieurs cas sont alors possibles :
    - Si ce temps est inférieur à la deadline, le robot est en mode normal : il avance normalement
    - Sinon, le robot passe en mode dégradé et le robot avance à vitesse réduite.
    - Après 30 secondes, si le robot est toujours en mode dégradé, il passe en mode arrêt d'urgence et s'arrête.
    - Le robot utilise un compteur pour connaître le nombre d'ordres consécutifs dont le temps de transmission respecte la deadline. Lorsque celui-ci atteint 5, le robot repasse en mode normal

Autrement dit, nous avions mis en place une application qui répondait au cahier des charges. Cependant, notre réseau était statique et nous souhaitions nous rapprocher d'un véritable réseau de capteurs, qui intègre généralement un routage dynamique. Nous avons alors intégré RPL à notre application.

6. Intégration du protocole de routage RPL : application finale

6.1. Introduction à RPL

Le protocole RPL (**R**outing **P**rotocol for **L**ow-Power and Lossy Networks) a été introduit pour l'internet des objets et est adapté aux réseaux de nœuds à faible débit et à pertes importantes. La base de ce protocole est la construction de graphiques acyclique appelés DAG (permettant d'éviter les boucle). Tous les nœuds feuilles sont contenus dans des chemins terminant vers un ou plusieurs nœuds racines. Un Destination-Oriented DAG (DODAG), est un DAG avec une racine unique que l'on appelle le DODAG root. Une instance RPL peut contenir un ou plusieurs DODAG. Le rang défini pour chaque nœud traduit la distance avec le DODAG root. Autrement dit, plus un nœud est loin du DODAG root plus son rang sera élevé. L'intérêt de ce protocole est qu'il permet de réaliser un routage dynamique entre nos nœuds. Il peut ainsi définir pour un nœud quel parent lui serait le plus approprié (selon des données traduisant la qualité de transmission) pour réaliser une communication afin d'atteindre une destination. Un nœud peut donc avoir plusieurs parents permettant alors de choisir différents chemins possibles pour réaliser une communication entre deux nœuds distants.

Pour mieux comprendre ce protocole, il est intéressant de comprendre comment se forme un DODAG. L'interaction entre les différents nœuds se fait au travers de RPL Control Messages qui sont des messages ICMPv6. Il en existe 3 types :

- **DAG Information Object (DIO)** - transmet des informations qui permettent à un nœud de découvrir une instance RPL, d'apprendre ses paramètres de configuration et de choisir des parents dans le DODAG
- **DAG Information Solicitation (DIS)** - sollicite un DIO à partir d'un nœud RPL
- **Destination Advertisement Object (DAO)** - utilisé pour propager les informations de destination vers le haut le long du DODAG.

Pour construire un DODAG, chaque nœud envoie périodiquement des messages DIO en multicast. De plus, chaque nœud peut utiliser des messages DIS pour solliciter un DIO. La fréquence d'envoi des

messages DIO dépend de la stabilité ou de la détection de problème de routage. Les nœuds écoutent les messages DIO et utilisent ces informations pour joindre un DODAG. En se basant sur les informations contenues dans les DIOs, un nœud peut ainsi choisir ses parents et réduire au maximum le nombre d'intermédiaire avec le DODAG root.

Afin de reconstruire le DODAG lors de problèmes de connexions, RPL effectue des DODAG Repair. Il réalise alors une opération de réparation globale en incrémentant le numéro de version du DODAG déclenchant ainsi une nouvelle version de DODAG. Les nœuds de la nouvelle version du DODAG peuvent choisir une nouvelle position dont le rang n'est pas limité par leur rang dans l'ancienne version du DODAG. Le message DIO spécifie les paramètres nécessaires tels que configurés et contrôlés par la stratégie du DODAG root.

Revenons maintenant sur la manière dont nous avons intégré ce protocole dans notre application.

6.2. Intégration dans l'application finale

Pour intégrer RPL dans notre application, il suffit d'ajouter dans le makefile de cette application les modules correspondant au protocole de routage :

```
USEMODULE += gnrc_rpl
USEMODULE += auto_init_gnrc_rpl
```

L'auto_init permet d'initialiser RPL sur un nœud et l'envoi périodique des messages DIO et également d'envoyer un message DIS pour chaque nœud. Mais pour que notre DODAG se construise, il a fallu définir un nœud comme DODAG root afin qu'il puisse alimenter les DIOs et ainsi permettre aux autres nœuds d'intégrer le DODAG : Riot propose la fonction **gnrc_rpl_root_init** qui permet de faire cela. Nous sommes ainsi parvenus à former l'architecture suivante:

Noeud_maître[dead:beef::3402] -----> ...[RPL_NODES]... -----> Noeud_robot[dead:beef::341e]

Nous avons ensuite réalisé quelques tests en disposant les nœuds de différentes manières dans l'espace pour observer le comportement du réseau. Nous nous sommes rendus compte que le nœud robot ne choisit pas forcément comme parent le nœud avec lequel il a une meilleure liaison. En effet, dans le système RIOT, le protocole RPL est partiellement intégré et ces observations nous ont permis de le démonter. Selon la RFC 6550, la définition du rang d'un nœud est le reflet de la distance de ce nœud par rapport au DODAG root (ici le donneur d'ordres). Ce rang est donc défini à partir d'un minimum hop rank increase (mhri). Dans RIOT, ce MHRI (d'une valeur de 256 correspondant au rang

du DODAG root) est ajouté à chaque saut pour calculer le rang. En revanche, RPL permet de mettre en place des métriques qui permettent à un nœud de choisir son parent préféré : or, cela n'est pas implémenter sous RIOT. Afin d'améliorer cela, nous avons décidé de calculer le rang d'un nœud à partir de paramètres qui nous renseignent sur la qualité de transmission du nœud avec ses parents. Chaque fois qu'un nœud reçoit un message, deux indicateurs de la qualité de transmission peuvent être récupérés : le LQI (Link Quality Indicator) et le RSSI (Received Signal Strength Indication). Nous avons décidé d'inclure ces paramètres dans les messages DIO.

A l'initialisation de RPL sur chaque nœud, un thread d'écoute de messages est lancé. Lorsque celui-ci reçoit un paquet ICMP de type DIO, nous récupérons les données de qualité de transmission et les ajoutons au DIO. La fonction de traitement de ce message est ensuite lancée. Cette fonction nous permet, dans le cas où notre nœud n'est associé à aucune instance RPL, d'ajouter l'émetteur dans la liste des parents du nœud actuel. Nous pouvons ensuite intégrer les données de qualités du DIO dans la structure de ce parent. Ensuite, le nœud choisi son parent préféré (celui de rang le plus faible) et calcule son propre rang à partir de la formule suivant :

$$Rang = rang\ de\ base + (rang\ de\ base - \frac{RSSI + LQI}{10})$$

Le rang de base est celui du DODAG root (soit 256). Ainsi, plus la qualité de la liaison entre le nœud et son parent est bonne, plus le RSSI et le LQI sont grands (avec $0 \leq RSSI \leq 28$ et $0 \leq LQI \leq 255$), donc plus le rang du nœud sera petit ce qui signifie que le nœud est proche de son parent. Nous divisons le dernier membre par 10 pour éviter qu'un nœud ne possède un rang plus faible que son parent (ce qui est impossible sous RPL). Voyons maintenant le résultat qu'il est possible d'obtenir sur un des nœuds grâce à notre modification :

```
rpl
# rpl
# instance table:       [X]
# parent table: [X]     [X]     [ ]
#
# instance [1 | Iface: 6 | mop: 2 | ocp: 0 | mhri: 230 | mri 0]
#       dodag [dead:beef::3402 | R: 715 | OP: Router | PIO: on | CL: 0s | TR(I=[8,20], k=10, c=0, TC=14s, TI=18s)]
#               parent [addr: fe80::3634:5110:3473:3762 | rank: 485 | lifetime: 38s]
#               parent [addr: fe80::3734:510b:330b:340a | rank: 486 | lifetime: 53s]
fibroute
# > fibroute
# Destination                             Flags           Next Hop                                Flags      Expires
# ::                                      0x00000000 H    fe80::3634:5110:3473:3762               0x00000001 28.7475
```

Fig. 9 : Exemple d'informations d'un nœud dans un réseau RPL

On peut voir que le nœud (d'un rang de 715) possède deux parents de rang 485 et 486. Et on remarque que seul le parent de rang le plus faible est intégré dans la table FIB du nœud : autrement dit, c'est avec ce parent que le nœud discutera.

Nous sommes ainsi parvenus à intégrer RPL dans notre application et à prendre en compte la qualité de transmission dans le choix du parent préféré afin d'améliorer la transmission des messages entres les différents nœuds du réseau. Il nous restait simplement à réaliser le robot qui servirait de vitrine à notre application.

6.3. Réalisation du robot

Le robot est composé de deux servomoteurs, d'une carte STM32F4 discovery, d'un module radio, d'une batterie et de quatre LEDs dont nous allons expliquer le fonctionnement. Nous l'avons réalisé avec du plexiglas à l'aide de la découpeuse laser :

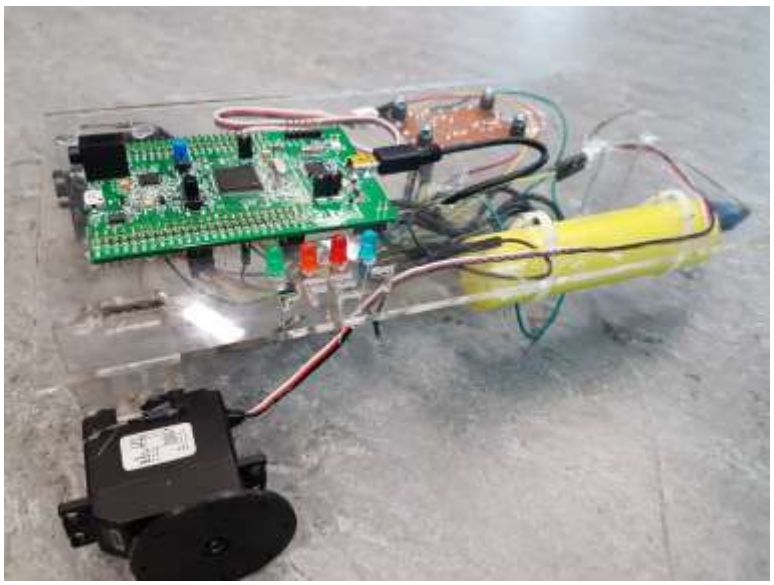


Fig. 10 : Robot utilisé pour notre application

Les LEDs permettent de nous renseigner sur l'état du robot :

- LED verte : signale le mode normal
- LED orange : signale le mode dégradé
- LED rouge : signale le mode arrêt d'urgence
- LED bleue : indique que le robot fait partie du DODAG

Nous avons également ajouté des LEDs pour le donneur d'ordre :

- LED bleue : indique que le DODAG est en place
- LED verte : lorsque le robot en mode normal, celle-ci clignote à chaque fois qu'un message est envoyé (toutes les deux secondes)
- LED orange : lorsque le robot est en mode dégradé ou arrêt d'urgence, le donneur d'ordres augmente sa fréquence d'envoi de messages (il en envoie toutes les secondes) et cette LED clignote à chaque envoi de message

Enfin, chaque relais possède une LED bleue pour indiquer qu'il fait partie du DODAG et une LED verte qui clignote quand un message transite par ce relais.

Nous avons ainsi réussi à mettre en place notre application finale. Nous allons maintenant faire un bilan de notre travail.

III. Bilan du projet

Nous allons ici faire un bilan du travail que nous avons effectué pendant ce projet.

Tout d'abord, il faut noter qu'il a été assez difficile pour nous de définir clairement le sujet. Nous nous sommes un peu dispersés dans nos recherches préliminaires et nous ne parvenions pas à bien séparer le travail. Toutefois, les différentes réunions avec nos encadrants nous ont permis de répondre à nos interrogations et à améliorer notre manière de travailler. Le fait de mettre en place un calendrier prévisionnel nous a également permis de respecter certaines échéances, et même s'il a fallu le modifier du fait des problèmes rencontrés, il représentait notre « guide » à suivre.

Nous avons décidé de ne pas tenter de mettre en place directement l'application finale, c'est-à-dire un réseau dynamique avec des notions temps réel. En effet, nous avons séparé l'aspect temps réel de l'aspect réseau et avons avancé pas à pas. Cela nous a permis d'avancer de manière efficace et de parvenir finalement à une solution fonctionnelle.

Nous savions que ce sujet était très ouvert et qu'il serait difficile, et nous savions aussi qu'il nous apporterait beaucoup de compétences. Ainsi, nous avons appris à designer une carte haute fréquence, à programmer sous un système d'exploitation inconnu jusque-là, et à mettre en place une méthode de gestion de projets. L'un de nos objectifs était également de permettre à celui qui reprendrait notre travail de le comprendre assez facilement et rapidement. Aussi, nous avons écrit un petit « guide utilisateur » pour prendre en main Riot rapidement, nous avons mis tous nos codes sources sur une archive GIT et enfin, nous avons édité un compte-rendu de chaque réunion avec nos encadrants dans le but de retracer le parcours que nous avons réalisé.

Bien sûr, nous n'avons pas pris toutes les problématiques en compte. Beaucoup d'améliorations peuvent donc être apportées à notre solution comme par exemple la gestion de l'énergie (dans notre cas, notre module est toujours actif) et la sécurisation des données (pour le moment, nos données circulent en clair le long de notre réseau).

Conclusion

Au terme de ce projet auquel nous avons consacré la majeure partie de notre temps, nous avons respecté les objectifs fixés en début de projet, à savoir la définition d'un cahier des charges et la mise en place d'une solution permettant d'y répondre. Nous avons eu plusieurs difficultés différentes car notre projet était multidisciplinaire et que plusieurs outils utilisés dans ce projet nous étaient inconnus auparavant. C'est pourquoi nous sommes heureux d'être parvenus à notre résultat.

Comme nous l'avons dit, ce projet est très dense et beaucoup d'améliorations peuvent être apportées. Nous pensons qu'il pourra être repris par d'autres étudiants pour l'étoffer.

Bibliographie

[1] T. WATTEYNE. « Proposition et validation formelle d'un protocole MAC temps réel pour réseaux de capteurs linéaires sans fils ». INSA Lyon, [En ligne]. Adresse URL : https://twatteyne.files.wordpress.com/2013/05/watteyne05masters.pdf

[2] V. GUNGOR, Ö. AKAN, I. AKIYLDIZ. « A Real-Time and Reliable Transport Protocol for Wireless Sensor and Actor Networks ». [En ligne]. Adresse URL : https://bwn.ece.gatech.edu/papers/2008/j5.pdf

[3] Z. TENG, K. KIM. « A Survey on Real-Time MAC Protocols in Wireless Sensor Networks ». Gyeongsang National University, Jinju, Korea, [En ligne]. Adresse URL : http://file.scirp.org/pdf/CN20100200003_69512435.pdf

[4] T. HE, J. STANKOVIC, C. LU, T. ABDELZAHER. « SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks ». University of Virginia, Washington University in St Louis, [En ligne]. Adresse URL : http://www.cse.wustl.edu/~lu/papers/icdcs03_speed.pdf

[5] B.K. SINGH. « A novel real-time MAC layer protocol for wireless sensor network applications ». University of Windsor, [En ligne]. Adresse URL : http://ieeexplore.ieee.org/abstract/document/5189639/

[6] B. DUPOUY. « Introduction aux bus et réseaux temps réel ». Télécom ParisTech, [En ligne]. Adresse URL : http://perso.telecom-paristech.fr/~dupouy/pdf/TRAM/5-BusTR-INF342.pdf

[7] K.E. GHOLAMI. « La gestion de la qualité de service temps-réel dans les réseaux de capteurs sans fil ». HAL, [En ligne]. Adresse URL : https://tel.archives-ouvertes.fr/tel-01158110/document

[8] M.R. KHAN. « Performance and route stability analysis of RPL protocol ». KTH Electrical Engineering, [En ligne]. Adresse URL : https://pdfs.semanticscholar.org/0062/d562d45934fdd8102299e6abbc083053eb9e.pdf

[9] Y.Q. SONG. « Ordonnancement temps réel dans les réseaux ». Université de Lorraine, [En ligne]. Adresse URL : https://members.loria.fr/YQSong/publis/presentation_40ans_tr_yqs.pdf

[10] N. KROMMENACKER. « Aptitudes Temps réel des réseaux sans fil ». Université de Nancy, [En ligne]. Adresse URL : http://www2.irccyn.ec-nantes.fr/ETR07/proceedings/Presentation-N.Krommenacker.pdf

[11] Y. WEI, Q. LENG, S. HAN, A.K. MOK, W. ZHANG, M. TOMIZUKA. « RT-WiFi: Real-Time High-Speed Communication Protocol for Wireless Cyber-Physical Control Applications ». [En ligne]. Adresse URL : https://pdfs.semanticscholar.org/dc9d/808b810ab6367ad34f63a7915878b357bef0.pdf

[12] B. ROMDHANI. « Exploitation de l'hétérogénéité des réseaux de capteurs et d'actionneurs dans la conception des protocoles d'auto-organisation et de routage ». HAL, [En ligne]. Adresse URL : https://tel.archives-ouvertes.fr/tel-00941099/document

[13] WIKIPEDIA. « Système d'exploitation pour capteur en réseau ». [En ligne]. Adresse URL : https://fr.wikipedia.org/wiki/Syst%C3%A8me_d%27exploitation_pour_capteur_en_r%C3%A9seau

[14] M. LENDERS. « Analysis and Comparison of Embedded Network Stacks ». Freien Universität Berlin, [En ligne]. Adresse URL : http://doc.riot-os.org/mlenders_msc.pdf

[15] WIKIPEDIA. « FreeRTOS ». [En ligne]. Adresse URL : https://fr.wikipedia.org/wiki/FreeRTOS

[16] T. WINTER, P. THUBERT, A. BRANDT, J. HUI, R. KELSEY, P. LEVIS, K. PISTER, R. STRUIK, J.P. VASSEUR, R. ALEXANDER. « RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks ». [En ligne]. Adresse URL : https://tools.ietf.org/html/rfc6550

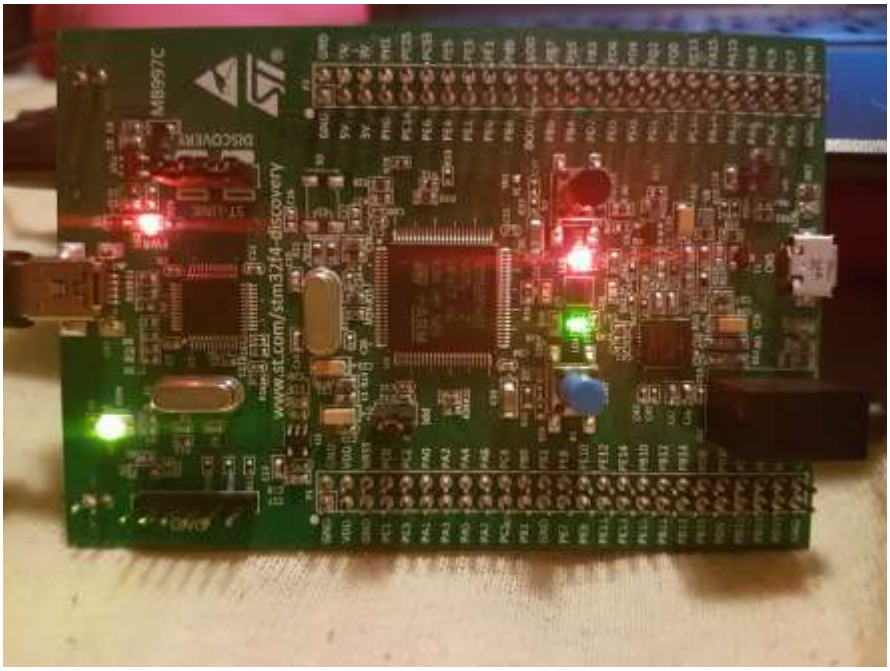
[17] J.P. VASSEUR, N. AGARWAL, J. HUI, Z. SHELBY, P. BERTRAND, C. CHAUVENET. « RPL: The IP routing protocol designed for low power and lossy networks ». IPSO Alliance. [En ligne]. Adresse URL : http://www.ipso-alliance.org/wp-content/media/rpl.pdf

[18] D. CULLER, R. KATZ. « An Analysis of the RPL Routing Standard for Low Power and Lossy Networks ». University of California. [En ligne]. Adresse URL : https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-106.pdf

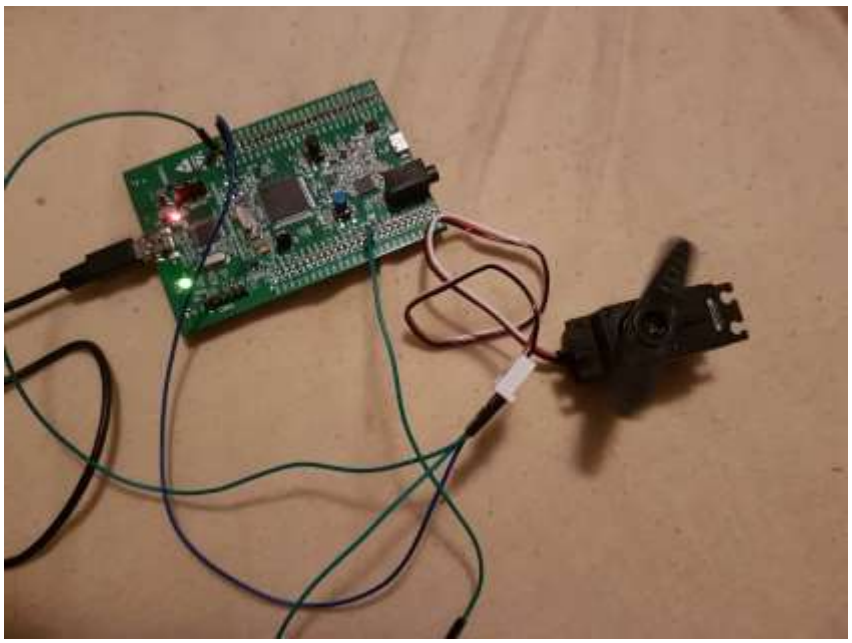
[19] WIKIPEDIA. « 6LoWPAN ». [En ligne]. Adresse URL : https://fr.wikipedia.org/wiki/6LoWPAN

[20] WIKIPEDIA. « Network Time Protocol ». [En ligne]. Adresse URL : https://fr.wikipedia.org/wiki/Network_Time_Protocol

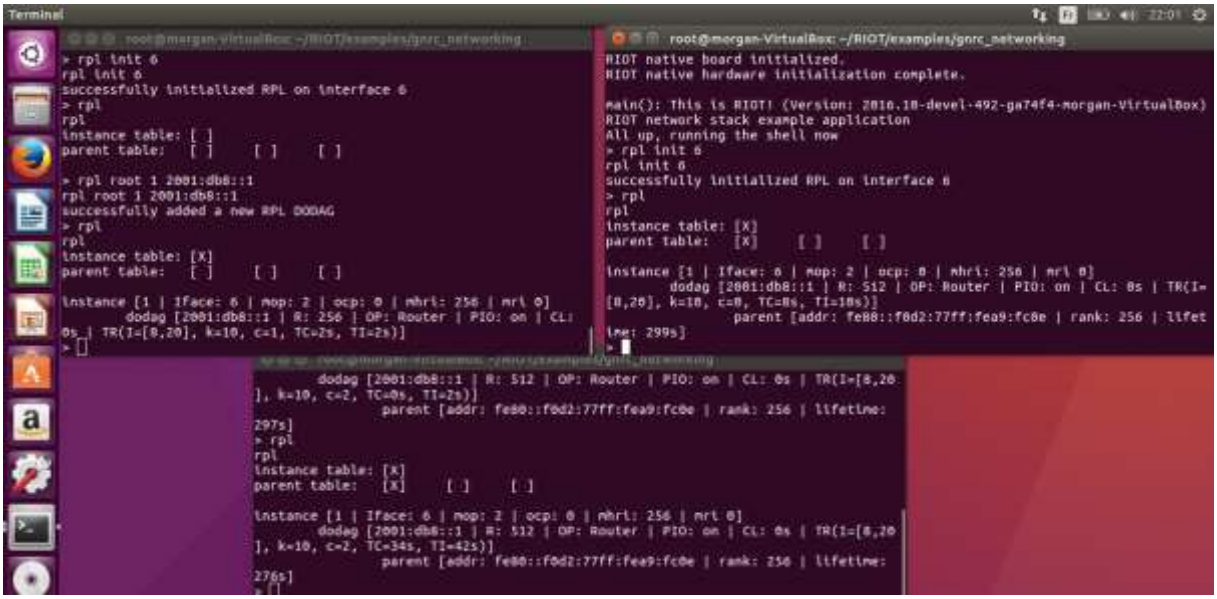
Annexe



Annexe 1 : Clignotement des LEDs en utilisant un timer

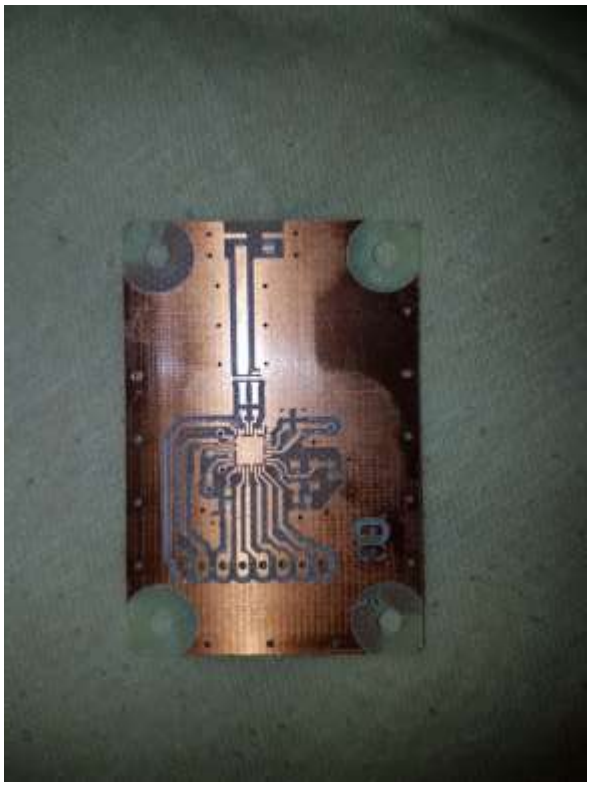


Annexe 2 : Pilotage d'un servomoteur depuis la carte STM32F4discovery

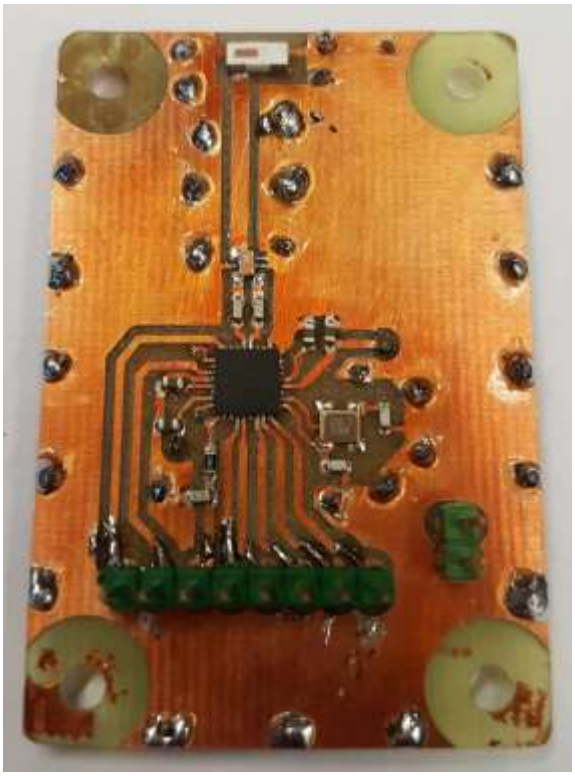


Annexe 3 : Exemple d'architecture de réseau utilisant RPL

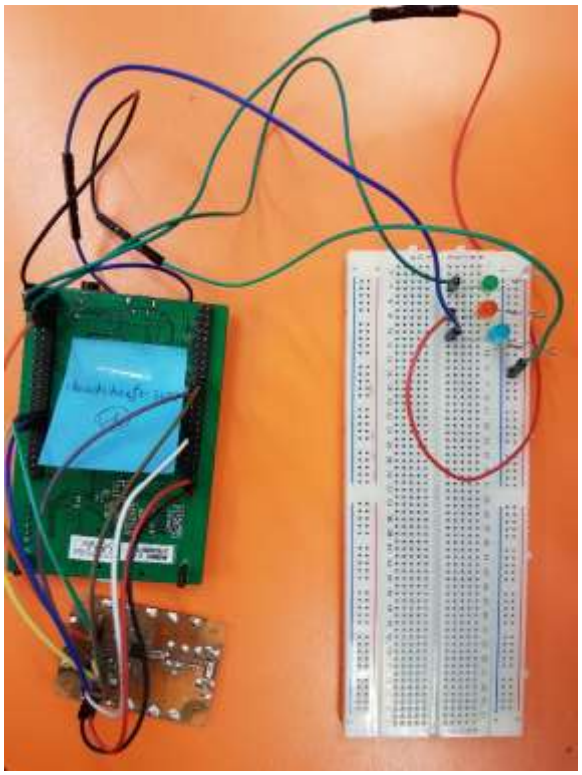
Annexe 4 : Communication par UART pour piloter un servomoteur



Annexe 5 : PCB du module radio (composants non-soudés)

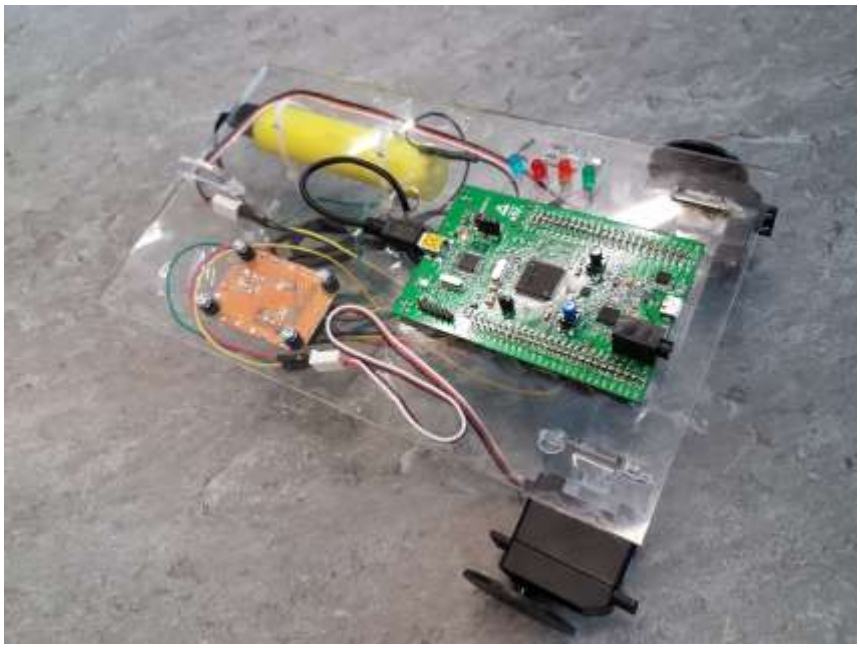


Annexe 6 : Module radio

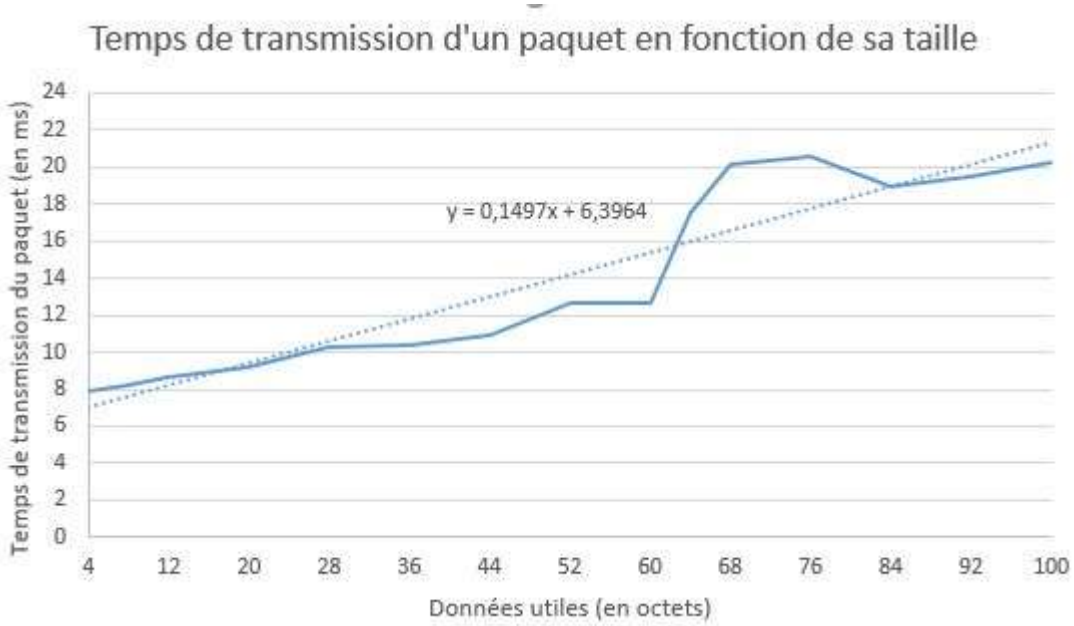


Annexe 7 : Nœud donneur d'ordres

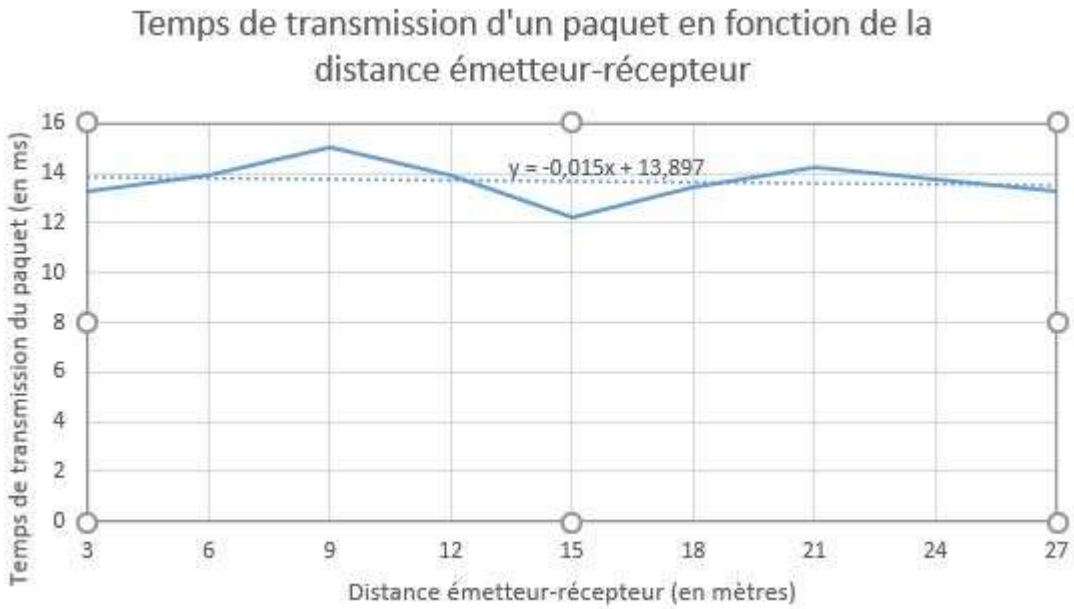
Annexe 8 : Nœud relais



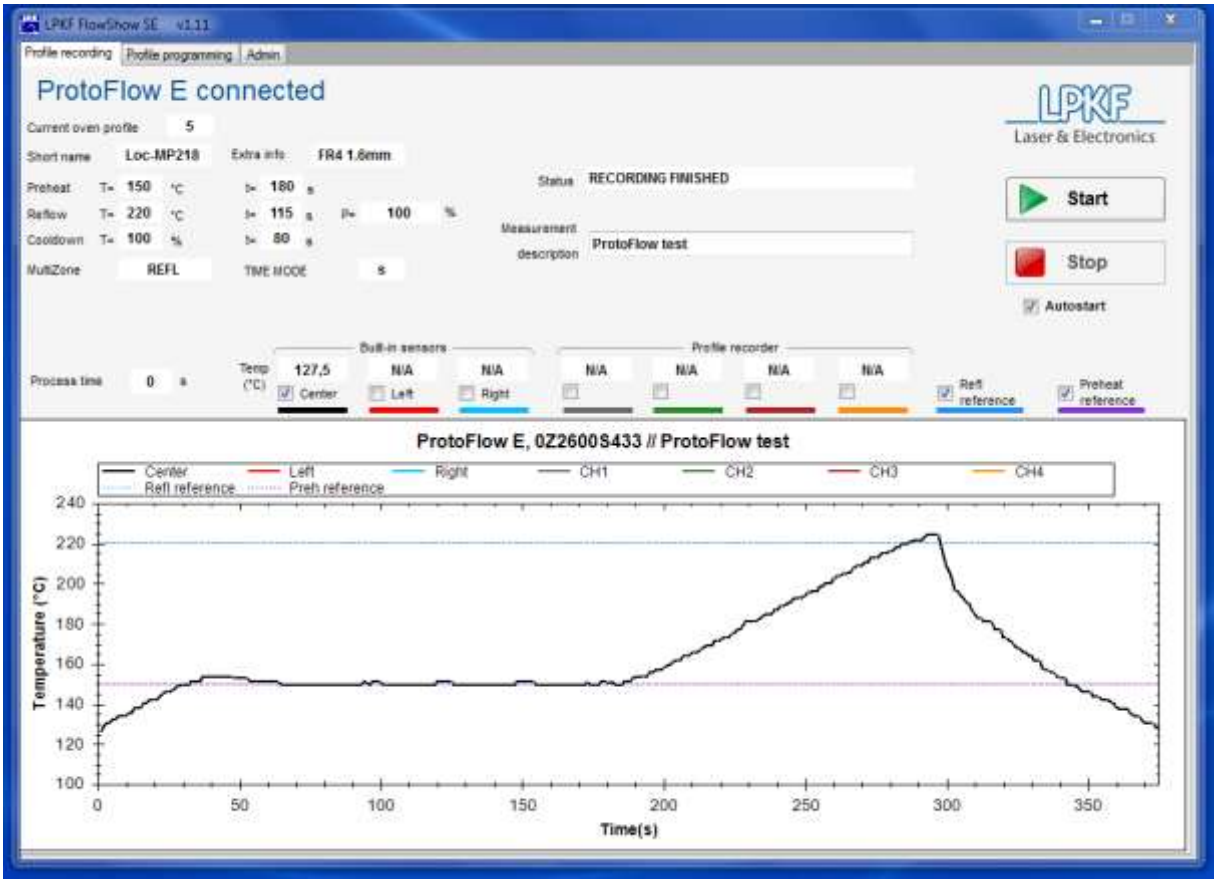
Annexe 9 : Robot réalisé à la découpeuse laser



Annexe 10 : Temps de transmission d'un paquet en fonction de sa taille



Annexe 11 : Temps de transmission d'un paquet en fonction de la distance émetteur-récepteur



Annexe 12 : Courbe représentant la température de chauffe du four en fonction du temps pour la soudure des composants