Rapport final

Développement d'un réseau de capteurs sans fil

David Rubino, Ludovic Schoepps (chef de projet) et Mickaël Thomas

Année 2014–2015

Projet industriel réalisé pour le Ministère de la Défense

Encadrant industriel : Antoine Moron

Encadrant universitaire : Marc Tomczak

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Rubino, David

Élève-ingénieur(e) régulièrement inscrit(e) en 3e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31010226

Année universitaire : 2014–2015

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Développement d'un réseau de capteurs sans fil

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 17 mars 2015

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Schoepps, Ludovic

Élève-ingénieur(e) régulièrement inscrit(e) en 3e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31212132

Année universitaire : 2014–2015

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Développement d'un réseau de capteurs sans fil

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 17 mars 2015

Signature :

Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Thomas, Mickaël

Élève-ingénieur(e) régulièrement inscrit(e) en 3e année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 31214076

Année universitaire : 2014–2015

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Développement d'un réseau de capteurs sans fil

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Villers-lès-Nancy, le 17 mars 2015

Signature :

Rapport final

Développement d'un réseau de capteurs sans fil

David Rubino, Ludovic Schoepps (chef de projet) et Mickaël Thomas

Année 2014–2015

Projet industriel réalisé pour le Ministère de la Défense

David Rubino
5 rue Jacques Callot
54500, VANDOEUVRE-LES-NANCY
+33 (0)6 20 04 57 03
david.rubino@telecomnancy.eu

Ludovic Schoepps
8 rue du château
57310, GUENANGE
+33 (0)6 74 04 98 11
ludovic.schoepps@telecomnancy.eu

Mickaël Thomas
4 rue de la paix
54000, NANCY
+33 (0)6 52 13 35 52
mickael.thomas@telecomnancy.eu

TELECOM Nancy
193 avenue Paul Muller,
CS 90172, VILLERS-LÈS-NANCY
+33 (0)3 83 68 26 00
contact@telecomnancy.eu

Ministère de la Défense
14 rue Saint-Dominique
75700, PARIS
+33 (0)1 80 50 14 00

Encadrant industriel : Antoine Moron

Encadrant universitaire : Marc Tomczak

Remerciements

Nous tenons à remercier chaleureusement notre encadrant industriel, Antoine Moron, pour nous avoir confié ce projet et nous avoir aidé à le mener à bien.

Nous voulons également remercier Marc Tomczak, notre encadrant universitaire, pour son aide et ses conseils précieux.

Table des matières

1 Introduction

Ce projet de développement d'un réseau de capteurs sans fil s'effectue dans le cadre des projets industriels de 3A à TELECOM Nancy. Ce projet est réalisé pour le ministère de la Défense.

L'objectif de ce projet est d'écrire les programmes permettant de mettre en place un réseau de capteurs sans fil utilisant la technologie ZigBee* [1]. A partir de cartes programmables fournies par le ministère, connectées à divers capteurs tels que bouton poussoir, capteur de luminosité, accéléromètre, notre travail consiste à développer un programme qui sera téléchargé sur ces cartes et permettra de remonter les données fournies par ces capteurs à une interface graphique sur une tablette Windows qui sera également réalisée par nos soins. Le projet final sera ainsi constitué d'un réseau de capteurs communiquant entre eux et permettant de remonter des données à une tablette Windows. De plus, selon les données remontées, notre réseau de capteurs devra être capable de déclencher des événements tel que l'allumage d'une LED sur une des cartes fournies.

Un exemple d'application concrète pourrait consister à surveiller à distance une pièce contenant des documents confidentiels. Ainsi, les capteurs pourraient être positionnés dans cette pièce, et en cas d'intrusion ils pourraient détecter la personne et transmettre un message au logiciel gérant ce réseau de capteurs.

Cette organisation est schématisée par la figure 1, dont l'architecture du réseau correspond au résultat attendu à la fin du projet. Une tablette Windows, qui permettra d'administrer et de surveiller le réseau, est connectée via un câble USB à un *Coordinateur**. Ce coordinateur communique avec plusieurs capteurs et actionneurs qui jouent le rôle d'*End-devices** dans la terminologie ZigBee.

Par exemple, afin de surveiller une salle contenant des documents confidentiels, il peut être intéressant de disposer d'un détecteur de porte ouverte, d'un détecteur de présence et/ou d'un détecteur de luminosité. Ces détecteurs pourraient par exemple déclencher une séquence d'enregistrement d'une caméra.

Afin de présenter ce projet plus en détails, nous allons voir dans une première partie une présentation rapide du ministère de la Défense, puis une description du projet avec ses spécifications suivi d'un état de l'art et d'une description de l'environnement logiciel et matériel. Une fois ces points abordés, nous rentrerons dans le cœur du sujet avec le déroulement de la partie embarquée suivi du déroulement de la partie Windows. Pour finir nous ferons un point sur l'organisation du projet avant de conclure ce rapport.

Nous allons donc tout d'abord parler du ministère de la Défense et de l'équipe où travaille notre encadrant industriel.

1. Les mots suivis d'une * sont définis dans le glossaire

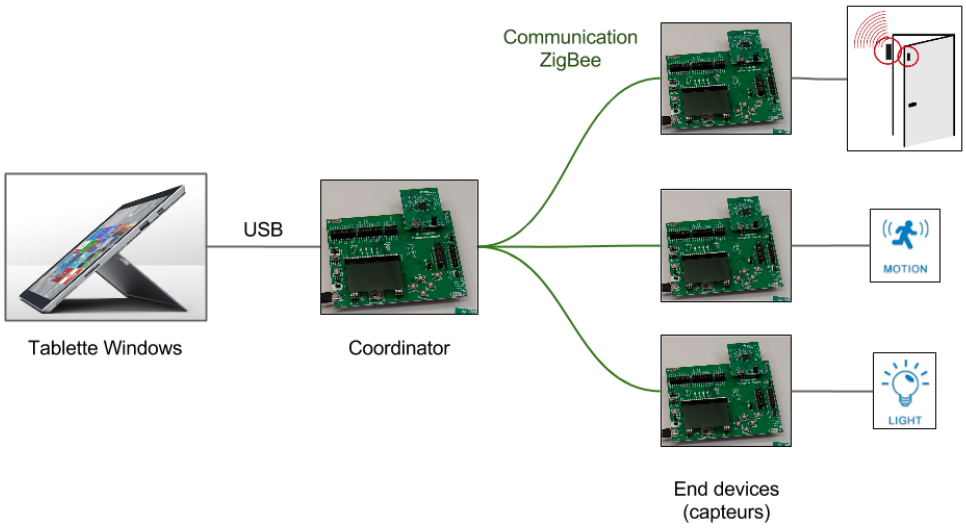


FIGURE 1 – Architecture attendue

2 Présentation du ministère de la Défense

Dans cette partie, nous allons présenter le ministère de la Défense pour lequel nous réalisons ce projet industriel.

Initialement créé en 1589 sous le nom de secrétaire d'Etat de la Guerre, le ministère de la Défense est l'organisme chargé de la préparation et de la mise en œuvre de la politique de défense française. Siégeant à Paris, son rôle s'étend à la fois dans le domaine international avec des opérations militaires à l'étranger et dans le domaine national, tel que la protection du territoire français. Le budget du ministère pour l'année 2015 s'élève à 42,02 milliards d'euros. Le ministère emploie plus de 285 000 personnes. Actuellement, il est dirigé par Jean-Yves Le Drian, ministre de la Défense. La figure 2 présente l'organisation du Ministère.

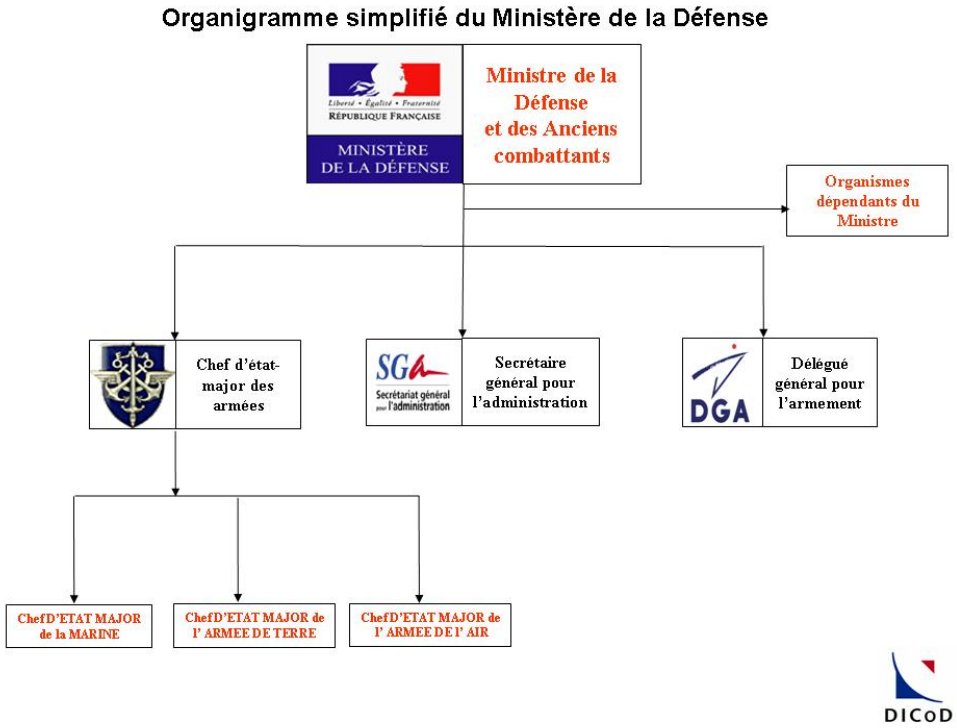


FIGURE 2 – Organigramme du ministère de la Défense [1]

Notre encadrant industriel travaille dans une équipe de 15 personnes, spécialisée dans les technologies embarquées. Ils travaillent sur des composants de petite taille et consommant peu de puissance. Leurs domaines d'expertise comprend les communications sans fil, la géolocalisation, le son, l'image, etc.

Cette équipe est affectée à la mission recherche, dont le but est l'apprentissage de nouvelles technologies au sein du ministère. Sur le budget total de la défense, celui de la mission recherche s'élève à 190 millions d'euros, ce qui, bien que ne représentant qu'une très faible partie du budget total, constitue tout de même une somme non négligeable.

Après cette présentation du client de notre projet, nous allons maintenant parler plus précisément des objectifs qui nous ont été confiés.

3 Description du projet

Le but de notre projet pour le ministère de la Défense est de mettre en place un système d'alarme basé sur des capteurs divers et variés communiquant (grâce à la technologie « ZigBee ») avec un « coordinateur » (un microcontrôleur chargé de récupérer toutes les informations transmises par les capteurs). Le « coordinateur » pourra déclencher des événements en conséquence et communiquera également avec un logiciel sur une tablette Windows. (Conformément aux spécifications du projet mais le logiciel en question pourrait très bien tourner sur un ordinateur sous Windows)

À partir de ce logiciel, l'utilisateur doit pouvoir visualiser les capteurs sur un plan ainsi que leur état (en alerte, déconnecté, etc.). Il doit également pouvoir prendre en compte de nouveaux capteurs rajoutés au plan en renseignant leurs caractéristiques (emplacement, type, ...) afin de les intégrer au réseau de capteurs. De même il doit pouvoir supprimer du plan des capteurs ne servant plus dans la réalité.

Le projet se décompose en deux parties : la programmation des cartes (partie embarquée) et la programmation de l'interface graphique pour Windows. Mickaël Thomas est responsable de la première partie (la partie embarquée) et David Rubino est responsable de la seconde (la partie Windows).

La partie embarquée comprend l'élaboration du réseau de capteurs, communiquant à travers la spécification ZigBee. Ces cartes nécessitent d'être programmées en langage C. Il s'agit d'un langage de programmation très répandu pour les programmes embarqués puisqu'il est populaire et qu'il est plus rapide et moins gourmand en mémoire que d'autres langages à but similaire.

Il faudra donc créer un programme à télécharger sur les cartes et qui répond aux fonctions suivantes :

- Prise en compte de différents types de capteurs dans le réseau (par exemple température, contact, luminosité, etc...).
- Déclencher des alertes en fonction de certains événements (ex : luminosité trop importante). Pour cela, il faut que l'utilisateur puisse définir une valeur de seuil au niveau de chaque type de capteur. Dès qu'une valeur mesurée est en dehors de ce seuil, le capteur déclenche un signal d'alarme sur le réseau qui sera signalé sur l'interface graphique sur la tablette Windows.
- Permettre l'appairage* automatique entre le coordinateur et les end-devices.

Concernant les fonctionnalités de l'interface, les objectifs suivants ont été établis suite à une discussion avec notre encadrant industriel. Tout d'abord, l'interface devra être ergonomique et intuitive : il est nécessaire que le logiciel ait un affichage très simple afin de ne pas chercher très longtemps comment rajouter un capteur sur la carte. Pour cela, il n'y aura qu'une seule vue séparée en 2 parties : à gauche, l'emplacement réservé pour le plan et qui occupera la plupart de la vue, et à droite un panneau d'affichage où se trouvent les différentes informations concernant les capteurs, les actionneurs et les règles de déclenchements. La barre d'outils situé au dessus du plan comprendra différents boutons de contrôles.

L'interface devra aussi inclure la possibilité d'importer un plan de l'endroit où l'on sou-

haite déployer les capteurs, et conserver la carte en mémoire lors de la réouverture du logiciel. L'interface doit également permettre la visualisation des capteurs et des actionneurs sur la carte. Ces derniers seront représentés par des icônes qui, selon l'état des capteurs ou des actionneurs, changeront. De plus, il faudra accéder aux caractéristiques des capteurs et des actionneurs : en cliquant sur chacun d'eux, les caractéristiques de celui-ci s'afficheront sur le panneau droit de l'interface avec ses différentes valeurs mesurées à l'instant t. L'interface permettra également de modifier un capteur (changer son nom ou sa valeur de seuil si applicable).

Maintenant que nous avons décris les objectifs principaux du projet, nous allons présenter l'environnement logiciel et matériel que nous utilisons pour la conduite de notre projet.

4 État de l'art

Les réseaux de capteurs sans fil existaient déjà depuis les années 1990 et sont en constante évolution grâce aux progrès dans le domaine des communications sans fil et grâce à des composants plus compacts, plus performants et consommant moins d'énergie.

Ces réseaux sont utilisés pour une variété d'applications incluant : acquisition de données à distance, surveillance et maintenance des machines, routes et bâtiments "intelligents", surveillance de l'environnement, sécurité de certains lieux, ... [6]

4.1 Protocoles de communication

Afin de pouvoir transmettre des informations à distance, il existe actuellement deux protocoles de communication qui sont les plus aptes à être exploités : le protocole Bluetooth et le protocole ZigBee*.

Ces deux protocoles sont similaires quant à la possibilité de transmettre des données à courte distance (et sans fil) mais le protocole Bluetooth, contrairement au protocole ZigBee, privilégie le débit des données face à la consommation énergétique (*cf* section 5.1). L'autonomie, pour un appareil muni de piles, avec le protocole Bluetooth serait de l'ordre de quelques mois alors que ZigBee permet une autonomie de plusieurs années. De plus, le Bluetooth ne peut communiquer qu'avec un maximum de 7 appareils à la fois (appelé "nœuds" dans un réseau) alors que ZigBee en supporte plusieurs milliers. Un autre avantage de ZigBee face au Bluetooth est le besoin en mémoire réduit ce qui fait qu'il est bien adapté aux petits appareils embarqués qui souvent ne disposent pas de beaucoup de mémoire.

Un autre protocole de communication populaire qui pourrait être utilisé est le Wi-Fi, mais bien qu'il permette un débit de données bien plus grand, il n'est pas adapté pour les réseaux de capteurs sans fil. En effet, celui ci consomme beaucoup trop d'énergie et demande trop de ressources pour des petits appareils embarqués.

Le tableau comparatif suivant permet d'y voir plus clair sur les différences entre ces 3 protocoles :

Caractéristique	**ZigBee**	**Bluetooth**	**Wi-Fi**
Besoins mémoire	4-32 Ko	250 Ko +	1 Mo +
Autonomie avec pile	Années	Mois	Jours
Nombre de nœuds	65 000+	7	256+
Vitesse de transfert	250 Kb/s	1 Mb/s	1000 Mb/s
Portée (environ)	100 m	10 m	300 m

Le réseau de capteur sans fil qui a été conçu pour ce projet ne nécessite qu'un débit très peu élevé puisque quelques valeurs sont transmises à intervalles réguliers et il est également essentiel d'avoir un réseau pouvant contenir un maximum de nœuds. Il était donc indispensable de recourir au protocole ZigBee pour bénéficier de ses nombreux avantages.

Bien que la portée de la communication via le protocole ZigBee n'est que de 100m (en

champ libre), elle peut être grandement améliorée selon la topologie (la forme) du réseau utilisé.

Il existe différentes topologies qui peuvent être utilisées. Dans la topologie en étoile, il existe un unique nœud de base et chaque nœud du réseau ne peut envoyer ou recevoir des messages que vers ou depuis cet unique nœud de base. L'avantage d'une telle topologie est sa simplicité et sa faible consommation d'énergie des nœuds mais l'inconvénient est la portée du réseau qui est limitée à au maximum 100m autour du nœud de base.

Dans la topologie « en toile » ou « en grille » (« mesh » en anglais), chaque nœud peut échanger des messages avec n'importe quel autre nœud du réseau. Si un nœud n'est pas à portée, le message pourra transiter par plusieurs nœuds intermédiaires. L'avantage ici est de pouvoir étendre la portée du réseau très simplement mais l'inconvénient est la surconsommation induite par l'utilisation de nœuds intermédiaires mais aussi la latence (délai entre l'envoi et la réception du message) induite. La topologie hybride consiste à désigner seulement certains nœuds comme pouvant être utilisés comme intermédiaires lors de la transmission d'un message. Ces derniers disposent en principe d'une source d'énergie externe alors que les nœuds autonomes en énergie ne permettront pas ce service.

C'est cette dernière topologie, qui est facile à mettre en place dans un réseau ZigBee car prévu pour, qui est utilisée pour ce projet industriel.

4.2 Solutions existantes

Parmi les réseaux de capteurs sans fil déjà existants dans le commerce, on y trouve toute la catégorie des systèmes d'alarme. En effet, dans un système d'alarme classique, plusieurs capteurs sont disposés à plusieurs endroits et communiquent avec ou sans fil avec une centrale d'alarme qui prendra la décision de déclencher ou pas l'alerte. Il est donc possible d'adapter ce projet pour qu'il réagisse de manière analogue ce qui le rend donc comparable aux multitudes de systèmes d'alarme existants dans le commerce.

Face à ces solutions qui existent déjà, on peut donc se poser la question de l'intérêt de ce projet face aux autres solutions. On va donc voir quels en sont les avantages et les inconvénients.

Tout d'abord, ce projet n'est pas moins cher que les systèmes d'alarme existants. L'achat des cartes programmables ZigBee reste coûteux et ne justifie donc pas l'utilisation de ce projet face à des systèmes d'alarme probablement moins chers.

En revanche, les systèmes d'alarme classiques existants dans le commerce présentent plusieurs inconvénients. Ceux-ci sont beaucoup moins polyvalents et ne sont compatibles qu'avec un nombre restreint de capteurs. La solution de ce projet permet de pouvoir s'adapter à n'importe quel type de capteur puisque qu'il est possible de programmer une carte de façon à ce qu'elle s'adapte au capteur auquel elle est connectée. Ceci est d'autant plus important pour ceux, comme le ministère de la Défense, qui peuvent avoir besoin de capteurs qui ne se trouvent pas dans le commerce. Dans ce cas, aucun système d'alarme classique ne conviendrait. De plus, certains systèmes d'alarme dépendent d'une ligne téléphonique ce qui rend ces systèmes extrêmement dépendants et très risqués. La polyvalence de notre solution permet de ne pas dépendre d'un seul moyen d'alerte mais on peut

l'adapter pour utiliser plusieurs moyens différents (transmission GSM par exemple).

Un autre avantage très important est la sécurité. En effet, il existe des systèmes d'alarme pour lesquels on a découvert des failles qui permettraient à des personnes malveillantes de pouvoir désactiver le système d'alarme, d'entrer dans une maison, faire ce qui bon leur semble et sortir comme si aucune effraction n'avait eu lieu[7]. Pour réussir une telle action, le technique consiste principalement à intercepter les informations, avec un sniffer*, qui sont transmises à distance et à les décrypter. Sachant ceci, il est bien probable que d'autres failles qui n'ont pas encore été découvertes ou même dévoilées existent parmi la multitude de systèmes d'alarme existants. Il est donc essentiel, lors de l'achat d'un tel système, de pouvoir faire confiance au fournisseur si l'on veut être à l'abri du moindre risque.

La solution proposée par ce projet industriel intègre de base un réseau sécurisé (*cf* section 6.7). Néanmoins, il faut tout de même faire confiance à ce système qui n'est pas à l'abri d'une faille non connue à ce jour. C'est pourquoi l'avantage de notre solution, qui n'est pas possible dans des solutions classiques, c'est la possibilité de rajouter une surcouche de sécurité, c'est-à-dire de pouvoir modifier le programme actuel afin de mieux chiffrer les données notamment. C'est un moyen pour des clients comme le ministère de la Défense de pouvoir appliquer leur propre système de sécurité dans lequel ils ont bien plus confiance qu'un autre.

5 Environnement logiciel et matériel

Dans cette partie, nous allons présenter les différents outils que nous avons utilisés pour le développement de notre projet.

5.1 Protocole de communication ZigBee

ZigBee est un protocole de communication sans fil similaire au protocole Wi-Fi. Contrairement à ce dernier, ZigBee est conçu pour être utilisé avec des systèmes embarqués peu puissants et disposant de peu d'énergie (systèmes alimentés par une pile par exemple).

Outre les réseaux de capteurs, ZigBee est par exemple utilisé dans certaines télécommandes (comme celle de la FreeBox v6). Les avantages par rapport à la technologie infrarouge couramment utilisée sont la grande portée ainsi que l'inutilité d'orienter la télécommande.

Il existe différents profils standardisés pour les applications diverses et variées offertes par ZigBee. Par exemple, le profil *Home Automation* est destiné aux applications domotiques et permet l'interopérabilité entre tout capteur ou actionneur implémentant ce profil, quel qu'en soit le constructeur.

Un réseau ZigBee se compose au moins d'un coordinateur et d'un ou plusieurs *end-devices* avec optionnellement un ou plusieurs routeurs. Le rôle du coordinateur est de faire exister le réseau, de gérer quels périphériques sont autorisés à en faire partie, de le configurer (clé de chiffrement, ...) et de l'entretenir. Les routeurs, quant à eux, ont pour but d'étendre la portée du réseau (les données pouvant passer par plusieurs routeurs avant d'atteindre leur destination). Ils sont également responsables de la conservation des messages en attente de remise à un *end-device*. Les *end-devices*, contrairement au coordinateur et aux routeurs, n'ont pas besoin d'être constamment allumés. Ils peuvent entrer en sommeil. Dans cet état, ils ne communiquent plus et ne consomment donc pratiquement pas d'énergie. Ils pourront se réactiver lors d'un événement spécifique (appui sur un bouton poussoir, capteur qui se déclenche, etc.) ou au bout d'une certaine durée, afin de traiter les messages reçus au cours de leur sommeil.

5.2 Kit de développement embarqué

Afin de réaliser la partie embarquée, le ministère nous fournit à chacun un kit constitué notamment :

— de deux cartes de développement (voir fig. 3) avec écran LCD, boutons poussoirs, capteur de température, capteur de lumière et accéléromètre, sur lesquelles sont connectées un module amovible qui contient un processeur TI CC2538 et un module ZigBee avec antenne intégrée. Son rôle est d'assurer la communication sans-fil avec les autres périphériques ZigBee du réseau et de leur transmettre les valeurs mesurées par les capteurs qui lui sont connectés.

— d'un petit périphérique USB ressemblant à une clé USB (appelé *sniffer**) permettant, une fois relié à un ordinateur, d'observer et de décoder à l'aide du logiciel fourni

par TI les communications ZigBee dans l'environnement radio proche. Cet outil peut être très utile pour vérifier que les données qui transitent sur notre réseau correspondent bien à ce que l'on attend.

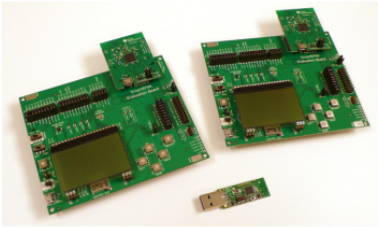


FIGURE 3 – Kit de développement CC2538

5.3 Pile logicielle Z-Stack

Z-Stack, de Texas Instruments est un ensemble de codes sources, de documentations et de logiciels permettant, sur les microprocesseurs ZigBee de la marque (comme le CC2538 que nous utilisons), d'implémenter les différents composants d'un réseau ZigBee (coordinateur, routeur, end-device). Il en existe plusieurs variantes en fonction du type d'application voulue.

Tout d'abord nous avons Z-Stack Mesh, destiné aux applications spécifiques ne nécessitant pas forcément d'interopérabilité entre constructeurs.

Ensuite, on trouve différentes variantes, basées sur Z-Stack Mesh, destinées à des applications courantes et permettant l'interopérabilité entre constructeurs : Z-Stack Home pour la domotique, Z-Stack Lighting pour l'éclairage, Z-Stack Energy pour le suivi de la consommation énergétique.

Dans notre projet nous avons seulement considéré Z-Stack Home (car notre application ressemble à un système domotique) et Z-Stack Mesh (qui nous permet de sortir du cadre imposé par Z-Stack Home et de nous libérer de la contrainte d'interopérabilité qui n'est pas nécessaire pour notre projet).

5.4 Environnement de développement IAR

IAR Embedded Workbench de IAR Systems est un environnement de développement comprenant éditeur, compilateur C/C++ et débogueur pour le développement sur divers processeurs embarqués (en particulier pour le CC2538 utilisé dans notre projet). Une licence nous est fournie par le ministère de la Défense et nous permet d'utiliser le logiciel avec toutes ces fonctionnalités. Nous utilisons IAR pour développer les programmes embarqués (coordinateur et end-devices) sur les modules ZigBee que nous utilisons.

5.5 Présentation du logiciel Visual Studio

Microsoft Visual Studio est un environnement de développement (IDE) publié par Microsoft et qui permet de faire du développement dans de très nombreux langages (C#, C++, JavaScript...). Il est donc très approprié dans le cadre de notre application puisqu'il offre de nombreuses facilités de développement avec C#, en particulier ce qui concerne l'interface graphique. En effet, Visual Studio permet de dessiner l'interface directement en faisant un drag-and-drop des composants voulus. Cela évite donc d'écrire le code manuellement, ce qui permet de gagner un temps important dans le développement.

Ces différents outils nous ont donc permis de travailler sur ce projet dont nous allons maintenant vous présenter l'avancement.

6 Déroulement de la partie embarquée

Nous allons maintenant vous présenter le déroulement du projet pour la partie embarquée d'une part puis pour la partie Windows dans la section suivante. Avant toute chose, étant donné que personne n'avait auparavant travaillé avec les technologies à mettre en œuvre, nous avons dû nous familiariser avec l'environnement matériel (la carte embarquée) et l'environnement logiciel (Z-Stack Home) ainsi que le standard ZigBee en lui-même. Pour cela, nous avons lu les très nombreuses documentations de Texas Instruments et de la ZigBee Alliance. [2] [3] [4]

6.1 Prototype initial

Après avoir installé les différents logiciels (notamment IAR) et documentations, nous avons pu tester les différents exemples de programmes fournis dans Z-Stack Home et fonctionnant sur la carte embarquée, ceci afin de nous familiariser avec le logiciel et nous imprégner du fonctionnement interne de Z-Stack.

Parmi ces exemples, nous avons choisi celui qui se rapprochait le plus de notre besoin tout en étant le plus simple possible, ceci afin de nous permettre de commencer à développer un prototype de coordinateur (SampleLight) et de capteur (SampleSwitch).

Le programme d'exemple *SampleLight* représente une ampoule qui est matérialisée par une LED sur l'une des cartes. Celle-ci peut-être commandée à distance, depuis une autre carte exécutant le programme complémentaire *SampleSwitch* qui représente un interrupteur et est matérialisé par un bouton poussoir sur la carte.

Nous avons ainsi modifié le programme SampleLight, pour qu'il soit capable de communiquer, au travers de sa liaison USB, l'état de la LED à l'application Windows à travers la liaison sans-fil ZigBee.

Ce premier prototype nous a permis de nous rendre compte de la complexité du fonctionnement de la couche Home Automation et nous avons décidé de nous simplifier la tâche en utilisant seulement le protocole ZigBee sans la partie Home Automation, c'est-à-dire en nous basant sur Z-Stack Mesh plutôt que Z-Stack Home.

L'intérêt de Home Automation est, en respectant certaines normes (types de capteurs et de données par exemple), de pouvoir simplifier l'installation de nouveaux capteurs à un réseau existant et de pouvoir lier facilement certains événements à certaines actions. Cependant, pour ce projet, nous ne nous servons pas des différentes fonctionnalités offertes et l'utilisation de cette couche Home Automation ne ferait qu'alourdir inutilement nos programmes.

Nous allons maintenant détailler la réalisation de l'application embarquée finale, qui fait suite au prototype initial.

6.2 Application finale

Nous avons choisi de baser notre application sur un autre exemple, *GenericApp* que l'on trouve dans la Z-Stack Mesh et qui permet d'envoyer et de recevoir un message à travers le réseau. Dans un premier temps, nous l'avons adapté à notre application pour envoyer l'état d'une LED d'un côté et le transmettre à l'application de l'autre. De plus, le programme nécessitait initialement de synchroniser le coordinateur avec le capteur en appuyant sur un bouton sur les deux cartes (appairage* manuel). Nous l'avons adapté afin que cette étape de synchronisation se fasse toute seule sans intervention manuelle.

Ensuite, une étape essentielle fut de mettre en place une structure pour stocker, sur une mémoire non volatile du coordinateur, toutes les données définissant le réseau (nom des capteurs, adresse IEEE*, types, règles de déclenchement des alertes et des événements, ...). L'idée est que l'application Windows doit être le reflet de l'état du coordinateur. De cette manière, le coordinateur reste indépendant de l'application Windows.

La mise en place de cette structure de stockage permettra d'établir un protocole de communication entre les capteurs et le coordinateur d'une part, puis entre le coordinateur et le logiciel Windows d'autre part. C'est ce protocole qui permettra de communiquer efficacement les différentes informations circulant sur le réseau.

Nous allons maintenant voir la structure de stockage qui a été mise en place.

6.3 Choix de la mémoire de stockage

Tout d'abord, nous avons dû faire un choix parmi plusieurs possibilités quant à la mémoire sur laquelle nous allions stocker les données. Une première possibilité était d'utiliser une carte micro SD branchée sur le coordinateur via un connecteur prévu à cet effet. L'autre solution était d'écrire directement sur la mémoire Flash sur laquelle est notamment stocké le programme.

La première solution d'utiliser une carte SD amène différents avantages et inconvénients.

L'inconvénient majeur est que l'API offerte par Z-Stack ne nous offre qu'une méthode primitive d'accès à cette mémoire (lire et écrire une donnée à telle adresse), c'est-à-dire sans implémentation d'un système de fichiers comme FAT32 ou exFAT qui permet de structurer les données à l'aide de fichiers et dossiers par exemples. Nous aurions ainsi dû, soit implémenter nous-mêmes un système similaire (en moins complexe) ou utiliser une implémentation libre et l'adapter nous-mêmes.

Les avantages sont quant à eux de disposer d'une plus grande capacité de stockage, et de pouvoir facilement remplacer la mémoire puisque la carte SD peut-être retirée (en cas de défaillance par exemple).

En ce qui concerne la deuxième solution d'utiliser la mémoire Flash interne, elle possède également ses avantages et inconvénients.

Du côté des inconvénients, tout d'abord il s'agit d'une mémoire interne qui est directement soudée à côté du processeur, elle n'est donc pas remplaçable et est susceptible de se détériorer avec le temps, jusqu'à devenir inutilisable. Elle est également très limitée en

taille (512 Ko pour contenir tout le programme et les données).

Son avantage majeur est sa simplicité d'utilisation, en effet, bien qu'il n'y ait pas d'implémentation d'un système de fichiers "lourd" comme FAT32, la Z-Stack nous offre une API (l'API NV) qui permet de stocker des données de taille variable sous une forme analogue à des fichiers : chaque donnée est associé à un numéro identifiant que l'on peut utiliser pour récupérer ou modifier la donnée plus tard, sans se préoccuper de la façon dont cette donnée sera stockée en mémoire. L'API NV gère ainsi automatiquement le placement des données dans la mémoire qui lui est attribuée (12 Ko de mémoire Flash).

Nous pensions initialement utiliser la carte SD pour y stocker les données utiles à notre application, mais après concertation avec notre encadrant, nous avons décidé par souci de simplicité d'utiliser plutôt la mémoire interne avec l'API NV.

Une fois cette question de quelle mémoire utiliser résolue, il nous a fallu ensuite définir la structure des données que nous allions y stocker. Il fallait une structure qui puisse s'adapter à un grand nombre d'appareils dans le réseau avec la possibilité d'en enlever et d'en supprimer simplement. De plus, il nous a été demandé de ne pas limiter artificiellement le nombre de capteurs qui peuvent êtres gérés dans le réseau, c'est-à-dire de ne pas explicitement écrire dans le programme qu'on ne souhaite pas gérer plus de 50 capteurs si on a aucune raison de ne pas pouvoir en gérer plus en réalité.

6.4 Structuration des données

Nos données sont constituées presque exclusivement de listes (liste de capteurs, d'actionneurs, de règles de déclenchement, associées chacune à une liste de capteurs provoquant leur déclenchement). Plusieurs structures de listes existent, dont les principales sont les listes chaînées (et leur variantes), les tableaux associatifs, et les tableaux statiques.

Nous avons choisi d'utiliser des listes chaînées. Leur avantage majeur est la non nécessité de définir à l'avance le nombre d'éléments (contrairement aux tableaux statiques) et de pouvoir très facilement ajouter et supprimer des éléments, puisque ces deux opérations consistent simplement à changer un lien entre les éléments qui peuvent être disposés n'importe où en mémoire. L'inconvénient principal est d'être obligé de parcourir toute la liste pour accéder à un élément précis. Mais le nombre d'éléments maximal restant de l'ordre de la dizaine voir de la centaine au maximum, les réductions de performances restent acceptables.

Une autre possibilité aurait été d'utiliser un tableau associatif. Son avantage est l'accès très rapide à un de ses éléments à partir, par exemple, de l'adresse IEEE d'un capteur ou d'un actionneur (qui est un cas qui se présente souvent). L'inconvénient est la nécessité de disposer d'une grande zone contiguë en mémoire, or il n'est pas possible d'écrire facilement cette zone contiguë dans la Flash en utilisant l'API NV. En effet, l'idéal serait de pouvoir l'écrire en une fois mais ce n'est pas possible comme l'écriture dans la Flash via l'API NV limite la quantité de données que l'on peut écrire en une seule fois. Une solution possible serait de fragmenter cette structure à plusieurs endroits de la mémoire mais cela complexifierait beaucoup la gestion de la mémoire et demanderait bien plus de temps pour mettre en place cette structure alors qu'il ne s'agit pas d'une fonctionnalité essentielle pour ce projet.

De ce fait, les listes chaînées sont très bien adaptées aux contraintes de ce projet.

6.4.1 Implémentation des listes chaînées

Nous avons ainsi développé notre propre implémentation de liste chaînée, baptisée *NvList* (liste chaînée construite sur l'API NV). En effet, il ne nous était pas possible d'utiliser une implémentation existante du fait de la spécificité de notre structure de stockage. Dans une liste chaînée classique, un élément pointe vers son successeur via un pointeur (c'est-à-dire qu'il contient l'adresse mémoire de son successeur). Dans notre application, les données sont stockées sur la mémoire sous forme d'éléments distincts portant chacun un numéro identificateur (ID), et l'accès à ces éléments doit se faire par l'appel des fonctions *osal_nv_read()* et *osal_nv_write()* plutôt que via un simple accès mémoire.

Les prototypes des fonctions de notre implémentation se trouvent en annexe B.2

Une *NvList* se compose d'une racine portant un ID propre, et pointant soit vers une autre ID qui est le premier élément de la liste, ou la valeur zéro si la liste est vide. Chaque élément contient le numéro de son successeur ou zéro si il s'agit du dernier élément de la liste, suivi des données propres à cet élément (voir fig 4 ci-dessous).

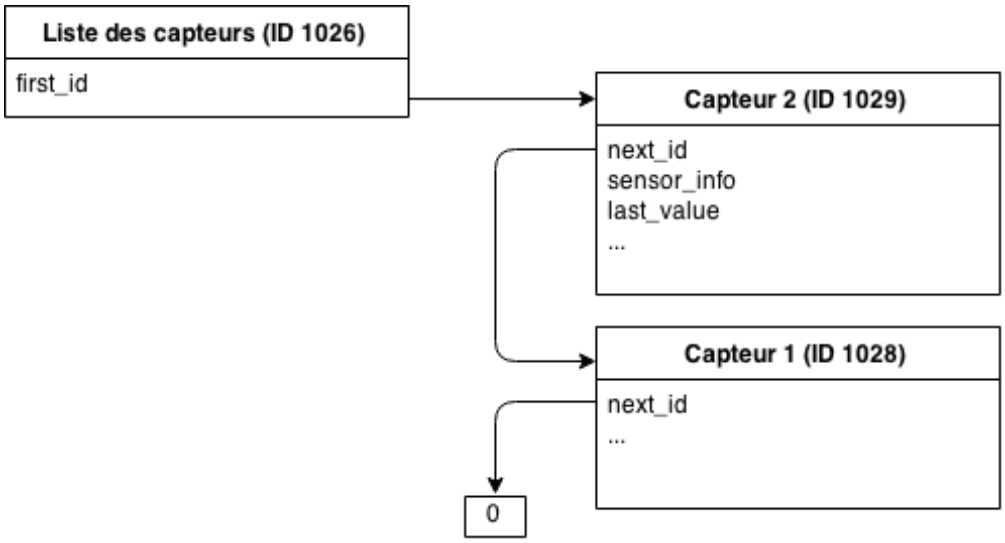


FIGURE 4 – Structure d'une *NvList*

Cela se traduit par les structures C (*list_head_t* pour la racine et *list_item_t* pour les éléments) ci-dessous.

Structure des listes

```
17 /* List layout in NVRAM */
18 typedef struct {
19   uint16 first_id;
20 } list_head_t;
21
22 /* Item layout in NVRAM */
23 typedef struct {
24   uint16 next_id;
25   uint8 data[0];
```

```
} list_item_t;
```

Parmi les fonctions que nous avons implémentées, nous allons détailler les plus importantes qui sont :

— *nvlist_create()* qui réserve un identifiant à une donnée en vue de l'ajouter à la liste
— *nvlist_append()* qui ajoute un élément en tête de liste
— *nvlist_delete()* qui supprime l'élément en tête de liste

6.4.1.1 Fonction *nvlist_create()*

Pour réserver un élément, la fonction *nvlist_create()* détaillée ci-dessous scanne l'ensemble de la plage des IDs utilisables par l'application (paramètres *search_start_id* et *search_count*) puis renvoie le premier ID libre après y avoir copié les données initiales.

Fonction nvlist_create()

```
uint16 nvlist_create(uint16 search_start_id, uint16 search_count, const
    list_item_t* data, int len) {
  uint16 current_id;

  if (data == NULL || len < sizeof(list_item_t)) {
    return 0;
  }

  for (current_id = search_start_id; current_id < (search_start_id +
      search_count); current_id++) {
    if (osal_nv_item_len(current_id) == 0) {
      break;
    }
  }

  /* We didn't find any free ID */
  if (current_id == (search_start_id + search_count)) {
    return 0;
  }

  if (osal_nv_item_init(current_id, len, (void*) data) != NV_ITEM_UNINIT) {
    return 0;
  }

  return current_id;
}
```

6.4.1.2 Fonction *nvlist_append()*

Une autre fonction intéressante à étudier est la fonction *nvlist_append()* donnée ci-dessous qui ajoute un élément en début de liste. On aurait pu également choisir de faire une inser-

tion en fin de liste, mais comme l'ordre des éléments n'a pas vraiment d'importance, nous avons préféré faire les insertions en début car cela est à la fois plus simple et plus rapide (pas besoin de parcourir la liste en entier). L'ajout se fait en deux étapes : on modifie l'élément ajouté en définissant son successeur (variable *item.next_id* de manière à le faire pointer vers le premier élément de la liste, puis on modifie ce dernier pour qu'il pointe vers l'élément ajouté. On peut noter que tout accès (lecture ou écriture) à la mémoire passe par une fonction spécifique (*nvlist_get()* et *nvlist_set()*, ce qui explique pourquoi nous avons dû faire notre propre implémentation. Un schéma explicatif est donné ci-dessous (fig 5)

Fonction nvlist_append()

```
88  bool nvlist_append(uint16 list_id, uint16 item_id) {
89    list_item_t item;
90    list_head_t list;
91
92    if (item_id == 0 || list_id == 0) {
93      return false;
94    }
95
96    if (osal_nv_item_len(item_id) == 0) {
97      return false;
98    }
99
100   /* Make the new item point to the first element of the list before the insertion
        */
101   item.next_id = nvlist_first(list_id, NULL, 0);
102   if (!nvlist_set(item_id, &item, sizeof(list_item_t))) {
103     return false;
104   }
105
106   /* Make the first element of the list the new element */
107   list.first_id = item_id;
108   return nvlist_set(list_id, (list_item_t*)&list, sizeof(list_head_t));
109 }
```

6.4.1.3 Fonction *nvlist_delete()*

Enfin, la fonction *nvlist_delete()* ci-dessous a pour but de supprimer un élément situé à une position arbitraire de la liste. Pour cela, la fonction doit d'abord récupérer l'identifiant de l'élément qui précède celui à supprimer (*prev_id*) ainsi que celui qui le suit (*next_id*). Ensuite, l'élément est supprimé et on réarrange les pointeurs entres ces trois éléments. L'élément précédant (*prev_id*) est modifié de sorte à pointer vers l'élément suivant celui à supprimer (*next_id*). Si l'élément à supprimer était le premier élément de la liste, alors la racine de la liste est modifiée pour pointer sur son successeur. Un schéma explicatif est donné ci-dessous (fig 6)

Fonction nvlist_delete()

```
111 bool nvlist_delete(uint16 list_id, uint16 deleted_id) {
112   uint16 first_id = nvlist_first(list_id, NULL, 0);
113   uint16 pred_id = nvlist_get_prev(list_id, deleted_id, NULL, 0);
```

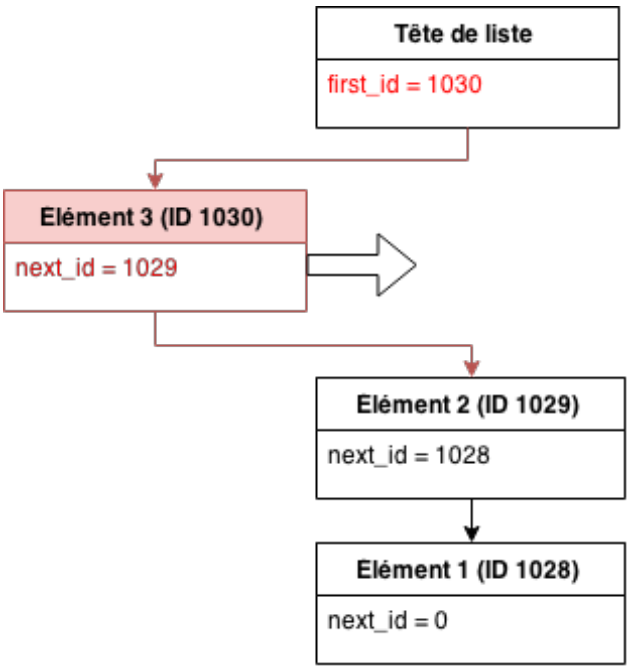


FIGURE 5 – Ajout d'un élément en tête de liste, la couleur rouge indique ce qui est modifié.

```
114   uint16 deleted_len = osal_nv_item_len(deleted_id);
115   uint16 next_id = nvlist_get_next(deleted_id, NULL, 0);
116
117   if (list_id == 0 || deleted_len == 0 || deleted_id == 0) {
118     return false;
119   }
120
121   if (osal_nv_delete(deleted_id, deleted_len) != SUCCESS) {
122     return false;
123   }
124
125   if (pred_id == 0 || deleted_id == first_id) {
126     /* If the deleted item was the first element of the list,
127      make the list start with its successor */
128     osal_nv_write(list_id, 0, sizeof(uint16), &next_id);
129   } else {
130     /* Make the predecessor of the deleted item point to its successor */
131     osal_nv_write(pred_id, 0, sizeof(uint16), &next_id);
132   }
133
134   return true;
135 }
```

Une fois cette implémentation de liste fonctionnelle, nous avons mis en place les différentes structures relatives aux données que nous souhaitions stocker.

Les données que nous avons à sauvegarder sont :

— une liste de capteurs/actionneurs, avec pour chaque élément sa configuration (seuil de déclenchement, type, adresse IEEE), sa dernière valeur connue, sa dernière acti-

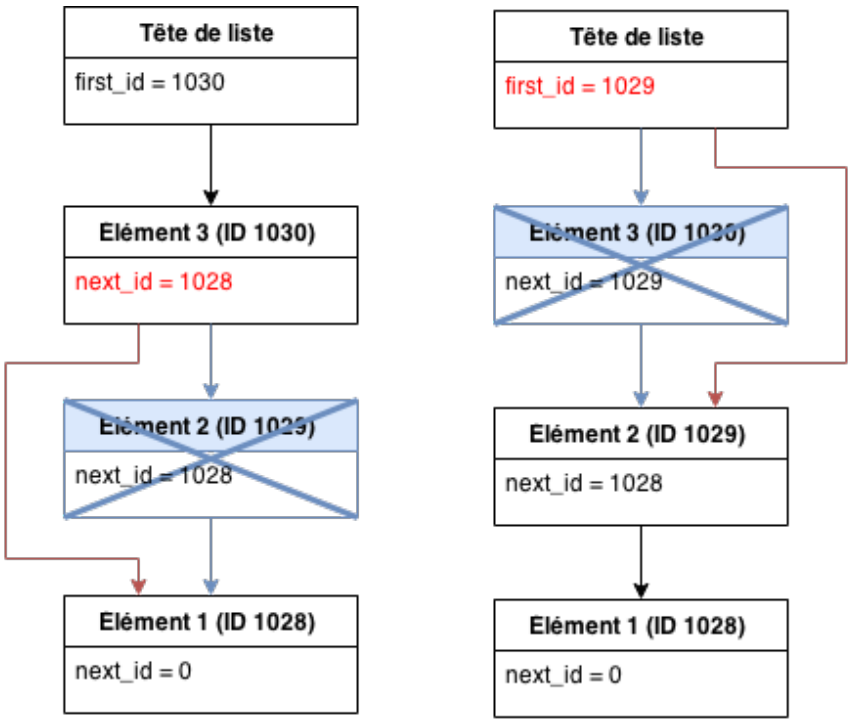


FIGURE 6 – Suppression d'un élément, la couleur rouge indique ce qui est modifié, la couleur bleue ce qui est supprimé.

vité et la qualité du lien radio avec cet élément.

— une liste de règles de déclenchement, chacune comprenant l'adresse IEEE de l'actionneur concerné et une liste des adresses IEEE des capteurs qui doivent tous avoir franchi leur seuil de déclenchement pour déclencher cet actionneur.

6.4.2 Structure *sensor_info_t*

La configuration d'un capteur est représentée sous la forme de la structure C suivante :

Structure *sensor_info_t*

```
29 /** Static information about a sensor, stored in the NVRAM inside sensor_item_t
       structure */
30 __packed typedef struct {
31   ZLongAddr_t ieee_addr;        // Unique 64 bits address
32   sensor_type_t type;
33   sensor_datatype_t datatype;
34   sensor_val_t threshold;
35   threshold_type_t threshold_type;
36 } sensor_info_t; //8 + 1 + 1 + 4 + 1 = 15 bytes
```

Le champ *ieee_addr* désigne l'adresse IEEE du capteur (tableau de 8 octets défini dans le type ZLongAddr_t). Viennent ensuite le type (capteur de luminosité, contact, actionneur, etc.) et le type de données (int, float, bool) qui sont définis dans des énumérations (*sensor_type_t*, *sensor_datatype_t*) détaillées en annexe B.3. Enfin, le champ *threshold* contient la

valeur de seuil utilisée pour déterminer si le capteur est en alarme, associée au mode de franchissement (champ *threshold_type*) de ce seuil ($>$, $\geq$, $<$, $\leq$, $=$ ou désactivé). Le type *sensor_val_t* permet de contenir soit un entier (int), un booléen (bool) ou un réél (float) tout en n'occupant en mémoire que la place requise par le plus grand des trois, ici 4 octets (type union en langage C).

6.4.3 Compactage des structures

On peut remarquer l'attribut *__packed* qui est une extension du compilateur IAR pour spécifier que la structure doit être compactée.

Le compactage d'une structure permet de limiter l'espace perdu entre les champs lorsque ceux ci sont alignés par le compilateur. L'alignement permet de gagner en performances à l'exécution en faisant en sorte qu'un mot de 4 octets par exemple soit placé à une adresse multiple de 4. Dans le cas contraire, le compilateur doit décomposer l'opération en deux accès mémoire pour récupérer l'ensemble de la donnée, puis la copier dans un registre intermédiaire avant d'y effectuer le traitement voulu.

Voici un exemple comparant une même structure, avec et sans alignement (fig 7)

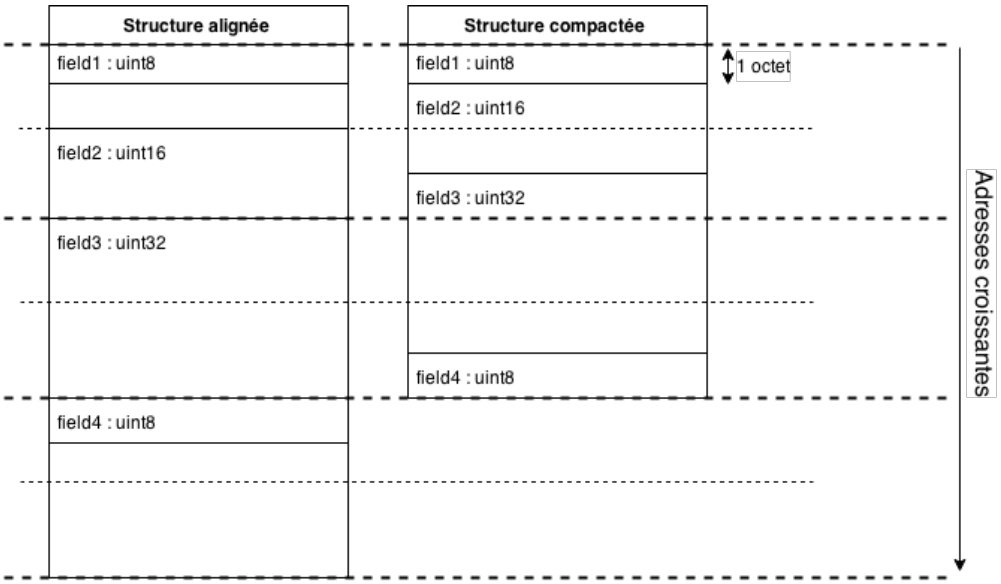


FIGURE 7 – Comparaison de l'espace mémoire occupé par une structure avec et sans alignement

Dans cet exemple, le champ 2 de taille 2 octets a été déplacé de sorte à ce qu'il commence à une frontière de demi-mot (représentées par des lignes pointillées légères). De même, le champ 3 de taille 4 octets est aligné sur une frontière de mot (représentées par des lignes pointillées forte). Enfin, la structure dans son ensemble s'aligne sur une frontière du type le plus volumineux contenu dans celle-ci (ici 4 octets), c'est pourquoi il y a un espace vide après le champ 4.

Pour plus d'informations sur l'alignement des structures, vous pouvez consulter ce lien [2]

2. http://www.catb.org/esr/structure-packing/

Notre structure *sensor_info_t* doit être stockée dans la mémoire Flash qui est en quantité limitée, c'est pourquoi nous avons préféré économiser cette dernière au détriment de la perte de performance induite (qui est de toute manière négligeable).

6.4.4 Liste des capteurs

La liste des capteurs se compose d'éléments stockés en mémoire Flash qui suivent la structure suivante :

Structure sensor_item_t

```
46 /** Sensor state
47   * contains static info (sensor_info_t) as well as last value and last
         communication time */
48 __packed typedef struct {
49   uint16 next_id;
50   sensor_info_t info;
51   uint32 last_seen_sec;      // Time of last communication with the sensor (in
         seconds,
52                              // measured relative to the coordinator start-up (0))
53   sensor_val_t last_val;     // last data communicated by the sensor
54   uint8 LQI;                 // Link quality indicator
55 } sensor_item_t; //2 + 15 + 4 + 4 + 1 = 26 bytes
```

Comme tout élément d'une NvList, cette structure débute par un champ contenant l'ID de l'élément suivant (ou 0 s'il s'agit du dernier). On retrouve ensuite notre structure *sensor_info_t* puis la date de dernière activité, la valeur actuelle du capteur ou actionneur et le LQI (Link Quality Indicator) qui donne une indication sur la puissance du signal. La date de dernière activité est exprimée en secondes écoulées depuis l'allumage du coordinateur (ce dernier fournit à l'interface graphique une commande permettant de savoir combien de temps s'est écoulé depuis son allumage pour "synchroniser" les horloges).

6.4.5 Liste des règles de déclenchement

La liste des règles de déclenchement contient des éléments suivant la structure *trigger_event_t* suivante :

Structure trigger_event_t

```
64 __packed typedef struct {
65   uint16 next_id;
66   uint16 addr_list_id;
67   ZLongAddr_t ieee_addr_target;
68 } trigger_event_t;
```

On retrouve encore une fois l'ID de l'élément suivant (*next_id*) suivi d'un champ contenant l'ID dans lequel sera stocké la liste des capteurs qui doivent être en alarme pour déclencher la règle. Vient enfin l'adresse IEEE de l'actionneur qui sera déclenché (notons qu'il est possible de faire plusieurs règles pour un actionneur donné, auquel cas il suffira

qu'une seule soit vérifiée pour que l'actionneur soit déclenché).

Après avoir vu la façon dont sont stockées les données, nous allons maintenait nous intéresser à la logique de traitement des messages provenant des capteurs.

6.5 Traitement des messages depuis les capteurs

La fonction *handle_message* présentée ci-dessous est appelée dans le corps du coordinateur à la réception d'un message.

Fonction de traitement des messages depuis les capteurs

```
33 void handle_message(afIncomingMSGPacket_t *pkt) {
34   sensor_item_t item;
35   ZLongAddr_t ieee_addr;
36   sensor_state_t *state;
37   uint16 item_id;
38   bool changed = false;
39
40   if (pkt->srcAddr.addrMode == afAddr16Bit) {
41     APSME_LookupExtAddr(pkt->srcAddr.addr.shortAddr, ieee_addr);
42   } else if (pkt->srcAddr.addrMode == afAddr64Bit) {
43     memcpy(ieee_addr, pkt->srcAddr.addr.extAddr, Z_EXTADDR_LEN);
44   } else {
45     // Other addressing modes not handled
46     return;
47   }
48   state = (sensor_state_t *) pkt->cmd.Data;
49
50   if (!(item_id = get_sensor_item_from_ieee(ieee_addr, &item))) {
51     changed = true;
52     memset(&item, 0, sizeof(sensor_item_t));
53     memcpy(item.info.ieee_addr, ieee_addr, Z_EXTADDR_LEN);
54     //We create another sensor_item in the nvram
55     item_id = nvlist_create(FREE_START_ID, FREE_ID_COUNT, (list_item_t *) &item,
          sizeof(sensor_item_t));
56     //We append this item at the beginning of the list
57     nvlist_append(SENSOR_ITEM_LIST_ID, item_id);
58     nvlist_get(item_id,(list_item_t *) &item, sizeof(sensor_item_t));
59   } else if (memcmp(&item.last_val, &state->val, sizeof(sensor_val_t)) != 0) {
60     changed = true;
61   }
62
63   item.LQI = pkt->LinkQuality;
64   item.info.datatype = state->datatype;
65   item.info.type = state->type;
66   item.last_val = state->val;
67   item.last_seen_sec = osal_getClock();
68   nvlist_set(item_id, (list_item_t *) &item, sizeof(sensor_item_t));
69
70   if (is_sensor_alarmed(&item)) {
71     HalLedSet(HAL_LED_1, HAL_LED_MODE_BLINK);
```

```
72    }
73
74    if (changed) {
75      evaluate_triggers();
76    }
77  }
```

Elle commence ainsi par convertir l'adresse source du message en adresse longue (IEEE 64 bits) si nécessaire à l'aide de la fonction *APSME_LookupExtAddr*.

Une fois l'adresse récupérée, on recherche dans la liste des capteurs à l'aide de la fonction *get_sensor_item_from_ieee* le capteur correspondant et le stocke dans la variable *item*.

Si l'adresse n'existe pas (première branche du *if*), on l'ajoute à la liste. Pour cela on initialise un élément vide (en le remplissant de zéros avec la fonction*memset*) puis on y recopie son adresse (*memcpy*) pour enfin le stocker dans un ID libre de la mémoire. Enfin, on l'ajoute à la liste des capteurs à l'aide de la fonction nvlist_append() détaillée en section 6.4.1.2). Comme cette dernière va modifier l'item ajouté pour le faire pointer vers son successeur, il est nécessaire de refaire une lecture depuis la mémoire avec la fonction *nvlist_get()* afin de ne pas écraser le successeur lors de la future écriture. On met également à vrai un booléen *changed* pour indiquer que la valeur du capteur a été mise à jour (puisqu'il vient d'être créé).

Dans le cas où l'adresse existe, on compare la valeur actuelle du capteur avec la nouvelle valeur, et on met le booléen *changed* à vrai si ces deux valeurs différentes.

Après cela, on met à jour l'élément avec les nouvelles données reçues du capteur puis on fait clignoter brièvement une LED si le capteur a franchi son seuil de déclenchement. Enfin, si la valeur a été mise à jour, on évalue les règles de déclenchement en appelant la fonction *evaluate_triggers()* que nous allons maintenant étudier.

6.5.1 Évaluation des règles de déclenchement

La fonction *evaluate_triggers()* est donnée ci-dessous.

Fonction de traitement des règles de déclenchement

```
79  void evaluate_triggers() {
80    trigger_event_t rule;
81    uint16 rule_id = nvlist_first(TRIGGER_EVENT_LIST_ID, (list_item_t*)&rule,
          sizeof(rule));
82    uint8 matched_count = 0;
83
84    while (rule_id != 0) {
85      addr_list_t addr;
86      uint16 addr_id = nvlist_first(rule.addr_list_id, (list_item_t*)&addr,
            sizeof(addr));
87
88      bool all_triggered = true;
89
90      while (addr_id != 0) {
```

```
91        sensor_item_t sensor;
92        uint16 sensor_id = get_sensor_item_from_ieee(addr.ieee_addr, &sensor);
93
94        if (sensor_id == 0 || !is_sensor_alarmed(&sensor)) {
95          all_triggered = false;
96          break;
97        }
98
99        addr_id = nvlist_get_next(addr_id, (list_item_t*)&addr, sizeof(addr));
100     }
101
102     if (all_triggered) {
103       matched_count++;
104     }
105
106     sensor_item_t actuator_item;
107     uint16 actuator_id = get_sensor_item_from_ieee(rule.ieee_addr_target,
          &actuator_item);
108
109     if (actuator_item.info.datatype != BOOL || actuator_item.last_val.bool_val !=
          all_triggered) {
110       Coordinator_SendMessage(rule.ieee_addr_target, (uint8*)&all_triggered, 1);
111     }
112     rule_id = nvlist_get_next(rule_id, (list_item_t*)&rule, sizeof(rule));
113   }
114
115   HalLcdWriteStringValue( "Active triggers: ", matched_count, 10, HAL_LCD_LINE_4 );
116 }
```

Elle parcourt la liste des règles, puis elle vérifie si tous les capteurs concernés par une règle sont bien en état d'alerte (variable *all_triggered*) ce qui donne la valeur d'activation de l'actionneur concerné par la règle. Si cette valeur est différente de la valeur précédente, on envoie alors un message à l'actionneur pour lui communiquer son nouvel état d'activation.

6.5.2 Programme gérant les capteurs et actionneurs

La gestion des capteurs et actionneurs est assurée par un programme à part, nommé *Sensor*. Il peut fonctionner à la fois en tant qu'end-device ou en tant que routeur en changeant la configuration du projet dans IAR. Le choix du type de capteur se fait en modifiant une variable avant la compilation de celui-ci (il n'y a qu'un seul programme pour tous les types de capteurs et d'actionneurs).

Il est constitué principalement d'une fonction d'envoi appelée périodiquement (toutes les 5 secondes) et d'une fonction appelée à la réception d'un message depuis le coordinateur.

La fonction appelée à la réception d'un message est montrée ci-dessous. Elle n'est utile que pour les actionneurs puisqu'un capteur n'a pas de message à recevoir du coordinateur.

Fonction de réception d'un message

```
443 static void Sensor_MessageMSGCB( afIncomingMSGPacket_t *pkt )
444 {
445   if (pkt->clusterId != SENSOR_CLUSTERID) {
446     return;
447   }
448
449   /* Ignore messages not coming from the coordinator */
450   if (pkt->srcAddr.addrMode != afAddr16Bit || pkt->srcAddr.addr.shortAddr != 0) {
451     return;
452   }
453
454   switch (Sensor_State.type)
455   {
456   case LED_ACTUATOR:
457     /* we're a LED actuator, just read a boolean value and set our LED accordingly
          */
458     bool on = *((bool*)pkt->cmd.Data);
459     bool previous = (HalLedGetState() & HAL_LED_1) != 0;
460     if (on != previous) {
461       HalLedSet(HAL_LED_1, on ? HAL_LED_MODE_ON : HAL_LED_MODE_OFF);
462       Sensor_SendTheMessage(); /* immediately send our new state back to the
            coordinator */
463     }
464     break;
465   }
466 }
```

Elle commence par vérifier que le message provient bien du coordinateur avant tout autre traitement puis décide du traitement à appliquer en fonction du type d'actionneur (il n'y en a ici qu'un seul qui est une LED qui peut s'allumer ou s'éteindre). Dans le cas de la LED, le message reçu est directement convertit en valeur booléenne (0 ou 1) indiquant si la LED doit s'allumer (1) ou s'éteindre (0). Enfin on renvoie le nouvel état au coordinateur afin de confirmer que la modification a bien été prise en compte.

Intéressons-nous maintenant à la fonction d'envoi d'un message appelée toutes les 5 secondes :

Fonction d'envoi d'un message

```
477 static void Sensor_SendTheMessage( void )
478 {
479   switch (Sensor_State.type) {
480   case SWITCH_SENSOR:
481   case LED_ACTUATOR:
482     Sensor_State.val.bool_val = HalLedGetState() & HAL_LED_1;
483     HalLcdWriteStringValue("Val: ", Sensor_State.val.bool_val, 10, HAL_LCD_LINE_6);
484     break;
485   case LIGHT_SENSOR:
486     Sensor_State.val.int_val = alsRead();
487     HalLcdWriteStringValue("Val: ", (uint16) (Sensor_State.val.int_val), 10,
          HAL_LCD_LINE_6);
```

```
488     break;
489   }
490
491   if ( AF_DataRequest( &Sensor_DstAddr, &Sensor_epDesc,
492                        SENSOR_CLUSTERID,
493                        sizeof(sensor_state_t),
494                        (byte *) &Sensor_State,
495                        &Sensor_TransID,
496                        AF_DISCV_ROUTE, AF_DEFAULT_RADIUS ) == afStatus_SUCCESS )
497   {
498     // Successfully requested to be sent.
499     HalLedSet ( HAL_LED_2, HAL_LED_MODE_BLINK );
500     HalLedSet ( HAL_LED_4, HAL_LED_MODE_OFF );
501   }
502   else
503   {
504     // Error occurred in request to send.
505     HalLedSet ( HAL_LED_4, HAL_LED_MODE_FLASH );
506   }
507 }
```

Tout comme pour la fonction de réception, son comportement dépend du type de capteur (ou d'actionneur).

Dans le cas où il s'agit d'un capteur ou actionneur booléen (représenté par une LED), on stocke directement la valeur actuelle de la LED dans la variable *Sensor_State.val*. On en profite aussi pour afficher la valeur sur l'écran LCD de la carte.

S'il s'agit d'un capteur de luminosité, la valeur est récupérée à l'aide de la fonction *alsRead()* qui permet de faire une lecture de l'entrée analogique connectée au capteur de luminosité. Elle est ensuite stockée également dans la variable *Sensor_State.val*.

La variable *Sensor_State.val* contient donc maintenant la nouvelle valeur correspondant au type de capteur ou actionneur. Il ne reste alors plus qu'à envoyer cette valeur au coordinateur à l'aide de la fonction *AF_DataRequest()*.

Maintenant que nous avons traité le fonctionnement du coordinateur du point de vue réseau, nous allons maintenant nous intéresser à la partie communication avec l'interface graphique.

6.6 Protocole de communication

Nous allons maintenant voir le protocole de communication qui a été établi entre le coordinateur et le logiciel d'une part, puis entre le coordinateur et les capteurs d'autre part.

6.6.1 Entre le coordinateur et le logiciel

Plusieurs fonctions sont essentielles pour la communication entre le coordinateur et le logiciel Windows : obtenir la liste des capteurs et actionneurs, ajouter/modifier/suppri-

mer un capteur ou un actionneur et ajouter/supprimer des règles de déclenchement mais aussi obtenir le temps actuel sur le coordinateur et pouvoir répondre à un "PING". Nous allons maintenant voir plus en détails ces différentes fonctions et de quelle manière elles sont utilisées.

Afin de communiquer les différentes données entre le coordinateur et le logiciel, la communication se fait via un port série* virtuel (qui utilise la liaison USB du coordinateur). Via ce port série, le logiciel Windows envoi un octet (ou byte) de commande tel que décrit dans l'énumération suivante :

Liste des codes de commande

```
71 /** Commands that can be sent to the coordinator through the serial port */
72 typedef enum {
73   CMD_GET_SENSOR_LIST = 0,
74   CMD_DELETE_SENSOR = 1,
75   CMD_MODIFY_SENSOR = 2,
76   CMD_DUMP_SENSORS = 3,
77   CMD_ADD_RULE = 4,
78   CMD_GET_RULE_LIST = 5,
79   CMD_DELETE_RULE = 6,
80   CMD_DUMP_RULES = 7,
81   CMD_GET_TIME = 8,
82   CMD_SEND_DATA = 9,
83   CMD_NVRAM_INFO = 10,
84   CMD_PING = 0x42,
85   CMD_CLEAR_ALL = 0xA0,
86 } operation_code_t;
```

Une fois cet octet de commande envoyé, le logiciel va envoyer les arguments de la fonction si besoin. Par exemple, la fonction *delete_sensor* demande l'adresse IEEE en argument sous la forme de 8 octets successifs.

En réponse, le coordinateur va renvoyer plusieurs octets selon la commande envoyée (*cf* les fonctions décrites plus loin). Finalement, le coordinateur va envoyer un octet pour le code d'erreur, tel que définit dans l'énumération ci-dessous, afin de savoir si tout s'est bien passé ou si une erreur quelconque a eu lieu. Dans le cas des fonctions qui ne font qu'envoyer des données sans affecter les données du coordinateur, comme *send_sensor_list* ou *send_rule_list*, aucun code d'erreur n'est envoyé via le port série (puisque celui ci serait toujours *ERROR_SUCCESS* si on devait en mettre un).

Liste des codes d'erreur

```
89 typedef enum {
90   ERROR_SUCCESS = 0,       /* no error */
91   ERROR_INVALID_FORMAT = 1, /* some field was missing or had an invalid value */
92   ERROR_NOT_FOUND = 2,     /* requested item was not found */
93   ERROR_INTERNAL = 3,      /* internal error */
94   ERROR_UNKNOWN = 0xFF,    /* unknown error */
95 } error_code_t;
```

Nous allons maintenant voir un peu plus en détail les différentes fonctions implémentées

et leurs utilités. La plus intéressante est sûrement la fonction *send_sensor_list*, cette fonction est appelée lors de la réception du byte de commande *CMD_GET_SENSOR_LIST*. Cette fonction ne demande pas d'argument et va d'abord renvoyer, sur le port série, un octet pour le nombre de capteurs puis, pour chaque capteur, elle va écrire une série d'octets représentant la structure *sensor_info_t* (détaillée plus haut en section 6.4.2), puis quatre octets pour la date de dernière activité, plusieurs octets pour la structure *sensor_val_t*, un octet pour le statut d'alerte et enfin un octet pour la qualité du signal. Cette fonction va donc permettre de connaître l'état de tous les capteurs et est très utile pour surveiller en permanence l'état du réseau.

Une autre fonction similaire est *send_rule_list*, celle ci est appelée lors de la réception du byte de commande *CMD_GET_RULE_LIST*. Cette fonction ne demande pas non plus d'argument et va renvoyer en premier le nombre de règles de déclenchement qui ont été définies suivi, pour chaque règle, de l'identifiant de cette règle (pour pouvoir la supprimer), de l'adresse IEEE de l'actionneur concerné, du nombre de capteurs concernés par la règle suivi de l'adresse IEEE de chacun de ces capteurs.

Pour ajouter ou modifier un capteur ou un actionneur sur le réseau. Il suffit d'envoyer le byte de commande *CMD_MODIFY_SENSOR* qui exécutera la fonction *modify_sensor*. Cette fonction prend en argument la structure *sensor_info_t* envoyée comme une série de bytes via le port série. Cette structure comprenant l'adresse IEEE du capteur à ajouter ou modifier, la fonction n'aura pas de mal à faire les changements au bon endroit. Finalement, cette fonction renverra le code d'erreur *ERROR_SUCCESS* si tout s'est bien passé ou *ERROR_INVALID_FORMAT* si une erreur au niveau des données reçues s'est présentée.

Pour supprimer un capteur ou un actionneur, il s'agit de la commande *CMD_DELETE_SENSOR* associée à la fonction *delete_sensor*. Elle prend l'adresse IEEE du capteur à supprimer sous la forme de 8 octets successifs. Cette fonction renvoi un byte d'erreur qui peut-être *ERROR_SUCCESS* si il n'y a pas d'erreur, *ERROR_INVALID_FORMAT* si le coordinateur n'a pas pu récupérer 8 octets en provenance du port série, *ERROR_NOT_FOUND* si le capteur à supprimer n'a pas été trouvé ou a déjà été supprimé ou *ERROR_INTERNAL* dans le cas exceptionnel où la suppression du capteur n'aurait pas été possible sur le coordinateur. Ce dernier cas n'est pas censé arriver mais pourrait peut-être arriver dans le cas d'erreurs de lecture ou écriture dans la mémoire par exemple (qui pourrait être la cause d'une mémoire défectueuse).

Les fonctions *add_rule* et *delete_rule* ont un comportement similaire aux fonctions *add_sensor* et *delete_sensor*. Il reste deux fonctions intéressantes qui sont *ping* et *send_time*. La première est une fonction très simple, répondant à la commande *CMD_PING*, qui va envoyer la chaîne de caractère *"PONG !"* sur le port série (sous la forme de 5 octets successifs). Cette fonction sert tout simplement à identifier que la communication sur un port série se fait bien avec le coordinateur et qu'on ne s'est donc pas trompé de port série. La deuxième fonction, répondant à la commande *CMD_GET_TIME*, va renvoyer, sur quatre octets, le nombre de secondes qui se sont écoulées depuis le démarrage du coordinateur. Cette fonction est importante pour synchroniser la date de dernière activité avec le temps réel sur le logiciel Windows.

D'autres commandes comme *CMD_DUMP_SENSORS*, *CMD_DUMP_RULES*, *CMD_SEND_DATA*, *CMD_NVRAM_INFO* et *CMD_CLEAR_ALL* sont des commandes qui ont été créées pour le débogage. Ces commandes n'ont donc pas été implémentées dans le logiciel Windows

mais nous avons utilisé le logiciel RealTerm [3] pour pouvoir écrire manuellement et recevoir des bytes sur le port série.

6.6.2 Entre le coordinateur et les capteurs

La communication entre le coordinateur et les capteurs / actionneurs est beaucoup plus simple puisque il y a beaucoup moins d'informations à transmettre. Dans un premier temps, nous nous sommes contentés de simplement envoyer la valeur correspondante à l'état du capteur (valeur de la luminosité pour le capteur de luminosité et un booléen pour l'interrupteur) et de l'actionneur (booléen selon si il est déclenché ou pas). Puis nous nous sommes aperçus qu'il était intéressant de communiquer le type d'appareil dont il s'agissait (capteur de luminosité, interrupteur ou actionneur). Cette donnée est importante puisqu'elle permet de définir quelle est le type de données (entier ou booléen ?) que l'on va recevoir et le traitement de ces données aura des conséquences immédiates (on ne peut pas manipuler un booléen comme un entier).

En effet, à la base, via l'interface du logiciel Windows, on devait indiquer manuellement de quel type d'appareil il s'agissait pour pouvoir définir les seuils (définissant si un capteur est en alerte ou non) adéquats. Mais ceci nous a semblé dommage de le définir manuellement alors que le capteur connaît, via le programme qu'on lui a transféré, son type et on pourrait très bien l'imposer à l'utilisateur. Après tout, si l'utilisateur en venait à se tromper dans la sélection du type, on aurait des problèmes de compatibilité entre les données reçues et le traitement de ces données.

Donc finalement, périodiquement, les capteurs et actionneurs communiquent 3 données (*cf* l'annexe B.1) :

— Leur valeur pouvant être un booléen, un entier ou un nombre à virgule. Celle ci est codée sur 4 bytes qui correspondent à la taille maximum, celle du nombre à virgule, que peut prendre cette valeur.

— Leur type codé sur un byte. Celui ci peut-être un interrupteur, un capteur de luminosité ou un actionneur à LED.

— Le type de données codé sur un byte. Celui ci représentant, fidèlement à sa valeur, le type booléen, entier ou nombre à virgule.

Une fois ce protocole de communication mis en place, il ne faut pas négliger un détail plus ou moins important selon l'utilisation qui est faite de ce réseau de capteurs sans fil. Il s'agit la sécurité du réseau que nous allons maintenant aborder.

6.7 Sécurité du réseau

Sans un réseau sécurisé, il est très facile, à l'aide d'un sniffer* adapté, de pouvoir récupérer l'ensemble des informations qui transitent sur le réseau. Par exemple, si une utilisation d'un réseau de capteur sans fil consiste à détecter la présence d'individus dans un bâtiment, la non-sécurisation de ce réseau engendrerait la possibilité pour une personne mal

3. RealTerm : http://realterm.sourceforge.net/

intentionnée de savoir lorsque les lieux sont déserts afin d'y commettre un délit quelconque.

En sécurisant un réseau, les données qui circulent seront chiffrées et des individus mal intentionnés ne pourront théoriquement pas les déchiffrer. En effet, on est jamais à l'abri d'une faille de sécurité ou d'une fuite des clés de chiffrement utilisées. Mais il est possible de minimiser les risques en utilisant des systèmes qui semblent de confiance et en multipliant les moyens de sécuriser ces données.

Pour ce projet, nous avons choisi de mettre en place le système proposé de base par ZigBee. Ce système utilise une variante de la méthode de chiffrement CCMP basée sur AES (un algorithme de chiffrement symétrique). Cette méthode utilise un compteur couplé à un système d'authentification [12]. Le compteur permet de ne jamais chiffrer de la même façon les messages et l'authentification permet de vérifier que le message n'a pas été altéré. Aucune faille n'a été découverte à ce jour.

Pour pouvoir configurer ce système dans notre projet, il a fallu tout d'abord rajouter une directive de compilation *SECURE=1*. Ensuite plusieurs choix nous étaient possibles. Le premier concerne la *network key*. Celle ci est la clé qui est utilisé pour chiffrer toutes les communications au sein du réseau. Si quelqu'un venait à mettre la main dessus, il serait donc capable de déchiffrer toutes ces communications. ZigBee nous donne le choix d'utiliser une clé par défaut sur tous les appareils ZigBee ou de la communiquer au moment où un appareil rejoint le réseau. L'inconvénient de la première est qu'il n'est plus possible de changer la *network key* sans faire attention à ce que les nouveaux capteurs ou actionneurs voulant rejoindre le réseau aient la bonne clé par défaut. L'inconvénient de la deuxième possibilité est qu'il existe un potentiel moment de vulnérabilité lorsque la clé est transmise à un nouvel appareil souhaitant rejoindre le réseau. Mais l'avantage est qu'il est possible pour le coordinateur de changer régulièrement la clé de réseau afin d'augmenter davantage la sécurité du réseau.

Un autre choix à faire concerne la *tc_link_key*. Cette clé est utilisée afin de chiffrer la *network_key* pour les appareils souhaitant rejoindre le réseau. Par défaut sa valeur correspond à la chaîne de caractère *"ZigBeeAlliance09"* pour être compatible avec des réseaux de capteurs sans fil existants. En laissant cette valeur par défaut, on aura effectivement un moment de vulnérabilité, comme expliqué dans le paragraphe précédent, puisque il sera possible de déchiffrer la *network_key* avec cette clé par défaut. Mais comme pour ce projet, nous n'avons pas d'intérêt à garder la valeur par défaut, et avons donc décidé de la changer d'une manière aléatoire.

Une autre possibilité est d'utiliser un type différent pour les *Link Keys*. Celui par défaut est le type *Global* alors que l'autre type est dit *Unique*. Ce dernier est similaire au premier sauf qu'il n'utilise pas qu'une seule *tc_link_key* mais il en utilise une par appareil différent. Bien que cette deuxième solution semble plus sécurisée, elle est aussi plus contraignante et elle n'a pas été choisie pour ce projet.

De ce fait, nous avons choisi de ne pas utiliser de *network_key* par défaut, de changer la *tc_link_key* par défaut et de garder le type *Global*. De cette manière, la *tc_link_key* est connue de tous les appareils ZigBee dès le départ, celle ci assure qu'un appareil tentant de rejoindre le réseau ne pourra pas le faire si il n'a pas la bonne clé pour déchiffrer la *network_key* qu'il va recevoir. On peut donc être sûr qu'un appareil réussissant à rejoindre le réseau est bien un appareil qui est autorisé à le faire.

Nous avons vu comment s'est déroulé la partie embarquée de ce projet, nous allons maintenant voir ce qu'il en est de la partie Windows.

7 Déroulement de la partie Windows

En plus de la partie embarquée avec la programmation sur cartes, nous avons développé un logiciel pour tablette Windows en C#, permettant de superviser le réseau de capteurs. L'environnement de développement utilisé était Microsoft Visual Studio.

Nous allons dans un premier temps nous pencher sur les avantages de ce langage.

7.1 Avantages du langage C#

Le C# est un langae créé par Microsoft et qui est apparu sur le marché en 2000. Il s'agit d'un langage orienté objet, c'est-à-dire basé sur le concept d'objets. Concrètement, cela signifie que chaque objet possède des attributs et des propriétés, ainsi que des méthodes permettant d'interagir avec d'autres objets du programme, le tout afin de former une entité globable répondant aux attentes des utilisateurs. D'autres langages sont basés sur ce principe, notamment /em Java.

Les langages orientés objets se prettent bien pour des projets de grande taille, où des objets physiques doivent être représentés. C'est pour cela qu'un langage de cette catégorie a été choisi pour l'application (analogie entre les composants réels et leur représentation sous forme d'objets dans le logiciel).

Ce qui a poussé au choix du langage C# était la spécification par l'industriel que l'application devrait tourner sur tablette Windows. Or, le langage se prêtant le mieux pour déployer des solutions logicielles sur des composants Microsoft est C#. En effet, ce langage a été concue pour développer des applications sur PC, puis dans les dernières années a été utilisé pour coder les logiciels de la tablette *Microsoft Surface* et du téléphone *Windows Phone*. Notre projet devant être déployé sur tablette, il s'agissait donc de la meilleure solution logicielle à utiliser.

Après avoir détaillé le choix du langage, nous allons passer à la description des objectifs attendus par le client pour l'interface.

7.2 Objectifs de l'interface

Les buts fixés pour le développement de cette interface étaient principalement d'implémenter une solution au design simple, intuitive, et facile d'utilisation, tout en étant ergonomique. Une quelconque personne devrait être capable, dès qu'elle aurait l'interface sous ses yeux, de savoir s'en servir immédiatement et sans aucune formation. Ainsi, l'ergonomie a été une notion extrèmement importante lors du développement.

Par rapport aux choix faits en rapport à l'ergonomie, il fallait que tous les éléments contribuent à ce but-là. Qu'il s'agisse des icônes choisies, ou bien encore de la disposition des différents composants, tout devait être calculé afin d'être le plus simple possible. Une vue de l'interface est présentée sur la figure 8.

Les différentes parties de l'interface vont être détaillées dans les paragraphes qui suivent.

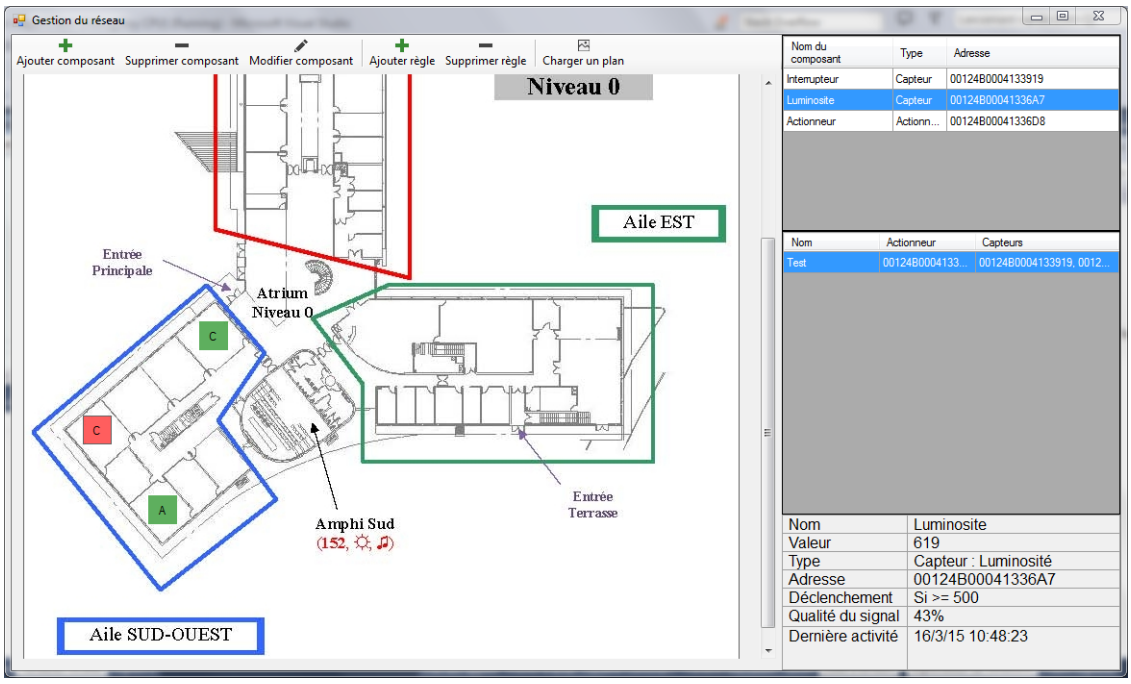


FIGURE 8 – Screenshot de notre interface

Bien évidamment, le logiciel possède égalament une partie pour gérer le déclenchement des divers évènements, aussi bien avec l'utilisateur qu'avec le coordinateur. Ces différents élèments seront détaillées après la présentation sur le design de l'interface.

7.3 Design de l'interface

Tout d'abord, il faut préciser que nous avons dans un premier temps développé le design de l'interface graphique par rapport à la communication avec les cartes, car il fallait d'abord établir le protocole sur les cartes avant d'écrire le code correspondant pour l'interface. Bien que le design et l'implémentation des composants sur l'interface aient légèrement changés au cours du projet, les principales idées de départ se retrouvent sur le design final.

Dans un premier temps, il est important de noter les différentes fonctions que permet cette interface. Elle permet de créer les composants (capteurs et actionneurs) se trouvant sur le réseau ZigBee et d'interagir avec. Il est bien entendu possible de modifier ou supprimer ces capteurs. L'interface permet aussi de créer des alertes, c'est-à-dire qu'un certain actionneur sera activé si un ou plusieurs capteurs renvoient une valeur dépassant un certain seuil. On peut par exemple prendre le cas d'une caméra qui serait déclenché si une personne franchit deux portes consécutivement. Comme pour les composants, il est possible de supprimer les règles. Enfin, l'interface permet de charger un plan de l'endroit où les capteurs seront disposés afin de visualiser géographiquement le réseau ZigBee dans son ensemble.

Chaque composant est caractérisé par des propriétés spécifiques. Ainsi, un actionneur possède un nom, un type, une adresse IEEE, une qualité du signal mesurable en fonction de la distance, et aussi une date et heure permettant de savoir quel a été le dernier moment où le capteur a donné signe de vie. Un capteur possède les mêmes propriétés, mais en plus

il inclut la possibilité de déclencher une alarme.

Au niveau des alertes, celles-ci sont caractérisées par un nom, un actionneur à déclencher, et le ou les capteurs qui seraient déclenchés si le seuil d'alerte des capteurs est atteint. Toutefois, pour que l'alerte soit déclenchée, il faudra que tous les seuils des capteurs concernés par l'alerte soit atteint.

Notre première démarche, tel que suggérée par l'industriel, était de concevoir une interface en deux grandes parties : une pour l'affichage de la carte où les capteurs seraient déployés, et une deuxième pour la visualisation des données de chaque capteur. À partir de cette hypothèse de départ, nous avons développé un squelette d'interface implémentant ces deux aspects, qui servira de base pour la suite du projet.

La partie gauche de l'interface est réservée à l'affichage du plan de la salle où l'on souhaite déployer les capteurs. Dans le cas de l'exemple, le plan utilisé est celui du rez-de-chaussée de TELECOM Nancy [5], car c'est à cet étage que les soutenances se feront. Ainsi, cela permettra de faire une démonstration en direct du réseau de capteurs.

La partie droite de l'interface est occupée par un panneau permettant de lister les différents composants (capteurs et actionneurs) se trouvant sur la carte, ainsi que la liste de tous les événements associés. Une partie est également réservée en bas pour l'affichage des différentes données du composant sélectionné.

Enfin, une barre d'outils, en haut de l'interface, permet d'effectuer les différentes actions sur l'interface, à savoir l'ajout et la suppression de composants et événements. Nous détaillerons ces différentes actions dans la suite de cette partie.

Deux différences importantes au niveau du design sont à noter par rapport à la version intermédiaire du logiciel présentée en décembre. Tout d'abord, nous avons rajoutés la liste des événements qui n'existait pas avant. Enfin, nous avons décidé de modifier le positionnement des icônes permettant d'effectuer des actions sur l'interface. Cette décision a été prise suite à une concertation avec l'industriel concernant l'ergonomie du logiciel. En effet, puisqu'il était nécessaire de rajouter la liste d'événements sur l'interface, il fallait modifier le design de la partie droite. Or, dans tout logiciel, les barres d'outils permettant d'effectuer des actions sur le logiciel se trouvent la plupart du temps en haut de l'interface. Cette nouvelle solution apparaissait donc comme beaucoup plus intuitive et ergonomique, rentrant donc bien dans nos objectifs principaux concernant le design de l'interface.

Les différentes actions possibles pour l'utilisateur interagissant avec le logiciel se trouvent dans la barre d'outils décrite précédemment. Le premier permet d'ajouter la représentation d'un composant. L'utilisateur sélectionnera le type de composant voulu, à savoir un capteur ou un actionneur. Ce composant est ensuite automatiquement positionné au centre de la carte. Pour le déplacer, il suffit de cliquer dessus et de le déposer à l'endroit voulu (drag and drop). La boîte de dialogue permettant de sélectionner les options est montrée en figure 9.

Cette boîte de dialogue permet de choisir un nom, une adresse et un type. Lorsque des composants sont déjà connectés au réseau, le logiciel détecte automatiquement les adresses IEEE des composants déjà présents sur le réseau, et l'utilisateur n'a qu'à sélectionner l'une d'elles. Cependant, il est aussi possible de rentrer l'adresse manuellement.

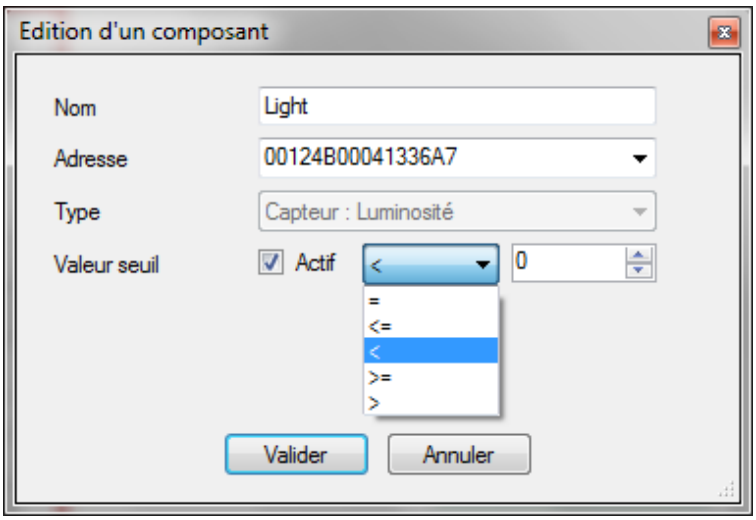


FIGURE 9 – Ajout d'un composant

Trois types de composants sont gérés par le réseau : les interrupteurs (capteur), les capteurs de lumière, et les LEDs (actionneur). Dans le cas où l'un des deux types de capteurs est sélectionné, il est possible de préciser si l'état d'alerte est actif, c'est-à-dire si ce capteur peut générer une alerte sous condition que sa valeur seuil soit atteinte.

Dans le cas d'un interrupteur, il n'y a pas de valeur à préciser, car c'est le fait de déclencher l'interrupteur qui est pris en compte. En revanche, dans le cas du capteur de luminosité, l'utilisateur pourra choisir une valeur de seuil à laquelle déclencher l'alerte. Il pourra également choisir si l'alerte est déclenché lorsque la valeur est égale, supérieure, supérieure ou égale, inférieure, ou bien inférieure ou égale au seuil.

Une fois le composant créé, l'utilisateur peut facilement avoir accès à ses données en cliquant dessus, après quoi celles-ci s'afficheront dans le coin de l'interface prévu à cet effet. Chaque composant est aussi ajouté dans le tableau à droite de l'interface permettant de lister les capteurs présents sur l'interface. Les informations résumées sont le nom, le type, et l'adresse.

Il est important de noter que le fait de cliquer sur le capteur dans le tableau permet aussi d'avoir accès aux informations. L'utilisateur peut également visuellement voir la distinction entre un capteur et un actionneur, car la première lettre du type est indiquée sur l'icône du composant. Ceci permet visuellement de reconnaître très rapidement les capteurs des actionneurs.

Si l'utilisateur souhaite supprimer un capteur, il lui suffit de le sélectionner, soit en cliquant sur son icône respective, soit sur la ligne correspondante dans le tableau, puis d'utiliser le contrôle "Supprimer composant" de la barre d'outils.

S'il le souhaite, l'utilisateur pourra modifier le capteur, mais seulement dans une certaine mesure. En effet, l'adresse du composant et le type étant fixe, il n'est pas possible de les modifier. Seuls le nom et la possibilité de déclencher une alerte peuvent être modifiés, de même que la valeur de seuil (selon le type de capteur choisi). De même, pour un actionneur, seul le nom pourra être modifié.

Une fois que l'utilisateur aura au minimum créé un capteur et un actionneur, il pourra

créer une alerte, ou règle de déclenchement, à l'aide du bouton correspondant dans la barre des tâches. La boîte de dialogue montrée en figure 10 s'affichera alors.

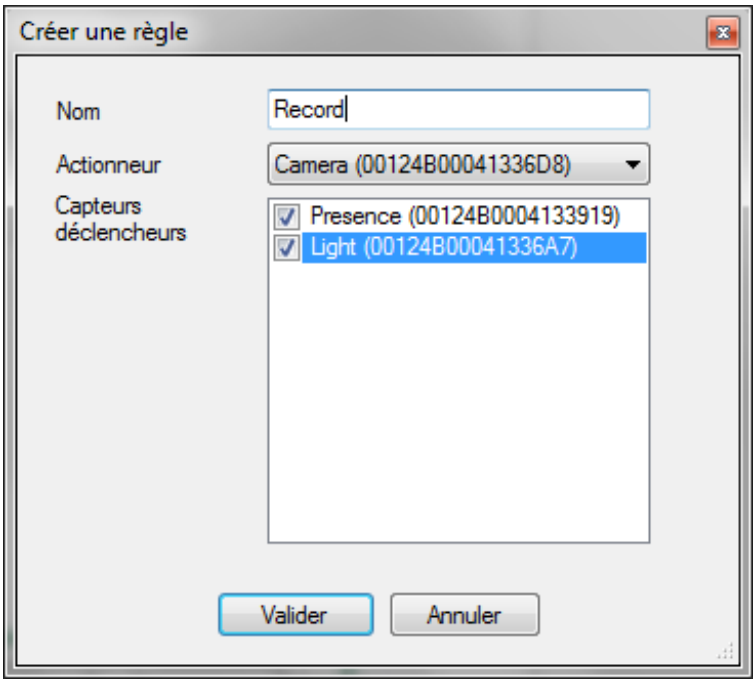


FIGURE 10 – Ajout d'une règle

Une règle est caractérisée par trois éléments : son nom, l'actionneur qui sera déclenché et les capteurs qui seront à l'origine du déclenchement de l'actionneur. Par exemple, dans le cas d'une salle fermée, si deux des interrupteurs sont ouverts, une LED pourra être allumée. En revanche, s'il n'y a qu'un seul des interrupteurs qui est activé, la LED ne sera pas allumée.

Comme pour les composants, la liste des différents événements est affichée dans un tableau récapitulatif sur l'interface avec le nom, l'actionneur, et les différents composants affectés à la règle. Il est également possible de supprimer une règle en cliquant sur le contrôle prévu à cet effet dans la barre d'outils.

Il faut également noter la mise en place d'un code couleur permettant de montrer visuellement si un composant est déconnecté du réseau. Lorsque l'utilisateur ajoute un capteur ou un actionneur, celui-ci est gris par défaut, c'est-à-dire qu'il n'est pas reconnu par le coordinateur. Lorsqu'il est ajouté au réseau, il devient vert. Enfin, si jamais le seuil d'alerte est atteint, il deviendra rouge, ce qui permettra à l'utilisateur de voir les capteurs qui ont été à l'origine d'une alerte.

Enfin, le dernier bouton de la barre d'outils permet de charger le plan de la salle où sont déployés les capteurs. Ce plan est sauvegardé en mémoire et est automatiquement chargé lors de le réouverture de l'interface. De même, la position des composants sur la carte, ainsi que tous les composants et règles, sont conservées en mémoire, et n'ont donc pas besoin d'être saisis de nouveau lors d'une prochaine utilisation.

Nous allons maintenant nous interésser à la partie suivante traitant des différents événements déclanchés par l'utilisateur en interagissant avec l'interface.

7.4 Gestion des événements déclenchés par l'utilisateur

Un aspect important du langagege C# est de pouvoir faire de la programmation événementielle, c'est-à-dire déclencher une procédure si l'utilisateur effectue une action spécifique sur l'interface. Ces actions sont appelées événements et sont essentielles pour le fonctionnement de l'application.

Chaque action (click sur un bouton, déplacement d'un composant...) fait appel à une méthode spécifique. Prenons par exemple le cas du déplacement d'un capteur. Après avoir créé un capteur, l'utilisateur peut le déplacer à l'endroit du plan qui le convient. Lorsqu'il va cliquer avec le bouton gauche de sa souris sur le capteur, le programme va appeler l'événement *sensor_MouseDown* sur le capteur courant, et effectuer les actions spécifiées dans la méthode (le code est donné ci-après).

```
1 private void sensor_MouseDown(object sender, MouseEventArgs e)
2 {
3    isDragging = true;
4    currentX = e.X;
5    currentY = e.Y;
6 }
```

Dans le cas de cette méthode, le programme autorise le composant à être déplacé, puis il récupère la position actuelle où se trouve le capteur et l'instancie comme étant les nouvelles coordonnées du composant.

Il est important de noter les deux arguments en paramètres de la méthode. Chaque méthode événementielle en possède deux. Le premier, le *sender*, référence l'objet sur lequel porte l'événement, à savoir ici l'objet capteur. Le deuxième argument, qui est ici de type *MouseEventArgs*, fait référence au type d'événement, c'est-à-dire un événement déclenché par la souris, le clavier...

Visual Studio est une grande aide pour ce type de programmation. Lorsqu'on crée un composant, l'IDE nous donne la liste de tous les événements possibles pour un composant. En cliquant dessus, il génère automatiquement l'en-tête de la méthode, ce qui est un gain de temps important pour le développeur.

Beaucoup d'événements ont été utilisés lors du développement de l'interface. Ainsi, à chaque bouton de la barre des tâches est associé un événement pour gérer les actions qui sont déclenchées lorsque l'utilisateur appuie dessus. De même, lorsque l'utilisateur clique sur un capteur ou le déplace, des événements sont appelés afin que sa position graphique suive le déplacement de la souris de l'utilisateur, ou encore que les données du capteur puissent être lues (le click simple déclenchant l'événement qui permet d'appeler la méthode interrogeant le coordinateur afin de connaître les donnéés du capteur).

Un autre événement dont il faut prendre note est celui permettant de mettre à jour les capteurs connus par l'interface. Celui-ci interagit avec la classe *Communication.cs* (dont le fonctionnement sera détaillé dans la partie suivante) et permet de récupérer la liste des composants connus par l'interface. Le code pour cet événement est présenté ci-après.

```
1 void com_SensorsChanged(object sender, EventArgs e)
2 {
```

```
3     try
4     {
5        Invoke((MethodInvoker)(() =>
6        {
7           foreach (Sensor sensor in sensorList)
8           {
9              Communication.SensorItem item = com.sensors.FirstOrDefault(s =>
                   Communication.IeeeArrayToString(s.info.ieee_addr) ==
                   sensor.Address);
10             sensor.updateFromItem(item);
11          }
12       }));
13    }
14    catch (InvalidOperationException)  {  }
15 }
```

Cette méthode appelle ainsi la fonction *IeeeArrayToString*, qui permet de récupérer la listes des adresses des capteurs connues par le coordinateur. Cette méthode, ainsi que d'autres, essentielles pour faire le lien entre l'interface et le réseau ZigBee, se trouvent dans la classe *Communication.cs*. C'est à ces méthodes que nous allons nous intéresser maintenant.

7.5 Interaction avec le réseau ZigBee

Au tout début du projet, il a fallu d'abord apprendre à se servir de la technologie ZigBee avant de pouvoir réfléchir à un protocole de communication entre les cartes et l'interface. Du côté logiciel, nous avons donc commencé par le développement graphique de l'interface. Ainsi, c'était surtout des données virtuelles qui étaient manipulées par l'interface au tout début du projet.

Cependant, une avancée majeure dans notre phase de réflexion a été l'élaboration d'un processus basique de communication entre l'une des cartes et l'interface que nous avons implémentée et testé. Il s'agissait simplement de changer la couleur d'une fenêtre sur l'interface lorsqu'une personne appuyait sur l'un des boutons de la carte. Cette opération, certes basique, nous a cependant donné les premières idées pour établir le protocole de communication avec l'interface.

Suite aux spécifications données par notre encadrant industriel, nous avons fixé le protocole d'échange suivant : le coordinateur est relié à l'interface par l'intermédiaire d'une liaison port série, et tous deux communiquent entre eux. Ainsi, l'interface ne communique pas avec les autres capteurs/actionneurs, cela relèvant du domaine du coordinateur qui connaît la liste des capteurs et actionneurs présents sur le réseau, ainsi que leur état. Le coordinateur envoie les données de chaque composant à l'interface qui peut alors les lire et gérer le réseau. Le coordinateur a donc un rôle primordial, puisque qu'il permet de faire le lien entre les différents capteurs et actionneurs du réseau, et l'interface.

Concernant les aspects plus techniques, la classe *Communication.cs* détaille le protocole en détail. L'interface gère, entre autres, deux listes : une de composants (capteurs et actionneurs), et une autre d'événements. Ces listes sont ainsi stockées en mémoire et permettent de retrouver le réseau dans son état lorsque l'application est fermée puis ouverte ultérieu-

rement.

Concernant le réseau, il était nécessaire de sauvegarder son état dans un fichier de configuration afin que, lors de la réouverture du logiciel, toutes les données ne soient pas perdues. Cela nécessitait donc une réflexion quant à savoir si la sauvegarde devait s'effectuer sur le coordinateur ou du côté logiciel. Après concertation avec l'encadrant industriel, il a été décidé d'effectuer la sauvegarde dans un fichier de configuration sur l'ordinateur ou tablette où serait lancé le logiciel. Cette décision fut prise pour une plus grande cohérence. Dans un système d'information, les fichiers de configuration sont la plupart du temps sauvegardés sur l'ordinateur utilisant le logiciel.

L'autre point important est la sécurité. Si les capteurs contiennent la configuration du réseau et viennent à être endommagés lors de leur déploiement, il sera plus gênant de récupérer la configuration du réseau que si ceux-ci se trouvaient du côté client. Enfin, si une personne souhaite manager le même réseau sur un ordinateur différent, il suffit juste de lui donner le fichier de configuration via clé USB par exemple.

La première étape à faire est d'analyser les différents ports de la tablette ou du PC afin de localiser celui sur lequel est branché le coordinateur. Cela est nécessaire car les ports peuvent changer, et il est donc quasi-impossible de le fixer dans l'interface. Voici un exemple de code permettant de faire une sélection sur les ports afin de localiser celui étant connecté au coordinateur.

```
public bool initiateConnection() {
   foreach (string portName in SerialPort.GetPortNames()) {
      Console.WriteLine("Portname : " + portName);
      port = new SerialPort(portName, 115200);
      port.Handshake = Handshake.RequestToSend;
      port.ReadTimeout = TIMEOUT_CHECK_CONNECTED;
      try {
         port.Open();
      } catch (System.UnauthorizedAccessException e) {
         Console.WriteLine("Unauthorized access");
         continue;
      }
      sendCmd((byte)operation_code.CMD_PING);
      try {
         byte[] buff = readNBytes(PONG.Length);
         String pong = System.Text.Encoding.UTF8.GetString(buff);
         Console.WriteLine("Ping ? " + pong);
         if (pong == PONG) {
            return true;
         }
      } catch (System.TimeoutException e) {
         Console.WriteLine("Timeout lors de l'initialisation");
         continue;
      }
      port.Close();
   }
   return false;
}
```

Tous les ports de l'ordinateur sont ainsi passés en revue jusqu'à tomber sur le port connecté au coordinateur. Cette méthode est appelée dans le constructeur de la classe lors de son initialisation dans l'interface. Si le coordinateur n'est pas connecté à l'ordinateur, un message d'erreur est retourné, et il est impossible d'accéder à l'interface. Ainsi, l'accès à l'interface passe par le pré-requis d'avoir le coordinateur relié au PC ou à la tablette. Une fois la carte reliée à l'interface, les méthodes pour la gestion des capteurs peuvent être utilisées.

Tout d'abord, il est important de noter l'utilisation de threads lors de l'ajout et suppression d'un composant ou d'une règle. Ces threads sont des processus légers qui sont utilisés afin de lancer l'exécution de plusieurs tâches en parallèle. Cela permet d'augmenter la rapidité d'exécution de l'application et est nécessaire au bon maintien de celle-ci. Ainsi, chaque composant aura son propre thread, ce qui permettra de fluidifier leur manipulation sur la carte.

Les différentes méthodes d'ajout et de suppression fonctionnant sur la même base pour les composants comme pour les règles, une seule de ces méthodes sera détaillée ici. Par exemple, intéressons-nous à la méthode *addSensor*, qui permet, comme son nom l'indique, d'ajouter un capteur ou un actionneur par l'envoi de données, en se conformant au protocole de communication établi. Le code pour cette méthode est donné ci-après.

```
internal void addSensor(Sensor sensor) {
   sensor_info info = sensor.getInfo();

   new Thread(() =>
   {
      List<byte> bytes = new List<byte>();
      bytes.Add((byte)operation_code.CMD_MODIFY_SENSOR);
      bytes.AddRange(StructureToByteArray(info));
      lock (port)
      {
         port.DiscardInBuffer();
         sendBytes(bytes.ToArray());
         checkAnswer();
      }
   }).Start();
}
```

Cette méthode prend un objet de type *Sensor* en paramètre, et n'est autre que la représentation sur l'interface d'un composant. Le principe de cette méthode est de récupérer les données du composant, les ajouter dans une liste, et envoyer cette dernière au coordinateur. Il est important de noter la méthode de verrouillage du port série afin de pouvoir envoyer les données sur la même sortie.

Il est intéressant de remarquer le code d'opération, dans ce cas-ci *CMD_MODIFY_SENSOR*. Il s'agit d'un élément extrait d'une énumération déclarée au début de la classe, et qui permet d'associer à chaque élément un code hexadécimal qui est envoyé au coordinateur. C'est la manière de communiquer afin de pouvoir transmettre les instructions des commandes. Les différents éléments de cette énumération sont présentés ci-après.

```
enum operation_code : byte
```

```
{
   CMD_GET_SENSOR_LIST = 0,
   CMD_DELETE_SENSOR = 1,
   CMD_MODIFY_SENSOR = 2,
   CMD_DUMP_SENSORS = 3,
   CMD_ADD_RULE = 4,
   CMD_GET_RULE_LIST = 5,
   CMD_DELETE_RULE = 6,
   CMD_DUMP_RULES = 7,
   CMD_GET_TIME = 8,
   CMD_SEND_DATA = 9,
   CMD_NVRAM_INFO = 10,
   CMD_PING = 0x42,
   CMD_CLEAR_ALL = 0xA0,
}
```

Ces différentes instructions vont permettre d'envoyer toutes les instructions nécessaires au coordinateur pour interagir avec l'interface. De la même manière, d'autres énumérations permettent de connaître le type d'erreur, le type du capteur, le type de seuil, et le type des données. Pour plus d'informations, ces différentes énumérations sont présentées en annexe C.

Une autre méthode primordiale présente dans cette classe est *updateData* qui permet la mise à jour fréquente des listes de capteurs et événements, ceci afin que tout nouvel élément présent sur le réseau soit rapidement détecté. Elle fait appel à deux autres méthodes, *updateSensorItems* pour la mise à jour des capteurs, et *updateRules* pour la mise à jour des règles de déclenchement. Ces deux méthodes fonctionnant sur les mêmes principes, seule la méhode *updateSensorItems* sera détaillée ici. Le code de cette méhode se trouve ci-dessous.

```
//Update the list of sensor_item according to the communication protocol
private void updateSensorItems()
{
   sendCmd((byte)operation_code.CMD_GET_SENSOR_LIST);
   int count = port.ReadByte();
   List<SensorItem> new_sensors = new List<SensorItem>();
   for (int i = 0; i < count; i++)
   {
      SensorItem sensor = new SensorItem();
      sensor.info = readStruct<sensor_info>();
      UInt32 last_act = readStruct<UInt32>();

      if (last_act != 0)
         sensor.last_activity = new DateTime(DeltaTime + last_act *
             TimeSpan.TicksPerSecond);
      else
         sensor.last_activity = null;

      sensor.last_val = readStruct<sensor_val>();
      sensor.alarmed = (readStruct<Byte>() != 0);
      sensor.lqi = (byte)port.ReadByte();
```

```
21          new_sensors.Add(sensor);
22      }
23
24      sensors = new_sensors;
25
26      if (SensorsChanged != null)
27      {
28          SensorsChanged(this, EventArgs.Empty);
29      }
30  }
```

Dans le cas de la liste de composants, l'interface envoie la commande *CMD_GET_SENSOR_LIST*, qui retourne la liste des capteurs qu'il connaît sur le réseau. À partir de cela, le coordinateur répond en renvoyant la liste des capteurs, ainsi que leur dernier état d'activité. Cette notion permet de savoir quel a été la dernière fois où l'un des composants a donné signe de vie.

Pour la mise à jour des règles de déclenchement, il s'agit du même principe, cette fois-ci en utilisant le code de commande *CMD_GET_RULE_LIST*.

Ceci conclut donc la partie consacrée au développement de l'interface graphique. Maintenant, nous allons nous intéresser aux aspects de gestion du projet.

8 Organisation

Dans cette partie, nous allons parler de l'organisation du projet en détaillant les outils utilisés, les méthodes de travail appliquées, ainsi que les livrables.

8.1 Le logiciel VMproject

Pour ce projet, ainsi que pour tous les projets industriels, un outil de gestion de projet, intitulé « VMproject », a été imposé. Il s'agit d'un logiciel accessible via une plate-forme web, et qui est développé par la société Versusmind.

Cet outil propose plusieurs fonctionnalités comme la planification de projets à l'aide d'un diagramme de Gantt qui inclue la répartition des charges. Cela permet d'avoir un planning très précis du projet et de pouvoir visuellement savoir où l'on en est.

Un autre atout est la possibilité d'organiser des réunions et de rédiger leurs comptes-rendus. VMproject permet également de partager des documents et de suivre l'activité de chacun des membres du projet. Chaque membre peut remplir des fiches de temps et être notifié par e-mail à la moindre activité. Cela permet ainsi de connaître l'avancement de tous les membres du groupe sur le projet, mais permet également à notre encadrant industriel, Antoine Moron, de suivre notre activité et de pouvoir réagir aux comptes-rendus de réunion.

À la date de rendu de ce rapport, nous avons comptabilisé 715 heures sur VMproject ce qui représente environ 240 heures par membre de l'équipe. Sur les 250 heures par personne exigée pour ce projet, il reste encore environ 10 heures par personne. Ces 10 heures seront justement consacrées à la préparation de la soutenance du 19 mars et également à la finalisation d'un poster et d'un site web qui ont été demandés par l'école.

Nous allons maintenant nous intéresser aux réunions avec nos encadrants qui ont permis notre suivi fréquent du projet.

8.2 Réunions avec nos encadrants

Concernant les réunions avec notre encadrant universitaire, Marc Tomczak, nous avons convenu de procéder à une réunion toutes les deux semaines. Ces réunions permettaient de faire le point sur ce qui avait été fait et sur ce qui restait à faire. VMproject est donc très utile dans ce cas-là, puisqu'il permet de planifier la réunion à l'avance et d'interagir pour la rédaction du compte-rendu une fois cette réunion terminée.

Quant aux réunions avec notre encadrant industriel, nous avons privilégié des échanges par e-mail lorsque cela était nécessaire. En effet, ce dernier habitant sur Paris, il ne pouvait pas se déplacer très souvent, ainsi un contact à distance était la meilleure option. Cependant, il a eu l'occasion de venir sur Nancy à des moments-clés de notre avancement, notamment au tout début du projet pour nous donner le matériel et les principales directives, et également avant la soutenance intermédiaire afin de faire un point sur l'avancement.

Enfin, sa dernière visite, environ un mois et demi avant la fin du projet, a permis de présenter notre avancement, de discuter des choix techniques envisagés, mais a surtout été d'une grande aide pour l'interface. En effet, il a pu partager avec nous ses idées quant aux améliorations qui pouvaient être faites au sujet de l'ergonomie. Nous avons pu, grâces à ses précieux conseils, retravailler le design de notre application afin qu'elle puisse parfaitement correspondre à l'attente du client.

Ainsi, les réunions avec nos deux encadrants étaient donc primordiales lors de notre avancement, tant pour nous aider sur le plan technique que sur l'aspect gestion de projet. Maintenant, nous allons passer à l'aspect organisationnel au sein de notre équipe.

8.3 Organisation au sein de l'équipe

Au niveau de l'organisation de notre équipe, nous sommes trois à travailler sur le projet. Ludovic Schoepps, le chef de projet, s'occupe de gérer le bon déroulement du projet et d'interagir avec nos encadrants, tout en participant au développement de l'application, au niveau embarqué tout comme au niveau Windows. Mickaël Thomas, quant à lui, est responsable de la partie embarquée. Enfin, David Rubino est responsable de la partie Windows.

Cette répartition a été choisie en fonction des approfondissements suivis par les membres de l'équipe à l'école. Ainsi, cela nous a aidé à acquérir de nouvelles connaissances dans des domaines très proches de ceux que nous étudions durant notre cursus à TELECOM Nancy.

Après cette présentation des membres de l'équipe, nous allons mettre en avant le système de gestion des sources mis en place durant le projet.

8.4 Gestion des sources

Notre encadrant nous a laissé le champ libre en ce qui concerne la gestion des sources en nous précisant que le code n'avait pas un caractère confidentiel. Nous avons donc choisi d'utiliser Git car c'est un très bon outil que nous avons l'habitude d'utiliser. Git est un logiciel de gestion de versions. Il permet donc de sauvegarder les différentes versions d'un projet dans un fichier afin qu'en cas de problème en cours de développement, il soit possible de revenir à une version antérieure fonctionnelle.

Concernant la sauvegarde des fichiers, celle-ci s'effectue à la fois sur l'ordinateur personnel de chacun des membres du groupe, mais aussi en ligne grâce à une plate-forme d'hébergement de code source. En effet, Git fournit la commande *push* permettant de déposer le code source en ligne. Ainsi, chaque membre du groupe peut, à l'aide de *pull*, une autre commande de Git, récupérer le travail des autres et à son tour poursuivre l'avancement sur le projet.

De nombreux sites web proposent ce service d'hébergement ; dans notre cas nous avons choisi Bitbucket, simplement parce que nous étions déjà tous inscrits sur le site, mais aussi car cette plate-forme possède un avantage majeur : le dépôt de code source est gratuit et privé, ainsi une personne externe au projet ne pourra pas voir le code.

Cette plate-forme de gestion a donc permis de sauvegarder le code depuis le début du projet. Maintenant, intéressons-nous au planning du projet défini dans le diagramme de Gantt.

8.5 Diagramme de Gantt

Au tout début du projet, notre première tâche fut d'établir un diagramme de Gantt prévisionnel afin de déterminer les principales étapes du projet ainsi que le temps associé à celles-ci. La réalisation de ce diagramme de Gantt était une étape importante du projet puisqu'elle permettait de bien cibler les dates de rendu de livrables. La version initiale de ce diagramme est présentée en annexe A.1.

Au fil du projet, il est cependant arrivé que certaines étapes prennent plus de temps que prévu, et qu'il faille décomposer certaines étapes en sous-étapes. La mise à jour de notre diagramme de base était donc nécessaire. Ainsi, nous avons établi un nouveau diagramme de Gantt en fonction de l'avancement réel du projet. Comme son prédécesseur, celui-ci est présenté en annexe A.2.

Les diagrammes de Gantt ont permis d'établir la liste des livrables, ainsi que la date où le rendu devait être effectué. Ainsi, dans la prochaine partie, nous allons nous intéresser à ces différents livrables.

8.6 Livrables

Plusieurs livrables ont été identifiés pour le projet. Au nombre de 3, ils sont répartis de manière égales sur la durée du projet. Ces différents livrables correspondent à des fins de phases repérables sur le diagramme de Gantt A.2.

Le premier de ces livrables, en date du 3 novembre 2014, clôture la phase 2, à savoir la rédaction de la note de cadrage. Comme son nom l'indique, il comprend la note de cadrage qui permet de définir les tâches à accomplir lors du projet. Grâce à elle, il a été possible de définir les objectifs précis du projet. Ce premier livrable était donc très important, puisqu'il permettait de définir la direction que prendrait le projet dans les prochains mois.

Un second livrable, à la date du 9 janvier 2015, correspond à la fin des phases 5, 12, et 16 sur le diagramme de Gantt. En effet, ce livrable arrivait comme un point intermédiaire du projet. Il aurait théoriquement été possible de livrer une version fonctionnelle du code de l'interface et des cartes à l'encadrant industriel ; cependant, ce dernier n'attendait pas de produits intermédiaires. Le seul livrable effectif fut donc le rapport intermédiaire, à destination de notre encadrant universitaire. Il est également important de préciser qu'à ce point-là du projet, plus de 40% avait déjà été réalisé, avec un volume horaire atteignant 313 heures sur 750.

Enfin, le livrable final, en date du 19 mars 2015, correspond à la fin des phases 9, 15, 18, et 20. Il comprendra l'application fonctionnelle, à savoir le code source de l'application C#, le code source du programme embarqué pour les cartes, ainsi qu'un rapport présentant le détail de notre démarche lors du développement de l'application. De plus, un poster

résumant les principaux objectifs du projet sera également confectionné. Ce livrable correspond également à la date de la soutenance finale, durant laquelle le projet sera présenté devant un jury.

Une étape non prévue à la base s'est inscrite dans le projet : celle d'un audit. La partie suivante va donc traiter de cet aspect important dans la gestion de projet.

8.7 Audit

Durant le mois de janvier, nous avons eu l'opportunité de participer à un audit de notre projet. Cette étape, non initialement prévue dans le planning, a permis à une personne extérieure de porter un regard objectif sur la gestion du projet. Cette expérience nous a permis de prendre en considération l'importance de l'aspect gestion de projet qui ne doit pas être négligé.

L'échange a permis de discuter de la note de cadrage et de la progression du projet par rapport au Gantt initial, en voyant les dérives éventuelles au niveau des périodes de temps. L'un des points forts qui est ressorti de cet audit était la notion de fin de période sur le diagramme de Gantt, qui devait forcément être marqué par un livrable. Dans notre cas, cependant, cela était légèrement différent car le client n'attendait pas de livrables intermédiaires. Néanmoins, cela nous a permis de réfléchir aux éventuels livrables qui auraient pu correspondre à ces fins de période, à savoir des versions fonctionnelles de nos programmes.

De plus, le développement est constamment testé en parallèle ; ainsi, nous sommes sûrs qu'à chaque fin de période, les programmes sont fonctionnels et ont un comportement connu. En définitive, s'il avait fallu livrer les résultats obtenus à la fin de chaque période du diagramme de Gantt, nous aurions eu des versions fonctionnelles de nos différentes itérations à présenter.

Cet audit a donc été fructueux pour notre apprentissage en tant que futurs ingénieurs, et nous a fait bien prendre conscience de la gestion de projet.

Cette partie finie notre rapport et nous amène donc à la conclusion.

9 Conclusion

Lors de ce projet, nous avons pu développer une application permettant de créer un réseau de capteurs communiquant au travers de la technologie ZigBee. Pour ce faire, il a fallu développer un programme embarqué permettant de créer le réseau. Développé en C, il permet de prendre en considération des capteurs et des actionneurs qui interagissent entre eux, gérés par le coordinateur, un capteur dédié à planifier le réseau.

Ce coordinateur, qui possède toutes les informations sur le réseau, les transmet ensuite à un logiciel Windows, que nous avons développé en C#, et qui permet à un utilisateur de gérer le réseau en récupérant toutes les informations envoyées par le coordinateur.

Ensemble, ces deux programmes (embarqué et Windows) permettent de construire le réseau de capteurs et de l'utiliser dans de nombreux cas, par exemple pour surveiller une zone délimitée d'un bâtiment. A l'heure actuelle, le livrable que nous fournissons au client, à savoir le Ministère de la Défense, permet d'établir ce réseau en téléchargeant les différents programmes sur les cartes embarqués, et en lançant l'interface sur un PC ou une tablette Windows.

De par l'application que nous avons fournie, nous répondons parfaitement à la demande du client, qui était de développer un réseau de capteurs sans fils avec une interface pour tablette Windows permettant de gérer ce réseau. Nous pouvons donc affirmer que les objectifs du projet ont été atteints.

Des améliorations éventuelles pourraient toutefois être possibles. Par exemple, la gestion de plus de types de capteurs et d'actionneurs pourrait être prise en compte. Il serait aussi possible d'imaginer une version mobile de l'application pour Windows Phone, afin que le contrôle à distance du réseau puisse se faire encore plus facilement.

Enfin, au niveau personnel, ce projet, qui s'inscrivait dans le cadre de notre formation d'ingénieur en tant qu'élèves à TELECOM Nancy, nous a permis de prendre conscience des types de projets menés en entreprise. De plus, il nous a permis d'être initié à la gestion de projet avec des outils et des méthodes appliqués dans le monde professionnel, ce qui, en tant que futurs ingénieurs, nous sera extrêmemet utile par la suite.

L'aspect pluridisciplinaire du projet, mêlant à la fois de l'embarqué avec du développement plus classique, montre bien les interactions entre les différents domaines de l'informatique dans la vie professionnelle et a donc été une préparation, à la fois pour notre stage de fin d'étude que pour notre vie future d'ingénieurs.

10 Références

[1] Ministère de la Défense, *Organigramme simplifié du Ministère de la Défense* 21 Mai 2012. http://www.defense.gouv.fr/portail-defense/ministere/organisation/organisation-du-ministere-de-la-defense/organigramme-simplifie-du-ministere-de-la-defense/organigramme-simplifie-du-ministere-de-la-defense-et-des-anciens-combattants (Accès le 12-01-2015).

[2] Texas Instruments, Inc., *Z-Stack Developer's Guide* 2006-2011, pp.3 ;14 ;25

[3] Texas Instruments, Inc., *Z-Stack User's Guide* 2010

[4] Texas Instruments, Inc., *CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee®/ZigBee IP® Applications* Avril 2012–Révision Mai 2013

[5] TELECOM Nancy, *Plan du bâtiment, niveau 0, aile Nord* https://www-intranet.telecomnancy.univ-lorraine.fr/lib/exe/fetch.php/infos_services/plans/n0.jpg (Accès le 03-03-2015)

[6] A Review of Wireless Sensor Technologies and Applications in Agriculture and Food Industry : State of the Art and Current Trends, http://www.mdpi.com/1424-8220/9/6/4728 (Accès le 12-03-2015)

[7] How thieves can hack and disable your home alarm system, http://www.wired.com/2014/07/hacking-home-alarms/ (Accès le 13-03-2015)

[8] ZigBee, http://fr.wikipedia.org/wiki/ZigBee (Accès le 10-01-2015)

[9] Réseau de capteurs sans fil, http://fr.wikipedia.org/wiki/Réseau_de_capteurs_sans_fil (Accès le 12-03-2015)

[10] ZigBee Home Automation, http://www.zigbee.org/zigbee-for-developers/applicationstandards/zigbeehomeautomation (Accès le 10-01-2015)

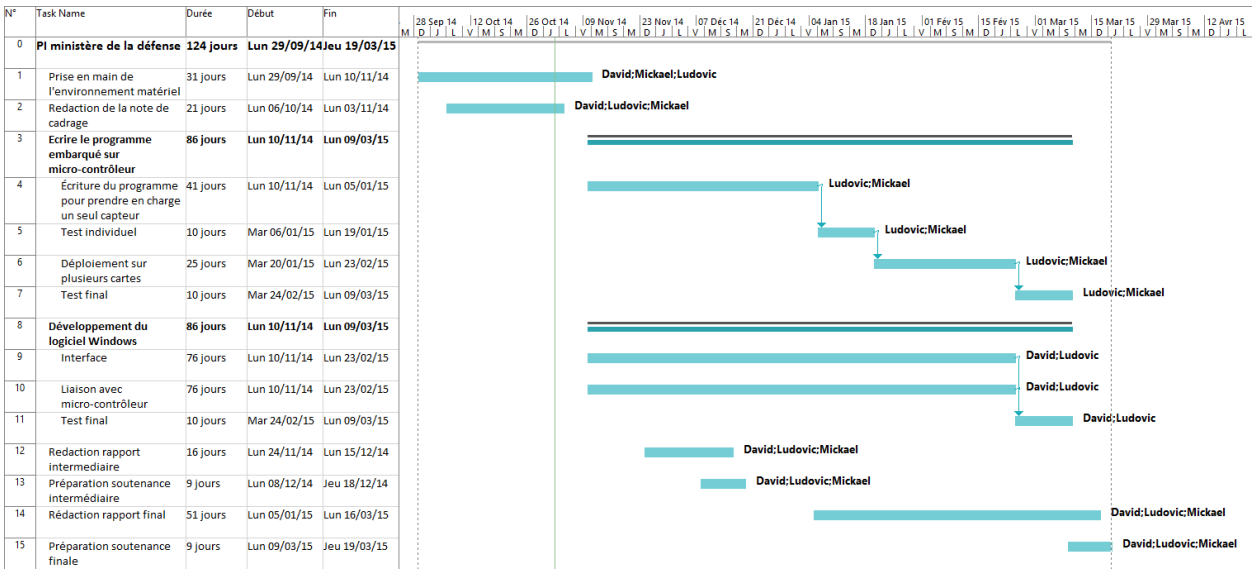
[11] RS-232, http://fr.wikipedia.org/wiki/RS-232 (Accès le 12-03-2015)

[12] Counter-Mode/CBC-Mac protocol, http://fr.wikipedia.org/wiki/Counter-Mode/CBC-Mac_protocol (Accès le 17-03-2015)

Liste des illustrations

Annexe A Diagrammes de Gantt

A.1 Diagramme prévisionnel



N°	Task Name	Durée	Début	Fin
0	**PI ministère de la défense**	**124 jours**	**Lun 29/09/14**	**Jeu 19/03/15**
1	Prise en main de l'environnement matériel	31 jours	Lun 29/09/14	Lun 10/11/14
2	Redaction de la note de cadrage	21 jours	Lun 06/10/14	Lun 03/11/14
3	**Ecrire le programme embarqué sur micro-contrôleur**	**86 jours**	**Lun 10/11/14**	**Lun 09/03/15**
4	Écriture du programme pour prendre en charge un seul capteur	41 jours	Lun 10/11/14	Lun 05/01/15
5	Test individuel	10 jours	Mar 06/01/15	Lun 19/01/15
6	Déploiement sur plusieurs cartes	25 jours	Mar 20/01/15	Lun 23/02/15
7	Test final	10 jours	Mar 24/02/15	Lun 09/03/15
8	**Développement du logiciel Windows**	**86 jours**	**Lun 10/11/14**	**Lun 09/03/15**
9	Interface	76 jours	Lun 10/11/14	Lun 23/02/15
10	Liaison avec micro-contrôleur	76 jours	Lun 10/11/14	Lun 23/02/15
11	Test final	10 jours	Mar 24/02/15	Lun 09/03/15
12	Redaction rapport intermediaire	16 jours	Lun 24/11/14	Lun 15/12/14
13	Préparation soutenance intermédiaire	9 jours	Lun 08/12/14	Jeu 18/12/14
14	Rédaction rapport final	51 jours	Lun 05/01/15	Lun 16/03/15
15	Préparation soutenance finale	9 jours	Lun 09/03/15	Jeu 19/03/15

FIGURE 11 – Diagramme de Gantt prévisionnel

A.2 Diagramme réel

N°	Task Name	Durée	Début	Fin
0	**PI ministère de la défense**	**124 jours**	**Lun 29/09/14**	**Jeu 19/03/15**
1	Prise en main de l'environnement matériel	31 jours	Lun 29/09/14	Lun 10/11/14
2	Redaction de la note de cadrage	21 jours	Lun 06/10/14	Lun 03/11/14
3	**Ecrire le programme embarqué sur micro-contrôleur**	**86 jours**	**Lun 10/11/14**	**Lun 09/03/15**
4	Prototype	6 jours	Lun 10/11/14	Dim 16/11/14
5	Première version pour le coordinateur et le capteur	14 jours	Lun 17/11/14	Jeu 04/12/14
6	Définition et mise en place d'un protocole de communication et d'un format de stockage des données	12 jours	Ven 09/01/15	Dim 25/01/15
7	Implémentation des différents types de capteurs	11 jours	Lun 26/01/15	Dim 08/02/15
8	Mise en place des règles de déclenchements des événements et des	11 jours	Lun 09/02/15	Dim 22/02/15
9	Finalisation et test final	11 jours	Lun 23/02/15	Lun 09/03/15
10	**Développement du logiciel Windows**	**86 jours**	**Lun 10/11/14**	**Lun 09/03/15**
11	Design d'un prototype d'interface	19 jours	Lun 10/11/14	Jeu 04/12/14
12	Liaison de base avec micro-contrôleur	19 jours	Lun 10/11/14	Jeu 04/12/14
13	Modification du design et amélioration de l'ergonomie	31 jours	Lun 12/01/15	Lun 23/02/15
14	Récupération des données envoyées par le micro-contrôleur	31 jours	Lun 12/01/15	Lun 23/02/15
15	Finalisation et test final	10 jours	Mar 24/02/15	Lun 09/03/15
16	Redaction rapport intermediaire	35 jours	Lun 24/11/14	Ven 09/01/15
17	Préparation soutenance intermédiaire	9 jours	Lun 08/12/14	Jeu 18/12/14
18	Rédaction rapport final	31 jours	Lun 02/02/15	Lun 16/03/15
19	Préparation soutenance finale	9 jours	Lun 09/03/15	Jeu 19/03/15
20	Confection d'un poster	11 jours	Lun 02/03/15	Dim 15/03/15

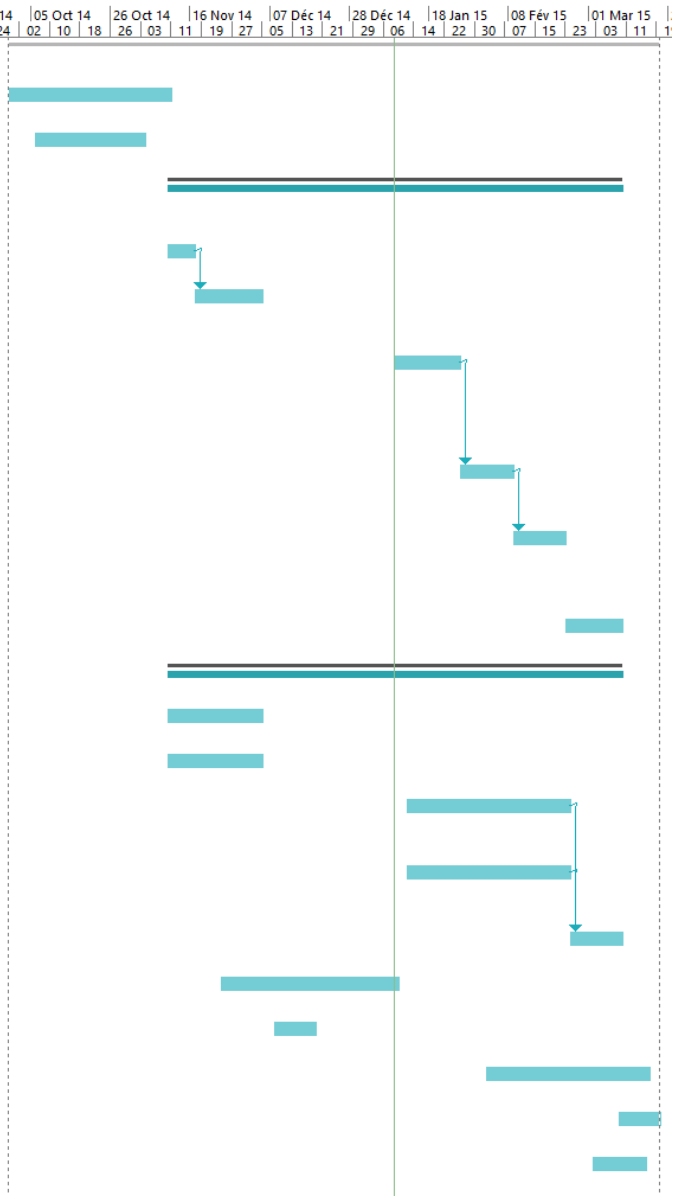


FIGURE 12 – Diagramme de Gantt réel

Annexe B Structures / Protocole de communication

B.1 Structures communes aux capteurs, actionneurs et au coordinateur

Common.h

```
#ifndef COMMON_H
#define COMMON_H

/** Sensor (or actuator) type */
enum sensor_type {
  SWITCH_SENSOR = 0,
  LIGHT_SENSOR = 1,
  LED_ACTUATOR = 2
};
typedef uint8 sensor_type_t;

enum sensor_datatype {
  BOOL = 0,
  INT = 1,
  FLOAT = 2
};
typedef uint8 sensor_datatype_t;

/** Union for storing the current value or threshold of a sensor/actuator */
typedef union {
  int int_val;
  float float_val;
  bool bool_val;
} sensor_val_t; //4 bytes

/** Data packet sent from a sensor/actuator to the coordinator to report its
    current value */
__packed typedef struct {
  sensor_val_t val;
  sensor_type_t type;
  sensor_datatype_t datatype;
} sensor_state_t;

#endif
```

B.2 Strucutures et fonctions pour la gestion des listes chaînées dans la NVRAM

NvList.h

```
#ifndef NV_LIST_H
#define NV_LIST_H

#include "ZComDef.h"

/* Linked list implementation using NVRAM items for storage
 * Internal structure :
 *  - Elements and lists are represented with their NVRAM ID
 *  - A list consists of a list_t struct which contains the ID of the first element of the list
 *    or 0 in case the list is empty
 *  - A list element is a list_item_t structure (with variable size) which contains data and
 *    a pointer to the ID of the next element (or 0 if it's the last item in the list)
*/

/** Base list item structure */

/* List layout in NVRAM */
typedef struct {
  uint16 first_id;
} list_head_t;

/* Item layout in NVRAM */
typedef struct {
  uint16 next_id;
  uint8 data[0];
} list_item_t;


/** Create a new item in NVRAM by scanning the specified ID range for an unused ID
  * @param search_start_id first ID to scan for
  * @param search_count number of IDs to try before giving up
  * @param data pointer to initial data
  * @param len length of the list_item_t structure
  * @return the new item ID, or 0 if no free item ID was found
  */
uint16 nvlist_create(uint16 search_start_id, uint16 search_count, const list_item_t* data, int len);


/** Write data to a list item
  * @param item_id item ID to write data into
  * @param data pointer to the data to write into the NVRAM
  * @param len length of the list_item_t structure
  * @return true if the item was found, false otherwise
```

```
44    */
45  bool nvlist_set(uint16 item_id, const list_item_t* data, int len);
46
47
48  /** Read data from a list item
49    * @param item_id item ID to write data into
50    * @param data pointer to write the data from the NVRAM into
51    * @param len length of the list_item_t structure
52    * @return true if the item was found, false otherwise
53    */
54  bool nvlist_get(uint16 item_id, list_item_t* data, int len);
55
56
57  /** Read the successor of an item
58    * @param item_id item ID for which we're looking for a successor
59    * @param next_data pointer to store the next item data into
60    *                 (can be NULL if only interested in the ID)
61    * @param len length of the list_item_t structure
62    * @return ID of the next element
63    */
64  uint16 nvlist_get_next(uint16 item_id, list_item_t *next_data, int len);
65
66
67  /** Read the predecessor of an item in the list
68    * @param list_id ID of the the list
69    * @param item_id ID of the item for which we're looking for a predecessor
70    * @param pred_data pointer to store the predecessor item data into
71    *                 (can be NULL if only interested in the ID)
72    * @param len length of the list_item_t structure
73    * @return ID of the predecessor, 0 if not found
74    */
75  uint16 nvlist_get_prev(uint16 list_id, uint16 item_id, list_item_t *pred_data,
       int len);
76
77
78  /** Append an element to the beginning of the list
79    * @param list_id ID of the the list
80    * @param item_id ID of the item to append
81    * @return true if the element was successfully added, false otherwise
82    * @note this changes the content of the item, so make sure not to override it
         with the previous
83    *       value after calling this function.
84    */
85  bool nvlist_append(uint16 list_id, uint16 item_id);
86
87
88  /** Delete an element from a list
89    * @param list_id ID of the the list
90    * @param deleted_id ID of the item to delete
91    * @return true if the element was successfully deleted, false otherwise
92    */
93  bool nvlist_delete(uint16 list_id, uint16 deleted_id);
```

```
94
95
96  /** Destroy a list, deleting all its items (and optionally the list itself)
97    * @param list_id ID of the the list
98    * @param only_items true to delete only the items and not the list itself
99    * @return true if all items were successfully destroyed, false otherwise
100   */
101 bool nvlist_destroy(uint16 list_id, bool only_items);
102
103
104 /** Count the number of items in a list
105   * @param list_id ID of the list
106   * @return the number of items in the list
107   */
108 int nvlist_count(uint16 list_id);
109
110 /** Dump the list to serial port for debugging purposes
111   * @param list_id ID of the list
112   */
113 void nvlist_dump(uint16 list_id);
114
115
116 /** Check if the list contains the specified item
117   * @param list_id ID of the list
118   * @param item_id item ID to search for
119   */
120 bool nvlist_contains(uint16 list_id, uint16 item_id);
121
122
123 /** Get the ID of the first element of a list
124   * @param list_id ID of the list
125   */
126 uint16 nvlist_first(uint16 list_id, list_item_t *data, int len);
127
128 #endif
```

B.3 Structures et fonctions pour le stockage et la transmission des données

Utils.h

```
#pragma once
#include <string.h>

#include "ZComDef.h"
#include "hal_types.h"
#include "af.h"

#include "NvList.h"
#include "Common.h"


#define MAGIC_NUMBER_ID 0x0401
#define SENSOR_ITEM_LIST_ID 0x0402 //At this id, the nvram contain an uint16 which is the id of the beginning of the linked list of sensor_item
#define TRIGGER_EVENT_LIST_ID 0x0403 //Same as above for the list of trigger event
#define FREE_START_ID 0x0404 //From here, we can start searching for free id
#define FREE_ID_COUNT (0xFFF - FREE_START_ID + 1)


enum threshold_type {
  NONE = 0, /* No threshold set */
  EQ,       /* Equal */
  LTE,      /* Less than or equal (valid only for FLOAT and INT types) */
  LT,       /* Less than (strict, valid only for FLOAT and INT types) */
  GTE,      /* Greater than or equal (valid only for FLOAT and INT types) */
  GT        /* Greater than (strict, valid only for FLOAT and INT types) */
};
typedef uint8 threshold_type_t;

/** Static information about a sensor, stored in the NVRAM inside sensor_item_t structure */
__packed typedef struct {
  ZLongAddr_t ieee_addr;        // Unique 64 bits address
  sensor_type_t type;
  sensor_datatype_t datatype;
  sensor_val_t threshold;
  threshold_type_t threshold_type;
} sensor_info_t; //8 + 1 + 1 + 4 + 1 = 15 bytes
/* WARNING: Do not forget to increment MAGIC_NUMBER in Coordinator.h if you change this structure */

/*********************************************************************
 * NVRAM elements
 * Those structures represent elements of an NvList stored in the NVRAM to keep track
 * of the network state. They all follow list_item_t convention (first next_id field points
```

```
 * to the next item ID or 0 at the end)
 */

/** Sensor state
  * contains static info (sensor_info_t) as well as last value and last communication time */
__packed typedef struct {
  uint16 next_id;
  sensor_info_t info;
  uint32 last_seen_sec;      // Time of last communication with the sensor (in seconds,
                             // measured relative to the coordinator start-up (0))
  sensor_val_t last_val;     // last data communicated by the sensor
  uint8 LQI;                 // Link quality indicator
} sensor_item_t; //2 + 15 + 4 + 4 + 1 = 26 bytes
/* WARNING: Do not forget to increment MAGIC_NUMBER in Coordinator.h if you change this structure */

__packed typedef struct {
  uint16 next_id;
  ZLongAddr_t ieee_addr;
} addr_list_t;
/* WARNING: Do not forget to increment MAGIC_NUMBER in Coordinator.h if you change this structure */

__packed typedef struct {
  uint16 next_id;
  uint16 addr_list_id;
  ZLongAddr_t ieee_addr_target;
} trigger_event_t;
/* WARNING: Do not forget to increment MAGIC_NUMBER in Coordinator.h if you change this structure */

/** Commands that can be sent to the coordinator through the serial port */
typedef enum {
  CMD_GET_SENSOR_LIST = 0,
  CMD_DELETE_SENSOR = 1,
  CMD_MODIFY_SENSOR = 2,
  CMD_DUMP_SENSORS = 3,
  CMD_ADD_RULE = 4,
  CMD_GET_RULE_LIST = 5,
  CMD_DELETE_RULE = 6,
  CMD_DUMP_RULES = 7,
  CMD_GET_TIME = 8,
  CMD_SEND_DATA = 9,
  CMD_NVRAM_INFO = 10,
  CMD_PING = 0x42,
  CMD_CLEAR_ALL = 0xA0,
} operation_code_t;

/** Status codes used in reply to serial commands */
typedef enum {
```

```
  ERROR_SUCCESS = 0,       /* no error */
  ERROR_INVALID_FORMAT = 1, /* some field was missing or had an invalid value */
  ERROR_NOT_FOUND = 2,     /* requested item was not found */
  ERROR_INTERNAL = 3,      /* internal error */
  ERROR_UNKNOWN = 0xFF,    /* unknown error */
} error_code_t;

/** Get the sensor_item matching the IEEE address
  * @param ieee_addr IEEE address
  * @param data pointer to store the sensor_item into (can NOT be NULL)
  * @return the id of the item if a match was found, 0 otherwise
  */
uint16 get_sensor_item_from_ieee(const ZLongAddr_t ieee_addr, sensor_item_t
    *data);

/** Handle an incoming message from a sensor
  * @param pkt the packet to handle
  */
void handle_message(afIncomingMSGPacket_t *pkt);

/** Check if a sensor is alarmed (value over threshold)
  */
bool is_sensor_alarmed(const sensor_item_t *sensor);

/** Send the list of all known sensors
  * Serial Output : - number of sensors (uint8)
  *                 - For each sensor :
  *                     - sensor info (sensor_info_t)
  *                     - time of last communication (uint32)
  *                     - last value (sensor_val_t)
  *                     - alarm status (bool)
  *                     - Link Quality (uint8)
  */
void send_sensor_list(void);

/** Modify a sensor
  * Serial input : - sensor_info_t
  */
void modify_sensor();

/** Dump sensors in ASCII form for debugging purposes */
void dump_sensors();

/** Delete a sensor
  * Serial input : - IEEE address
  */
void delete_sensor();


/** Ping ? Pong!
  * Serial output : PONG!
  */
```

```
141 void ping();
142
143 /** Init our structures in NVRAM (list of sensors and trigger events)
144   */
145 void init_nvram();
146
147 /** Clear all data in the coordinator (factory reset)
148   */
149 void clear_all();
150
151 /** Add a new trigger rule (IEE
152   * Serial Input :
153   *  - number of triggering sensors (uint8, can be 0)
154   *  - IEEE address of the actuator (8 bytes)
155   * Serial Output :
156   *  - error code (error_code_t)
157   */
158 void add_rule();
159
160 /** Send the list of all trigger rules
161   * Serial Output :
162   *   - number of rules (uint8)
163   *   For each rule :
164   *     - identifier of the rule
165   *     - IEEE address of the actuator
166   *     - number of triggering sensors
167   *     For each triggering sensor:
168   *        - IEEE address of the sensor
169   */
170 void send_rule_list();
171
172 /** Delete a trigger rule
173   * Serial Input :
174   *   - identifier of the rule
175   *   - IEEE address of the actuator
176   * Serial Output :
177   *   - error code (error_code_t)
178   */
179 void delete_rule();
180
181 /** Evaluate trigger rules (this is called by the coordinator periodically) */
182 void evaluate_triggers();
183
184 /** Send local time (in secs since startup) for time calculation
185   * Serial Output : - time (4 bytes)
186   */
187 void send_time();
188
189 /** Send data to a device in the network
190   * Serial Input :
191   *   - IEEE address of the destination (8 bytes)
192   *   - number of bytes of data to send (1 byte)
```

```
193   *   - data to send
194   * Serial Output :
195   *   - error code (error_code_t)
196   */
197 void send_data();
198 
199 /** Dump NVRAM usage info over the serial port */
200 void nvram_info();
```

Annexe C Énumérations de la classe *Communication.cs*

```
enum operation_code : byte
{
   CMD_GET_SENSOR_LIST = 0,
   CMD_DELETE_SENSOR = 1,
   CMD_MODIFY_SENSOR = 2,
   CMD_DUMP_SENSORS = 3,
   CMD_ADD_RULE = 4,
   CMD_GET_RULE_LIST = 5,
   CMD_DELETE_RULE = 6,
   CMD_DUMP_RULES = 7,
   CMD_GET_TIME = 8,
   CMD_SEND_DATA = 9,
   CMD_NVRAM_INFO = 10,
   CMD_PING = 0x42,
   CMD_CLEAR_ALL = 0xA0,
}

enum error_code : byte
{
   ERROR_SUCCESS = 0,       /* no error */
   ERROR_INVALID_FORMAT = 1, /* some field was missing or had an invalid value */
   ERROR_NOT_FOUND = 2,     /* requested item was not found */
   ERROR_INTERNAL = 3,      /* internal error */
   ERROR_UNKNOWN = 0xFF,    /* unknown error */
}

public enum sensor_type : byte
{
   SWITCH_SENSOR = 0,
   LIGHT_SENSOR = 1,
   LED_ACTUATOR = 2
};

public enum sensor_datatype : byte
{
   BOOL = 0,
   INT = 1,
   FLOAT = 2
};

public enum threshold_type : byte
{
   NONE = 0, /* No threshold set */
   EQ,       /* Equal */
   LTE,      /* Less than or equal (valid only for FLOAT and INT types) */
   LT,       /* Less than (strict, valid only for FLOAT and INT types) */
   GTE,      /* Greater than or equal (valid only for FLOAT and INT types) */
   GT        /* Greater than (strict, valid only for FLOAT and INT types) */ };
```

Résumé

L'objectif du projet était de développer un réseau de capteurs sans fil utilisant la technologie ZigBee, avec système de remontée de données d'alertes sur une tablette Windows. Pour cela, la solution retenue fut de développer la partie embarquée en C sur des modules ZigBee dotées d'un processeur TI CC2538. L'IHM pour superviser et gérer le réseau a été développé en C#, du fait que le logiciel est déployé sur un environnement Windows.

Un protocole de communication entre les modules et l'interface a été défini. Chaque capteur et actionneur communique avec le coordinateur quiest chargé de superviser le réseau. Celui-ci est connecté à l'interface et permet de remonter au logiciel les données de chaque composant se trouvant sur le réseau.

Des règles de déclenchement peuvent être définies pour activer un actionneur sous condition qu'un ou plusieurs capteurs mesurent une valeur en dehors d'un dseuil défini par l'utilisateur.

Mots-clés : ZigBee, réseau de capteurs, système de surveillance

Abstract

The objective of the project was to develop a wireless sensor network using the ZigBee technology, with the ability to report data and alerts to a Windows tablet. In order to achieve that, an embedded program in C was implemented for ZigBee modules based on the TI CC2538 processor. The graphical user interface used for supervision and network management was implemented in C#, since this interface is intended for Windows components.

A communication protocol between the modules and the interface has been defined. Each sensor and actuator sends data to the coordinator which is responsible for monitoring the network. This coordinator is connected to the GUI in order to send data for each component located on the network.

Triggering rules can be defined in order to start an actuator if one or more sensors reach a threshold defined by the user on the interface.

Keywords : ZigBee, sensor network, monitoring system