Rapport d'élève ingénieur

Projet de deuxième année

Filière : Informatique des systèmes interactifs pour l'embarqué, la Robotique et le virtuelle

---

Mise en place d'un système de tandem afin d'améliorer la fiabilité d'un réseau en étoile

---

Présenté par : *Ismail KANDIL & Venceslas DUET*

Responsable ISIMA : *Mamadou Kanté*

Responsables entreprise : *Antonio Freitas & Michel Misson*

Date de la soutenance *: 24/03/2023*

Durée du projet : *60 heures*

Campus des Cézeaux. 1 rue de la Chébarde. TSA 60125. 63178 Aubière CEDEX

Remerciements

Nous tenons à remercier Messieurs Michel Misson, Antonio Freitas et Théophile Decaesteker en tant que tuteurs de notre projet pour l'aide, les conseils et le suivi qu'ils ont pu nous fournir tout au long de ce projet. Ils nous ont permis d'avoir à notre disposition tout l'équipement nécessaire, à savoir deux cartes ATSAMR21-XPRO et deux OpenMoteB. Nous remercions également Monsieur Emmanuel Mesnard et Michel Cheminat pour leur cours sur les systèmes embarqués qui nous a été très utile dans ce projet.

Table des figures

1 Carte SAM R21 XPLAINED PRO 3
2 Microchip Studio 5
3 ASF WIZARD 5
4 Diagramme de Gantt prévisionnel 7
5 Diagramme de Gantt final 8
6 PIN du LED utilisateur 9
7 Séquence type d'un périphérique en mode esclave 10
8 Protocole I2C 13
9 Protocole SPI 13
10 Protocole UART 14
11 Shéma du protocole choisi 15
12 Pin SDA et SCL 16
13 Interconnexion entre un seul contrôleur et une seule cible en I2C viii
14 Diagramme de transaction de base I2C. viii
15 Diagramme d'état de bus I2C ix

Résumé

Ce projet vise à réaliser une boite à outils permettant à plusieurs cartes, contenant le même programme, de fonctionner de manière à permettre une communication sans fil en étoile. Cette boîte à outils sera intégrée dans une base de code pré-existante et devra s'intégrer dans un environnement embarqué géré par le système d'exploitation Riot OS.

Il a été déterminé lors des réflexions que les cartes devront communiquer entre elles et a été envisagé l'utilisation du bus I2C. La communication de deux cartes nécessite aussi bien l'utilisation du bus en mode contrôleur que cible. Riot OS ne proposant qu'une interface vers le périphérique I2C en mode contrôleur, il a été décidé de séparer le projet en deux parties : le développement du pilote matériel pour permettre le fonctionnement du périphérique I2C en mode cible, géré par Venceslas et le développement des preuves de concept démontrant l'utilisabilité du bus I2C dans les tâches demandées géré par Ismaïl.

À ce jour, les cas pratiques ont pu être vérifiés et le pilote I2C possède une interface stable. Toutefois, le pilote n'est pas encore totalement fonctionnel et la boite à outils n'est qu'en stade de prototype. Il est envisagé de fournir un API rudimentaire qui pourra être utilisé par le développeur principal dans le projet final.

clés :Réseau en étoile, Embarqué, I2C, Riot OS

Abstract

This project aims to create a toolbox allowing multiple boards containing the same program to function in a way that enables wireless communication in a star topology. This toolbox will be integrated into an existing codebase and must be compatible with an embedded environment managed by the Riot OS operating system.

During the planning stage, it was determined that the boards must communicate with each other and the use of the I2C bus was considered. Communication between two boards requires the use of the bus in both controller and target modes. As Riot OS only provides an interface to the I2C device in controller mode, it was decided to divide the project into two parts : the development of hardware drivers to allow the I2C device to operate in target mode, managed by Venceslas, and the development of proof of concept demonstrating the usability of the I2C bus in the required tasks, managed by Ismaïl.

To date, practical cases have been verified, and the I2C driver has a stable interface. However, the driver is not yet fully functional, and the toolbox is only in the prototype stage. It is envisaged that a rudimentary API will be provided, which can be used by the lead developer in the final project.

Keywords : Redundancy, Star Network, Embedded, I2C, Riot OS.

Lexique

Microcontrôleur : Un circuit intégré rassemblant dans un même boitier un microprocesseur, plusieurs types de mémoires et des périphériques de communication (Entrées-Sorties).

C : Langage de programmation impératif généraliste, de bas niveau. Inventé au début des années 1970 pour réécrire UNIX.

IDE : Acronyme de Integrated Development Environment (Environnement de Développement) est un ensemble d'outils pour augmenter la productivité des programmeurs qui développent des logiciels. Il comporte, par exemple, un éditeur de texte destiné à la programmation, des fonctions qui permettent, par pression sur un bouton, de démarrer le compilateur ou l'éditeur de liens, ainsi qu'un débogueur en ligne, qui permet d'exécuter ligne par ligne le programme en cours de construction.

Interruption : un mécanisme permettant de déclencher une action spécifique lorsqu'un événement extérieur se produit. Il interrompt temporairement l'exécution du programme en cours pour exécuter le traitement associé à l'événement détecté.

Broche : un connecteur physique permettant d'interfacer des composants électroniques externes avec la carte. Les broches peuvent être utilisées pour transmettre des signaux d'entrée ou de sortie, tels que des données, des signaux de contrôle, des signaux d'horloge, etc.

Polynôme générateur : est un polynôme utilisé dans les algorithmes de codage de canal pour générer des codes de contrôle de redondance pour la détection ou la correction d'erreurs de transmission de données.

API : Acronyme d'Application Programming Interface est un ensemble de protocoles, de règles et de définitions de fonctions qui permettent à des applications logicielles de communiquer et d'échanger des données. Elle définit les interfaces de programmation qui permettent à une application d'accéder aux fonctionnalités et aux données d'une autre application, d'un système d'exploitation ou d'un service web de manière standardisée et sécurisée. L'API peut être utilisée pour développer des applications tierces, des plugins, des extensions et des intégrations de manière efficace et fiable.

Driver : Pilote informatique. Programme destiné à faire le lien entre un périphérique et l'ordinateur afin d'en utiliser les fonctionnalités.

AVR : famille de microcontrôleurs 8 bits créée par Atmel Corporation, maintenant une filiale de Microchip Technology.

ARM : Acronyme de Advanced RISC Machines est une architecture de processeurs RISC (Reduced Instruction Set Computing) utilisée dans les systèmes embarqués.

RISC : est une architecture de processeur qui utilise un jeu d'instructions limité pour améliorer les performances et réduire la complexité du processeur.

RIOT OS : est un système d'exploitation open-source et gratuit pour l'Internet des Objets (IoT) qui offre une plateforme de développement et de déploiement flexible pour les appareils connectés. Il est conçu pour être économe en énergie et en mémoire, et prend en charge une large gamme de périphériques et de protocoles de communication.

XOR : est un opérateur logique binaire qui retourne comme résultat, vrai (1) si les entrées sont différentes, et faux (0) si elles sont identiques. En d'autres termes, il renvoie 1 si une seule des deux entrées est vraie, mais pas les deux. L'opérateur XOR est souvent utilisé en cryptographie et en informatique pour la manipulation de bits.

SERCOM : est un registre matériel programmable qui contrôle les communications série synchrone et asynchrone avec des périphériques externes tels que des capteurs, des écrans ou des modules de communication. Il est utilisé pour configurer et contrôler les paramètres de communication, tels que la vitesse de transmission des données, le format de trame et le mode de synchronisation.

CNES : Centre National d'Études Spatiales est l'agence spatiale française chargée de la conception et de la mise en œuvre de programmes spatiaux pour la France.

IDLE : en I2C, c'est un état dans lequel les deux lignes de communication du bus I2C (SCL et SDA) sont toutes deux maintenues à un état haut (HIGH) lorsque le bus n'est pas utilisé pour une communication.

DMA : les processeurs de conception modernes utilisent un périphérique spécial, échangeant entre la mémoire principale et les périphériques utilisant traditionnellement les interruptions. Cela permet d'accélérer les opérations et de réduire l'utilisation du temps processeur utilisé.

Table des matières

Remerciements **i**

Table des figures **ii**

Résumé **iii**

Abstract **iii**

Lexique **iv**

Introduction **1**

1 Exposition du contexte initial **2**
1.1 Description du projet 2
1.2 Présentation du matériel 2
1.3 Présentation des outils de développement 4

2 Réalisation et conception **6**
2.1 Blinky project 8
2.2 Développement du pilote logiciel 9
2.2.1 API Riot OS 10
2.2.2 Communication avec le périphérique SERCOM 11
2.3 Mise en liaison des deux cartes 11
2.3.1 Choix du protocole de communication 11
2.3.2 Établissement du protocole 15
2.3.3 Premier resultat : partage de donnée 17
2.3.4 Inversion logicielle des rôles 18
2.3.5 Envoi d'une ancienne structure 19
2.3.6 Integration d'un code CRC 19
2.4 Gestion d'interruption 20
2.4.1 Protocole des interruptions 20
2.4.2 Première utilisation : Moniteur de battement de cœur 22
2.4.3 Deuxième utilisation : Inversion matérielle des rôles 23

3 Résultats et discussion **24**
3.1 Analyse des résultats 24
3.2 Extensions possibles 24

Conclusion **25**

Annexe **viii**

Introduction

Les lanceurs de fusée modernes utilisent une grande quantité de capteurs afin de garantir une connaissance exacte de l'environnement. Cette grande quantité de capteurs induisent l'utilisation d'une grande quantité de câbles pour relier directement le capteur au nœud de calcul.

Dans le but de réduire l'utilisation des câbles, le centre national d'études spatiales (CNES) envisage l'utilisation de capteurs sans fils. L'équipe gérée par Antonio Freitas fait partie des équipes explorant une piste d'implémentation de cette volonté consistant en l'utilisation d'un réseau en étoile.

Cette topologie en étoile possède l'avantage d'être simple à mettre en place, cependant, la totalité de la communication dépend d'une unique carte, appelé puits, et dont une panne entraîne la panne du réseau entier.

Étant une agence spatiale, le CNES ne peut pas accepter un système aussi fragile. Afin de rendre ce projet plus robuste, M. Freitas nous a mandaté dans le but de réfléchir par le biais de séances de remue-méninges ainsi que d'implémenter une solution capable de détecter des pannes et d'agir en conséquence.

Après donc trois ans de développement, l'équipe a donc fait appel à des étudiants de l'ISIMA afin de proposer une solution technique permettant une superivision et une mise en redondance de ce puits.

1 Exposition du contexte initial

1.1 Description du projet

Nous avons sélectionné notre projet parmi les options proposées par notre responsable de filière : "Tandem - Réseau de capteurs sans fil". Ce projet nous a attiré en raison de la combinaison entre programmation embarquée temps réel et les concepts de réseaux. De plus, ce sujet semble être un complément pertinent pour notre cours actuel sur les systèmes embarqués. Nous avons déjà travaillé avec M. MESNARD sur une platine EasyPic6 et nous avons étudié divers aspects de l'embarqué, tels que le bus de communication I^2C et la gestion des interruptions. Cependant, nous n'avons jamais eu l'occasion de travailler sur un projet impliquant la communication entre deux microcontrôleurs.

À la suite de ce choix, nous avons dû développer et programmer des bibliothèques logicielles permettant de programmer deux cartes en mode pilote et copilote pour sécuriser la communication avec un réseau de capteurs sans fil et les intégrer avec un code existant.

Le contexte initial du sujet découle du besoin du CNES* de remplacer les installations filaires dans les lanceurs par des installations sans fil. Ce besoin est lié à une question cruciale dans le monde de l'aéronautique, qui influe sur la conception des engins concernés : le besoin de réduire le poids.

Le code existant implique un protocole qui concerne les réseaux de capteurs sans fil. Dans la théorie, il s'agit d'un réseau en forme d'étoile, dans lequel chaque branche de l'étoile contient un capteur, et le centre de l'étoile est le puits qui commande les capteurs et récupère les données collectées.

Notre rôle était de renforcer la sécurité de ce protocole en ajoutant un deuxième puits, en parallèle au premier. Le deuxième puits avait pour mission de s'assurer que le premier avait bien collecté toutes les données provenant de tous les capteurs et que le puits fonctionnait correctement sans aucun problème matériel. Si le deuxième puits détectait un problème ou une divergence dans les données recueillies, il devait prendre le relais pour assurer le bon fonctionnement du système. Cette configuration permettait de garantir une sécurité accrue et une fiabilité maximale du système de capteurs sans fil dans les lanceurs.

Nous allons, à la suite de cette description du projet, présenter en détails l'ensemble du matériel qui nous a été nécessaire pour sa réalisation.

1.2 Présentation du matériel

Pour notre projet, le principal sujet d'étude est la carte SAM R21 Xplained Pro (Figure 1). Elle est fabriquée par la société Microchip Technology, qui a acquis l'activité

microcontrôleurs de la société Atmel en 2016. Le microcontrôleur sur la carte, le SAM R21, a été développé par Atmel, qui a été l'un des premiers fabricants à introduire des microcontrôleurs basés sur la technologie ARM Cortex-M.

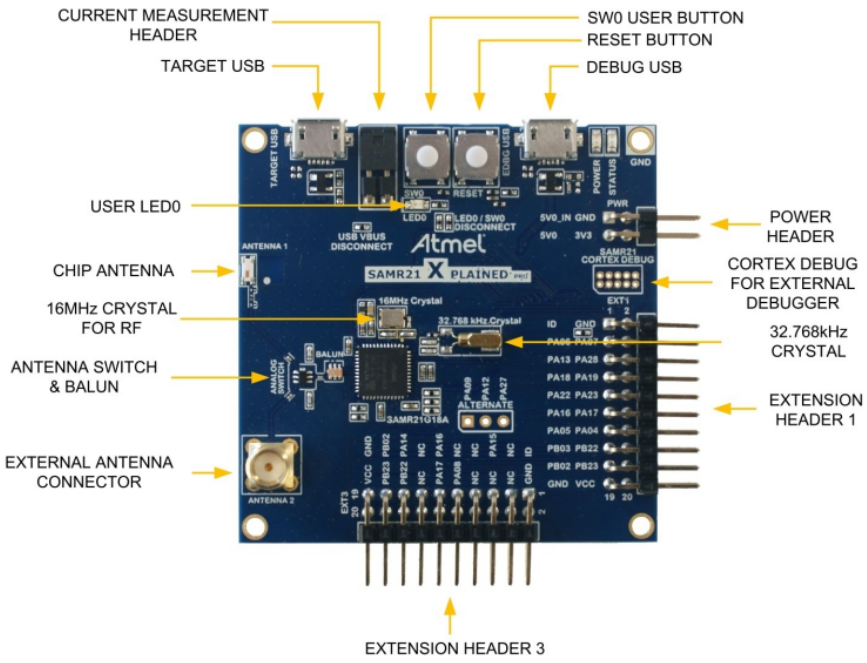


FIGURE 1 – Carte SAM R21 XPLAINED PRO

La carte SAM R21 Xplained Pro est un outil de développement puissant pour les applications basées sur le microcontrôleur ARM Cortex-M0+ de Microchip. Elle est dotée d'un microcontrôleur ATSAMR21G18A de 32 bits cadencés à 48 MHz avec une mémoire Flash 256 ko, mémoire statique de 32 ko, offrant une grande puissance de traitement pour les applications de capteurs sans fil.

La carte mesure 60 mm x 60 mm x 1.6 mm et dispose de nombreuses interfaces, notamment un port de programmation USB qui permet de programmer et déboguer le microcontrôleur de manière rapide et facile. Elle est également équipée d'un bouton de réinitialisation pour redémarrer le système, d'un autre bouton et d'une LED dédiée aux applications de l'utilisateur, et deux LED pour indiquer l'état de la carte.

La carte SAM R21 Xplained Pro possède également des connecteurs pour une antenne externe et des périphériques externes tels que des capteurs, des actionneurs et des cartes d'extension, ce qui permet de développer facilement des applications non filaires complexes.

L'alimentation de la carte peut être assurée par une source d'alimentation externe de 5V, ou par le port USB de l'ordinateur. La carte est compatible avec l'IDE de dévelop-

pement Microchip Studio (anciennement Atmel Studio) et fournit une variété d'exemples de code pour aider les développeurs à commencer rapidement leur travail.

En résumé, la carte SAM R21 Xplained Pro est un outil de développement riche en fonctionnalités pour les applications de capteurs sans fil. Elle offre une grande puissance de traitement grâce à son microcontrôleur SAM R21 de 32 bits, ainsi qu'une variété d'interfaces et de connecteurs pour une intégration facile avec des périphériques externes. Elle est largement utilisée dans le développement d'applications de capteurs sans fil grâce à sa fiabilité, sa flexibilité et sa facilité d'utilisation.

À la suite de cette présentation du matériel utilisé (hardware), nous allons exposer les outils de développement (logiciel) nécessaires à la création de notre application.

1.3 Présentation des outils de développement

Nos tuteurs nous ont initialement recommandé d'utiliser RIOT OS comme outil de développement pour notre projet. Cependant, nous avons rapidement réalisé que RIOT OS ne répondait pas totalement à nos besoins, notamment en termes de programmation en parallèle des deux cartes, car cet outil ne dispose pas des fonctionnalités nécessaires pour connecter deux microcontrôleurs. Nous avons donc cherché une solution alternative et avons découvert que le fabricant de notre carte proposait un environnement de développement appelé Microchip Studio (Figure 2).

Microchip Studio est un environnement de développement intégré (IDE) pour les microcontrôleurs Microchip AVR et ARM. Il est conçu pour aider les programmeurs à développer, déboguer et programmer des applications pour des microcontrôleurs Microchip.

Microchip Studio comprend plusieurs fonctionnalités, notamment un éditeur de code avancé, un compilateur de code source, un débogueur intégré et un programmateur de microcontrôleurs. Il est également équipé de fonctions telles que la coloration syntaxique, l'auto-complétion de code.

Le débogueur intégré de Microchip Studio permet aux programmeurs de surveiller les variables en temps réel, d'exécuter le code pas à pas, de mettre en place des points d'arrêt et de détecter les erreurs de code. Le programmateur de microcontrôleurs permet aux programmeurs de transférer le code compilé sur le microcontrôleur et de le tester.

Microchip Studio intègre aussi une fonctionnalité « ASF Wizard » (Figure 3) permettant d'accéder à l'ASF. Ce dernier est une collection de bibliothèques de logiciels prêtes à l'emploi pour les microcontrôleurs Atmel, intégrée dans l'environnement de développement

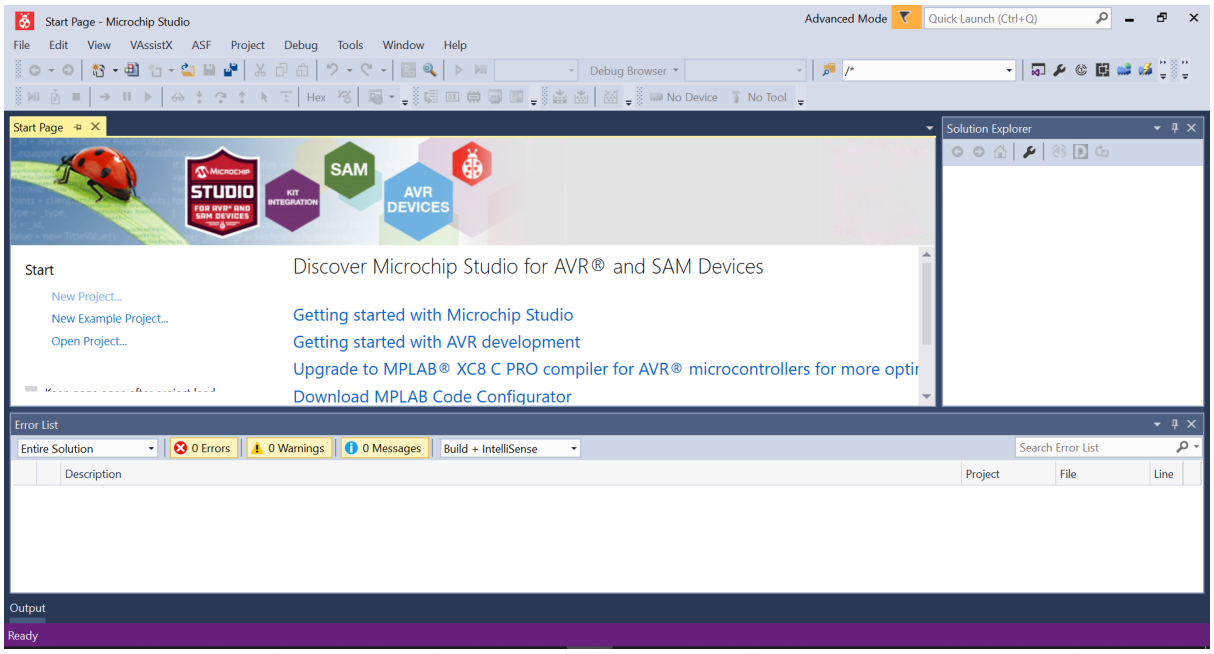


FIGURE 2 – Microchip Studio

Microchip Studio. Cette bibliothèque fournit des fonctionnalités pour le développement de projets embarqués avec les microcontrôleurs Atmel, tels que la gestion des communications, les contrôleurs de périphériques, les interfaces utilisateur et les fonctions de sécurité.

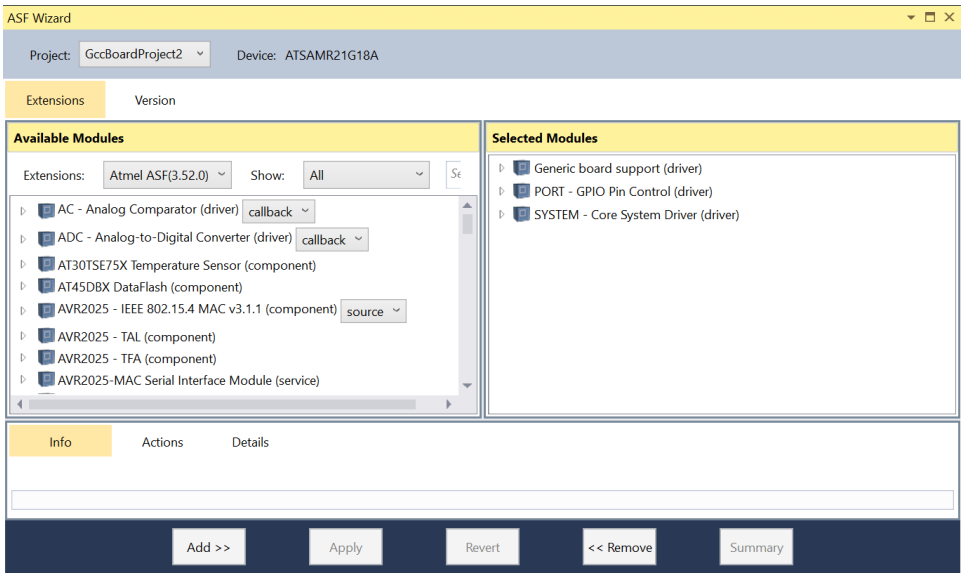


FIGURE 3 – ASF WIZARD

Nous avons aussi besoin d'un outil appelé LogicPort, qui est un simulateur graphique permettant de visualiser en temps réel le transfert de données entre deux cartes, dans le but de faciliter la visualisation de ce transfert de données.

2 Réalisation et conception

Initialement, le sujet de notre projet était formulé de la manière suivante : « *Nous allons considérer un réseau de capteurs sans fil organisé en étoile, le cœur de ce réseau interroge à tour de rôle l'ensemble des nœuds à sa portée selon un ordre convenu (ceci est le trajet). Une défaillance du nœud au cœur du réseau a un impact fort, c'est pourquoi il est prévu de mettre à cet endroit une solution basée sur deux cartes travaillant en tandem : l'une est le Pilote, elle organise les échanges, l'autre est le Passager, elle surveille les choix faits par le Pilote dans l'exécution de sa mission. Les deux cartes connaissent parfaitement la séquence des échanges à respecter. Le protocole mis en œuvre à ce niveau est déjà spécifié. Il s'agit pour la carte Passager de surveiller la bonne exécution de ce protocole et, en cas d'anomalie constatée, de demander au Pilote de suspendre son activité (pour subir une réinitialisation par exemple) et d'accepter une inversion des rôles* ».

Après notre première rencontre avec nos tuteurs, nous nous sommes sentis un peu perdus face à la complexité du protocole de communication avec les capteurs sans fil. Nous avons alors eu des doutes quant à notre choix de projet. Cependant, en tant que futurs ingénieurs, nous avons pris conscience de l'importance de relever ce défi et nous avons mis en place une méthodologie de travail. Tout d'abord, nous avons identifié les mots clés du sujet et avons étudié plusieurs rapports sur le code existant pour éclaircir tout point d'interrogation. Ensuite, nous avons divisé notre projet en sous-projets, en suivant l'intitulé du sujet, afin d'obtenir rapidement des résultats concrets et d'adapter nos attentes au fur et à mesure de l'avancement des travaux. Ces sous-projets étaient les suivants :

- Blinky project ;
- Mise en liaison des deux cartes ;
- Protocole d'inversion des rôles ;
- Traitement des cas d'erreur ;

Suite de cette première décomposition, nous avons réalisé un diagramme de Gantt prévisionnel (Figure 4) :

Nous avions établi une méthodologie de travail en commençant par un mini-projet simple, qui consistait à faire clignoter une LED de la carte ATSAMR21-XPRO. Dans un second temps, nous avions prévu de mettre en place un protocole de communication entre les deux cartes, une notion que nous n'avions jamais abordée auparavant, étant donné que nous n'avions communiqué qu'avec d'autres périphériques tels que la mémoire, le GLCD, le LCD ou encore le capteur de température. Une fois cette étape franchie, notre objectif était de traiter les cas d'erreurs que la deuxième carte devait détecter et de développer le

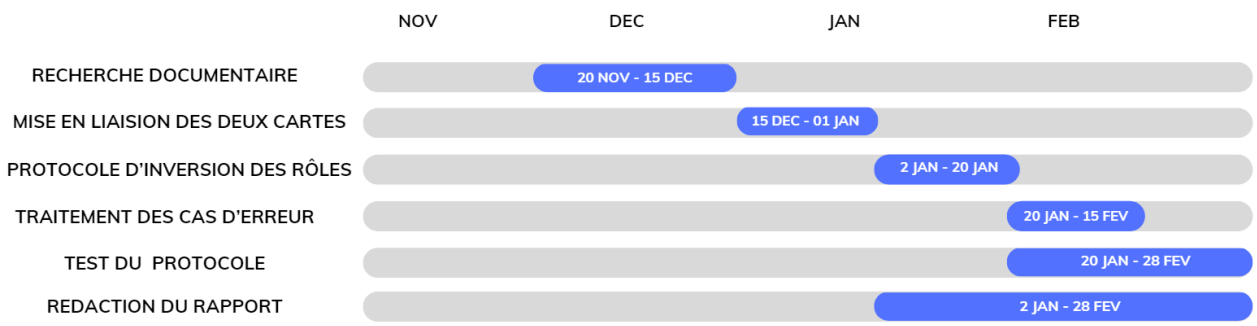


FIGURE 4 – Diagramme de Gantt prévisionnel

protocole d'inversion de rôles, c'est-à-dire la capacité de la deuxième carte à demander à la première de ne plus être le pilote.

À la suite de recherches plus approfondies et de consultations auprès de notre tuteur, notre décomposition s'est avérée ne pas être assez pertinente. Comme on a expliqué auparavant, l'api RIOT OS ne permet pas de lier deux cartes. On a du trouvé une autre solution rapidement pour qu'on puisse avancer sur notre projet. De plus, le cahier de charges final était le suivant :

- Proposer un protocole d'échange de données à partir d'une structure définie dans le code existant. ;
- Mettre un fil d'état capable de faire en sorte que la deuxième carte puisse interrompre la carte principale ;
- Échanger les rôles ;
- Développement d'un pilote logiciel pour Riot OS ;
- Développement d'une boite à outils ;

Il a alors fallu ajouter des sous-projets concernant tout d'abord la gestion d'interruption du fil extérieur, la prise en compte du code existant pour réussir à envoyer des données en respectant le protocole de communication que nous avons établie. Enfin, il fallut penser à toutes les méthodes qui pourraient être un plus pour cette recherche.

Au cours du projet, nous avons élaboré plusieurs diagrammes de Gantt, y compris le deuxième, le troisième et plusieurs autres jusqu'à obtenir la version définitive présentée dans la figure ci-dessous (Figure 5) :

Concernant la répartition du travail, les parties de recherche ont été faites en commun, tandis que les parties de développement ont été effectuées en parallèle selon deux démarches distinctes : Ismail KANDIL qui utilise l'API ASF et Microchip Studio, et Venceslas DUET qui est resté sur RIOT OS tout en essayant d'écrire les pilotes nécessaires afin d'utiliser ce dernier dans la communication de plusieurs cartes.

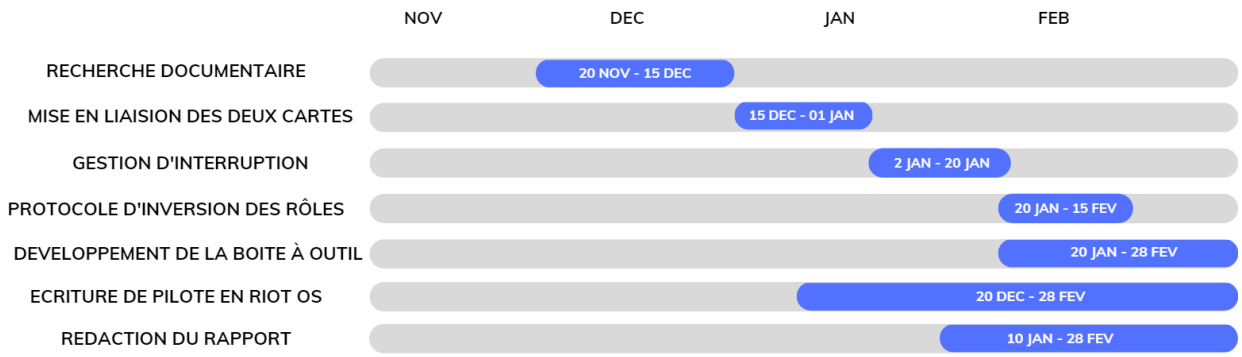


FIGURE 5 – Diagramme de Gantt final

Après avoir exposé l'ensemble des sous-projets, nous allons maintenant les détailler séparément dans les parties suivantes.

2.1 Blinky project

Afin de commencer à utiliser Microchip Studio, on a décidé à réaliser un premier projet simple, mais aussi classique pour tout programmeur en informatique embarqué qui est le projet "Blinky Project". L'objectif de ce dernier est de clignoter la LED d'utilisateur qui se trouve au niveau de notre carte. À ce stade, nous avons commencé à lire les documents liés au microcontrôleur de la carte afin de savoir à quel port de sortie est attaché cette LED, et aussi de chercher dans les documents liés à l'environnement Microchip studio afin de savoir quelles fonctions qui nous serons utiles.

Avant de commencer ce projet, nous avons tout d'abord récupéré nos deux cartes cibles auprès de nos tuteurs lors de notre première rencontre à LIMOS, à l'IUT d'informatique d'Aubière. Au cours de cette rencontre, nous avons discuté avec un développeur informatique qui était avec nos tuteurs et qui a développé une bonne partie du protocole principal de ce grand projet. L'idée était qu'il échange avec nous les différentes documentations qui pourraient être utiles pour notre projet. Il nous a même envoyé ses rapports de stage et de projet pour que nous puissions comprendre l'ancien protocole. Cela nous a donné une solide base pour poursuivre notre projet avec confiance et assurance.

Après avoir fait une première lecture générale de la documentation fournie avec notre carte, nous avons effectué d'autres recherches sur le site du concepteur de la carte afin de trouver la documentation du design de la carte. À partir de celle-ci, nous avons pu savoir quel port de sortie pourrait nous permettre de faire clignoter la LED de l'utilisateur. Cette dernière est liée au port PA19, qui doit être configuré en sortie (Figure 6).

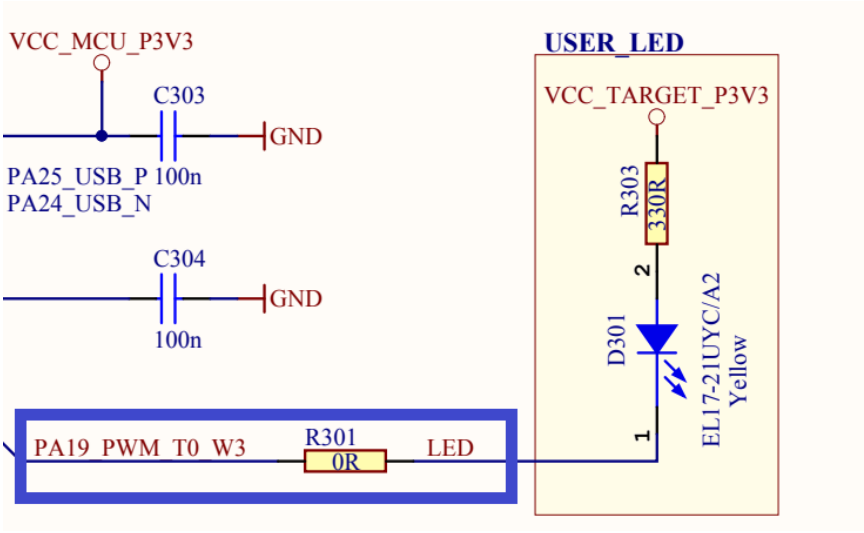


FIGURE 6 – PIN du LED utilisateur

À partir de la documentation de l'API ASF, on a appris que la fonction "port-pin-set-output-level" permet de configurer le port de la LED en sortie, c'est-à-dire de l'allumer. Le nom de ce port doit être inclus en tant que premier paramètre pour cette fonction. Cette dernière appartient au driver "PORT" qui a été automatiquement ajouté dès la création de notre projet. Ce dernier gère toutes les entrées/sorties de la carte et sera utile au cours de ce projet. Enfin, nous avons réussi à faire clignoter la LED. Ce projet sera utilisé après pour savoir à quelle partie on est arrivé dans notre code.

À travers ce mini-projet, nous avons compris comment utiliser l'ASF WIZARD de Microchip Studio et comment inclure n'importe quel driver dans les projets. De plus, nous avons compris comment utiliser les différentes fonctionnalités de Microchip Studio et la conception de code, ce qui est intéressant pour la suite du projet.

Grâce à notre familiarisation avec l'environnement MICROCHIP STUDIO, ainsi qu'à nos recherches documentaires à l'aide de ce premier sous-projet, nous sommes désormais prêts à aborder des sous-projets plus complexes présentés dans la suite de notre apprentissage.

2.2 Développement du pilote logiciel

Il a été remarqué assez tôt que Riot OS ne supportait pas de base l'utilisation du périphérique I2C en mode esclave. Il a été donc entrepris de mettre en place un pilote logiciel permettant cela et plus de développer les tests en parallèle en utilisant l'ASF.

Ce développement est séparé en deux parties :

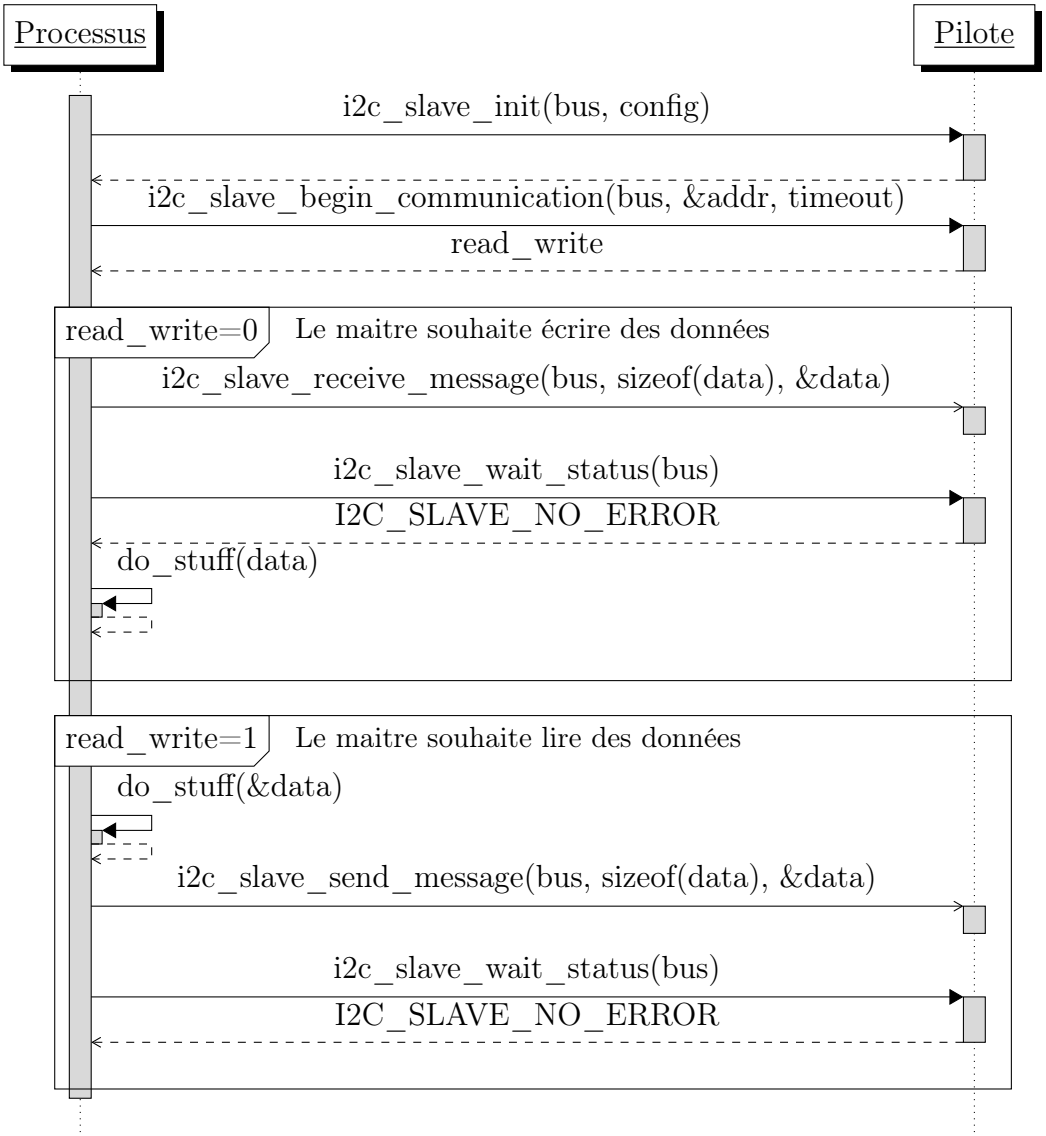
- Le développement de l'API tenant compte des spécificités de Riot OS

- Le développement du pilote à proprement parlé, limitant l'utilisation de code nouveau et tenant compte des spécificités du microcontrôleur.

2.2.1 API Riot OS

Riot OS est un système d'exploitation embarqué basé sur un ordonnanceur et permettant l'exécution de plusieurs processus légers. De ce fait, il devient intéressant de ne pas utiliser des fonctions de rappel, mais plutôt des fonctions bloquant le fil d'exécution du processus en cours.

FIGURE 7 – Séquence type d'un périphérique en mode esclave



Cela permet de ne pas avoir à écrire du code exécuté par les interruptions en tant que développeur dans l'espace utilisateur et ainsi réduire le risque de problème logiciel lors de l'exécution.

La méthode utilisée pour rendre cela possible est actuellement l'utilisation d'attente active, mais la configuration de l'ordonnance est envisagée pour rendre cette partie efficiente, si le temps le permet.

2.2.2 Communication avec le périphérique SERCOM

Le code travaillé étant un pilote matériel, il est nécessaire de pouvoir communiquer avec les périphériques SERCOM du microcontrôleur.

Le SERCOM, pour SERial COMmunication interface, est le périphérique développé par la société Atmel et intégré dans leurs microcontrôleurs 32 bit. Il supporte les bus série USART, SPI en enfin I2C.

Le support de l'I2C par le périphérique SERCOM propose toutes les fonctionnalités normales fonctionnalités du bus dont les différents modes de vitesse, le fonctionnement maitre esclave ainsi que la reconfiguration en fonctionnement. Cela le rend donc théoriquement utilisable pour notre projet, ce qui est vérifié dans le projet fait avec ASF.

Il existe plusieurs modes d'utilisation du périphérique SERCOM :

— Le mode classique utilisant les interruptions ainsi que le registre DATA pour récupérer les données émises

— Le mode utilisant le périphérique DMA afin d'optimiser l'utilisation du processeur.

La solution actuellement envisagée est l'utilisation du mode classique qui est plus simple à implémenter de notre côté (il n'est pas nécessaire de manipuler les registres du DMA)

L'idée donc du pilote est de manipuler dans l'API des registres internes au pilote paramétrant la boucle d'interruption.

La boucle d'interruption elle-même vérifie les registres d'états du périphérique SERCOM, d'un côté, ainsi que de l'API afin de pouvoir effectuer les actions demandées

2.3 Mise en liaison des deux cartes

2.3.1 Choix du protocole de communication

Dès la première lecture du sujet de projet, le fait de programmer les deux cartes en parallèle nous a donné l'idée qu'il faut un bus de communication pour envoyer et partager les données lors de la communication avec le réseau de capteurs sans fil. En effet, après plusieurs réflexions, nous avons confirmé qu'il faut avoir ce bus de communication et donc que notre projet repose sur celui-ci.

Cependant, cette tâche ne s'avérait pas aussi simple que prévu. En effet, parmi les contraintes imposées par nos tuteurs, il était exigé que nous développions un protocole de communication qui permette l'établissement de cette communication entre plusieurs

cartes, pour une éventuelle extension du projet. Cette exigence rendait notre travail encore plus complexe, car nous devions non seulement développer un protocole de communication pour deux cartes, mais également tenir compte de l'évolutivité du projet.

Malheureusement, au cours de nos TP et cours, nous n'avons jamais abordé la partie qui traite de la communication entre deux microcontrôleurs en liaison série. Nous avons donc effectué plusieurs recherches, que ce soit au niveau des forums ou des ouvrages, pour connaître les différentes options et techniques et pour comprendre les différents avantages de chacune pour atteindre cet objectif.

Après avoir effectué des recherches, nous avons identifié trois types de communication série qui peuvent être utilisés pour établir une communication :

- I2C (Inter-Integrated Circuit) est un protocole de communication série synchrone utilisé pour la communication de données entre différents circuits intégrés sur une carte électronique.
  Le protocole I2C permet la communication entre des composants électroniques tels que des microcontrôleurs, des capteurs, des mémoires, des interfaces d'affichage, etc. Il utilise deux lignes de communication : la ligne de données (SDA) et la ligne d'horloge (SCL).
  La communication se fait sous forme de trames de données, avec des adresses d'identification des périphériques qui permettent à plusieurs composants de communiquer sur le même bus I2C. Les données sont transmises de manière synchrone avec l'horloge du bus, ce qui permet d'assurer la synchronisation entre les différents composants.
  Le protocole I2C offre plusieurs avantages, tels que la possibilité de connecter plusieurs périphériques sur le même bus, la gestion des conflits de bus grâce à une méthode de maître/esclave, la gestion de l'énergie grâce à la possibilité de désactiver les périphériques en mode veille, et une faible consommation d'énergie(Figure 8).

- SPI (Serial Peripheral Interface) est un protocole de communication série synchrone utilisé pour la communication de données entre des circuits intégrés sur une carte électronique.
  Le protocole SPI utilise quatre fils de communication : une ligne de données bidirectionnelle (MOSI - Master Out Slave In), une ligne de données bidirectionnelle (MISO - Master In Slave Out), une ligne d'horloge (SCK - Serial Clock) et une ligne de sélection de périphérique (SS - Slave Select). La ligne de sélection de périphérique permet de sélectionner le périphérique avec lequel la communication doit être

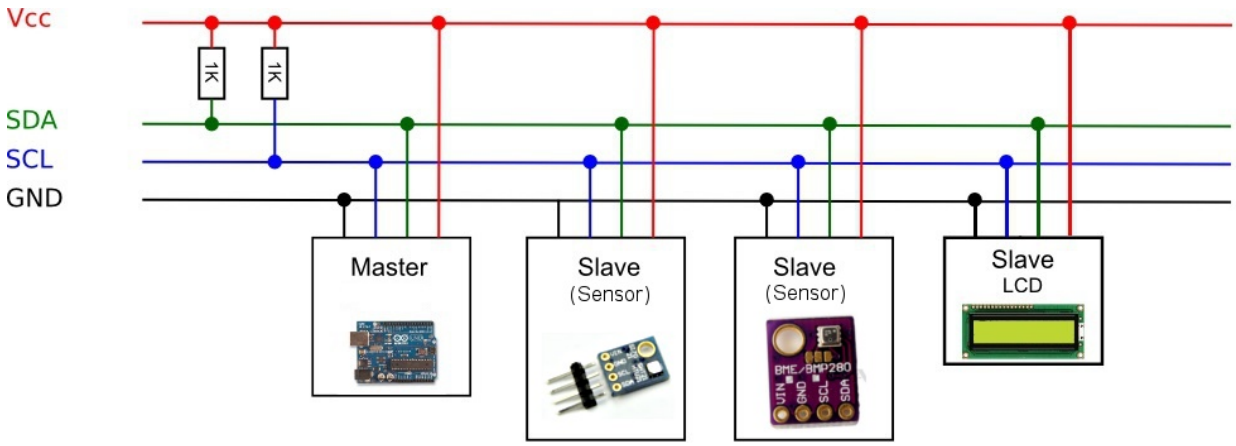


FIGURE 8 – Protocole I2C

établie.

La communication SPI est basée sur des échanges de trames de données, avec des formats de données prédéfinis pour les commandes et les réponses. Les trames de données sont transmises de manière synchrone avec l'horloge du bus, et les données sont transférées simultanément dans les deux sens sur les lignes MOSI et MISO.

Le protocole SPI offre plusieurs avantages, tels que la vitesse élevée de communication, la possibilité de transférer des données de manière simultanée dans les deux sens, et la possibilité de connecter plusieurs périphériques sur le même bus SPI. En revanche, SPI utilise plus de fils de communication que d'autres protocoles tels que I2C, ce qui peut limiter son utilisation dans des systèmes avec des contraintes de connectivité (Figure 9).

FIGURE 9 – Protocole SPI

- UART (Universal Asynchronous Receiver/Transmitter) est un protocole de communication série asynchrone utilisé pour la transmission de données entre des composants électroniques. Le protocole UART utilise deux fils de communication : une ligne de données bidirectionnelle (TX - Transmitter) et une ligne de données bidirectionnelle (RX - Receiver).

  La communication UART est asynchrone, ce qui signifie qu'il n'y a pas de signal d'horloge dédié pour synchroniser la transmission et la réception de données. Au lieu de cela, la transmission et la réception de données sont basées sur un ensemble

de règles prédéfinies pour la transmission et la réception des données.
Lorsque des données sont transmises via la ligne TX, elles sont transmises à une vitesse prédéfinie, généralement exprimée en bits par seconde (bauds). Le récepteur RX reçoit ensuite les données et effectue une opération de désérialisation pour les convertir en données compréhensibles par le système. Pour faciliter la synchronisation, un bit de départ et un bit d'arrêt sont ajoutés à chaque transmission de données.
Le protocole UART offre plusieurs avantages, tels que la simplicité de mise en œuvre, la faible utilisation des ressources matérielles et la grande compatibilité avec une large gamme de dispositifs. Cependant, la communication UART est relativement lente par rapport à d'autres protocoles tels que SPI ou I2C, et elle ne permet pas la transmission de données simultanées dans les deux sens (Figure 10).

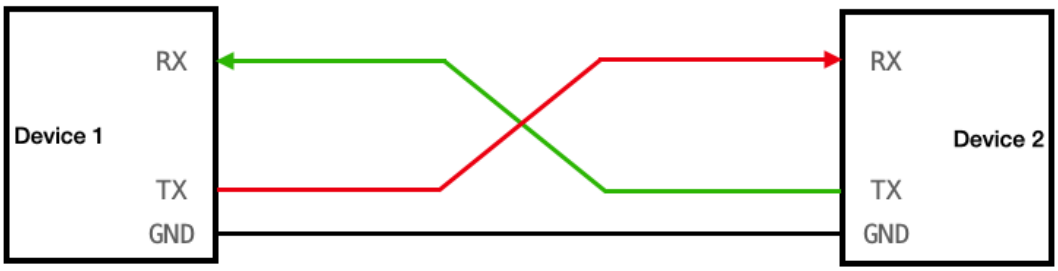


FIGURE 10 – Protocole UART

Le choix d'un protocole de communication dans un projet diffère d'un projet à un autre selon plusieurs motifs, par exemple le nombre de périphériques à connecter, la vitesse et la fiabilité de la communication, etc.

Pour le cas de notre projet, notre principale contrainte est d'établir un protocole qui permet non seulement la communication entre seulement deux cartes SAMR21, mais aussi en consommant le moins d'énergie possible et en minimisant l'utilisation des fils. Nous souhaitons également choisir un protocole facile à mettre en place pour réduire le temps de développement et les coûts associés.

Après avoir effectué des recherches sur les différents protocoles de communication, nous avons décidé de choisir le protocole I2C pour ce projet (Figure 11). Comme nous l'avons mentionné précédemment, l'I2C respecte toutes nos contraintes. Ce dernier utilise un minimum de fils de communication et permet de ne pas gaspiller d'énergie. De plus,

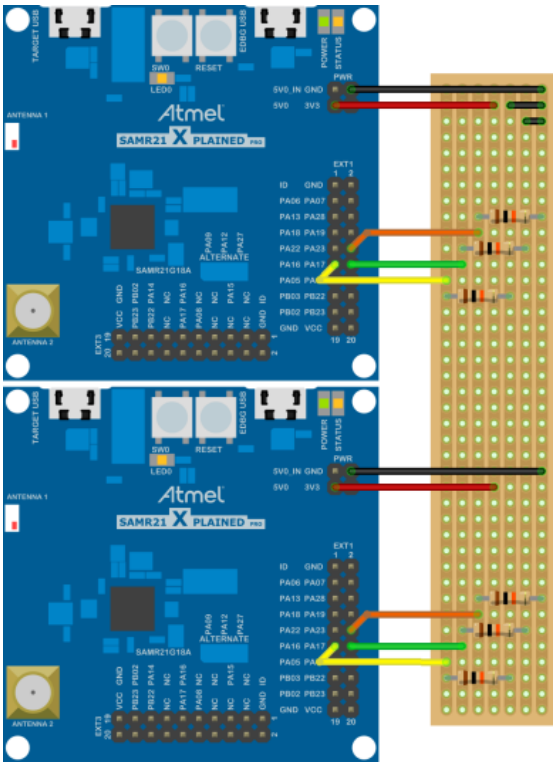


FIGURE 11 – Shéma du protocole choisi

nous avons trouvé plusieurs bibliothèques en ASF gérant ce protocole, ce qui facilitera encore plus notre développement.

De plus, l'I2C est un protocole de communication fiable et bien établi dans l'industrie électronique, avec une grande disponibilité de composants compatibles sur le marché. Il permet une communication à une vitesse modérée, allant jusqu'à plusieurs centaines de kilobits par seconde, ce qui peut être suffisant pour la plupart des projets.

En somme, le choix de l'I2C pour notre projet s'explique par sa simplicité, sa faible consommation d'énergie, sa fiabilité et sa grande disponibilité de composants compatibles. Ces avantages font de l'I2C un protocole de communication idéal pour établir une communication entre deux ou plusieurs microcontrôleurs dans notre projet.

2.3.2 Établissement du protocole

Le protocole I2C permet la communication entre un dispositif contrôleur et un dispositif cible. Habituellement, le dispositif contrôleur est un microprocesseur, tandis que le dispositif cible est un périphérique non programmable tel qu'une mémoire. Toutefois, dans notre situation particulière, le dispositif cible sera également un microcontrôleur. **Problématique : Est-ce que RIOT OS permet de programmer une carte en mode esclave ?**

Après avoir commencé ce sous-projet, nous avons réalisé que l'API proposée par nos tuteurs, à savoir RIOT OS, ne comportait pas de fonctions permettant de programmer une carte en mode cible. Nous avons entrepris des recherches dans la communauté RIOT OS pour trouver une solution à ce problème, mais malheureusement, nous avons découvert que cela était impossible, non seulement pour le protocole I2C, mais également pour tous les autres protocoles de communication. Face à cette difficulté, nous avons décidé d'explorer d'autres options pour avancer dans le projet. Nous avons ainsi choisi de travailler avec Microchip Studio et d'écrire en parallèle les pilotes nécessaires pour rester compatible avec RIOT OS. Cette approche nous a permis de continuer à avancer dans le développement de notre projet malgré les contraintes rencontrées.

Avant de commencer à programmer ce protocole de communication, il faut tout d'abord chercher les bonnes broches à connecter. Ces derniers servent exactement à connecter les deux fils SDA (DATA) et SCL (Clock) du protocole i2c pour que le microcontrôleur ATSAMR21G18A puisse envoyer et recevoir des données. On peut les trouver simplement soit dans le document du design, soit dans le datasheet du microcontrôleur (Figure 12).

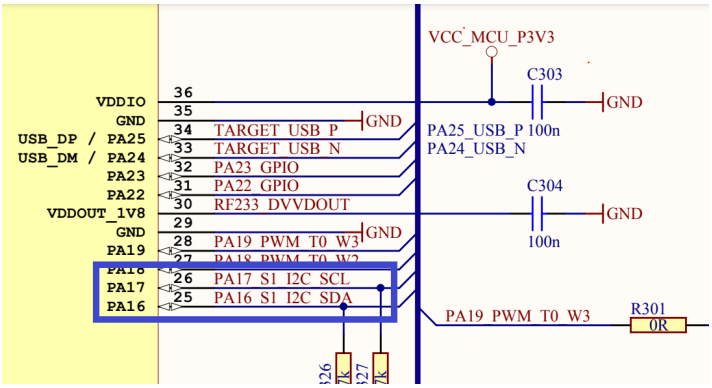


FIGURE 12 – Pin SDA et SCL

Le protocole I2C nécessite au moins deux appareils, dont au moins un doit être un microcontrôleur pour pouvoir être un contrôleur, et l'autre doit être une cible. Une configuration préalable doit être effectuée au niveau des registres SERCOM de ces périphériques en activant soit le mode "contrôleur", soit le mode "cible". Enfin, le plus important est de définir une adresse pour chaque cible afin que le contrôleur puisse les choisir.

Une fois que les deux cartes sont configurées et que les broches SDA et SCL sont correctement connectées, il est possible de commencer la communication via le protocole I2C. Le maître envoie une demande à l'esclave, qui répond avec les données requises. Cette communication peut être répétée plusieurs fois, selon les besoins de l'application.

En utilisant Microchip Studio, il était plus simple de mettre en place une communication entre deux cartes grâce à la bibliothèque ASF. Pour commencer, il fallait ajouter les

drivers « I2C master mode » et « I2C slave mode » au projet, puis programmer chaque carte en fonction de son rôle dans la communication.

Carte au rôle contrôleur :

Tout d'abord, on déclare une structure config-i2c-master de type i2c-master-config, qui sera utilisée pour stocker les paramètres de configuration du module I2C contrôleur, puis on appelle la fonction i2c-master-get-config-defaults(&config-i2c-master) qui initialise la structure config-i2c-master avec les valeurs par défaut. Cela permet de s'assurer que toutes les options de configuration sont initialisées avec des valeurs sûres avant d'y apporter des modifications spécifiques. Puis, on définit les broches utilisées pour la communication I2C. Dans notre projet, il s'agit des broches PA16 et PA17 du microcontrôleur SAMR21, qui sont connectées aux broches SDA et SCL du bus I2C (config-i2c-master.pinmux-pad0 = PINMUX-PA16D-SERCOM3-PAD0 ; config-i2c-master.pinmux-pad1 = PINMUX-PA17D-SERCOM3-PAD1 ;). Enfin, on appelle la fonction i2c-master-enable(&i2c-master-instance) qui est utilisée pour activer le module I2C. Elle définit l'état du bus I2C sur IDLE, c'est-à-dire qu'elle met le bus en attente d'une nouvelle transaction, avec i2c-master-instance est une structure qui est utilisée pour conserver les informations d'état du logiciel d'une instance de module matériel associé.

Carte au rôle cible :

Le programme de cette carte utilise une structure similaire à celle du maître pour communiquer via le protocole I2C. Toutefois, il est essentiel de définir l'adresse de l'esclave pour que le maître puisse le détecter et communiquer avec lui. La fonction "configure-i2c-slave" est utilisée pour définir cette adresse et configurer le périphérique I2C en mode esclave. Ensuite, le programme peut recevoir des données du maître et envoyer des données en réponse en utilisant les fonctions I2C appropriées.

2.3.3 Premier resultat : partage de donnée

Avant d'avancer sur le projet, il fallait être sûr que la communication a été établie, pour ceci, on doit simplement envoyer des données. Ces derniers seront stockés dans un tableau codé sur un octet, il serait envoyé depuis le contrôleur à l'aide de la fonction i2c-master-write-packet-wait. Dans cette dernière, on définit l'adresse du périphérique cible qui va de son côté utiliser la fonction i2c-slave-read-packet-wait pour recevoir ce tableau.

On s'est basé sur plusieurs exemples cités dans Microchip studio pour pouvoir savoir comment utiliser le protocole I2C. Ce dernier est un peu complexe et sans ces exemples, on ne sera pas capable de savoir comment utiliser les fonctions de la bibliothèque ASF. Mais enfin, nous avons réussi à envoyer ces données.

Au début, nous avons utilisé la LED de l'utilisateur du projet BLINKY afin de savoir si le code est bien exécuté, mais cela n'était pas suffisant. Heureusement, nous avons pu emprunter un appareil appelé LogicPort qui permet de visualiser les données envoyées dans l'I2C. Cela nous a permis de confirmer le succès de ce sous-projet.

2.3.4 Inversion logicielle des rôles

Avant de poursuivre ce projet, il était essentiel de s'assurer qu'il était possible de réaliser une inversion de rôles entre le maître et l'esclave. En effet, l'idée centrale du projet est que lorsque le maître, qui est responsable de piloter le système, commet une erreur, que ce soit d'ordre matériel ou logiciel, l'esclave doit lui envoyer un message l'invitant à arrêter de prendre les commandes afin que l'esclave puisse les prendre à sa place.

Pour l'instant, dans cette partie du projet, nous ne prendrons pas en compte le traitement de ce message envoyé par l'esclave au maître. Nous allons plutôt nous concentrer sur la mise en place d'une inversion logicielle : dès que le maître est en mesure d'envoyer une donnée, telle que le tableau précédent par exemple à l'esclave, il va désactiver lui-même sa fonction de maître. Ensuite, le processus sera inversé, avec le maître devenant esclave et l'esclave devenant maître pour renvoyer la donnée dans l'autre sens.

En somme, il s'agit de mettre en place un système de rotation des rôles entre le maître et l'esclave, permettant ainsi de garantir une plus grande flexibilité et une meilleure réactivité en cas d'erreur ou de dysfonctionnement dans le système.

Une fois que les adresses des deux cartes ont été définies, il a été facile de programmer le code correspondant. La première carte a été configurée avec l'adresse 0x14 et a joué initialement le rôle de maître, avant de passer en mode esclave. La seconde carte a quant à elle été configurée avec l'adresse 0x12 et a joué le rôle d'esclave, avant de devenir maître.

Au niveau de la programmation, il y avait deux blocs distincts, l'un pour le mode maître et l'autre pour le mode esclave. Les deux cartes devaient alterner entre ces deux blocs en effectuant quelques modifications pour assurer la transition en douceur entre les deux rôles.

En fin de compte, nous avons réussi à établir un premier échange de données entre les deux cartes, ce qui nous sera très utile dans la suite du projet. Cette expérience nous a permis de mieux comprendre comment mettre en place une inversion de rôles entre deux cartes et de l'appliquer de manière efficace dans le cadre d'un projet plus complexe.

2.3.5 Envoi d'une ancienne structure

Pour l'instant, nous avons réussi à envoyer un exemple simple qui est un tableau de données codé sur 8 bits. Le bus I2C ne permet en général que d'envoyer des paquets de 8 bits. L'ancien code que nous devons compléter génère, après une phase appelée la phase de découverte de voisinage, une liste chaînée dont chaque bloc représente les informations reçues d'un secteur donné.

Notre travail était de trouver une solution afin d'envoyer cette liste. La solution est de la sérialiser, c'est-à-dire de découper cette structure en une suite d'informations plus petites et de les stocker dans un tableau codé sur 8 bits.

Cette structure contient une adresse codée sur 16 bits, un autre entier sur 8 bits et enfin un tableau dont les éléments sont de type, une autre structure dont tous les éléments sont codés sur 8 bits. Même si la structure de cette information est complexe, il était facile de la sérialiser. Un simple parcours permet d'ajouter chaque élément de cette structure à un tableau de 8 bits, mais le problème se trouve au niveau de l'adresse codée sur 16 bits. La solution de cette dernière est plutôt simple, on l'a divisée en deux : les 8 premiers bits de poids fort, puis les autres 8 bits restants.

Après avoir fait des tests, la fonction marche très bien, mais le problème se trouve plutôt au niveau de la récupération de cette structure. Un autre travail est généré qui est la dé-sérialisation de ce tableau reçu. Ceci veut dire qu'il faut reconstruire à nouveau la structure de l'ancien code dans la carte cible afin de respecter la programmation en parallèle. Pour cela, il fallait simplement faire un retour en arrière.

Enfin, la sérialisation et la dé-sérialisation étaient une manière efficace pour envoyer cette structure, ceci nous permettrait d'avoir des résultats efficaces lorsqu'on inverse les rôles des deux cartes.

2.3.6 Integration d'un code CRC

Dans notre projet, il est important de ne pas développer un code sans le sécuriser. L'envoi de données via le protocole I2C sans intégrer un mécanisme de détection des erreurs n'est pas acceptable. En revanche, il existe plusieurs protocoles qui permettent la détection des erreurs en cas de transmission erronée, et parmi eux, nous avons choisi d'utiliser le CRC. Le CRC (Cyclic Redundancy Check) est une méthode de détection d'erreurs dans les données numériques qui est couramment utilisée dans les systèmes de communication numériques. Il s'agit d'un code de vérification de redondance cyclique qui consiste à ajouter un ensemble de bits supplémentaires (le CRC) à un message ou à un fichier. Les bits CRC sont calculés en utilisant un algorithme spécifique qui utilise

les données du message pour générer une somme de contrôle qui est ensuite ajoutée au message. Le récepteur peut alors recalculer la somme de contrôle à partir des données reçues et la comparer à celle qui a été envoyée avec le message pour détecter les erreurs de transmission.

Pour ceci, nous avons codé une fonction qui utilise un polynôme générateur fixe (POLY) de valeur 0x07 pour effectuer le calcul du CRC. Le calcul du CRC est effectué à l'aide d'une boucle qui parcourt l'ensemble des octets de données d'un tableau codé en 8 bits passé en paramètre. Pour chaque octet, le CRC est modifié en effectuant un XOR avec l'octet en question. Ensuite, pour chaque bit de l'octet (soit 8 bits), une opération de décalage vers la gauche («) est effectuée sur le CRC. Si le bit le plus significatif (bit 7) du CRC est à 1, un XOR avec le polynôme générateur est effectué. Cette opération est répétée pour chaque octet de données, jusqu'à ce que l'ensemble des octets ait été traité. Finalement, le CRC calculé est retourné en sortie de la fonction. Le récepteur peut alors recalculer la somme de contrôle à partir des données reçues et la comparer à celle qui a été envoyée avec le message pour détecter les erreurs de transmission.

2.4 Gestion d'interruption

2.4.1 Protocole des interruptions

Parmi les contraintes de l'I2C, il y a le fait qu'il ne permet pas à la carte cible d'envoyer des messages ou des commandes à la carte qui contrôle la communication. En effet, cela pose un grand problème pour notre projet. La question posée est : comment la carte cible qui joue le rôle de copilote pourrait-elle envoyer un message à la carte contrôleur qui est le pilote pour lui dire d'arrêter la communication quand elle commet une faute ? Après plusieurs discussions avec nos tuteurs, nous sommes arrivés à une solution : l'interruption. Cette dernière référence à une technique de programmation utilisée dans les systèmes informatiques temps-réel pour traiter des événements ou des tâches en temps opportun. Lorsqu'un événement survient tel qu'une entrée utilisateur ou un signal de périphérique, le processeur interrompt immédiatement l'exécution de la tâche en cours pour traiter l'événement. Le traitement de l'interruption consiste généralement à sauvegarder l'état actuel de la tâche en cours, à exécuter une routine de traitement spécifique à l'interruption et à restaurer ensuite l'état de la tâche précédente.

Ce sous-projet consiste à ce qu'une carte puisse envoyer une interruption à une autre carte, et que cette dernière puisse la détecter d'une manière unique, cette interruption sans aucun bruit extérieur. En effet, nous avons discuté avec nos tuteurs que les bruits électromagnétiques sont parmi les contraintes principales dans les fusées spatiales. Ainsi,

il faut sécuriser l'envoi et la réception de ces interruptions pour ne pas générer des effets de bord qui auront une grande influence sur le projet.

Le programme est divisé à nouveau en deux : Un pour la carte qui envoi l'interruption, et l'autre pour la carte qui la reçoit.

Du côté expéditeur, on a créé une fonction init-irq-pin () qui initialise une broche d'entrée/sortie (IO) qui sera utilisée pour générer une interruption. La fonction utilise la bibliothèque ASF (Advanced Software Framework) pour la configuration de la broche. Elle configure la direction de la broche en tant que sortie, puis la met à un niveau logique bas. Le choix de cette broche n'est pas important puisqu'il y a plusieurs d'entrée/sortie, plusieurs broches la carte. Puis une autre fonction send-interrupt (), envoie une interruption en émettant un signal sur la broche préalablement configurée. La faite d'envoyer un signal sous forme d'un pique n'a pas été suffisant. En effet, qu'on touche la broche réceptionniste de cette interruption, on observe qu'elle reçoit une interruption, donc on n'a pas répondu à l'une de nos problématiques. La solution qu'on a essayée est d'utiliser les résistances de pull-up et pull-down afin de fixer le niveau logique d'une broche d'entrée lorsque cette broche n'est pas connectée à une source de signal. En fixant le niveau logique de la broche, les résistances de pull-up et de pull-down peuvent aider à réduire les niveaux de bruit générés. Cependant, elles ne peuvent pas éliminer complètement le bruit. Mais ceci n'est pas accepté, on doit être sûr que la transmission est sécurisée. Enfin, on a modifié la fonction qui envoi l'interruption. En effet, on a utilisé une boucle pour envoyer 110 impulsions. À chaque itération, elle met la broche de la carte expéditrice à un niveau logique haut, puis bas, avec une pause de 50 microsecondes entre chaque changement de niveau. C'est-à-dire, on a fixé un motif qui doit être détecté par la carte réceptionniste.

Du côté réceptionner, on a créé une fonction init-irq-interrupt () qui permet d'initialiser la broche à laquelle on va envoyer l'interruption. Pour cela, on utilise la bibliothèque ASF et le pilote qui permet la gestion des interruptions. La fonction spécifie le numéro de broche associé à l'interruption IRQ, la configuration de la broche pour être utilisée en tant que canal d'entrée de l'interruption externe, la résistance de tirage de la broche (pull-up ou pull-down dont on a parlé avant), le critère de détection de l'interruption soit niveau haut ou bas. Après avoir rempli la structure de configuration, la fonction extint-chan-set-config () est appelée pour configurer le canal d'entrée d'interruption externe avec les valeurs de configuration définies dans la structure. Ensuite, une autre fonction extint-register-callback() est utilisée pour enregistrer une fonction de rappel (irq-handler, du paragraphe suivant) qui sera exécutée lorsque l'interruption externe sera détectée sur le canal d'entrée spécifié (dans le nôtre, c'est le canal 6). Cette fonction de rappel doit être définie ailleurs dans le code et sera exécutée de manière asynchrone lorsque l'interruption externe est détectée. Enfin, La fonction extint-chan-enable-callback () est utilisée pour

activer la détection d'interruption sur le canal d'entrée spécifié (canal 6) avec le type de rappel EXTINT-CALLBACK-TYPE-DETECT. Cela signifie que la fonction de rappel irq-handler sera appelée lorsque la condition de détection d'interruption spécifiée dans la structure de configuration (EXTINT-DETECT-RISING) sera remplie sur la broche spécifiée.

La fonction de rappel irq-handler est appelée lorsqu'une interruption externe est détectée sur le canal d'entrée spécifié dans le code, dans notre cas, c'est le canal 6. Cette fonction est généralement utilisée pour effectuer une action spécifique en réponse à l'interruption. Le rôle précis de cette fonction dépend de l'application pour laquelle elle est utilisée. En général, elle est utilisée pour traiter l'événement qui a déclenché l'interruption externe. Cette dernière nous servira beaucoup lorsqu'on envoie les interruptions.

En utilisant ces fonctions, le code est en mesure de configurer le système pour détecter une interruption sur une broche d'E/S spécifique et d'exécuter une fonction de rappel spécifique lorsque l'interruption est détectée. Cela permet au microcontrôleur de surveiller les événements externes tels que les signaux provenant d'autres périphériques (dans notre cas, la deuxième carte), ce qui peut être utilisé pour déclencher des actions ou des tâches spécifiques.

2.4.2 Première utilisation : Moniteur de battement de cœur

Dans ce projet, une autre problématique est posée : Comment on peut détecter un problème matériel au niveau du code principal ? En effet, la carte pilote est en train de communiquer avec le réseau de capteur, et elle est en train de partager les données collectées avec la carte copilote. Cette dernière doit être sûre que la carte pilote fonctionne très bien au niveau matériel. Si elle détecte un problème matériel, elle doit envoyer une interruption à la carte pilote pour qu'elle puisse échanger les rôles. Une solution qu'on a proposée est un moniteur de battement de cœur. Ce dernier permet la surveillance de présence de la carte principale. Cette technique consiste à ajouter une fonctionnalité logicielle sur la carte copilote, qui vérifie que l'autre carte pilote est toujours en vie. Si la carte électronique pilote ne répond pas, cela signifie qu'elle a peut-être planté ou qu'elle ne fonctionne pas correctement. Dans ce cas, la carte copilote doit déclencher une action corrective. Cette action doit être une interruption envoyée à la carte pilote pour qu'elle puisse lâcher la communication et deviennent copilote (inversement de rôle). Pour ceci, on utilise le protocole d'interruption déjà cité. Pour la carte pilote, elle doit envoyer une interruption sécuriser toutes les secondes à la carte copilote, ceci représente son battement de cœur. Pour la carte copilote, il y a un minuteur qui s'incrémente toutes les millisecondes. Si le copilote parvient à détecter l'interruption du pilote dans une durée inférieure à une

seconde, alors dans ce cas, elle initialise ce minuteur à l'aide de la fonction de rappel irq-handler citée dans le paragraphe précédent. Sinon, elle envoie une interruption au pilote pour échanger les rôles.

2.4.3 Deuxième utilisation : Inversion matérielle des rôles

Dans cette partie, nous devons développer un protocole qui permet d'inverser le rôle des deux cartes au niveau de la communication en I2C, c'est-à-dire inverser les rôles de contrôleur et de cible, mais cette fois de manière matérielle et non logicielle. Comme mentionné précédemment, la carte cible, qui est le copilote, doit générer une interruption à la carte ayant le rôle de contrôleur (pilote) lorsqu'une erreur se produit, afin qu'elle puisse arrêter sa communication. À ce moment-là, nous pouvons utiliser ce signal pour que la carte pilote ne soit plus le contrôleur de la communication et se transforme en cible, et vice versa. Pour cela, la carte contrôleur, après avoir reçu l'interruption, libère le bus I2C et l'autre carte prend la main.

3 Résultats et discussion

3.1 Analyse des résultats

Le projet avait pour objectif de mettre en place un réseau de capteurs sans fil sécurisé. Pour ce faire, nous avons développé une boîte à outils qui permet de mettre en place un protocole de communication entre deux cartes embarquées en mode pilote et copilote. Grâce à cette boîte à outils, nous avons pu créer une communication fiable et sécurisée entre les deux cartes, ce qui permet de garantir une transmission de données sécurisée.

Nous avons également commencé le développement d'un driver en RIOT OS pour pouvoir utiliser cette API dans le projet. Le développement de ce driver a été une étape cruciale, car il s'agissait de notre premier obstacle. Nous avons donc consacré beaucoup de temps et d'efforts pour développer un driver fiable qui répondait à nos besoins spécifiques.

En mettant en place un protocole de communication sécurisé et en développant un driver en RIOT OS, nous avons pu garantir que les données collectées par les capteurs étaient transmises de manière fiable et sécurisée. Ceci permet de garantir une surveillance efficace et précise du réseau de capteurs sans fil, en évitant les risques de perte ou de vol de données sensibles.

3.2 Extensions possibles

Le projet vise à faciliter la communication entre deux cartes en créant une boîte à outils qui contient des fonctions pour inverser les rôles, envoyer des données, etc. Le code existant est écrit en RIOT OS, tandis que le nouveau code est écrit à l'aide de la bibliothèque ASF. Il serait intéressant d'approfondir le développement du driver en RIOT OS pour pouvoir l'intégrer dans le code existant. Cependant, il est également possible d'intégrer directement le code développé en ASF, notamment parce qu'il est écrit avec une bibliothèque ouverte qui permet le développement et la modification du code source en fonction des besoins.

En choisissant d'intégrer le code en ASF, il faut prendre en compte les différences entre les deux systèmes d'exploitation et les bibliothèques utilisées. Cela peut nécessiter des ajustements pour assurer une compatibilité et une cohérence entre les deux codes. D'autre part, l'approfondissement du driver en RIOT OS peut nécessiter une certaine expertise dans ce système d'exploitation.

Conclusion

Notre objectif était d'aller le plus loin possible de la mise en place d'un système de supervision du puits d'un réseau en étoile. Cela est passé par la discussion et la proposition de solution techniques durant les nombreuses réunions avec nos tuteurs de projet côté LIMOS, le développement d'une boîte à outils simplifiant l'intégration de la solution technique au sein du projet déjà en place ainsi que l'intégration finale de ce dernier dans le code existant. À la rédaction de ce présent document, nous avons réussi à proposer et valider une solution technique en plus de créer les fondations de la boite à outils. Cependant, nous n'avons pas encore terminé le pilote permettant le fonctionnement du périphérique I2C en mode cible et donc la finalisation de la boite à outils ainsi que son intégration dans le code déjà existant.

Lors de ce projet, nous avons développé nos compétences en gestion de projet ainsi qu'en gestion de réunion où nous avons su mettre en pratique nos connaissances techniques afin de proposer des solutions techniques cohérentes.

De plus, nous avons également amélioré nos compétences en développement dans le langage C pour l'embarqué. Le développement d'un pilote nous a permis de découvrir la manipulation de registres de configuration et la lecture de documentation constructeur, notamment pour le périphérique SERCOM embarqué dans le microcontrôleur de la carte utilisé.

Le projet, ayant plutôt été envisagé par nos tuteurs comme un test de notre avancement, la production d'un programme fonctionnel en fin de projet n'était pas requis. Il reste encore beaucoup de points, particulièrement axés sur le développement logiciel, afin de le finaliser.

Annexe

- **Annexe 1 :** Interconnexion entre un seul contrôleur et une seule cible en I2C.

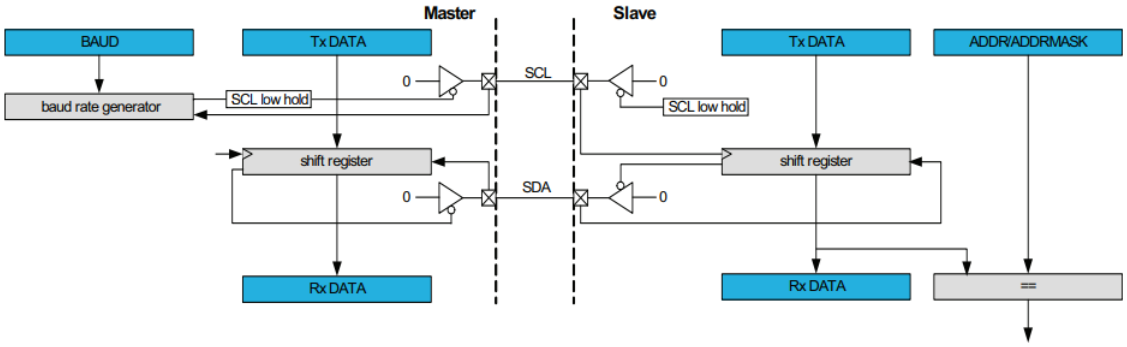


FIGURE 13 – Interconnexion entre un seul contrôleur et une seule cible en I2C

- **Annexe 2 :** Diagramme de transaction de base I2C.

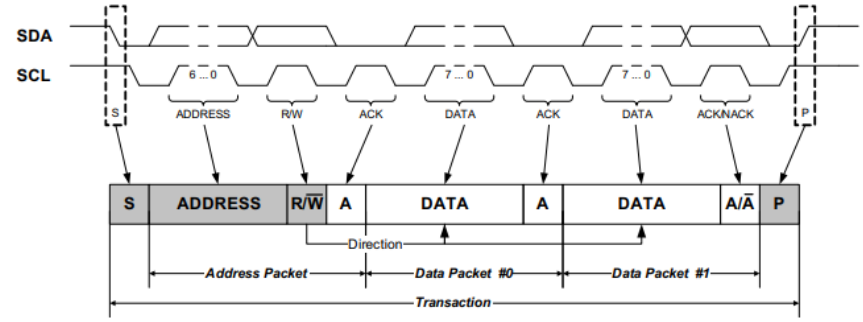


Figure 26-3. Transaction Diagram Syntax

Bus Driver:
- Master Drives Bus
- Slave Drives Bus
- Either Master or Slave Drives Bus

Special Bus Conditions
- S — START Condition
- Sr — Repeated START Condition
- P — STOP Condition

Data Packet Direction:
- R — Master Read — "1"
- W̄ — Master Write — "0"

Acknowledge:
- A — Acknowledge (ACK) — "0"
- Ā — Not Acknowledge (NACK) — "1"

FIGURE 14 – Diagramme de transaction de base I2C.

- **Annexe 3 :** Diagramme d'état de bus I2C.

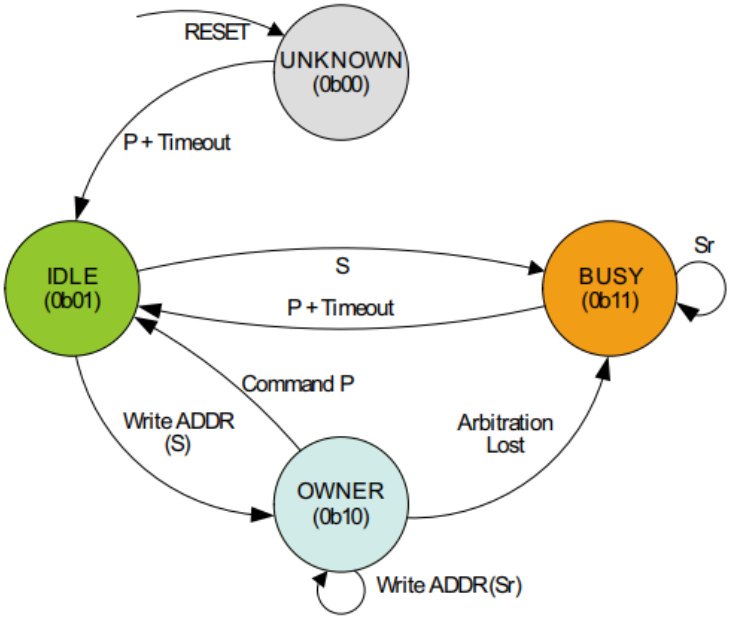


FIGURE 15 – Diagramme d'état de bus I2C

- **Annexe 4 :** Fonctions principales utilisées dans la plupart des algorithmes appliqué dans le projet :

```
1  //Structure à partager entre les deux carte :
2  typedef struct gnrc_waye_conn_table_sector_info {
3          bool found;
4          int8_t rssi_waye_min;
5          int8_t rssi_waye_max;
6          int8_t rssi_hello_min;
7          int8_t rssi_hello_max;
8          uint8_t lqi_waye_min;
9          uint8_t lqi_waye_max;
10         uint8_t lqi_hello_min;
11         uint8_t lqi_hello_max;
12         int8_t calc;
13 } gnrc_waye_conn_table_sector_info_t;
14
15 typedef struct gnrc_waye_conn_table_entry {
16         uint16_t addr;
17         uint8_t best_sector;
18         gnrc_waye_conn_table_sector_info_t sector_info[DATA_LENGTH];
19         struct gnrc_waye_conn_table_entry *next;
20 } gnrc_waye_conn_table_entry_t;
21
22
```

```
23  //Fonction permettant la configuration d'une carte en mode contrôleur :
24  void configure_i2c_master(void)
25  {
26          struct i2c_master_config config_i2c_master;
27      i2c_master_get_config_defaults(&config_i2c_master);
28      config_i2c_master.pinmux_pad0    = PINMUX_PA16D_SERCOM3_PAD0;
29          config_i2c_master.pinmux_pad1    = PINMUX_PA17D_SERCOM3_PAD1;
30      i2c_master_init(&i2c_master_instance, SERCOM3, &config_i2c_master);
31          i2c_master_enable(&i2c_master_instance);
32  }
33
34
35
36  //Fonction permettant la configuration d'une carte en mode cible :
37  void configure_i2c_slave(void)
38  {
39          struct i2c_slave_config config_i2c_slave;
40          i2c_slave_get_config_defaults(&config_i2c_slave);
41          config_i2c_slave.address         = SELF_SLAVE_ADRESS;
42          config_i2c_slave.address_mode    = I2C_SLAVE_ADDRESS_MODE_MASK;
43          config_i2c_slave.pinmux_pad0     = PINMUX_PA16D_SERCOM3_PAD0;
44          config_i2c_slave.pinmux_pad1     = PINMUX_PA17D_SERCOM3_PAD1;
45          config_i2c_slave.buffer_timeout = 1000;
46          i2c_slave_init(&i2c_slave_instance, SERCOM3,&config_i2c_slave);
47          i2c_slave_enable(&i2c_slave_instance);
48  }
49
50
51
52  //Fonction qui permet de calculer le CRC d'un tableau à envoyer:
53   uint8_t calculate_crc(uint8_t* data, uint8_t len)
54  {
55      const uint8_t POLY = 0x07;
56      uint8_t crc = 0x00;
57      uint8_t i, j;
58
59      for (i = 0; i < len; i++) {
60          crc ^= data[i];
61          for (j = 0; j < 8; j++) {
62              if (crc & 0x80) {
63                  crc = (crc << 1) ^ POLY;
64              } else {
65                  crc <<= 1;
66              }
67          }
68      }
69
```

```
70      return crc;
71  }
72
73
74
75  //Fonction permettant l'envoi d'un tableau en I2C :
76  void send_table_over_i2c(uint8_t * buffer) {
77          uint16_t timeout = 0;
78          uint16_t statu = 1;
79          struct i2c_master_packet packet = {
80                  .address = SLAVE_ADDRESS,
81                  .data_length = 200,
82                  .data = buffer,
83                  .ten_bit_address = false,
84                  .high_speed      = false,
85                  .hs_master_code  = 0x0,
86          };
87          if (i2c_master_write_packet_wait(&i2c_master_instance, &packet)
88                  == STATUS_OK) {
89
90                  send_interrupt();
91                  }
92
93  }
94
95
96
97  //Fonction permettant la sérialisation du tableau
98  //de la structure à envoyer en I2C :
99  void serialize_gnrc_waye_conn_table_entry_t(gnrc_waye_conn_table_entry_t
100
101         int i;
102
103         buffer[0] = entry->addr >> 8;
104         buffer[1] = entry->addr & 0xFF;
105         buffer[2] = entry->best_sector;
106
107         for (i = 0; i < TABLE_SIZE; i++) {
108                 buffer[3 + i*10] = entry->sector_info[i].found;
109                 buffer[4 + i*10] = entry->sector_info[i].rssi_waye_min;
110                 buffer[5 + i*10] = entry->sector_info[i].rssi_waye_max;
111                 buffer[6 + i*10] = entry->sector_info[i].rssi_hello_min;
112                 buffer[7 + i*10] = entry->sector_info[i].rssi_hello_max;
113                 buffer[8 + i*10] = entry->sector_info[i].lqi_waye_min;
114                 buffer[9 + i*10] = entry->sector_info[i].lqi_waye_max;
115                 buffer[10 + i*10] = entry->sector_info[i].lqi_hello_min;
116                 buffer[11 + i*10] = entry->sector_info[i].lqi_hello_max;
```

```
117                 buffer[12 + i*10] = entry->sector_info[i].calc;
118         }
119 }
120
121
122
123 //Fonction permettant la désérialisation du tableau
124 //de la structure à reçu en I2C :
125 gnrc_waye_conn_table_entry_t
126                         deserialize_gnrc_waye_conn_table_entry_t(uint8_t *buffer) {
127     gnrc_waye_conn_table_entry_t entry;
128
129     entry.addr = (buffer[0] << 8) | buffer[1];
130     entry.best_sector = buffer[2];
131
132     for (int i = 0; i < TABLE_SIZE; i++) {
133         entry.sector_info[i].found = buffer[3 + i*10];
134         entry.sector_info[i].rssi_waye_min = buffer[4 + i*10];
135         entry.sector_info[i].rssi_waye_max = buffer[5 + i*10];
136         entry.sector_info[i].rssi_hello_min = buffer[6 + i*10];
137         entry.sector_info[i].rssi_hello_max = buffer[7 + i*10];
138         entry.sector_info[i].lqi_waye_min = buffer[8 + i*10];
139         entry.sector_info[i].lqi_waye_max = buffer[9 + i*10];
140         entry.sector_info[i].lqi_hello_min = buffer[10 + i*10];
141         entry.sector_info[i].lqi_hello_max = buffer[11 + i*10];
142         entry.sector_info[i].calc = buffer[12 + i*10];
143     }
144
145     return entry;
146 }
147
148
149 //Code qui permet d'initialiser le Pin de
150 //l'interruption et l'envoi de cette interruption: :
151 void init_irq_pin(void)
152 {
153         struct port_config config_port;
154         port_get_config_defaults(&config_port);
155         config_port.direction = PORT_PIN_DIR_OUTPUT;
156         port_pin_set_config(ITR_PIN_MASTER, &config_port);
157         port_pin_set_output_level(ITR_PIN_MASTER, false);
158 }
159 void send_interrupt(void)
160 {
161         for (uint8_t i = 0; i < 110; i++){
162                 port_pin_set_output_level(ITR_PIN_MASTER, true);
163                 delay_us(50);
```

```
164                 port_pin_set_output_level(ITR_PIN_MASTER, false);
165                 delay_us(50);
166         }
167 }
168
169 //Fonction qui permet d'initialiser la broche à
170 //laquelle on va reçevoir l'interruption:
171 void init_irq_interrupt(void)
172 {
173     struct extint_chan_conf config_extint_chan;
174     extint_chan_get_config_defaults(&config_extint_chan);
175     config_extint_chan.gpio_pin           = IRQ_PIN;
176     config_extint_chan.gpio_pin_mux       = MUX_PA22A_EIC_EXTINT6;
177     config_extint_chan.gpio_pin_pull      = EXTINT_PULL_NONE;
178     config_extint_chan.detection_criteria = EXTINT_DETECT_RISING;
179         config_extint_chan.filter_input_signal = true;
180     extint_chan_set_config(6, &config_extint_chan);
181     extint_register_callback(irq_handler, 6, EXTINT_CALLBACK_TYPE_DETECT);
182     extint_chan_enable_callback(6, EXTINT_CALLBACK_TYPE_DETECT);
183 }
184
185
186
187 //fonction qui permet de reçevoir une interruption et
188 //de réagir selon le cas d'utilisation
189 //(Dans ce cas il initialise le miniteur de
190 //l'algorithme "battement de cœur"):
191 void irq_handler(void)
192 {
193         irq_count++;
194         if (irq_count>=100){
195                 irq_count = 0;
196                 DELAY = 0;
197                 }
198 }
```

Listing 1