

Banc de test logiciel pour pile OCARI



Calypso Barnes
Encadrée par
Jean-Marie Cottin

Du 2 avril au 30 août 2013

Remerciements

Je voudrais consacrer cette page à toutes les personnes qui m'ont soutenue au cours de ce stage.

J'aimerais particulièrement remercier mon maître de stage, Jean-Marie COTTIN, pour sa disponibilité, le savoir qu'il m'a transmis, la patience dont il a fait preuve à mon égard ainsi que sa sympathie et ses encouragements.

J'aimerais également remercier Tuan DANG, chef de projet à EDF R&D, pour son aide, ses conseils, et sa gentillesse, ainsi que Dimitri RZEPSKI, chef du groupe Contrôle Commande d'EDF R&D, pour m'avoir accueillie au sein de son groupe.

Mes remerciements vont également à tous mes professeurs de l'université de Nice-Sophia Antipolis qui m'ont tant appris. Je remercie notamment Cécile BELLEUDY, coordinatrice du Master, Jean-Marc RIBERO, directeur du département d'électronique, Jean-Pierre FOLCHER, sans qui je ne serais pas arrivée jusqu'au Master et Daniel GAFFE, qui m'a enseigné la programmation.

Je tiens également à remercier Valérie CONTESSO et Brigitte TORREGROSSA pour leur amabilité et leur soutien dans toutes les procédures administratives du stage.

Table des matières

1 Introduction	4
2 Présentation de l'entreprise	5
2.1 Le groupe EDF	5
2.2 EDF R&D	6
2.3 STEP	7
2.4 Le groupe P1A : « Contrôle Commande »	7
3 Contexte du stage : le projet OCARI.....	8
3.1 Présentation du projet	8
3.2 EOLSR	9
3.3 OSERENA.....	10
4 L'objectif du stage.....	11
4.1 L'intérêt du simulateur.....	11
4.2 Modélisation d'une plateforme hardware.....	12
4.3 Modélisation de l'interaction entre les capteurs	13
4.4 Un programme en licence libre...	14
5 Travail réalisé pendant le stage.....	15
5.1 Sélection d'un simulateur pour le MCS-51.....	15
5.2 La programmation du modèle du CC2530	17
5.2.1 Ajout dans le simulateur des mémoires spécifiques au CC2530.....	20
5.2.2 Programmation des périphériques.....	23
5.3 Interfaçage entre deux simulateurs	37
5.4 Bilan	38
6 Perspectives.....	39
Bibliographie	40
Annexes	42



1 Introduction

D'après le magazine Technology Review de MIT, le réseau de capteurs sans fil est l'une des dix nouvelles technologies qui bouleverseront le monde ainsi que notre manière de vivre et de travailler. Il répond à l'émergence ces dernières décennies de l'offre et d'un besoin accru d'observation et de contrôler des phénomènes physiques et biologiques de nombreux domaines (industriel, technique, domotique, sécurité, transports...).

Dans ce rapport sera présenté un stage de fin d'études effectué au sein d'EDF R&D à Chatou sur une durée de 5 mois. L'objectif de ce stage est de mener une étude de faisabilité de la méthode de validation d'un protocole, en prenant pour objet d'étude la couche MaCARI qui fait partie du protocole de communication OCARI pour les réseaux de capteurs sans fil (standard IEEE 802.15.4). OCARI est développé par EDF R&D et ses partenaires académiques depuis 2007, et rentre actuellement en phase préindustrielle.



Figure 1 - Le CC2530 de Texas Instruments, module RF sur lequel est portée la pile OCARI

2 Présentation de l'entreprise



2.1 Le Groupe EDF

Le groupe EDF est un des leaders sur le marché de l'énergie en Europe et a une dimension mondiale. Premier producteur d'électricité en Europe, il dispose en France de moyens de production essentiellement nucléaires et hydrauliques fournissant à 95% une électricité sans émission de CO₂.

L'ambition du groupe EDF est de devenir le premier électricien dans le monde. Son objectif est de fournir, dans tous les pays où EDF est implanté, une électricité sûre et respectueuse de l'environnement à des prix abordables, mais aussi de dynamiser une stratégie internationale axée sur les pays en forte croissance.

1er producteur mondial d'électricité nucléaire, 1er producteur hydroélectrique européen, EDF est leader dans les énergies décarbonées, qui est l'un des axes majeurs de sa stratégie de développement. Les énergies nouvelles, comme l'éolien, le solaire photovoltaïque, la biomasse..., occupent en outre une part croissante dans le mix énergétique du Groupe, notamment via sa filiale EDF Energies Nouvelles. EDF est aussi un acteur majeur de la production électrique à partir d'énergies fossiles. Les savoir-faire reconnus d'EDF s'étendent à la conception, la construction, la maintenance et la réhabilitation ou la déconstruction de tous les types d'unités de production, mais également à l'ingénierie de projets complexes à l'image des 1ers réacteurs EPR en construction en France et en Chine, ou encore du grand ouvrage hydroélectrique de Nam Theun, au Laos.

Chiffres clés

- **Production : 630,4 TWh produits dans le monde**
- **Clients : 37 millions de clients dans le monde dont 28 millions en France.**
- **Salariés : 161 560 salariés dans le monde.**
- **Chiffre d'affaires : 65,2 milliards d'euros dont 44,5% hors de France**

Dates historiques

- **8 avril 1946 : Création d'EDF avec le statut d'entreprise publique à caractère industriel et commercial (EPIC)**
- **9 août 2004 : Transformation de l'EPIC en société anonyme (SA).**
- **1^{er} juillet 2007 : Ouverture totale à la concurrence.**

2.2 EDF R&D

La R&D (Recherche et Développement) a été créée dès 1946 sous le nom de la direction des Etudes et Recherche. Sur le plan mondial, elle se situe au premier rang des centres de recherche industriels. Elle bénéficie d'une expertise technologique de référence en matière de modélisation et de simulation numérique. EDF R&D est au service de la performance du Groupe EDF.



Figure 2 - La R&D en chiffres

Les missions d'EDF R&D se situent à différentes échelles de temps du court terme, moyen terme et long terme :

- Contribuer à la performance des unités opérationnelles,
- Réaliser des études, développer des méthodes et des outils pour les différentes branches et entités du Groupe,
- Eclairer l'avenir et préparer les relais de croissance du groupe EDF.

La R&D mobilise ses compétences pour appuyer les entités opérationnelles dans tous les domaines d'activités du Groupe :

- La production d'électricité par les filières classiques : nucléaire, thermique à flamme, hydraulique ;
- Le management d'énergie, qui s'attache notamment à garantir à tout instant l'équilibre entre production et consommation ;
- Les réseaux de transport et de distribution d'électricité ;
- Le développement commercial : marketing et vente d'électricité, de gaz et de services liés à ces énergies ;
- Les énergies renouvelables (solaires, éolien, biomasse, etc.) et l'environnement.

2.3 Le département STEP

STEP (Simulation et Traitement de l'information pour l'Exploitation des systèmes de Production) aide l'exploitant en charge des moyens de production du Groupe (nucléaire, hydraulique, thermique et EnR). Il accompagne dans la conduite, la surveillance et le maintien des installations existantes et à venir, en prenant en compte le cadre réglementaire évolutif en termes de sûreté et d'environnement. Ses activités visent à mettre à disposition de l'exploitant et de l'ingénierie support, des outils et méthodes qui permettent d'améliorer la performance des moyens de production.

Les groupes du département STEP

Code	Sigle	Nom	Effectif	Agents	Stagiaires
P10		Membres du groupe P10	12	11	1
P1A	CC	Contrôle Commande	32	27	5
P1B	SDTI	Systèmes Dynamiques et Traitement de l'Information	39	26	13
P1C	FC	Fonctionnement et Conduite	30	25	5
P1D	SIS	Systèmes d'Information et de Surveillance	26	21	5
P1E	MPR	Mesures Physiques et Radioprotection	29	24	5
TOTAL			168	134	34

2.4 Le groupe P1A : « Contrôle Commande »

Les compétences du groupe Contrôle Commande sont les suivantes :

- Sûreté de fonctionnement des systèmes programmés
- Systèmes de Contrôle Commande
- Informatique industrielle

La mission du groupe P1A est d'être un centre d'expertise sur les architectures, les technologies, les méthodes et outils d'ingénierie, de vérification et qualification des systèmes de Contrôle Commande des centrales de production d'électricité.

Le groupe P1A est également fournisseur de solutions en matière de services d'information en temps réel aux acteurs pour la conduite et la supervision des procédés.

Les objectifs techniques clés du groupe sont les suivants :

- Contribuer à la durée de vie des systèmes de Contrôle Commande en mettant au point des méthodes et techniques de rénovation partielles
- Réussir l'EPR en appui au CNEN
- Proposer une démarche outillée pour la qualification fonctionnelle de composants électroniques programmée critiques.
- Concevoir les architectures pour les salles de commandes de demain.
- Concilier l'optimisation globale du parc de production diversifié et réparti avec la gestion locale des moyens de production et spécifiés associés.
- Promouvoir l'usage de standards dans le domaine de l'I&C. Identifier les standards de demain et favoriser leur émergence



figure 3 - Le site de la R&D à Chatou

3 Contexte du stage : le projet OCARI

3.1 Présentation du projet

Le stage a été réalisé dans le cadre du projet OCARI « Optimisation des Communications Ad hoc pour les Réseaux Industriels ». Ce projet dirigé par EDF a pour objectif de mettre en œuvre des équipements et des logiciels permettant le déploiement de réseaux maillés de capteurs sans fil à bas coût dans les environnements industriels. OCARI est développé depuis 2007 en partenariat avec la DCNS, l'INRIA, LATTIS, LIMOS, LRI, et Telit-RF Technologies.



Les réseaux de capteurs sans fil sont un type particulier de réseaux ad hoc sans fil principalement conçus pour acheminer des données de capteurs à l'aide de technique de communication numérique radio fréquence (RF). Ils sont soumis à plusieurs contraintes :

- Le fonctionnement sur pile ou batterie pour une longue durée.
- La quantité limitée d'espace mémoire et de stockage sur chaque nœud.
- Des ressources de traitement limitées sur chaque appareil.
- Une bande passante limitée.

Le premier objectif du projet OCARI est l'autonomie. Les capteurs de réseaux utilisant le protocole ZigBee ont une autonomie limitée à quelques jours car ils sont actifs en permanence. L'objectif d'OCARI est d'étendre considérablement la durée de vie du réseau sur batterie à plusieurs mois ou plusieurs années, tout en conservant la robustesse : le réseau se reconfigure automatiquement et rapidement lors de la perte, de l'introduction ou de la réapparition d'un nœud. Un autre objectif qui est le corollaire de la robustesse est la mobilité. Un nœud collecteur de données peut bouger, figurant un intervenant humain traversant une installation.

Le projet OCARI tente de proposer des mécanismes pour atteindre ces objectifs :

- MaCARI, une méthode d'accès déterministe au medium RF
- EOLSR, une stratégie de routage proactive à basse consommation
- OSERENA, un algorithme d'ordonnancement des activités basé sur la coloration des nœuds

Ces mécanismes sont expliqués brièvement dans les parties 3.2 et 3.3. Une représentation de l'architecture de la pile logicielle OCARI peut être trouvée en annexe.

3.2 EOLSR

EOLSR («Energy efficient Optimized Link State Routing») est le protocole de routage utilisé dans la pile protocolaire OCARI. Il construit un arbre de routage qui prend ses racines dans le puits de collecte, où les routes d'énergie ont le coût énergétique le plus faible et sont construites à partir de routeurs à plus haute énergie. EOLSR comprend deux modules :

- La découverte du voisinage : EOLSR permet à chaque nœud de découvrir ses voisins à travers l'échange de messages « Hello ».
- La construction des routes : Etant donné un puits de collecte, EOLSR construit un arbre à basse consommation qui prend racine dans ce nœud.

EOLSR est une extension du protocole de routage OLSR. Cependant il a été adapté aux applications de collecte de données où les capteurs collectent des données et les transmettent à un nœud appelé puits de collecte ou nœud stratégique. Ce nœud stratégique utilise un arbre de collecte de données qui prend racine dans lui-même pour récupérer les données des capteurs. La racine de l'arbre est appelé le CPAN.

EOLSR garde des principes d'OLSR mais ajoute des simplifications concernant les données maintenues et les règles de traitement :

- Pour augmenter la mise à l'échelle, tous les nœuds ne stockent pas leurs voisins à 2 sauts. Par conséquent, le routage n'est plus basé sur les relais MultiPoints.
- La route sélectionnée est celle ayant un coût énergétique minimum et qui évite les routeurs à basse énergie résiduelle.
- Etant donné que maintenir et échanger une route avec chaque autre nœud du réseau coûte cher en termes de stockage de données, de bande passante et d'énergie, EOLSR ne maintient sur chaque nœud qu'une route par puits.
- Le format du message utilisé est simplifié. Fusionner plus d'un message n'est plus possible. Par conséquent, l'en-tête de ces messages est simplifié.

Un nouveau message est introduit (message du statut de l'arbre). Ce message est routé vers la racine de l'arbre pour l'en informer de la stabilité des liens radio. Si le lien est stable, la racine peut déclencher la coloration des nœuds par OSERENA. Une option backup est aussi introduite : dans le cas général, tout nœud maintient un parent, qui est le prochain saut pour atteindre la racine de l'arbre. Si l'option backup est utilisée, chaque nœuds doit aussi maintenir un backup de son parent. Cela accélère la réparation de la route quand le lien avec le parent ne peut plus être utilisé.

3.3 OSERENA

OSERENA (« Optimized SchEduling of RoutEr Node Activity») est un algorithme qui ordonne l'activité des nœuds. Pour chaque nœud de capteur, il définit des périodes d'activité pour la transmission et la réception de données, ainsi que des périodes d'inactivité durant lesquelles la radio est éteinte pour économiser la batterie. Cet algorithme est basé sur la coloration des nœuds. Le défi d'OSERENA est de s'adapter aux ressources limitées dans les réseaux de capteurs sans fil. Les capteurs ont une faible capacité de stockage et une bande passante limitée. C'est pourquoi l'objectif a été de réduire la taille des données stockées et des messages échangés pour faire la coloration.

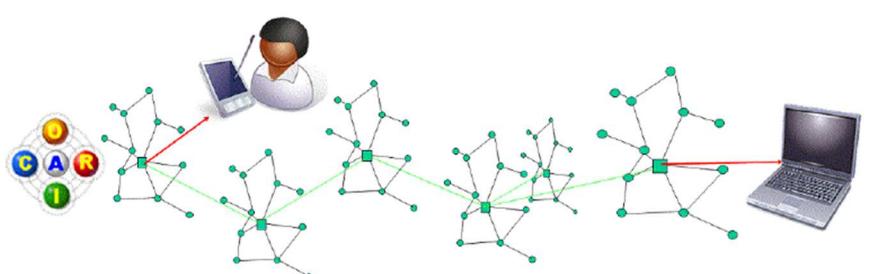
L'objectif de la coloration des nœuds est d'assigner une couleur à chaque nœud de capteur du réseau tel que deux nœuds n'aient la même couleur que s'ils ne peuvent interférer et que le nombre total de couleurs utilisées soit minimisé.

OSERENA est utilisé sur un médium d'accès à slots, basé sur un cycle composé de time slots. Chaque couleur est associé à un time slot.

- Pendant ce time slot, tous les nœuds ayant la couleur associée peuvent transmettre des données.
- Tout nœud est réveillé pendant ses time slots pour transmettre des données, et pendant les time slots de ses voisins pour pouvoir recevoir les données qui lui sont destinées.
- Le nœud peut dormir (éteindre sa radio) le reste du temps.

Cet ordonnancement offre les bénéfices suivants :

- Un gain de bande passante grâce à la réutilisation spatiale de la RF : les nœuds de mêmes couleurs peuvent transmettre simultanément. De plus, aucune bande passante n'est perdue à cause des retransmissions après collisions.
- Un gain en énergie puisque un nœud peut dormir pendant les time slots où ni lui ni ses voisins à un saut ne transmettent. Aucune énergie n'est perdue dans les retransmissions.
- Un gain dans les délais de collecte de données grâce à un coloriage intelligent adapté aux applications de collecte de données.



4 Objectif du stage

4.1 L'intérêt du simulateur

Le protocole de communication OCARI rentre actuellement en phase préindustrielle et a donc besoin d'être validé.

L'augmentation de la complexité des protocoles de communication des réseaux de capteurs sans fil et la variabilité des environnements industriels dans lesquels ils sont déployés font que la simulation en laboratoire ne suffit plus pour étudier de manière réaliste le comportement du protocole et du réseau. Les méthodes classiques de validation reposent sur une mise en situation réelle de plateformes RF embarquant le logiciel et en l'application d'un protocole de test fastidieux. Ces opérations de validation du produit final sont nécessaires mais elles sont très coûteuses, longues, n'offrent que peu de visibilités de la trace de chacun des nœuds, et certains bugs ne peuvent pas être reproduits à cause de leur nature non déterministe. Les trames échangées peuvent être observées grâce à des sniffers, mais quand un problème est détecté dans les échanges, n'ayant aucune visibilité de l'exécution du code sur la plateforme matérielle, il est très difficile de remonter à la source de ce problème. La validation du protocole devrait être facilitée par une validation du logiciel avant intégration sur la plateforme matérielle.

C'est pour cela qu'EDF R&D souhaiterait déboguer la pile logicielle d'OCARI, couche par couche, sur un simulateur. Le fait de faire fonctionner un protocole de communication sur une plateforme matérielle simulée permettrait d'explorer un nombre de scénarios plus grand que ceux effectivement accessibles sur une plateforme matérielle donnée. Ce simulateur permettrait également une observation du comportement du logiciel OCARI afin d'en valider non seulement les fonctionnalités mais aussi d'en effectuer le profilage du code source (quelles fonctions sont le plus sollicitées ou consomment le plus de ressources CPU) et d'en optimiser les performances, en optimisant le code source de ces fonctions. L'objectif du stage est de mener une étude de faisabilité de la méthode de validation d'un protocole.

4.2 Modélisation d'une plateforme hardware

Le code OCARI est actuellement porté sur la plateforme CC2530 de Texas Instruments. L'objectif est donc de faire tourner le binaire du code OCARI sur un simulateur ayant les mêmes caractéristiques et imitant le comportement du CC2530.

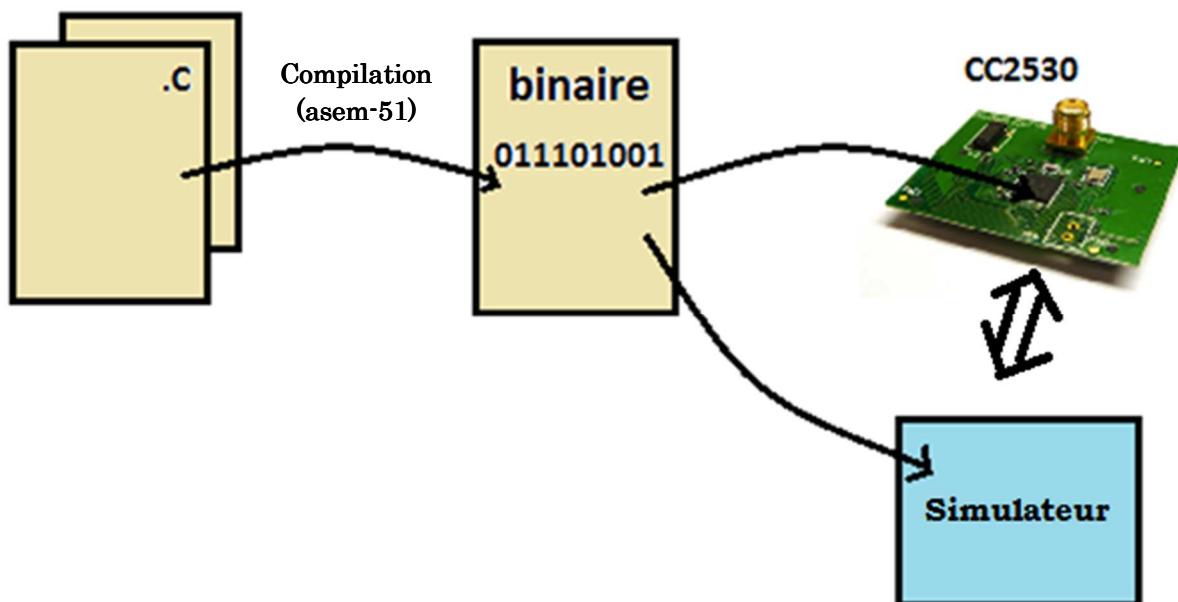


Figure 4 - Concept du simulateur du CC2530

Le CC2530 est un SoC (system-on-chip) utilisé pour les implémentations du standard IEEE 802.15.4 (LR-WPAN). Il permet la construction d'un réseau robuste avec un coût matériel très réduit. Il possède aussi différents modes d'opération, le rendant très adapté pour des systèmes qui requièrent une consommation d'énergie très faible, comme OCARI.

Le CPU (Central Processing Unit) utilisé dans le CC2530 est le MCS-51 d'Intel. Chargé de l'exécution des instructions des programmes, le CPU ou microprocesseur est l'élément principal du SoC. C'est donc la partie qu'il a fallu modéliser en premier en recherchant un simulateur préexistant en licence libre. Ensuite, pour modéliser le SoC complet, il a fallu coder les blocs autour du CPU (mémoires, timers...) qui constituent le CC2530 (voir le diagramme en bloc du CC2530 en annexe).

Une fois la plateforme modélisée, elle exécute le programme d'OCARI, simule sur la sortie radio les trames qui seraient transmises par la plateforme matérielle, ainsi que l'entrée radio pour recevoir des trames d'autres simulateurs CC2530.

4.3 Modélisation de l'interaction entre les capteurs

Une fois le simulateur du CC2530 terminé, pour pouvoir observer le bon fonctionnement d'OCARI, Il faut le connecter via un lien radio simulé à une deuxième instance du même simulateur. Un simulateur pourrait exécuter le code OCARI d'un nœud coordinateur, alors que l'autre pourrait exécuter le code OCARI pour un nœud routeur. Les deux simulateurs de CC2530 pourraient alors échanger des trames.

Un oracle (observateur) devra être implémenté pour observer les trames échangées entre les deux capteurs, et ainsi pouvoir vérifier si ce sont bien celles attendues.

L'atout du modèle logiciel du simulateur est aussi de pouvoir déboguer le programme grâce à GDB (GNU Debugger) qui permet de suivre et altérer l'exécution du programme. Si l'oracle détecte un problème sur les trames échangées, GDB permettra d'en découvrir l'origine.

Le simulateur principal doit être relié à un simulateur de réseau de haut niveau, qui ne contiendrait non pas un grand nombre d'implémentations du simulateur, mais qui serait chargé d'envoyer des messages au simulateur du CC2530, en simulant ainsi des trames reçues d'un grand réseau, incluant aussi des trames qui ne lui seraient pas destinés. Le simulateur du CC2530 pourrait lui aussi envoyer des trames vers le reste du réseau.

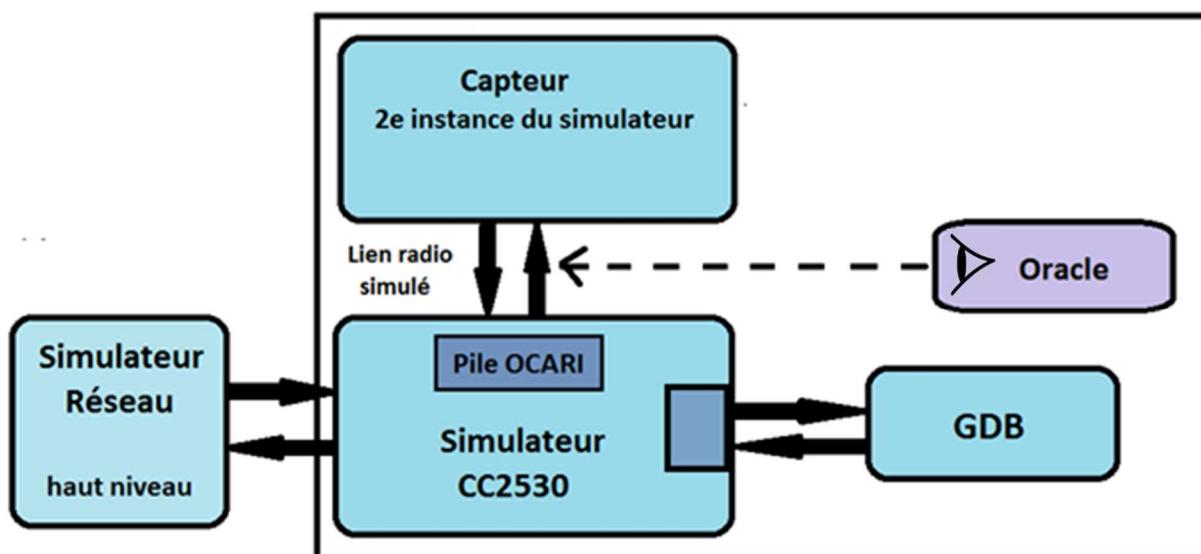


Figure 5 - Concept du simulateur de réseau

4.4 Un Programme en licence libre...

Mon stage a été effectué dans l'idée que le simulateur du réseau de capteurs sans fil soit en licence libre.

Ucsim, le programme utilisé pendant le stage est protégé par la licence GNU GPL, ce qui signifie que ce programme a été obtenu gratuitement, qu'il est possible de le modifier, et qu'il pourra être redistribué gratuitement ensuite.

La Licence Générale Publique de GNU (GPL) est conçue pour protéger des logiciels libres. Elle donne la liberté de copier ou d'adapter les programmes qui possèdent cette licence. Chaque personne qui possède une copie est libre de modifier cette copie et de la redistribuer. Les copies redistribuées sont aussi bien sûr protégées par la GPL. C'est pour cela que cette licence est dite contaminante.

5 Travail réalisé pendant le stage

5.1 Sélection d'un simulateur pour le MCS-51

La première partie du stage à consisté à trouver un simulateur pour le microcontrôleur du CC2530. Le 8051 est un microcontrôleur 8 bits qui a été développé par Intel et autour duquel, par la suite, ont été conçus d'autres microcontrôleurs qui sont dits de la famille MCS-51. Cette famille est l'une des plus prolifiques parmi celles des microcontrôleurs.

Plusieurs simulateurs ont donc été testés pour ce microcontrôleur, avec la contrainte que le simulateur soit en licence libre, car le simulateur du CC2530 est supposé l'être aussi.

OVPsim :

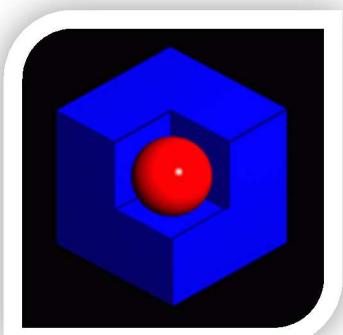
OVPsim offre une infrastructure qui permet de décrire des plateformes avec un ou plusieurs processeurs qui contiennent des mémoires partagées, des bus à topologie arbitraire et des modèles périphériques. Les plateformes sont créées en utilisant une simple API (Application Programming Interface) qui spécifie comment des composants hardware devraient interagir entre eux. OVPsim possède des modèles pour de nombreux processeurs standards dont le 8051, mais aussi du microprocesseur Arm Cortex M3 qui a aussi été considéré pour la simulation. En effet, le code d'OCARI sera prochainement porté sur une plateforme Atmel possédant un Cortex M3 pour laquelle il faudra également concevoir un simulateur logiciel.



OVPsim a donc été téléchargé et installé, et son code source a été étudié. Malheureusement, il s'avère qu'OVPsim n'est pas entièrement libre, et certaines parties des programmes nécessitent une licence payante pour être accédés et modifiés. OVPsim était donc inutilisable pour ce stage.

MIDE-51 :

MIDE-51 est une IDE (Integrated Development Environment), autrement dit une suite de programmes servant au développement de logiciels et destinés à être utilisés ensemble. MIDE-51 simule le comportement du 8051.



Il compile des fichiers en assembleur et permet de les tester et de les déboguer.

Après avoir téléchargé et installé MIDE-51, quelques problèmes ont été rencontrés lors du tutoriel. Le fichier fourni pour tester le studio MIDE-51 ne fonctionnait pas, il y avait un bug dans MIDE-51 dont l'origine n'a pu être déterminée. Il était donc préférable de chercher un autre simulateur.

QEMU :

QEMU est un simulateur qui fonctionne par compilation du code émulé vers le code natif de la machine sur laquelle QEMU tourne (du x86 en l'occurrence). La prise en main et la modification de ce type de simulateur est assez délicate : le code est complexe et plus difficile à déboguer. QEMU sera gardé pour l'émulation du processeur ARM Cortex-M3 sur lequel sera porté prochainement le code d'OCARI.



UCSIM :

Ucsim est un simulateur logiciel pour les microcontrôleurs de la famille des MCS-51. C'est un logiciel libre sous la licence GNU GPL. Il interprète les instructions à partir d'un fichier binaire de format Intel Hex (.hex). Le CPU simulé possède des espaces mémoire, le stockage est simulé par des Chips de mémoire, et les décodeurs d'adresse les connectent entre eux. Le simulateur peut être contrôlé par une interface en lignes de commande acceptant des commandes simples.



Ucsim est le logiciel qui a été finalement choisi pour le simulateur du CC2530. Il correspondait au critère du logiciel libre, les tests fonctionnaient et le code source semblait moins complexe que ceux des autres simulateurs testés.

```
calypso@calybuntu:~$ s51 /home/calypso/Desktop/CC2530/cc2530test1.hex
uCsim 0.5.4, Copyright (C) 1997 Daniel Drotos, Talker Bt.
uCsim comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
264 words read from /home/calypso/Desktop/CC2530/cc2530test1.hex
0> n
0x00 00 1c ad d6 fc 30 67 5b ....0g[
000000 00 . ACC= 0x00 0 . B= 0x00 DPTR= 0x0000 @DPTR= 0x00 0 .
00001c d8 . PSW= 0x00 CY=0 AC=0 OV=0 P=0
    0x0003 75 aa 02 MOV aa,#02
0> [REDACTED]
```

Figure 6 - Lancement du programme Ucsim original dans la console

5.2 La programmation d'un modèle du CC2530

Après avoir installé Ucsim, il a fallu étudier le fonctionnement du code source écrit en C++. Le User's Guide du CC2530 de Texas Instruments a servi de base pour la programmation du modèle, il peut être téléchargé en suivant ce lien :

www.ti.com/general/docs/lit/getliterature.tsp?literatureNumber=swru191d& fileType=pdf.

Il décrit en détail le fonctionnement de tous ses composants. Les outils utilisés pour la programmation du simulateur sont décrits en annexe.

Le « main » du code source

Le « main » du programme instancie la classe du simulateur et la classe de l'application. A l'initialisation du simulateur, la fonction *mk_controller* est appelée. Cette fonction instancie le CPU (*cl_51core*) et tous les périphériques du CC2530 (instanciés dans le *cl_51core*). A l'initialisation de l'application, un traitement des arguments du main est effectué, puis le set de commandes et un nouveau *commander* sont instanciés puis initialisés pour permettre une interface qui permettra à l'utilisateur de contrôler le simulateur. Le « main » appelle ensuite la fonction *application->run()*, ce qui fera entrer le programme dans une boucle *while* dont il ne sortira que lorsqu'il recevra un signal d'arrêt de la simulation (comme la commande *quit* donnée par l'utilisateur). Dans cette boucle *while*, le *commander* attend une entrée de l'utilisateur pour la traiter ensuite. Par exemple, après la commande *next*, le simulateur traitera la prochaine instruction en mémoire avant d'attendre la prochaine entrée de l'utilisateur. Si la commande *run* est entrée, le simulateur traitera toutes les instructions à la suite sans ne plus attendre une entrée.

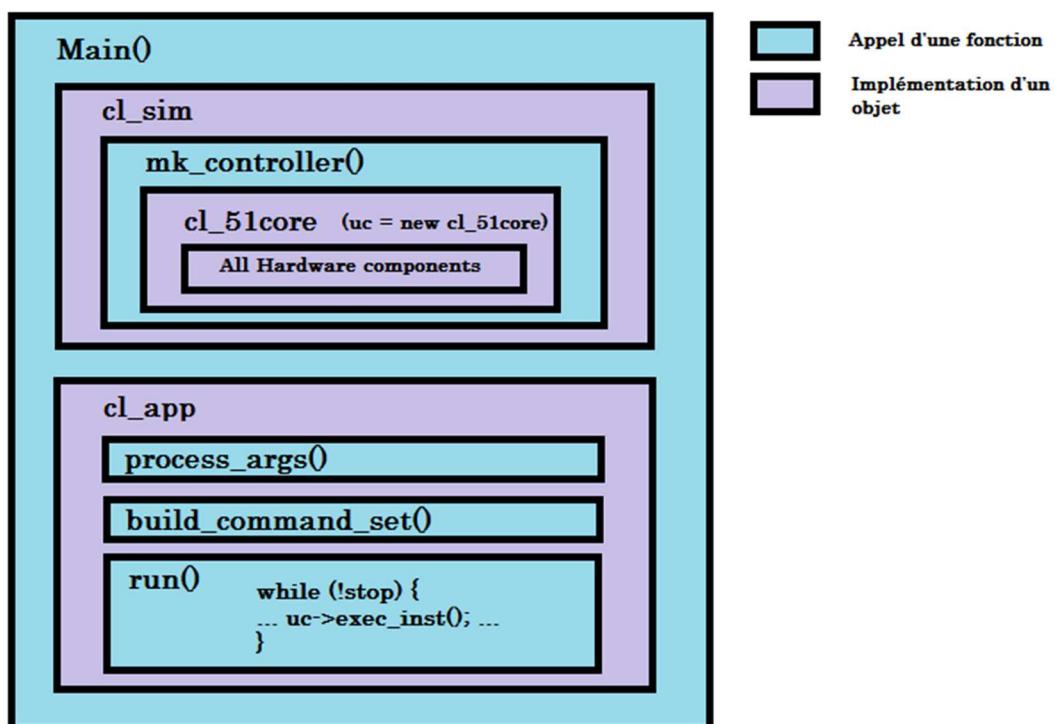


Figure 7 - Explication schématique du fonctionnement d'Ucsim

La simulation du temps écoulé

Le temps que prend le simulateur à exécuter les instructions du programme OCARI ne correspond évidemment pas au temps réel que met la plateforme matérielle pour exécuter les mêmes instructions. La vitesse que met le simulateur à interpréter les instructions d'OCARI dépend de la vitesse du CPU de la machine sur laquelle tourne le simulateur, ainsi que de du code source du simulateur.

Connaître le temps réel écoulé est très important pour la validation d'OCARI, car il faut vérifier que les délais sont bien respectés, et que certaines fonctions ne prennent pas un temps réel anormalement long à s'exécuter.

Afin de simuler le temps écoulé, le simulateur se base sur les instructions exécutées : connaissant le nombre de cycle que prennent les instructions et la fréquence du système, le simulateur peut en déduire le temps écoulé depuis le dernier Reset. Lorsque la fréquence du système est modifiée, le simulateur retient le temps au moment du changement, puis recommence à compter les cycles à 0. Pour connaître le temps écoulé, il lui suffit d'ajouter le temps mémorisé en mémoire au temps écoulé depuis le changement de fréquence (nombre de cycles comptés / fréquence).

Tests du bon fonctionnement du simulateur

Le simulateur interprète les commandes à partir d'un fichier en format .hex. Pour tester le bon fonctionnement des composants hardware qui ont été programmés pour le CC2530, des programmes ont été écrits en assembleur pour le MCS-51 en format .a51. Ceux-ci sont ensuite convertis par le logiciel Asem-51 en format .hex. Le fichier .hex est passé en argument lors du lancement d'Ucsim et les instructions en hexadécimal sont chargées dans la mémoire flash du simulateur. Le CPU simulé interprète ensuite ces instructions et les exécute.

A l'exécution de la simulation, il est possible de vérifier que le fonctionnement est correct en observant les mémoires, ou grâce aux fonctions printf() insérées dans le code du simulateur lui-même, qui permettent d'afficher des informations dans la console et de tracer la simulation.

```

cc2530test1.a51 ✘
$INCLUDE (mcu/CC2530.mcu) ;include CC2530 SFR symbol def
MOV IEN0, #080H
MOV IEN1, #02H
MOV T1CC0H, #03H

MOV DPTR, #6100H ; Radio cell test
MOV A, #83H
MOVX @DPTR, A ;

MOV DPTR, #6101H ; Radio cell test
MOV A, #7DH
MOVX @DPTR, A ;

MOV DPTR, #6102H ; Radio cell test
MOV A, #50H
MOVX @DPTR, A ;

MOV MCON, #00H
MOV T1CTL, #02H ; Up/Down mode to T1CC0 and Tick Freq /1
MOV T1CCTL0, #04H; a)

cc2530test1.hex ✘
:1000000075A88075AA0275DB039061007483F09077
:100010006101747DF09061027450F075C70075E461
:100020000275E50475E60375DA0475DB0075D50025
:1000300075D48075D30075D29075D60375D70275C7
:10004000C41A7586C075C1CC75CC0475EE0475DC18
:100050001275DD0075CB5275CD0375ED0975EB1189
:1000600075EF0675C30075A2F575A3FF759C417405
:100070000CF275C33275A20975A30075A40375C788
:1000800008906180E09062727401F0906271740077
:10009000F09062707402F0906273740CF0F0740D62
:1000A000F0F0F0F0740CF0F0F0740EF0F0F0F00E
:1000B000740FF0F0F0F0740EF0F0F0F0740FF0F058
:1000C000F0F0740AF0F0F0F0740BF0F0F0F0740F50
:1000D000F0F090618175ED0375EB327402F0788376
:1000E000E2741175C60474117411741174117411D1
:1000F0007411741174117411741174117411D8
:080100007411E2E2E275D6037E b)
:00000001FF c)

```

Figure 8 - a) Programme test en assembleur ; b) Programme test converti en format hexadécimal; c) Vue du programme chargée dans la mémoire flash du simulateur

5.2.1 Ajout dans le simulateur des mémoires spécifiques au CC2530

Certaines mémoires étaient déjà programmées dans Ucsim. Les mémoires sont des instanciations d'une classe *cl_address_space* qui crée de nouveaux objets *cl_memory_cell*, qui sont des cellules mémoires. L'objet *cl_address_space* possède un tableau de pointeurs, chaque pointeur pointe vers une cellule mémoire qui possède une donnée sur 8 bits. L'index du pointeur dans le tableau correspond à l'adresse de la cellule en mémoire.

Les cellules de mémoire peuvent être classées en tant que cellules surveillées. Dans ce cas, si le programme accède aux données de la cellule en lecture ou en écriture, les composants hardware en sont immédiatement informés : les composants qui sont enregistrés en tant que surveillant cette cellule voient leurs fonctions *read* ou *write* appelées en fonction de l'accès.

Le contrôleur d'interruptions

Le contrôleur d'interruptions sert un total de 18 sources d'interruptions, divisées en six groupes, chacun desquels est associé à une des quatre priorités d'interruption.

Un mécanisme de contrôle des interruptions était déjà programmé dans Ucsim, il a suffi de rajouter les interruptions provenant des différents composants du CC2530, en faisant attention aux priorités, et en les classant par groupes. Comme il y a 18 sources d'interruptions, il a fallu complexifier un peu les fonctions qui leur sont liées, en raison d'un plus grand nombre de registres SFR à prendre en compte. Les interruptions sont gérées à la fin de l'appel de la fonction *tick* des composants hardware (correspondant à la détection d'un front d'horloge). L'interruption à la plus haute priorité est appelée en premier, le programme sauvegarde l'état de ses variables et de sa pile, avant de faire un saut à l'adresse en mémoire où est codé le programme à exécuter pour cette interruption. Le diagramme de synthèse des interruptions peut être trouvé en annexe.

La mémoire SRAM

La mémoire SRAM est mappée dans l'espace mémoire IRAM ainsi que dans des parties de la XRAM. Elle est créée en tant qu'objet de type *cl_address_space* de taille 8 Ko.

La mémoire FLASH

La FLASH fournit une mémoire non-volatile programmable dans le circuit. Elle est mappée dans les espaces mémoires ROM et XRAM. La FLASH du CC2530 utilisé a une taille de 256 Ko. En plus de contenir le code du programme d'OCARI et les constantes, la mémoire non volatile permet à l'application de sauver les données qui doivent être préservées pour être disponibles après redémarrage du système. La mémoire FLASH est composée de 8 banques FLASH de 32 Ko, l'accès auxquelles est contrôlé par le registre MCON. Les banques FLASH sont mappées au travers des mémoires ROM et XRAM. Chaque banque flash est une instantiation différente de la classe *cl_address_space*.

Les registres SRF

Les registres SFR (Special Function Registers) sont un espace de mémoire de 128 octets directement accessible par une simple instruction du CPU. Ils contrôlent certaines des fonctions du CPU et des périphériques.

Les registres XREG

Les registres XREG sont des registres supplémentaires dans l'espace mémoire de la XRAM. Ils sont principalement utilisés pour la configuration et le contrôle de la radio du CC2530.

L'arbitre de mémoire

L'arbitre de mémoire est au cœur du système, il connecte le CPU et le contrôleur du DMA (Direct Memory Access) avec les mémoires physiques et tous les périphériques au travers du bus SFR. L'arbitre de mémoire a quatre points d'accès mémoire. Ces accès peuvent être mappé vers une des trois mémoires physiques : La SRAM, la mémoire flash, et les registres SFR/XREG.

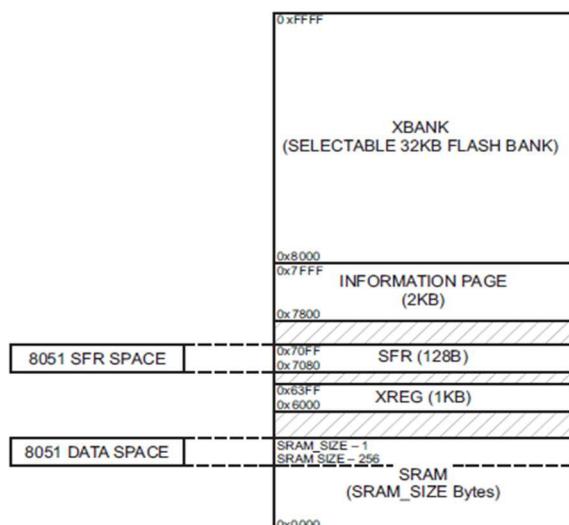


Figure 9 - Mapping de la mémoire XRAM

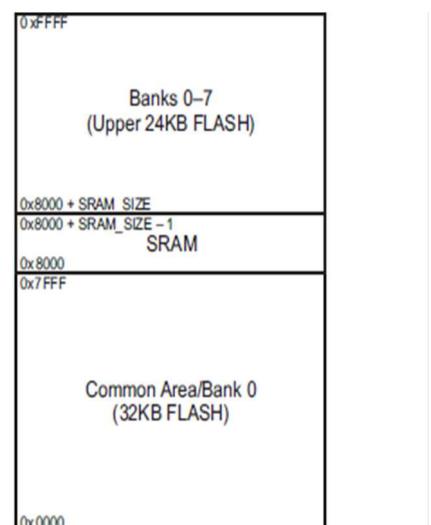


Figure 10- Mapping de la mémoire ROM

L'accès aux mémoires est géré par les fonctions *read*, *get*, *write* et *set* appartenant à la classe *cl_address_space*. En fonction de l'adresse donnée en argument, de la mémoire concernée, et des configurations des registres de contrôle de la mémoire, un accès à une mémoire différente est demandé, du fait que beaucoup de mémoires sont mappées dans d'autres. Par exemple, si un *read* de la XRAM est appelé à l'adresse 0x8000, la fonction *read* reconnaît l'identité XRAM de l'objet *cl_address_space*, ensuite la fonction vérifie dans quel intervalle se trouve l'adresse passée en argument. Après avoir reconnu que l'adresse est

supérieure ou égale à 0x8000, la fonction appelle la fonction *read* d'une des banques Flash de type *cl_address_space* et lui passera en argument l'adresse 0 (l'adresse 0x8000 dans la XRAM correspond au début de cette banque flash). La banque Flash est sélectionnée parmi les 8 banques Flash en fonction de la valeur dans le registre MCON (Memory CONtrol).

La commande « dump »

C'est la commande utilisée lors de l'exécution d'Ucsim pour observer l'état des mémoires. Elle affiche les données des cellules mémoires dans un intervalle d'adresse. Cette commande a été modifiée pour inclure les mémoires spécifiques au CC2530, et pour qu'elle affiche les sous parties des mémoires affichées: par exemple entre les adresses 0 et 0x2000 de la XRAM, l'accès est redirigé vers la SRAM, et les derniers 256 octets de la SRAM redirigent vers la IRAM. Donc, par exemple, commander un *dump* de la XRAM entre les adresses 0x1DDA et 0x1FFF rappelle la commande *dump* pour la SRAM entre les mêmes adresses 0x1DDA et 0x1FFF, qui à son tour appellera un *dump* de la IRAM entre les adresses 0 et 0xFF.

Cette commande est très utile pour pouvoir vérifier l'état des mémoires lorsque l'on teste le simulateur avec différentes configurations.

On peut observer sur la figure suivante un *dump* d'un espace mémoire de la XRAM dans laquelle est mappée une partie de l'espace mémoire de la IRAM.

```
0> dump xram 0x1FF0 0x2020
The flashbank mapped to xram is flashbank0.
Dump of xram (including end of iram).
0x1ff0 00 00 00 00 00 00 00 00
0x1ff8 00 00 00 00 00 00 00 00

*****End of iram section of xram.*****

0x2000 2d 04 77 0a fb 07 ff d6
0x2008 ec d5 ae 75 62 8e 18 77
0x2010 3c a6 c5 30 62 75 e5 a6
0x2018 f0 42 61 5b 9e f4 18 cb
0x2020 f8
```

5.2.2 Programmation des périphériques

Le système sur puce CC2530 inclut de nombreux périphériques différents. Cependant, tous ne sont pas programmés dans le simulateur, seuls ceux qui sont utiles au bon fonctionnement de la plateforme quand le programme OCARI est chargé, ou que le simulateur n'en a pas besoin. De cette façon, des périphériques comme le convertisseur numérique-analogique, le module qui gère l'encodage et le décodage des données, ou l'amplificateur opérationnel ne sont pas implémentés dans le simulateur.

Les composants hardware sont tous programmés en tant que classes héritées de la classe `cl_hw`. Les classes de ces composants sont toutes instanciées dans la classe du CPU (`cl_51core`).

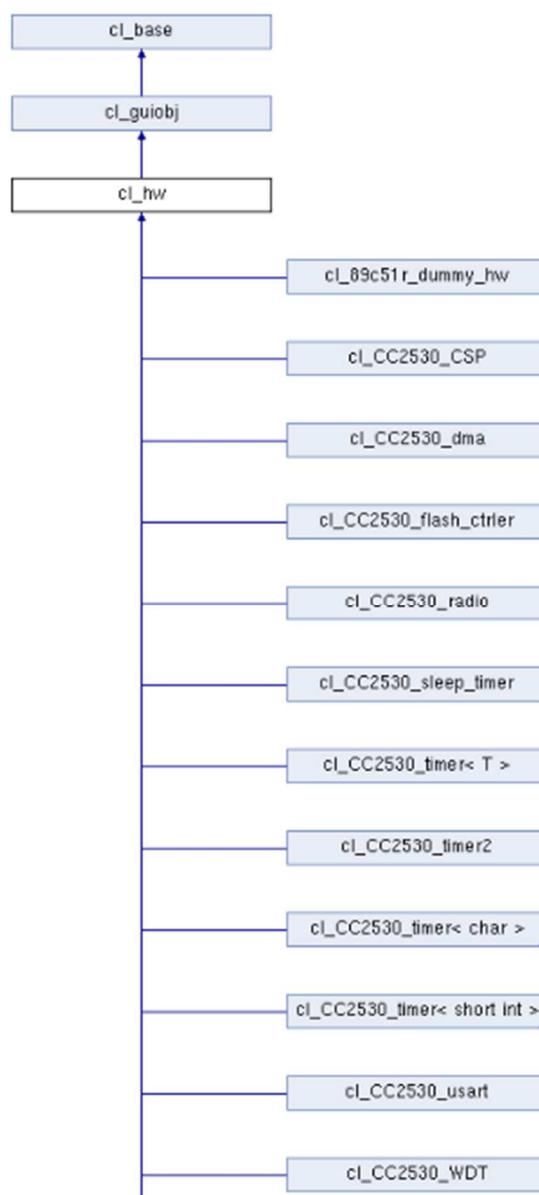


Figure 11 - Diagramme Doxygen montrant une partie des classes des composants hardware programmés pour le CC2530, et les classes dont elles sont héritées

Les évènements

Dans le programme du simulateur, il y a un système qui permet à un composant hardware de prévenir les autres composants hardware quand un évènement spécifique survient. Pour cela, il doit devenir « partenaire » avec les composants hardware qui doivent être informés de ses évènements. Quand un évènement se produit, les composants partenaires en sont informés grâce à la fonction *inform_partners(event)*. Un débordement du timer 1 par exemple est un des évènements qui peut déclencher un transfert du DMA, si un des canaux DMA est configuré pour être déclenché par ce *trigger*.

Le contrôleur DMA

DMA ou Direct Memory Access signifie accès direct à la mémoire. C'est un procédé où les données circulant de ou vers les périphériques sont transférées directement à la mémoire principale grâce à un contrôleur. Le contrôleur DMA peut donc être utilisé pour soulager le CPU dans la gestion des transferts de données, permettant d'atteindre de plus hautes performances. Le contrôleur assure aussi que les requêtes du DMA ont une priorité appropriée entre elles et les accès mémoires du CPU.

Le contrôleur DMA accède à la mémoire en utilisant l'espace mémoire de la XRAM, il a donc accès à toutes les mémoires physiques. Les caractéristiques de chaque canal (*trigger*, priorité, mode de transfert, mode d'adressage, pointeurs de source et destination, et compte de transfert) sont configurées avec des descripteurs DMA n'importe où en mémoire.

Les principales fonctionnalités du DMA sont les suivantes :

- 5 canaux DMA indépendants.
- 3 niveaux configurables de priorité des canaux DMA.
- 32 évènements *trigger* configurables pour déclencher les transferts DMA.
- Contrôle indépendant des adresses de source et de destination.
- Modes de transfert simple, en bloc ou répété
- Transferts de longueur variable.
- Peut opérer soit en mode octet soit en mode mot (2 octets).

Chaque canal du DMA peut déplacer des données d'un endroit dans la mémoire XRAM à un autre. Les canaux doivent être configurés avant de pouvoir être utilisés. Ils doivent aussi être armés avant l'initialisation du transfert. Le transfert démarre quand l'évènement *trigger* associé à son canal se produit. Un transfert peut aussi être forcé en écrivant dans le registre DMAREQ (DMA REQuest). Le diagramme décrivant le mode opératoire du DMA peut être trouvé en annexe.

Voici un exemple de transfert DMA lors d'un test du simulateur. Le DMA est configuré dans le programme de test en assembleur, notamment pour configurer les adresses où se trouvent les configurations des canaux (registres DMAxCFG). Dans la portion de programme assembleur suivant, des registres sont également configurés pour armer les canaux et pour forcer un transfert.

Portion de programme assembleur configurant les registres SFR liés au contrôleur DMA :

```
MOV DMA0CFGH, #0H
MOV DMA0CFGL, #80H
MOV DMA1CFGH, #0H
MOV DMA1CFGL, #90H
MOV DMAARM, #03H;
MOV DMAREQ, #02H
```

Les configurations des canaux 0 et 1 du DMA peuvent donc être trouvées dans la XRAM aux adresses 0x80 et 0x90 respectivement, comme indiqué dans les registres DMA0CFGH:DMA0CFGL et DMA1CFGH:DMA1CFGL. Les configurations pour le test sont les suivantes :

Canal	Source	Dest.	Long. Variable	Long.	Taille des données	Mode de transfert	Trigger	Inc. adr. Source	Inc. adr. Dest.
0	0x8003	0	2	10	2 octets	Block	10	+2 octets	+2 octets
1	0x8040	0x50	2	15	1 octet	Repeated Single	7	+1 octet	+1 octet

Cette configuration des canaux 0 et 1 signifie que lorsque surviendra le trigger numéro 10, qui correspond à une comparaison sur le canal 1 du timer 4, à chaque front d'horloge dans le contrôleur DMA, un mot de 2 octets sera transféré de la zone mémoire commençant à l'adresse 0x8003 dans la XRAM à la zone mémoire commençant à l'adresse 0, jusqu'à ce que 10 mots aient été transférés. A chaque transfert de mot, l'adresse de source et de destination sont incrémentées chacune de 2 octets, pour ne pas constamment lire et écrire sur le même octet. Le transfert sur le canal 1 sera déclenché par le trigger 7, soit la comparaison sur le canal 0 du timer 3. Etant donné que le mode de transmission est « repeated single », un seul octet sera transféré à chaque fois que le trigger surviendra, et non pas à chaque front d'horloge après le trigger, cela jusqu'à ce que 15 octets aient été transférés.

Lorsque le simulateur interprète l'instruction qui écrit la valeur 3 dans le registre DMAARM, les canaux 0 et 1 sont armés et les configurations pour ces canaux sont lues en mémoire. Voici ce que le simulateur affiche alors :

```
CC2530DMA.cc:163 in tick()
*****
** CC2530DMA ****
tick!
CC2530DMA.cc:258 in LoadConfig()
Transferring on channel 0: 0x0010 Words, Maximum length is: 0x0010
Repeated : no
CC2530DMA.cc:258 in LoadConfig()
Transferring on channel 1: 0x0015 Bytes, Maximum length is: 0x0015
Repeated : yes
```

Ensuite lorsque survient un évènement trigger pour un de ces canaux, des données commencent à être transférées au front d'horloge suivant. Dans ce cas, au front d'horloge précédent sont survenus les triggers pour les deux canaux :

```
***** CC2530DMA *****
tick!
CC2530DMA.cc:224 in transfer()
DMA transfer on channel 0: Moving 0x02 from 0x8005 to 0x0002
CC2530DMA.cc:233 in transfer()
DMA transfer on channel 0: Moving 0x75 from 0x8006 to 0x0003
CC2530DMA.cc:224 in transfer()
DMA transfer on channel 1: Moving 0xe5 from 0x8040 to 0x0050
```

Si on avance encore dans la simulation et qu'on observe l'état de la mémoire XRAM, on peut observer que des données sont bien en cours d'être copiées comme prévu dans la configuration :

<pre>0> dump xram 0 0x60 The flashbank mapped to xram is flashbank0. Dump of xram (including beginning of sram). *****Beginning of sram section of xram.*****</pre> <pre>0x0000 75 aa 02 75 db 03 90 61 0x0008 00 74 83 f0 90 61 01 74 0x0010 7d f0 90 61 02 74 50 f0 0x0018 90 61 03 74 00 00 00 00 0x0020 00 00 00 00 00 00 00 00 0x0028 00 00 00 00 00 00 00 00 0x0030 00 00 00 00 00 00 00 00 0x0038 00 00 00 00 00 00 00 00 0x0040 00 00 00 00 00 00 00 00 0x0048 00 00 00 00 00 00 00 75 0x0050 e5 00 00 00 00 00 00 00 0x0058 00 00 00 00 00 00 00 00 0x0060 00</pre>	<pre>0> dump xram 0x8000 The flashbank mapped to xram is flashbank0. Dump of xram (including beginning of flashbank0). *****Beginning of flashbank0 section of xram.*****</pre> <pre>0x8000 75 a8 80 75 aa 02 75 db 0x8008 03 90 61 00 74 83 f0 90 0x8010 61 01 74 7d f0 90 61 02 0x8018 74 50 f0 90 61 03 74 64 0x8020 f0 90 61 04 74 6e f0 90 0x8028 61 05 74 ab f0 90 61 06 0x8030 74 ce f0 90 61 07 74 f5 0x8038 f0 75 c7 00 75 e4 02 75 0x8040 e5 04 75 e6 03 75 d5 00 0x8048 75 d4 80 75 d3 00 75 d2</pre>
--	--

Le contrôleur de la mémoire Flash

Le contrôleur de la mémoire flash gère l'écriture et l'effacement de la flash. Celle-ci est constituée de 128 pages de 2 ko chacune.

Le contrôleur peut effectuer les fonctions suivantes :

- Programmation sur mots de 32 bits.
- Effacement d'une page de flash.
- Effacement de la flash entière.
- Bits verrous pour la protection de l'écriture et la sécurité du code.

La page est la plus petite unité effaçable en mémoire flash, et le mot de 32 bits est la plus petite unité qui peut être accédée pour l'écriture. Pendant les opérations d'écriture, l'adressage

de la flash est sur 16 bits (registres FADDRH :FADDRL) et renvoie à des mots de 4 octets, et non pas à un seul octet, comme lorsque cette mémoire est accédée par le CPU.

La flash est programmée en série sur une séquence d'un ou plusieurs mots de 32 bits. En général la page de flash doit être effacée avant l'écriture, ce qui remet tous les ses bits à 1. C'est aussi le seul moyen de mettre à 1 les bits dans cette mémoire. Quand on écrit un mot dans la flash, les bits 0 sont programmés à 0 et les bits 1 sont ignorés. Cette dernière caractéristique d'écriture de la flash n'est pas reproduite dans le simulateur. Les mots sont écrits en flash sans se préoccuper des 1 et 0. Le simulateur vérifie tout de même que l'écriture dans la flash suit l'algorithme d'écriture correct. Il y a un nombre limité de fois où l'on peut écrire un mot avant qu'il soit effacé, ainsi qu'un nombre limité de fois où l'on peut écrire dans une page. Le simulateur compte le nombre d'écritures de chaque mot et chaque page et signal une erreur si il y a eu un nombre trop important d'écritures.

L'algorithme à suivre pour écrire dans la flash est le suivant. Il faut d'abord configurer l'adresse du mot où l'on veut écrire (les 16 MSB de l'adresse sur 18 bits du premier octet du mot), puis démarrer l'écriture en écrivant dans le registre FCTL, écrire 4 fois dans le registre FWDATA (Flash Write DATA) en moins de 20 µs. Les données sur un octet écrites dans ce registre sont mémorisées dans un buffer de 4 octets. Lorsque celui-ci est plein, le mot de 4 octet est écrit dans la Flash. Une fois le buffer plein, il faut attendre que le registre FCTL signale que l'écriture est terminée pour pouvoir commencer une nouvelle écriture. L'effacement d'une page ou de la flash entière ne peut durer plus de 20 ms. Le simulateur signalera une erreur si c'est le cas, il signalera également si l'écriture d'un mot a pris plus de 20 µs.

Voici une partie du programme en assembleur qui a servi au test de l'écriture dans la flash, suivi de ce que le simulateur montre après 4 écritures dans le registre FWDATA. Les 4 octets écrits dans ce registre ont été 0x0C, 0x0C, 0x0D et 0x0D. Une fois que le buffer de 4 octets qui lit le registre est plein, les 4 octets sont écrits dans la flash, après que l'adresse d'écriture du mot ait été configurée. 0x100 est l'adresse configurée du mot (en mode adressage par 4 octets), 0x400 est donc l'adresse du premier octet (en mode adressage de chaque octet).

Programme test en assembleur :

```
MOV DPTR, #6272H ; faddrh
MOV A, #01H
MOVX @DPTR, A ;MOV FADDRH, #01H  setting flash write @
MOV DPTR, #6271H ; faddrl
MOV A, #00H
MOVX @DPTR, A ; MOV FADDRL, #00H
MOV DPTR, #6270H ; fctl
MOV A, #02H
MOVX @DPTR, A ; MOV FCTL, #02H flash write enabled
MOV DPTR, #6273H ; fwdata
MOV A, #0CH
MOVX @DPTR, A ;
MOVX @DPTR, A ;
MOV A, #0DH
MOVX @DPTR, A ;
MOVX @DPTR, A ;
```

Réponse du simulateur à l'exécution du programme :

```

Next instruction to be executed:
0x00bf f0      MOVX  @DPTR,A
0>
mov.cc:513 in inst_movx_Sdptr_a()
CC2530flashCtrler.cc:247 in write()
REGISTER value is 0x0d0d0c0c!
CC2530flashCtrler.cc:258 in write()
CC2530flashCtrler.cc:134 in flashWrite()
Flash Write of 0x0c to address 0x0400 in flashbank0!
Flash Write of 0x0c to address 0x0401 in flashbank0!
Flash Write of 0x0d to address 0x0402 in flashbank0!
Flash Write of 0x0d to address 0x0403 in flashbank0!
MOVX: Writing 0x0d in xram at @ 0x6273

```

Les timers

Le CC2530 possède de nombreux timers :

- Le timer 1 (16 bits)
- Le timer MAC ou timer 2 (24 bits)
- Les timers 3 et 4 (8 bits)
- Le sleep timer
- Le watchdog timer

Les timers 1, 3 et 4 possèdent de nombreuses fonctionnalités communes, il est donc paru plus judicieux de créer un template pour ces timers afin d'éviter le plus possible des répétitions de code.

Timer 1

Le timer 1 est un timer indépendant sur 16 bits aux fonctionnalités typiques d'un chronomètre/compteur telles que la capture de la valeur en entrée, la comparaison en sortie et les fonctions PMW (Pulse Width Modulation).

Le timer possède 5 canaux indépendants de capture et de comparaison, avec un pin E/S par canal. Ce timer est utilisé pour un éventail d'applications de contrôle et de mesure, et la disponibilité du mode de comptage croissant/décroissant avec 5 canaux permet, par exemple, des implémentations d'applications de contrôle de moteur.

Les caractéristiques du timer 1 sont les suivantes :

- 5 canaux de capture/comparaison
- Capture d'entrée sur front montant, descendant, ou tous les fronts
- Mise à 1, 0 ou alternance (*toggle*) sur le pin de sortie pour la comparaison
- Modes de fonctionnement en compteur libre, modulo, ou croissant/décroissant
- Diviseur de fréquence d'horloge par 1, 8, 32 ou 128.
- Requête d'interruption générée sur chaque capture, comparaison ou fin de compte
- Trigger pour le contrôleur DMA

Timers 3 et 4

Les timers 3 et 4 sont deux timers sur 8 bits. Chacun de ces timers possède deux canaux indépendants de capture et de comparaison et utilise un pin E/S par canal.

Les caractéristiques des timers 3 et 4 sont les suivantes :

- Deux canaux de capture ou comparaison.
- Mise à 1, 0 ou alternance (*toggle*) sur le pin de sortie pour la comparaison
- Diviseur de fréquence d'horloge par 1, 2, 4, 8, 16, 32, 64 ou 128.
- Requête d'interruption générée sur chaque capture, comparaison ou fin de compte
- Trigger pour le contrôleur DMA

Voici un exemple de ce que le simulateur montre pendant la simulation, lorsque le timer 3 atteint la valeur 3, laquelle est aussi configurée dans le registre de comparaison du canal 0. Lorsque cette comparaison survient, les informations sur le timer sont affichées. On observe la même chose pour le timer 4 qui a atteint la valeur 6, ce qui déclenche une comparaison sur son canal 1. On voit également que ces événements déclenchent un transfert DMA.

```
Next compare event in 0 Timer ticks...
CC2530timer3.cc:145 in CaptureCompare()
Compare mode? 4
Compare case: 0 Timer mode: 2
Event 7 triggered Channel 1 of the DMA
*****
CC2530timer3[1] Count: 0x0003 Modulo mode *****
Prescale value: 4 System clk division: 1
System Frequency: 3.2e+07 Hz CC2530 Crystal: 32000000 Hz
Time elapsed: 2.875e-06 s
CC2530timer3 IO Pins: Channel 0: 1 Channel 1: 0
*****  

CC2530timer3 Count: 3
Next compare event in 1 Timer ticks...
Next compare event in 0 Timer ticks...
CC2530timer4.cc:144 in CaptureCompare()
Compare mode? 4
Compare case: 0 Timer mode: 1
Event 10 triggered Channel 0 of the DMA
*****
CC2530timer4[1] Count: 0x0006 Down Mode *****
Prescale value: 1 System clk division: 1
System Frequency: 3.2e+07 Hz CC2530 Crystal: 32000000 Hz
Time elapsed: 2.875e-06 s
CC2530timer4 IO Pins: Channel 0: 0 Channel 1: 0
*****  

CC2530timer4 Count: 6
CC2530DMA.cc:163 in tick()  

*****
CC2530DMA *****
tick!
Radio tick. FSM state 0 Timer countdouwn: 0
*****
Register State Info: *****
R0= 0x19 R1= 0x1c R2= 0xad R3= 0xd6
R4= 0xfc R5= 0x30 R6= 0x67 R7= 0x5b
ACC= 0x0d 13 . B= 0x00
DPTR= 0x6273 @DPTR= 0x0d 13 .
```

Le Timer MAC

Le timer MAC est principalement utilisé pour la temporisation des algorithmes de CSMA/CA. Il fonctionne à la même fréquence que l'horloge du système, qui doit être 32 MHz lorsque le timer MAC fonctionne.

Les principales fonctions du timer MAC sont les suivantes :

- Un compteur sur 16 bits qui fournit, par exemple, une période symbole/trame de 16µs/320µs.
- Une période ajustable avec une précision de 31,25 ns.
- 2 fonctions de comparaison sur 16 bits.
- Un compteur d'overflows (débordements) sur 24 bits.
- Une fonction de capture start-of-frame-delimiter (SFD – voir le format des trames en annexe).
- Des interruptions générées pour les comparaisons et overflows.
- La possibilité de déclencher un transfert DMA.
- La possibilité d'ajuster la valeur du timer pendant le compte en introduisant un délai.

Il y a 6 événements possibles qui peuvent survenir dans le timer Mac (comme comparaison du compte d'overflow numéro 1 par exemple), et deux sorties T2_EVENT1 et T2EVENT2 qui peuvent chacune être associées à un évènement, et peuvent ensuite être utilisées pour déclencher un transfert du DMA, ou comme entrées pour la radio.

Le timer MAC possède également plusieurs registres multiplexés, dû au nombre limité de registres SFR. Certains registres internes au timer MAC peuvent donc être accédés indirectement en passant par des registres SFR, en fonction du registre SFR T2MSEL (Timer 2 multiplex Select).

Voici un exemple de ce que montre le simulateur lors de son exécution dans le cas d'un overflow du timer MAC (le timer a atteint le compte 0xFFFF et est repassé à 0), et si la sortie T2_EVENT1 est configurée pour passer à 1 quand la valeur du compte du timer MAC est égal à 0 (comparaison avec 0).

```
CC2530timer2.cc:337 in do_UpMode()
CC2530_MAC_timer overflow !

*****
CC2530_MAC_timer[1] Count: 0x0000      Up Mode      *****
Timer Frequency: 8e+06 Hz      CC2530 Crystal: 32000000 Hz
Time elapsed: 6.125e-06 s
*****



COMPARE 1 EVENT! Count: 0
MAC TIMER OUTPUTS: EVENT1: 1 (Compare 1)           EVENT2: 0
Timer MAC count: 0x00.
```

Le Sleep Timer

Le Sleep Timer est utilisé pour fixer une période durant laquelle le système entre et sort des modes de faibles puissances. Il est aussi utilisé pour maintenir le compte dans le timer MAC dans ces modes. C'est un timer sur 24 bits opérant à une fréquence de 32 kHz, il possède des fonctionnalités de capture et de comparaison sur 24 bits et peut être utilisé pour déclencher un transfert DMA.

Le watchdog timer

Le Watchdog Timer (WDT) est conçu pour être une méthode de récupération dans des situations où le CPU aurait un problème. Il provoque un reset du système quand un software ne le remet pas à 0 pendant un intervalle de temps sélectionné. Le WDT peut fonctionner soit en mode Watchdog, soit en mode timer, et peut générer des requêtes d'interruption en mode timer.

Le Sleep Timer et le Watchdog Timer ont été programmés mais n'ont pas été testé, faute d'un manque de temps, et surtout du fait que leur programmation n'était pas prioritaire, sachant qu'ils n'étaient pas indispensables pour la validation du code OCARI (dans un premier temps).

Les USARTs

USART 0 et USART 1 sont des interfaces de communication série (SPI) qui peuvent fonctionner séparément en mode UART asynchrone ou en mode SPI (Serial Peripheral Interface) synchrone.

Le mode UART est utilisé pour les interfaces asynchrones possédant les pins nommés RXD et TXD, et optionnellement les pins RTS et CTS. Dans ce mode, l'USART possède les caractéristiques suivantes :

- une taille de 8 ou 9 bits pour les données utiles
- une parité paire, impaire ou pas de parité
- des niveaux de bits *start* et *stop* configurables
- un transfert du MSB ou LSB en premier
- des requêtes d'interruption indépendantes en émission et réception
- des triggers pour le DMA indépendants en émission et en réception
- Un statut d'erreur de parité ou de trame

Le mode UART permet des transferts asynchrones en full-duplex : la synchronisation des bits dans le récepteur n'interfère pas avec la transmission. Quand l'UART transfert un octet, il transmet un bit *start*, 8 bits de données, un neuvième bit optionnel, un bit de parité optionnel, et un ou deux bits *stop*. Les opérations des USARTs sont contrôlées par les registres SFR UxUCR et UxUSR, où x correspond à 0 ou 1.

Le mode SPI est utilisé quand l'USART communique avec un système externe à travers une interface à 3 ou 4 pins. L'interface possède donc les pins MISO (Master Input Slave Output), MOSI (Master Output Slave Input), SCK (Serial Clock) et SSN (Slave Select).

Quand l'USART fonctionne en mode SPI, elle possède les caractéristiques suivantes :

- Les modes Master et Slave
- Une polarité et phase de SCK configurables
- Transfert du MSB ou LSB en premier

La fréquence d'envoi et de réception des bits en mode UART, ainsi que la fréquence de l'horloge SCK générée par le master SPI dépendent du facteur Baud (unité de mesure du nombre de symboles transmissibles par seconde), configuré dans les registres SFR.

Voici un test du mode UART de l'USART. Des données sont envoyées pendant que d'autres sont reçues. L'octet à transmettre est 0xCC. La première chose que fera l'UART est de calculer le bit de parité à transmettre avec cet octet. La présence du bit de parité ainsi que sa parité paire ou impaire sont déterminés d'après les configurations des registres SFR. Pour déterminer cette parité, l'UART effectue un xor de tous les bits de l'octet en question. Voici une partie du programme qui calcule la parité :

```
t_mem check = byte & 0x01;//Isolate LSB
for (int i =0; i<7; i++)
{
    byte >>= 1;//right shift
    check ^= (byte & 0x01);//LSB xor (previous xor result)
    fprintf(stderr, "Itération: %d\tByte: 0x%02x\tCheck: %d\n", i, byte, check);
}
```

Le résultat de la variable check (valeur de check après la dernière itération) correspond à une parité paire, qui doit être inversée si la parité à transmettre est configurée pour être impaire. C'est le cas ici. Voici ce que montre le simulateur lorsqu'il cherche la parité de l'octet à transmettre.

```
Parity CHECK !!!
BYTEToCheck: 0xcc      First Check: 0
Itération: 0    Byte: 0x66      Check: 0
Itération: 1    Byte: 0x33      Check: 1
Itération: 2    Byte: 0x19      Check: 0
Itération: 3    Byte: 0x0c      Check: 0
Itération: 4    Byte: 0x06      Check: 0
Itération: 5    Byte: 0x03      Check: 1
Itération: 6    Byte: 0x01      Check: 0
ODD Parity !!!
```

```
Parity to TX: 1
```

Ensuite les bits sont envoyés à partir du buffer de transmission (envoi du LSB ou MSB selon la configuration) en effectuant un shift droit ou gauche après chaque envoi (selon que ce soit le LSB ou le MSB qui soit envoyé). En même temps, le buffer de reception reçoit lui aussi un bit avant de faire un shift pour laisser la place au bit suivant. Cela se fait jusqu'à ce que l'octet entier soit envoyé et un autre reçu. Voici ce que nous montre le simulateur après deux et trois cycles d'envoi/réception de bits :

```
CC2530uart.cc:145 in BaudTick()
RX bits count: 2
IPIN: 1
RxReg: 0x01
TX bits count: 2
TxReg: 0xcc
CC2530uart.cc:273 in Shift_out()
TxReg's MSB: 1
TXpin: 1
```

```
CC2530uart.cc:145 in BaudTick()
RX bits count: 3
IPIN: 0
RxReg: 0x02
TX bits count: 3
TxReg: 0x98
CC2530uart.cc:283 in Shift_out()
TxReg's MSB: 1
TXpin: 1
```

Après avoir reçu le nombre attendu de bits, l'UART vérifie que l'octet reçu est correct grâce au bit de parité qu'il a également reçu. Il envoie aussi le bit de parité pour l'octet qu'il vient de finir de transmettre.

```
CC2530uart.cc:145 in BaudTick()
RX bits count: 10
IPIN: 1
RxREG: 0xaa
Parity CHECK !!!
BYTEToCheck: 0xaa      First Check: 0
Itération: 0    Byte: 0x55      Check: 1
Itération: 1    Byte: 0x2a      Check: 1
Itération: 2    Byte: 0x15      Check: 0
Itération: 3    Byte: 0x0a      Check: 0
Itération: 4    Byte: 0x05      Check: 1
Itération: 5    Byte: 0x02      Check: 1
Itération: 6    Byte: 0x01      Check: 0
ODD Parity !!!
```



```
Parity OK!
TX bits count: 10
TxReg: 0x00
CC2530uart.cc:292 in Shift_out()
ParityTx!
TXpin: 1
```

Le module Radio et le Processeur de commandes CSP

La radio a été la partie la plus complexe à modéliser. Un processeur simple (CSP) sert d'interface entre le CPU et la radio, ce qui permet au CPU de donner des commandes, de lire l'état, d'automatiser et de séquencer les évènements radio.

Le sous-module FSM (Finite State Machine) contrôle l'état de l'émetteur-récepteur RF, des FIFOs d'émission et de réception, ainsi que l'état de la plupart des signaux analogues contrôlés dynamiquement. Le FSM est utilisé pour fournir la séquence correcte d'évènements. Aussi, il permet de traiter étape par étape les trames reçues du démodulateur : la lecture de la taille de la trame, le compte du nombre d'octets reçus, la vérification du FCS, et optionnellement la gestion de la transmission des trames ACK (Acknoledge) après la bonne réception d'une trame. Le FSM effectue des tâches similaires pour la transmission, et il contrôle les transferts de données entre le modulateur et démodulateur, ainsi qu'entre la TXFIFO et la RXFIFO (FIFOs utilisées pour stocker les trames en transmission et réception) dans la RAM.

Le modulateur et le démodulateur ne sont pas simulés. Ils sont utilisés pour adapter les données au médium RF, cependant dans le simulateur, les données ne sont jamais converties en données analogiques, elles sont transférées en octet sur des bus qui connectent les capteurs. Le module radio possède une entrée et une sortie sur un octet (radio_in et radio_out), qui sont connectées aux entrées/sorties d'autres capteurs.

Le module radio s'occupe aussi du filtrage des trames et de l'identification des adresses. Ces deux fonctions utilisent un bloc de 128 octets de la RAM réservée à la radio pour stocker des informations sur les adresses locales ainsi que les configurations et résultats de l'identification des sources.

Pour pouvoir effectuer les fonctions de filtrage des trames et d'identification des adresses, le simulateur doit analyser les trames entrantes. Une fois qu'il a reçu le nombre d'octets spécifiés par le champ « Frame Length » de la trame, indiqué dans le premier octet après l'en-tête de synchronisation, il va analyser les différents champs de la trame. Des vues schématiques du format des trames du standard IEEE 802.15.4 peuvent être trouvées en annexe. L'analyse de la trame commence par la partie FCF (*Frame Control Field*), cela permet de déterminer entre autre le type de la trame (beacon, data, ack, ou commande MAC), de déterminer également si cette trame nécessite qu'une trame ACK soit renvoyée, mais aussi les modes d'adressage de source et de destination. Ce sont ces modes d'adressage qui vont permettre au simulateur de déterminer où trouver les adresses de source et de destination s'il y en a ainsi que leur taille : il peut y avoir soit des adresses longues sur huit octets, soit des adresses courtes sur deux octets qui sont associées à une adresse de PAN sur deux octets également.

Lors de la réception d'une trame, plusieurs étapes sont effectuées par la radio pour son traitement. Tout d'abord, la radio détecte et retire l'en-tête de synchronisation PHY (le préambule et le SFD (Start of Frame Delimiter)), puis reçoit le nombre d'octets spécifié par le champ Frame Length field. Ensuite la trame est filtrée, en fonction de la configuration des

registres réservés à la radio. La fonction de filtrage rejette les trames qui ne lui sont pas destinées. Elle filtre également sur le type de la trame, et sur les bits réservés du FCF. Cette fonction est contrôlée par les registres FRMFILT0 et FRMFILT1, ainsi que par les valeurs des adresses mémorisées dans la RAM. L'étape suivante est l'identification de la source. L'adresse de source est retrouvée dans la trame de réception puis comparée au tableau d'adresses stocké dans la RAM, qui peut contenir jusque 24 adresses courtes ou 12 adresses étendues IEEE.

Une structure « Frame » est créée dans le programme. Deux instanciations de cette structure sont faites, une pour contenir les informations de la trame de la FIFO de réception après que la radio l'ait décodée, l'autre contenant les informations de la trame de la FIFO d'émission.

Voyons un exemple de traitement d'une trame dans le simulateur. La trame dans la FIFO de réception est 408801003D0B71A87D8364500C0C0C...0C0C (en format hexadécimal). La radio va décoder cette trame, identifier si l'adresse de source fait partie de celles des capteurs auxquels elle est associée, vérifier si la trame lui est destinée, et enfin faire passer cette trame par un filtre. La radio ne se préoccupe pas du *payload* (données utiles) de la trame.

A la fin de la réception de la trame, voici ce que montre le simulateur pendant le traitement de cette trame :

```
Frame length: 64
Frame Control Field: 0x8801
Frame type: Data
Security Enabled: No
Frame Pending: No
Acknoledge Requested: No
Pan compression: No
Destination addressing mode: Short address
Source addressing mode: Short address
Source address: 0x6450
Source Pan ID: 0x7d83
Source Address Match Found!
Result Register: 0x01
Destination address: 0x71a8... Match!
Destination Pan ID: 0x3d0b... Match!
Frame has passed the filter without being rejected.
```

Cette trame est acceptée, puisque l'adresse de source correspond à la première dans le tableau des adresses de la radio (d'après le registre RESINDEX ou *result register* qui contient la valeur 1), l'adresse de destination correspond à l'adresse de la radio (la radio peut lire la valeur de son adresse dans les registres XREG), enfin le filtre n'a trouvé aucune raison de rejeter la trame.

Après le décodage de la trame, l'identification de la source et le filtrage, la radio vérifie ensuite que le FCS est correct, et rattache aux trames reçues des données sur la validité du FCS, et d'autres valeurs comme le RSSI et l'identification de la source. Enfin la radio renvoie une trame ACK automatiquement si cela est spécifié par la trame qu'elle vient de recevoir.

Des registres SFR de la radio sont aussi utilisés pour stocker des informations sur la qualité du lien radio, tels que le CCA (Clear Channel Assessment) qui indique si le canal est libre pour effectuer une transmission ou non, ou le bit RSSI_VALID (Received Signal Strength Indicator), indiquant la puissance du signal reçu. Dans le simulateur, ces valeurs sont des valeurs constantes déterminées lors de sa programmation, indiquant une bonne qualité de la liaison radio. Toutefois, ces valeurs peuvent être modifiées pour évaluer ce qu'il se passerait en cas d'une mauvaise liaison radio. Le FCS (Frame Check Sequence) n'est pas utilisé dans le simulateur, la simulation considérera par défaut et indiquera dans les résultats de la vérification que le FCS est toujours correct. Mais cela n'empêche pas la modification des résultats de la vérification du FCS pour tester les conséquences d'une trame reçue erronée.

Le command Strobe CSMA-CA Processor (CSP) sert d'interface pour la radio et le CPU grâce à quelques registres SFR et XREG, il possède entre autre dans les registres XREG 24 octets de mémoire pour stocker le programme qu'il exécutera, et 3 registres de données. Le CSP est aussi une interface avec le timer MAC car il observe les événements de ce dernier. Il permet au CPU de donner directement des commandes à la radio, et donc de contrôler les opérations de la radio.

Le CSP exécute des instructions sur un octet dans la mémoire d'instructions de 24 octets. Les accès en écriture à cette mémoire sont séquentiels et se font en écrivant dans le registre RFST. L'exécution des programmes commence lorsque le CPU donne l'instruction ISSTART au registre RFST, une commande qui est exécutée instantanément. Le diagramme montrant comment charger et exécuter un programme dans le CSP est donné en annexe. Le set d'instructions du CSP est composé de 46 instructions.

Voici un exemple d'une partie du code pour le décodage d'instructions (lecture du code en hexadécimal dans l'espace mémoire du CSP et appel de la fonction de l'instruction correspondante), suivi d'un exemple d'une instruction simple, DECX, qui décrémente la valeur contenue dans le registre XREG CSPX.

```
switch (code)
{
    case 0xB8: inst_wait_event1(code); break;
    case 0xB9: inst_wait_event2(code); break;
    case 0xBA: inst_int(code); break;
    case 0xBB: inst_label(code); break;
    case 0xBC: inst_waitx(code); break;
    case 0xBD: inst_rand_xy(code); break;
    case 0xBE: inst_set_cmp1(code); break;
    case 0xC0: inst_inc_x(code); break;

void cl_CC2530_CSP::inst_dec_x(uchar code)
{
    cell_cspx->set(cell_cspx->get() - 1);
    PC = PC + 1;
}
```

5.3 Interfaçage entre deux simulateurs

Pour simuler le fonctionnement concomitant de deux simulateurs, deux instances du CC2530 sont implémentées dans la fonction *main* du programme. Il y a donc deux instances de la classe simulateur, et deux instances de la classe application. L'interface en lignes de commandes permettant de contrôler le simulateur est supprimée. Le programme rentre dans une boucle *while* et y reste tant qu'aucun des deux simulateurs de CC2530 ne reçoit un signal de fin de simulation.

Une fois dans la boucle *while*, dans un premier temps, chacun des simulateurs reçoit un front d'horloge de fréquence 32 MHz qui est la fréquence maximum du CC2530. Cette horloge sert de référence pour le temps, mais aussi à s'assurer que les deux simulateurs restent synchronisés. En réalité, il est possible que les capteurs se désynchronisent, notamment s'ils ratent des trames de type *beacon*. Le simulateur ne pousse pas encore la simulation jusque-là, mais c'est un point qui pourra être travaillé par la suite. En fonction de la fréquence choisie dans les configurations des registres SFR de chaque CC2530 la fréquence de base peut être divisée par 2, 4, 8, 16, 32, 64 ou 128. Si la fréquence d'un des simulateurs est 32 Mhz divisé par 8, alors tous les 8 fronts montants de l'horloge de référence, ce simulateur exécutera une instruction.

Dans un second temps, si elles sont disponibles, les données sont échangées entre les simulateurs. Ces données sont en réalité des signaux modulés qui ont un débit de 2 Mchips/s, soit 62,5 ksymboles/s ou 31,25 ko/s. Il faut donc 32 µs pour transmettre un octet, ce qui correspond à 1024 cycles d'horloge à 32 MHz.

Pour plus de simplicité, les données sont transmises sur des bus de 8 bits entre les 2 simulateurs, un buffer est rajouté sur chaque bus pour retenir les données pendant 32 µs. Les variables (sémaphores) DataPresent et DataRead associées à chaque buffer permettent aux 2 simulateurs de savoir si ils peuvent écrire/lire des données dans ce buffer. Quand un simulateur écrit dans le buffer, DataPresent passe à 1. Quand l'autre simulateur lit les données, DataRead passe à 1. Au bout de 32 µs, le buffer se vide, et DataPresent ainsi que DataRead passent à 0.

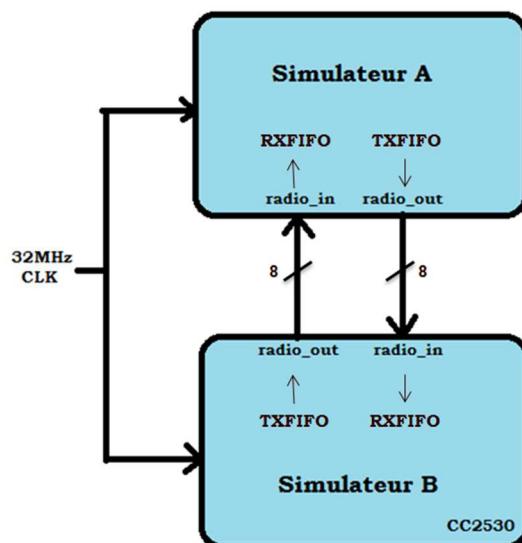


Figure 12 - Interface entre deux instances du simulateur

5.4 Bilan

Nous arrivons actuellement à faire fonctionner deux simulateurs de CC2530 en même temps. Toutefois, la communication entre les deux simulateurs n'a pas fini d'être implémentée, et n'a été ni testée, ni déboguée.

Le simulateur du CC2530 réussit à imiter les fonctions majeures de la plateforme réelle. Tous les périphériques qui sont nécessaires à OCARI ont été implémentés et testés. Certaines fonctions de ces périphériques n'ont par contre pas été implémentées, car il était peu probable que le code d'OCARI y accède. Il faudrait vérifier qu'OCARI n'accède effectivement pas aux registres liés à ces fonctions une fois que les deux simulateurs pourront échanger des messages, en exécutant chacun un binaire d'OCARI pour un type de capteur différent (coordinateur, routeur...).

Cependant, il reste quelques bugs à rectifier. Par exemple, le chargement d'un gros binaire dans la mémoire flash du simulateur ne se fait pas correctement. Au lieu que le binaire soit écrit sur les 8 banques de flash pour un total de 256 ko, il est écrit sur les deux premières banques, puis au lieu de continuer à être écrit sur la troisième, il est de nouveau écrit sur la première banque flash, écrasant les données qui s'y trouvent.

Il faudrait aussi tester plus extensivement toutes les fonctionnalités dans le simulateur, pour pouvoir détecter de nouveaux bugs qui n'apparaissent que dans des cas isolés. Le fait par exemple de rajouter une deuxième instance du CC2530 dans le simulateur a d'abord créé une erreur de type segmentation fault. Après avoir déboguer le programme à l'aide de GDB, il s'avère que cette erreur provient d'un pointeur non initialisé. Ce pointeur n'avait pourtant jamais causé de problèmes (appareils) lorsqu'il n'était utilisé que dans le simulateur contenant une seule instance du CC2530.

De nombreuses erreurs de type Segmentation Fault apparaissent au fur et à mesure que la programmation avance. Cela est en fait dû aux nombreuses variables non initialisées dans le programme original d'Ucsim. Le logiciel Valgrind a permis de détecter qu'il y avait dans le programme beaucoup de sauts conditionnels dépendant de pointeurs non initialisés. Pour remédier à ce problème, il suffit d'initialiser à des valeurs par défaut toutes les variables du programme. Valgrind a aussi détecté de nombreuses fuites de mémoire dans Ucsim. Il faut donc revoir les destructeurs.

Ces bugs ne sont pas difficiles à corriger, mais cela prend un certain temps dont nous manquions. La programmation d'un simulateur de réseau est complexe, mais une fois ces bugs rectifiés, nous devrions pouvoir commencer à observer les trames échangées entre 2 simulateurs.

Le programme d'Ucsim qui a été modifié pendant le stage ainsi que ses mises à jour peuvent être trouvés sur la page github.com/CassyBarnes accessible à tout le monde.

6 – Perspectives

Nous avons réussi pendant ce stage à créer un simulateur pour le CC2530 et à faire fonctionner deux instances du CC2530 simultanément, bien qu'il reste encore quelques bugs à rectifier.

5 mois de stage ne sont pas suffisants pour concevoir un simulateur de réseau complet. C'est pour cela qu'un autre stagiaire à EDF R&D a déjà pris le relai pour continuer à développer ce concept.

La prochaine étape sera de réussir à faire dialoguer deux modules radio de CC2530, et d'observer les trames échangées entre ces deux modules. Par la suite, il faudra rendre possible un dialogue entre un nombre quelconque de noeuds CC2530 en interaction. Une fois que les simulateurs communiqueront tous entre eux, il faudra créer un interfaçage avec un débogueur afin de contrôler le programme simulé. Un générateur aléatoire de scénarios sera nécessaire pour pouvoir tester tous les cas nécessaires à la validation du protocole, même les plus isolés. C'est alors que l'on pourra profiler le code simulé et enfin valider la pile protocolaire d'OCARI.

Ce principe de validation d'un protocole peut en théorie être étendu à tous types de communications sans fils et serait extrêmement pratique pour leur débogage. A la fin du stage, cette méthode de validation semble faisable.

Bibliographie

10 Breakthrough Technologies. (s.d.). Récupéré sur MIT Technology Review:

<http://www2.technologyreview.com/featured-story/401775/10-emerging-technologies-that-will-change-the/2/>

M-IDE Studio for MIDE-51. (2006). Récupéré sur

<http://www.mytutorialcafe.com/Microcontroller%20Compiler%20Assembly%20M-IDE%20Studio%20MCS-51.htm>

GNU Emacs. (2009). Récupéré sur GNU Operating System: <http://www.gnu.org/software/emacs/>

Tags Table. (2009). Récupéré sur GNU:

http://www.gnu.org/software/emacs/manual/html_node/emacs/Tags.html

Workshop de démonstration OCARI et Présentation de l'Alliance OCARI. (2011). *Presentation.*

Valgrind. (2012). Récupéré sur Valgrind: <http://valgrind.org/>

(2013). Récupéré sur Open Virtual Platforms: http://www.ovpworld.org/technology_ovpsim

(2013). Récupéré sur QEMU Open Source Processor Emulator: http://wiki.qemu.org/Main_Page

(2013). Récupéré sur VirtualBox: <https://www.virtualbox.org/>

(2013). Récupéré sur Ubuntu: <http://www.ubuntu.com/>

About. (2013). Récupéré sur Doxygen: <http://www.stack.nl/~dimitri/doxygen/>

C++ Templates. (2013). Récupéré sur TutorialsPoint:

http://www.tutorialspoint.com/cplusplus/cpp_templates.htm

CassyBarnes. (2013). Récupéré sur GitHub: <https://github.com/CassyBarnes>

Intel Hex. (2013). Récupéré sur Wikipedia: http://en.wikipedia.org/wiki/Intel_HEX

The GNU Project Debugger. (2013). Récupéré sur <https://www.gnu.org/software/gdb/>

Universal asynchronous receiver/transmitter. (2013). Récupéré sur Wikipedia:

http://en.wikipedia.org/wiki/Universal_asynchronous_receiver/transmitter

CC2530. (s.d.). Récupéré sur Texas Instruments:

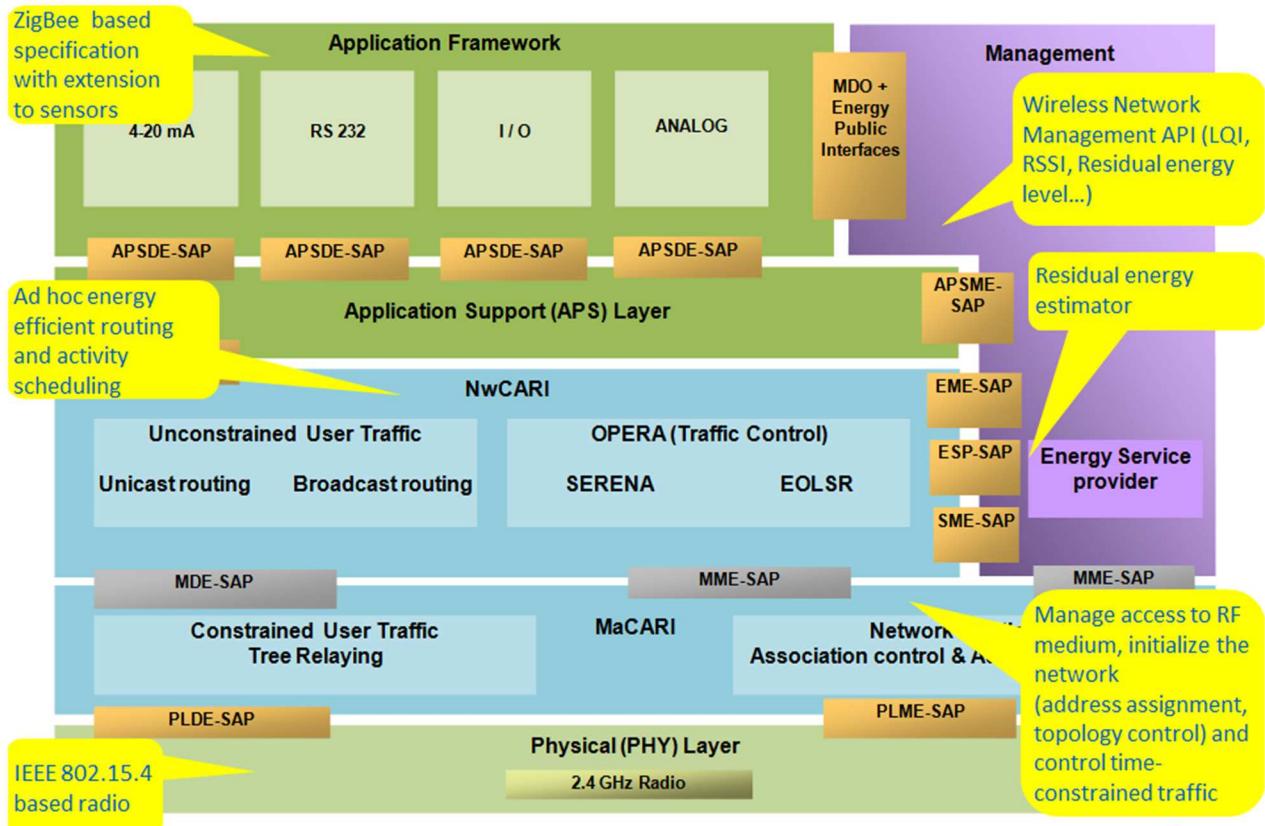
www.ti.com/product/cc2530?DCMP=PPC_Google_TI&k_clickid=0324f131-1398-fb68-e3de-0000565441f9&247SEM=

(2005). Chapter 15 - Memory Mapping and DMA. Dans J. Corbet, A. Rubini, & G. Kroah-Hartman, *Linux Device Drivers.*

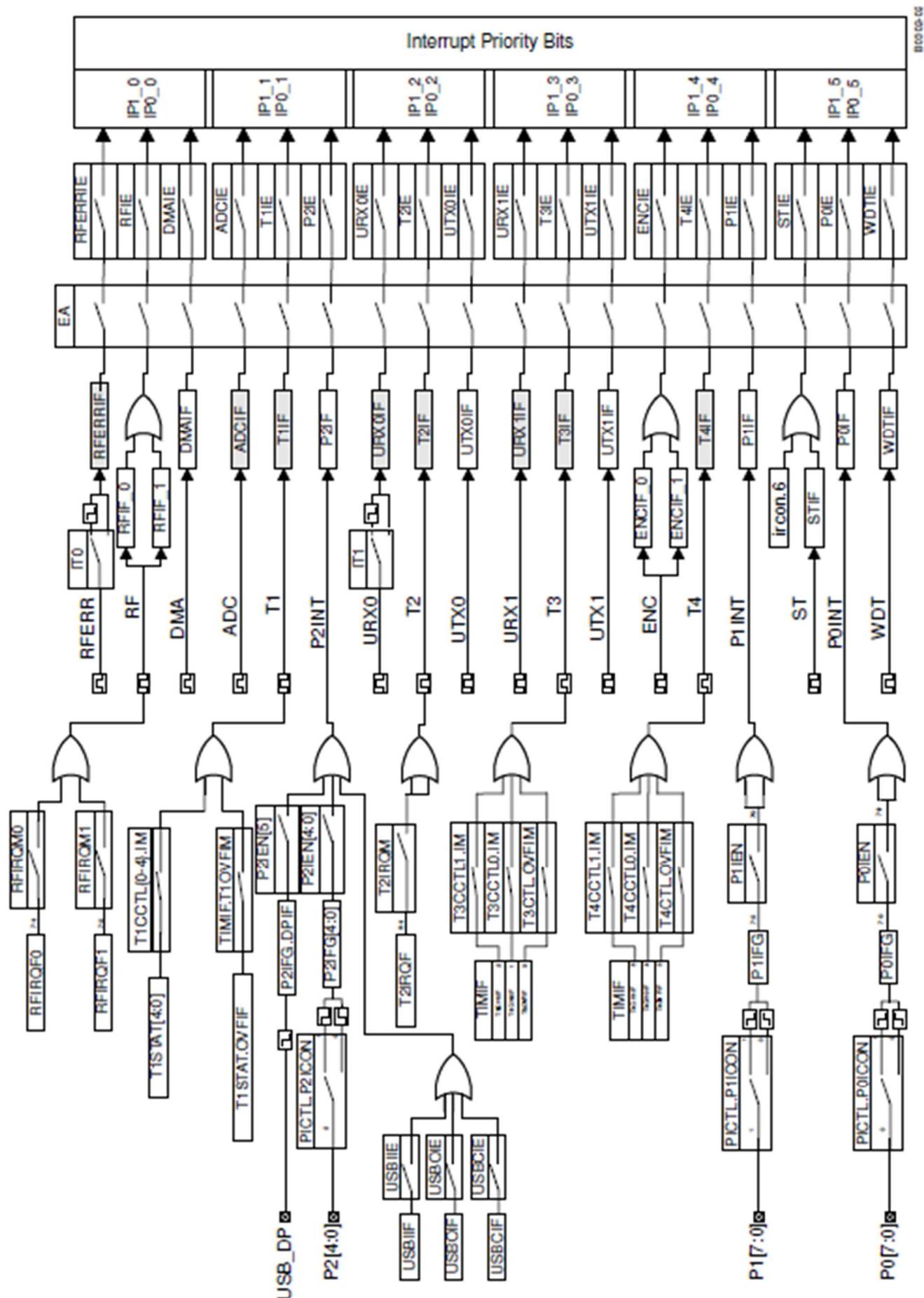
- Direct Memory Access.* (s.d.). Récupéré sur Wikipedia: http://en.wikipedia.org/wiki/Direct_memory_access
- Drotos, D. (1999). *Ucsim Software Simulator for Microcontrollers*. Récupéré sur <http://mazsola.iit.unimiskolc.hu/~drdani/embedded/ucsim/>
- EDF. (s.d.). Livret d'accueil EDF R&D.
- EDF. (s.d.). *Livret d'accueil nouvel embauché EDF*.
- EDF. (s.d.). STEP: Simulation et Traitement de l'Information pour l'exploitation des Systèmes de Production.
- GNU General Public Licence.* (s.d.). Récupéré sur GNU Operating System:
<http://www.gnu.org/licenses/gpl.html>
- IEEE. (2013). Récupéré sur IEEE Standards Association: <http://standards.ieee.org/about/get/802/802.15.html>
- INRIA. (2013). *Energy Efficient Routing Specifications - EOLSR*.
- INRIA. (2013). Node Coloring Specification - OSERENA.
- LIMOS. (2011). MaCARI . *Présentation*.
- OCARI Wireless Sensor Network Alliance.* (s.d.). Récupéré sur www.ocari.org
- Texas Instruments. (2013). CC253x System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and Zigbee Applications. *User's Guide*.
- Vijayvargiya, A. (2013). *An Idiot's Guide to C++ Templates - Part 1*. Récupéré sur Code Project:
<http://www.codeproject.com/Articles/257589/An-Idiots-Guide-to-Cplusplus-Templates-Part-1>

Annexes

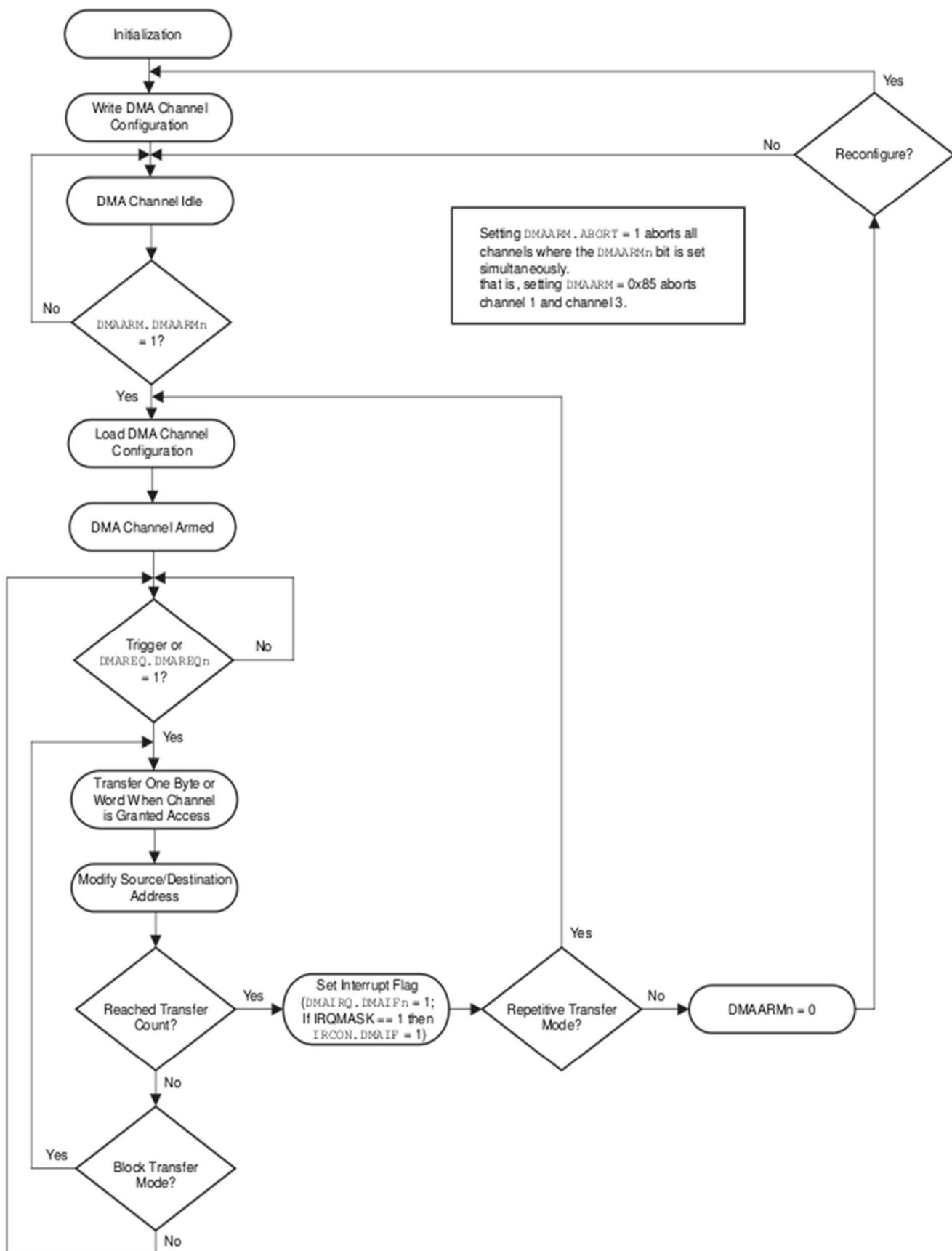
Annexe 1 – Architecture de la pile logicielle OCARI



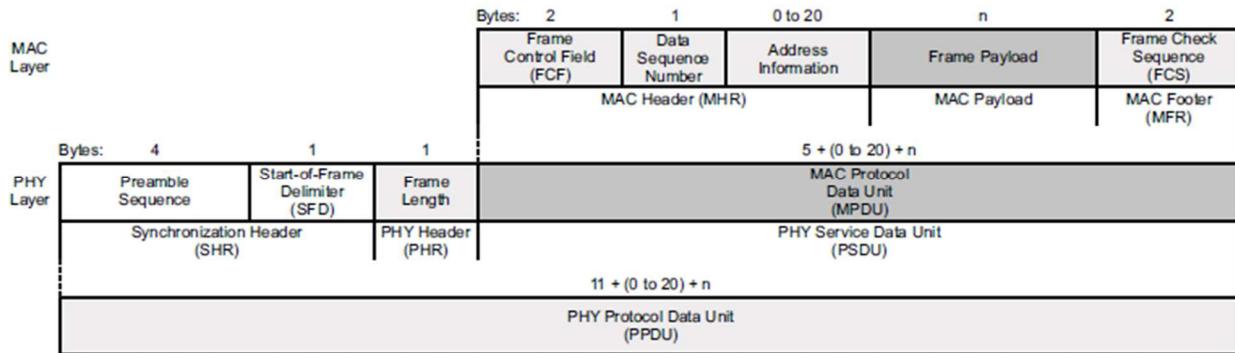
Annexe 2 – Revue des interruptions dans le CC2530



Annexe 3 – Opérations du contrôleur DMA pour le transfert de données



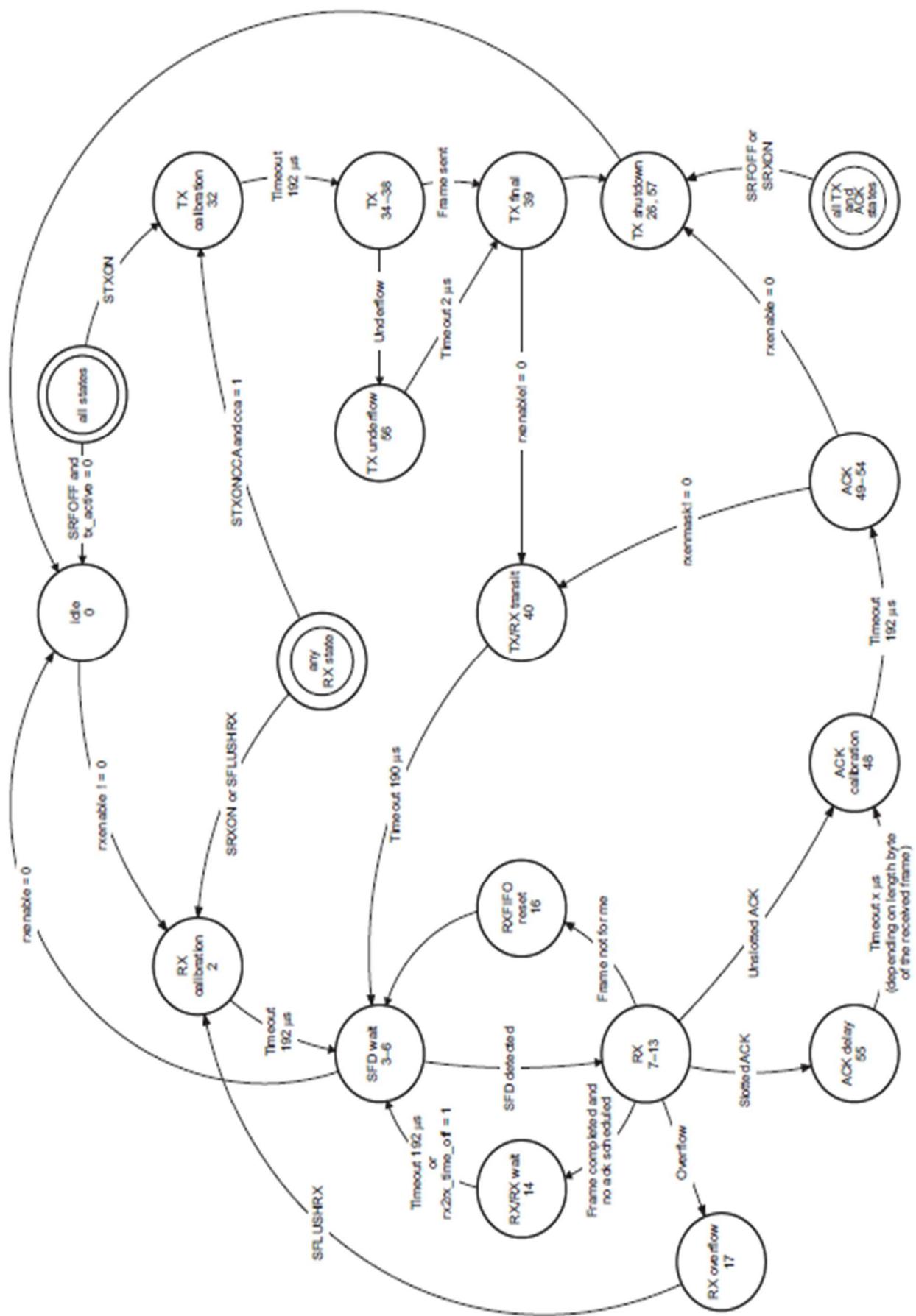
Annexe 4 – Format des trames d'après le standard 802.15.4



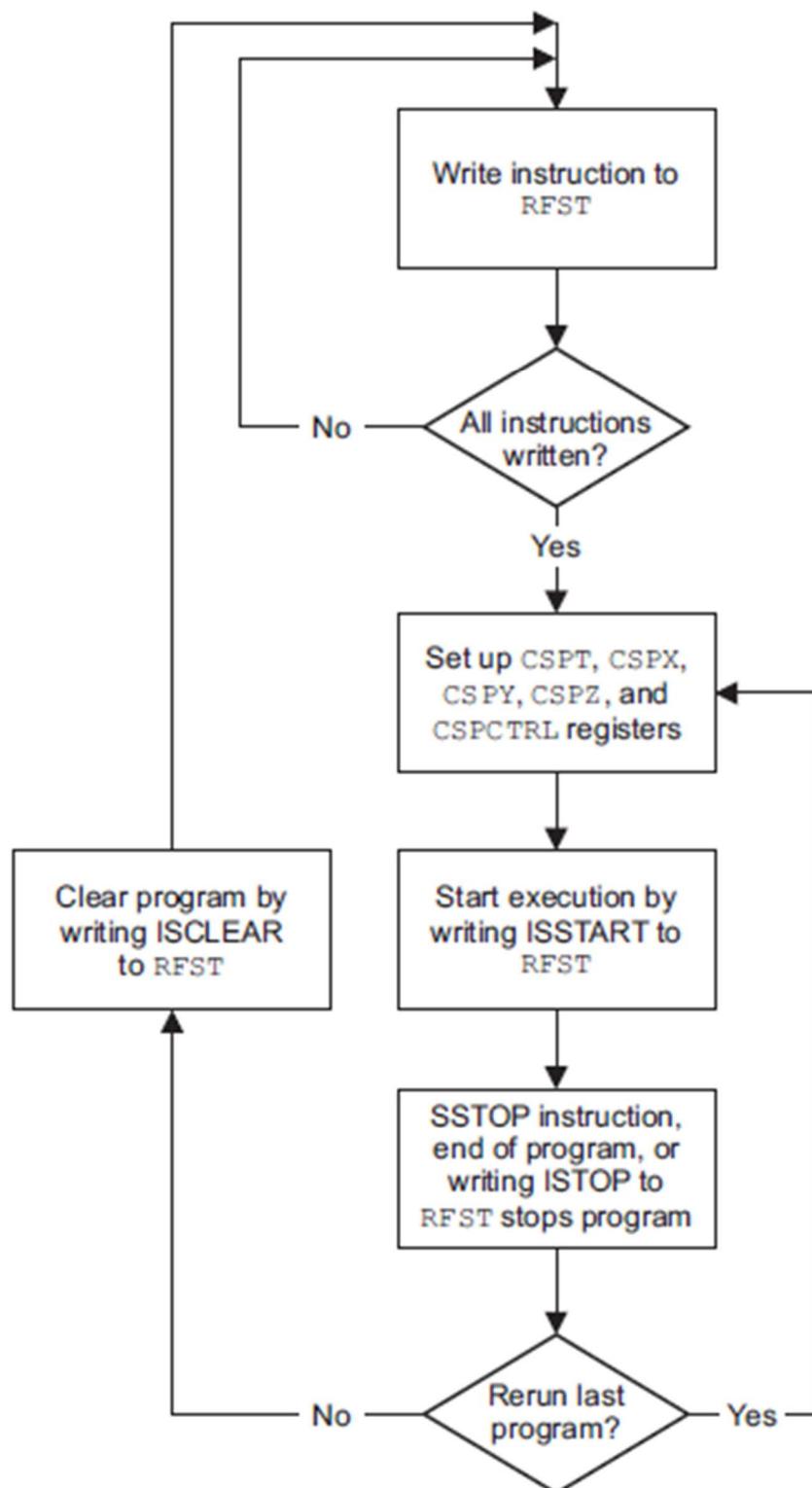
Annexe 5 – Format du Frame Control Field (FCF) d'après le standard 802.15.4

	Bits: 0–2	3	4	5	6	7–9	10–11	12–13	14–15
Frame type	Security enabled	Frame pending	Acknowledge request	Intra PAN	Reserved	Destination addressing mode	Reserved	Reserved	Source addressing mode

Annexe 6 – Finite State Machine Principal de la Radio du CC2530



Annexe 7 – Comment faire fonctionner un programme CSP



Annexe 8 - Les principaux outils utilisés durant le stage

Oracle VM VirtualBox

Oracle VM VirtualBox est un logiciel libre publié par Oracle. Il est installé en tant qu'application sur un OS (Operating System) hôte existant. Cette application sur l'hôte permet à des OS invités (Guest OS) d'être chargés et de tourner, chacun avec son propre environnement virtuel. VirtualBox a été utilisé pendant le stage pour avoir un environnement Linux sur l'ordinateur qui a été fourni par EDF avec pour OS windows.



Ubuntu

Ubuntu est un système d'exploitation libre fondé sur la distribution Linux Debian, et qui utilise Unity par défaut pour son Desktop. C'est la distribution de Linux la plus populaire sur les PC portables. Ubuntu est composé de nombreux packages logiciels, dont la majorité sont libres (sous la licence GNU GPL). Cet OS est idéal pour programmer puisqu'on peut facilement y installer des compilateurs comme GCC (GNU Compiler Collection), ou des logiciels comme EMACS. Ubuntu a donc été installé en tant que machine virtuelle sur l'ordinateur fourni par EDF grâce à VirtualBox.



Emacs

Emacs très populaire parmi les programmeurs. C'est une famille d'éditeurs de texte caractérisés par l'extensibilité de leurs fonctionnalités. C'est l'un des éditeurs de texte les plus puissants et les plus polyvalents, il propose des fonctions comme la manipulation des mots et des paragraphes, la coloration syntaxique et l'indentation pour faciliter la lecture, ou les macros au clavier pour exécuter n'importe quelle séquence définie par l'utilisateur. Emacs a donc été le logiciel que utilisé pour écrire les programmes du simulateur.



The GNU Project Debugger (GDB)

GDB est le débugger standard pour l'OS GNU. Toutefois son usage ne se limite pas seulement à GNU, C'est un débugger portable qui peut tourner sur beaucoup de systèmes de type Unix et qui fonctionne pour de nombreux langages de programmation comme C ou C++. GDB est un logiciel libre, protégé par la Licence Générale Publique de GNU (GPL), qui permet de voir ce qu'il se passe dans un autre programme pendant qu'il s'exécute, ou ce qu'un autre programme



faisait au moment de crasher. GDB offre beaucoup de facilité pour tracer et altérer l'exécution des programmes informatiques. L'utilisateur peut surveiller et modifier les valeurs des variables internes au programme, et même appeler des fonctions indépendamment du comportement normal du programme. GDB a été l'outil principal utilisé pour déboguer le programme du simulateur. Par exemple, quand le programme s'arrêtait en raison d'un *segmentation fault*, GDB permettait de retrouver les dernières instructions que le programme exécutait avant l'erreur, et donc d'en trouver beaucoup plus facilement la source.

Fichier TAGS

Un fichier TAGS est un index de définitions de fonctions, de variables et d'autres caractéristiques de syntaxe intéressante à partir d'un code source. Si on crée un fichier TAGS pour un projet, Emacs peut utiliser ce fichier pour trouver les définitions indexées très rapidement.

Pour la génération d'un tableau de TAGS à partir de tous les fichiers au format .cc et .h, la commande utilisée peut être la suivante :

```
find . \(-name \*.cc -o -name \*.h \) -print | xargs etags
```

Ensuite dans Emacs, lorsque l'on veut retrouver un élément dans le code, il suffit de taper la commande :

alt-x + tags-search + Nom à rechercher

En voici un exemple dans le code du simulateur : Si on recherche dans le fichier TAGS « ::do_interrupt », Emacs ouvrira le premier fichier où il en trouvera une référence. Il ouvre donc le fichier uc51.cc directement au niveau de la définition de la fonction cl_51core::do_interrupt().

Cela permet aussi de retrouver beaucoup plus rapidement d'où viennent certaines variables ou fonctions, et d'en retrouver rapidement la définition. On peut aussi retrouver tous les endroits où une fonction est appelée dans le code. Cela permet un gain de temps considérable, surtout sur un code source comme Ucsim qui est complexe à comprendre, notamment en raison de l'héritage des classes.

Doxygen

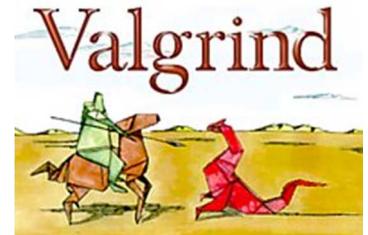
Doxygen est un logiciel gratuit permettant de générer de la documentation à partir du code source d'un programme. Doxygen peut générer de la documentation sur des fichiers, des *namespaces*, des classes, des structures, des *templates*, des variables, des fonctions, des *typedef*, des *enums* et des *define*. Les langages qui peuvent être documentés sont entre autres les langages C/C++, Java, Python, et VHDL. Doxygen génère automatiquement des diagrammes sur les classes et les collaborations en HTML et LaTex.



Doxxygen a été utilisé lors du stage pour mieux comprendre la structure du code d'Ucsim. Doxygen a beaucoup aidé à visualiser la hiérarchie des classes, ainsi que les variables et les fonctions contenues dans chaque classe. Ce logiciel a permis un gain de temps très important.

Valgrind

Valgrind est un système d'instrumentation pour construire des outils d'analyse. Il possède des outils qui peuvent automatiquement détecter des bugs dans la gestion de la mémoire et dans les threads. L'outil le plus utilisé de valgrind est Memcheck. Cet outil insère du code d'instrumentation supplémentaire autour de presque toutes les instructions, cela permet un suivi de la validité et de l'adressabilité de la mémoire.



Memcheck peut détecter entre autre :

- L'utilisation de mémoires non initialisées
- Les lectures et les écritures vers des mémoires qui ont déjà été libérées
- Des fuites de mémoires
- Des accès mémoires en dehors des blocks alloués par la fonction malloc.

Valgrind a été utilisé lors du stage pour trouver l'origine de certaines erreurs de type *Segmentation Fault* qui sont dues aux accès vers des mémoires non valides. Valgrind a permis de trouver qu'il y avait de très nombreux sauts conditionnels dans le programme qui dépendaient de pointeurs non initialisés, et qu'il y a beaucoup de fuites de mémoires dans le code original d'Ucsim.

Git

Git est un logiciel libre de gestion de versions décentralisé. Il a été utilisé lors du stage pour sauver les versions du programme à différentes étapes de sa programmation, chaque sauvegarde est accompagnée de commentaires qui expliquent les nouveaux changements. Il est ensuite possible de revenir à une ancienne version s'il y a un problème, ou de pouvoir observer les différences entre les deux versions du code. Il est possible aussi de créer plusieurs branches, c'est-à-dire que l'on peut avoir deux branches ou plus à partir d'un même code, chaque branche ayant des sauvegardes différentes de modifications du code d'origine. Git permet aussi de sauvegarder les versions de codes sur un site (github.com). Le programme d'Ucsim qui a été modifié pendant le stage pourra ainsi être trouvé sur la page github.com/CassyBarnes, où il est accessible à tout le monde.

