# Reconfiguration dynamique et simulation fine modélisée au niveau de transaction dans les réseaux de capteurs sans fil hétérogènes matériellement-logiciellement

Mihai Galos

ÉCOLE CENTRALE LYON

ÉCOLE DOCTORALE Électronique, Électrotechnique, Automatique

Institut de Nanotechnologie de Lyon

Année : 2012 Thèse numéro : 2012-43

# Thèse

pour obtenir le titre de

DOCTEUR D'ÉCOLE CENTRALE DE LYON

Présentée et soutenue par

Mihai GALOS

le Jeudi, 15 Octobre 2012

## Reconfiguration dynamique et simulation fine modélisée au niveau de transaction dans les Réseaux de Capteurs sans fil hétérogènes matériellement/logiciellement

*Thèse dirigée par Fabien Mieyeville et co-encadrée par David Navarro*

Jury :

| | | |
|---|---|---|
| **M. AUGUIN** | Directeur de Recherche Université de Nice-Sophia Antipolis | **Rapporteur** |
| **R. LEVEUGLE** | Professeur Laboratoire TIMA Grenoble | **Président** |
| **F. MIEYEVILLE** | Maître de Conférences HDR - Ecole Centrale de Lyon | **Examinateur** |
| **D. NAVARRO** | Maître de Conférences - Ecole Centrale de Lyon | **Examinateur** |
| **F. PECHEUX** | Maître de Conférences HDR - Université Pierre et Marie Curie Paris | **Examinateur** |
| **I. O'CONNOR** | Professeur - Ecole Centrale de Lyon | **Examinateur** |

Familiei mele, actuale şi viitoare.

# Remerciements

Je veux remercier mon équipe, Fabien Mieyeville, David Navarro et Ian O’Connor pour leur support continu et leur patience avec mon travail et mon français durant cette thèse. Merci à mes amis et anciens collègues Kotb Jabeur, Natalia Yakimets pour les questions plus profondes de la vie. Corrado Sciancalepore, Alexandra Pavlova, merci pour avoir été à mes côtés pendant les périodes les plus difficiles de ces trois ans.

Dernièrement, merci à la France pour avoir réuni les conditions qui ont mené au succès de ce travail.

# Résumé (Français)

Pendant les dernières années, des unités intelligentes, qui communiquent entre eux par radio-fréquence ont été développées. Ces unités ont la possibilité de capter une mesure analogique de l'environnement, éventuellement prendre une décision au niveau de région locale et la transmettre à ses voisins. Elles forment un réseau, connu dans la littérature comme Réseau de Capteurs sans Fil (WSN). Peu à peu, d'une unité à une autre, l'information peut atteindre l'utilisateur qui pourrait interpréter ces données.

Ces systèmes ont premièrement attiré l'attention du domaine militaire, mais ils commencent à être déployés dans les applications civiles. Les exemples incluent des réseaux capables d'aider les pompiers-sapeurs éteindre les incendies et peuvent même être présents dans les ponts dans les régions susceptibles aux tremblements de terre. Grâce à eux, la structure de résistance des ponts peut être vérifiée.

Les buts des WSNs sont les suivants : (i) déterminer la valeur d'un ou plusieurs paramètres physiques à un endroit, (ii) détecter occurrence des événements et estimer des paramètres des événements captés, (iii) classifier un objet détecté et (iv) suivre un objet. Par conséquent, les besoins particuliers d'un WSN sont les prochaines : (i) utilisation d'un grand nombre de capteurs, (ii) fixer des capteurs stationnaires, (iii) avoir une faible consommation énergétique, (iv) être capable de s'auto-organiser, (v) faire du calcul distribué sur le signal, (vi) possibilité de requête de calcul singulier.

Comme déjà dit, les unités peuvent être déployées ad-hoc, elles ont la propriété de pouvoir s'auto-organiser. Elles vont décider d'une manière indépendante

qui, quand et par quelle route va transmettre ses données, sans intervention humaine à posteriori.

Ces unités sont dénommées “noeuds” et sont typiquement constituées d’un microcontrôleur, un capteur, une interface radio et une batterie. La batterie est la ressource la plus importante sur le noeud puisque souvent elle est irremplaçable. Le noeud peut se trouver dans des endroits où l’accès physique serait difficile ou indésirable, comme dans le corps humain même, par exemple. C’est pour ainsi qu’une faible consommation est désirée.

Les critères de consommation faible et grand nombre des noeuds impliquent des architectures matérielles moins puissantes présentes sur les noeuds, ayant un coût inférieur. Ceci pose des limitations sur le logiciel qui sera exécuté sur ces noeuds.

Le logiciel qui s’exécute sur le microcontrôleur des noeuds doit parfois être modifié. A cause du fait que le noeud est considéré physiquement inaccessible, ceci doit être fait à travers l’interface radio. Dans la littérature, ce genre de mise-à-jour s’appelle “reconfiguration dynamique”. Pour modéliser comment certains logiciels se comporteraient dans un contexte plus grand qu’au niveau seul, la simulation au niveau du réseau serait nécessaire.

Cette thèse s’adresse aux problématiques de reconfiguration et simulation dans le contexte des WSN hétérogènes, amenant deux contributions : MinTax et IDEA1TLM.

MinTax est une nouvelle approche de reconfiguration dynamique qui est composée de deux parties : un nouveau langage de programmation et un compilateur qu’on a développé pour le langage. Ensuite, au lieu de compiler la mise-à-jour sur le PC et l’envoyer au noeud, on l’écrit directement dans MinTax et on l’envoie au noeud pour qu’il la compile lui-même et qu’il génère le code-objet correspondant à l’architecture cible.

IDEA1TLM a été développé en vue de quantifier les performances de MinTax au niveau de réseau WSN et supporte la modélisation du protocole 802.15.4 “slotted” et “undslotted”.

Le manuscrit de cette thèse est organisé en quatre chapitres.

Le premier chapitre offre une introduction dans le monde des WSNs, se concentrant surtout sur la partie logicielle. L’état d’art concernant la reconfiguration dynamique ainsi que sur le développement des compilateurs sont aussi introduits.

Le deuxième chapitre s’adresse à MinTax. Les besoins concernant la reconfiguration dynamique ont été premièrement identifiés et pris comme spécification pour ce langage. Une fois le langage établi on l’a utilisé comme prémise pour le compilateur. Les différentes clauses supportées par le compilateur, ainsi que sa structure interne, les architectures matérielles et les systèmes d’exploitation pour lesquels MinTax offre du support sont décrits dans ce chapitre.

Le troisième chapitre traite IDEA1TLM. Cette plate-forme d’exploration des WSN permet de simuler le comportement matériel des noeuds au niveau de cycle d’horloge de bus près, ainsi que le comportement du logiciel à travers un simulateur du jeu d’instructions intitulé BLISS. La structure interne du modèle logiciel du noeud est décrite ici, ainsi que des informations pertinentes à la simulation du logiciel.

Le quatrième chapitre représente les conclusions et perspectives et achèvent le manuscrit.

**Chapitre 1 : Introduction**

Les réseaux de capteurs sans fil (WSN), sont composées par des noeuds intelligents à une faible puissance de calcul et une faible consommation. Leur utilisation commence à se faire présente dans les applications civiles, pour capter des mesures analogiques de l’environnement, comme la température, le mouvement ou la vibration. La ressource la plus importante du noeud est la batterie. C’est pour ainsi que

la façon dont elles communiquent et se comportent peut affecter leur durée de vie. Ces paramètres sont gérés par l'unité de calcul de noeud, le microcontrôleur et dans une manière plus petite par l'interface radio.

La Section 1.1 présente la structure interne du noeud et les architectures matérielles les plus utilisées. La segmentation du logiciel qui s'exécute sur le noeud est présenté aussi ici.

Dans la Section 1.2 la reconfiguration dynamique est introduite comme étant la mise-à-jour dynamique, après le déploiement physique du noeud.

Les modèles de programmation sont traités dans la Section 1.3. Ils consistent en approches en code machine ou bytecode pour les Machines Virtuelles. Les approches en code machine peuvent être pour liées à un système d'exploitation modulaire ou monolithiques. Les Machines Virtuelles et l'interprétation du bytecode sont introduits et ensuite les approches basées sur la différence "Diff". La section s'achève avec une comparaison entre les solutions présentées.

La Section 1.4, 1.5 et 1.6 introduit le lecteur dans la thématique de compilation et argumente pourquoi une approche qui suppose la génération du code-objet directement sur le noeud serait plus efficace.

**Chapitre 2 : MinTax**

Ce chapitre débute avec les besoins particulières d'une solution pour la configuration dynamique. Ceci a été compris dans un cahier de charges, étant la base de notre travail.

La Section 2.2 offre les principes sur lesquels MinTax se base, ainsi que les clauses supportées par notre langage. Les principes concernant la coexistence des fonctionnalités entre plusieurs reconfigurations dynamiques sont présentés aussi.

Dans la Section 2.3 la structure interne du compilateur MinTax est présentée. Pour chaque bloc composant, les outils et les processus utilisés sont décrits en détail.

Pour que la mémoire utilisée pour la compilation soit à nouveau disponible aux applications finales, elle est allouée dynamiquement par le compilateur de MinTax. L'allocation classique de mémoire utilise la fonction malloc(). Un tel usage ne serait pas optimale, puisqu'on sait que le compilateur allouera et désallouera la mémoire dans un ordre prédéfini.

La Section 2.5 met en contexte les différences existantes entre les différentes architectures matérielles supportées, liées à l'allocation de mémoire.

Ensuite, 2.6 explique plus en détail comment la mémoire flash et RAM est utilisée pendant le processus de compilation et après.

L'Analyseur Lexicale, responsable de couper le fichier d'entrée en unité dénommées "atomes" est présenté dans la Section 2.7. Les différents types d'atomes possibles, ainsi que les paramètres maximaux "par défaut" sont présentés ici.

Le chapitre continue avec l'Analyseur Sémantique dans la Section 2.8. La grammaire de MinTax est présentée ici. Elle est responsable de l'ordre des atomes dans le fichier d'entrée. La grammaire est décrite à l'aide des règles écrites pour un générateur d'analyseurs sémantiques. Ces règles peuvent être récursives ou non pas.

Dans la Section 2.9, l'Alloueur de variables est décrit. Celui-ci va assigner un nom de variable du fichier d'entrée à un registre. S'il n'y a plus de registre disponible, la variable sera allouée dans la mémoire RAM.

Ensuite, dans la Section 2.10, le processus de génération de code est expliqué. Le résultat sera premièrement écrit dans une mémoire tampon de sortie, pour être écrite dans la mémoire flash à la fin du processus de compilation.

La Section 2.11 traite l'imbrication en MinTax, ou bien, l'emboîtement des clauses et comment le compilateur les interprète.

Ensuite, les Sections 2.12 - 2.18 explique les différentes clauses possibles en

MinTax, comme les clauses itératives ou conditionnels, l'accès direct aux entrées et sorties et la structure d'une fonction.

Dans la Section 2.19, on introduit les plate-formes matérielles (microcontrôleur et interface radio) et logicielles (systèmes d'exploitation) supportées par notre solution.

Étant une solution visant plusieurs architectures HW-SW, dans la Section 2.20 on démontre pour la première fois une approche qui offre du support complet hétérogène dans le domaine WSN.

Les résultats commencent avec la Section 2.21, où on fait la quantification des performances de notre approche à travers plusieurs algorithmes, optimisations et architectures matérielles. Pour démontrer que l'approche a bien des correspondances avec le but originaire, celui de minimiser le temps de transmission radio et grâce à cela, l'énergie pour une reconfiguration dynamique, les mesures d'oscilloscope sont présentées ici. Ensuite, on a comparé notre solution avec des autres existantes dans la littérature et démontré des performances supérieures à travers des algorithmes simples et complexes.

Le chapitre s'achève avec une conclusion et les perspectives pour le langage de programmation, ainsi que pour le compilateur. En vue de développement reste un traducteur de langage C à MinTax, vu les similarités de sémantiques et syntaxe entre les deux langages.

**Chapitre 3 : BLISS et IDEA1TLM**

Les mesures faites ont prouvé la supériorité de MinTax par rapport aux autres existantes. Ensuite, il était temps de simuler le comportement d'un noeud équipé avec ce compilateur dans le contexte d'un réseau. Pour ceci, on voulait utiliser un simulateur de réseau. Le candidat idéal était IDEA1, développé par notre équipe. Mais pour pouvoir simuler MinTax en IDEA1, il nous a fallu dans un premier temps in-

tégrer le comportement logiciel en IDEA1. De plus, on a décidé de simuler le comportement matériel au cycle d'horloge près. C'est pour ainsi qu'on a implémenté une nouvelle version d'IDEA1, basée sur l'ancienne, qu'on a intitulée IDEA1TLM.

La Section 3.4.1 présente l'aperçu global sur l'architecture logicielle d'IDEA1TLM. Les différents bloques composants, ainsi que la façon dont ils communiquent est expliquée ici.

Comme la Section 3.4.2 explique, IDEA1TLM offre du support pour deux cas : soit le microcontrôleur gère la communication par radio-fréquence, soit elle est gérée par l'interface radio elle-même.

Dans la prochaine section, le modèle pour l'émetteur-récepteur radio CC2420 est décrit dans la réception et transmission.

La Section 3.4.4 introduit le modèle de l'émetteur-récepteur MRF24J40 et présente ces états internes à l'aide d'un diagramme d'états et événements.

Le modèle pour le module de consommation est formulé dans la Section 3.4.5, pour achever avec la sortie de la simulation et donner des exemples des résultats produits avec IDEA1TLM.

BLISS, présenté dans la Section 3.5 est un simulateur de jeu d'instructions qui vise plusieurs architectures matérielles et qui est écrit en C++, pour offrir de la compatibilité avec IDEA1TLM. BLISS prend comme entrée la sortie d'un compilateur pour les architectures visée, en format ELF. Ensuite, il va décoder le fichier ELF, regarder à quelle instruction il a affaire et commencer à exécuter la fonctionnalité présente dans l'ELF. A la fin, il va produire un fichier de sortie qui comprend un tableau de correspondance fonction-nombre de cycle d'horloge. En utilisant ce tableau et connaissant la fréquence d'horloge du microcontrôleur, ces cycles peuvent être convertis en temps qu'on peut introduire dans la simulation pour un certain état.

Les résultats pour BLISS sont présentés dans la Section 3.6 et la Section 3.7 démontres des résultats BLISS en conjonction avec IDEA1TLM.

Le chapitre s'achève avec les conclusions et perspectives dans la Section 3.8.

**Chapitre 4 : Conclusion**

Le dernier chapitre du manuscrit est décomposé en deux parties, qui réitèrent les conclusions de chaque partie déjà présentée, une conclusion sur la reconfiguration dynamique et une sur la simulation dans les WSNs.

# Summary (English)

During the last years, intelligent units, which are able to communicate with one another using a radio interface have been manufactured. These units have the possibility to take analogue measurements from their environment, possibly make a decision pertaining to them (at local level) and transmit them to their neighbours. They form a network, the name of which is Wireless Sensor Networks in the literature. Slowly, from hop to hop, the information can reach the end-user who may interpret the results.

These systems have at first attracted the attention of the military field as immediate applications in trespasser monitoring, but they are now evermore present in civilian use. The examples include networks capable of assisting fire-fighters against bush fires (temperature monitoring) and may even be present in the structure of bridges in earthquake-sensitive areas. Thanks to them, the structural integrity of the object can be monitored and deemed safe or not.

The goals of WSNs are the following : (i) determine the value of one or multiple physical parameters of the immediate region, (ii) detect the occurrence of events and estimate the parameters of the observed events, (iii) classify a detected object and (iv) follow an object. As a consequence, the particular needs of WSNs are the following : (i) the use of a large number of sensors, (ii) implement stationary sensors, (iii) have a low energy footprint, (iv) being capable of auto-organising themselves, (v) make a distributed computation on the signal, (vi) implement the possibility of a single signal query.

As already mentioned, the units may be deployed ad-hoc, having the property of auto-organising themselves. They will decide independently who, when and through which route will sends its data, without human interference.

These units are called “nodes” and are typically comprised of a microcontroller, a sensor, a radio communication interface, and a battery. The battery is the most important resource the node has, because, as often is the case, it is irreplaceable. The nodes bearing a battery employ a fire-and-forget philosophy, meaning they won’t be physically accessible after deployment. The node may find itself in places where the physical access is difficult or undesirable, like inside the human body for example. This is why why the low-power consumption is not only desired, but also necessary.

The criteria for low-power consumption and a large node count imply less powerful hardware architectures, yet they also stand for a reduced price point. This, in turn, poses limitations to the software running on these types of nodes.

The software may need to be updated from time to time to account for new features or remove existing ones. Owing to the fact that the node is considered physically inaccessible, this has to be done using the radio interface. In the literature, this type of update is called “dynamic reconfiguration”. In order to model how certain types of software would behave in a much broader context, a network-broad simulation is required.

The PhD addresses the two topics at a time : first the dynamic reconfiguration, then the simulation in the context of heterogeneous WSNs, bringing forth two contributions : MinTax and IDEA1TLM.

MinTax is a new approach towards dynamic reconfiguration and is composed of two parts : a new high-level programming language specifically tailored for the task at hand and its corresponding compiler. Next, instead of compiling the update on the computer and sending it to the node, we write the update directly using

MinTax and send it to the node. The compiler will receive, compile and produce the associated object code for the target hardware-architecture.

IDEA1TLM was developed so as to quantify the performances of MinTax at network level and supports the modelling of 802.15.4 “slotted” and “unslotted” protocol.

The manuscript is organised in four chapters.

The first chapter offers an introduction to the world of WSNs and concentrates itself more on the software side. The state-of-the-art pertaining to the dynamic reconfiguration and compiler development is thoroughly analysed.

The second chapter addresses MinTax. First of all, the requirements relevant for the dynamic reconfiguration have been identified and have been considered as specification for the new solution. Secondly, the different clauses and constructs supported by the compiler, as well as its internal structure, the supported hardware platforms and operating systems are all described within this chapter.

The third chapter details IDEA1TLM. This WSN exploration platform allows users to simulate behavioural characteristics at hardware level with bus-cycle accuracy and at software level using an instruction-set simulator named BLISS.

The fourth an final chapter represents the conclusions.

**Chapitre 1 : Introduction**

Wireless Sensor Networks are composed of intelligent nodes, having low computational capabilities and a low energy footprint. Their use is starting to be present in civilian applications, to perform physical measurements on their environment, for example to measure temperature, movement or temperature. The most precious resource on the node is its battery. Which is why the way they communicate and behave may affect their lifespan. These parameters are handled by the computing unit of the node, most often a microcontroller and on a much lesser scale, the radio transceiver.

Section 1.1 presents the internal structure of a node and the supported hardware architectures. The decomposition of software in different logic levels is also presented here.

In Section 1.2, the concept of dynamic reconfiguration is introduced, being the dynamic software update, after the physical node deployment.

The different programming paradigms are presented in Section 1.3. They consist of approaches based on machine code which may or may not be linked to a modular or monolithic Operating System. Virtual Machines and bytecode interpretation is next introduced, followed by solutions based on difference, also known as “diff-based approaches”. The section finishes with a comparison between the different solutions.

Section 1.4, 1.5 and 1.6 offer the reader an introduction in the field of compiler construction and arguments why an approach based on the generation of object code would be better and more appropriate in this context.

**Chapitre 2 : MinTax**

This chapter starts out by presenting the specific requirements related to the solutions for dynamic reconfiguration. These have been summed up and constitute the basis of our solution.

Section 2.2 offers the foundation on which MinTax is based, as well as the clauses supported by it. The principles of software component coexistence across different dynamic reconfigurations on the same microcontroller are also laid out.

In Section 2.3, the internal structure of the MinTax component is presented. For each comprising block, the tools and the processes used to generate the intermediate files which ultimately lead to compiling the MinTax compiler are explained.

So as the RAM memory used to compile the update to be freed once the compilation is overdue, it is dynamically allocated by the MinTax compiler. Normally, the classic memory allocation function malloc() is used to perform dynamic

memory allocation. Such a doing would not suit the MinTax compiler to the best of its interests, because of the intermediate metadata which leaves “holes” in the RAM. Thus, Section 2.4 deals with the intricacies of malloc().

Section 2.5 puts into context the existing differences between the supported hardware architectures, in relation to the aforementioned section.

Afterwards, Section 2.6 explains in greater detail the flash and RAM memory maps during the compilation process and after.

The Lexical Analyser, responsible for splitting the input file into indivisible atoms is presented in Section 2.7. The possible atoms types are also presented here, alongside the “default” maximal parameters for this step.

The chapter continues with the Semantic Analyser in section 2.8. Here, the grammar for MinTax is laid out. It is responsible for the order of the atoms in the input file. The grammar is described using rules for a semantic analyser generator. These rules may or may not be recursive.

Section 2.9 ensues with the Variable Allocator. This will allocate a variable from the input file to a register. When no more register are available, the variable will be allocated to the RAM.

Next, in section 2.10, the process of code generation is explained. The result will at first be output to a temporary buffer in the RAM. Later, when the compilation has finished, it will be written to the flash memory.

Section 2.11 handles nesting in MinTax.

The following Sections (2.12 - 2.18) explain the different possible clauses that may come about in MinTax programs, like iterative or conditional clauses, direct port access to inputs and outputs.

In Section 2.19, the supported hardware (microcontroller and radio interface) as well as software platforms (operating systems supported by our solution) are

presented.

MinTax being a solution targeting multiple HW-SW architectures, in Section 2.20, we argue that for the first time, a truly fully-heterogeneous dynamic reconfiguration solution has been built for WSNs.

The results start out with Section 2.21, where we quantify the performances of our approach over multiple metric-algorithms, optimisations and hardware architectures. In order to show that this approach has in fact fulfilled its original purpose, that of minimising the radio transmission time and by doing so, the overall energy requirements for an dynamic-update, oscilloscope measurements are presented. Next, we compare our solution against others', present in the literature and show that our solution promises superior throughput across simple and complex algorithms.

The chapter ends with a conclusion and the perspectives for the MinTax programming language and its compiler. The next immediate step would be to write a C-to-MinTax transcoder, a trivial task thanks to the semantic and syntax similarities between the two languages.

**Chapitre 3 : BLISS and IDEA1TLM**

The physical measures taken with the oscilloscope have proven MinTax to be a superior solution to its competitors. Next, the behavioural simulation of the node equipped with the MinTax compiler was needed. A network simulator was proposed, it being IDEA1, developed by out team. But in order to be able to simulate MinTax in IDEA1, we first needed to integrate a software behavioural model in IDEA1. Furthermore, we decided to simulate the hardware behaviour at bus cycle accuracy. Which is why we opted out for a new version of IDEA1, based on the original, which we call IDEA1TLM.

Section 3.4.1 presents a global overview of the IDEA1TLM software architecture.

The different component blocks, as well as the fashion in which they communicate, is explained here.

Like Section 3.4.2 explains, IDEA1TLM offers support for two cases : (a) When the microcontroller handles the radio communication and (b) when the communication is handled by the radio interface.

In the next section, the model for the radio transceiver interface CC2420 is presented.

Section 3.4.4 introduces the model for the MRF24J4O radio transceiver and presents its internal state machine using state and event diagram.

The model for the consumption block is formulated in Section 3.4.5, to finish off with the results produced by the simulation.

BLISS, presented in Section 3.5 is an instruction set simulator that targets multiple hardware architectures and which is written in C++ so as to offer compatibility support with IDEA1TLM. BLISS takes as input the ELF file produced by the compiler. Next, it will decode the ELF, look at which instruction is currently in scope and start executing the functionality. At the end, BLISS will produce an output file consisting of a lookup table pair : name of function - number of corresponding clock cycles. Using this lookup table and knowing the clock frequency of the microcontroller, these cycles may be translated into wait periods which can in turn be inserted in the simulation, corresponding to a specific state in the statemachine.

The results for BLISS are tackled in Section 3.6 and Section 3.7 offers insight on results referring to BLISS in conjunction with IDEA1TLM.

The chapter finishes off with the conclusions and perspectives in Section 3.8.

**Chapitre 4 : Conclusions**

The last chapter of this manuscript is further broken down into two parts, which reiterate the partial conclusions od the previous chapters (dynamic reconfiguration

and simulation) so as to build a general conclusion.

# Liste des Publications

1. A Cycle-Accurate Transaction-Level Modelled Energy Simulation Approach for Heterogeneous Wireless Sensor Networks : M. Galos, D. Navarro, F. Mieyeville, I. O Connor, 10th IEEE International NEWCAS Conference 16-21 Juin 2012 - Montréal, Canada
2. Reconfiguring Hardware-Software Wireless Sensor Networks : M. Galos, F. Mieyeville, D. Navarro, I. O Connor, the 14th International Symposium on Wireless Personal Multimedia Communications 3-7 Oct 2011 - Brest, France
3. Energy-aware Software Updates in Heterogeneous Wireless Sensor Networks : M. Galos, D. Navarro, F. Mieyeville, I. O Connor, 9th IEEE International NEWCAS Conference 26-29 June 2011 - Bordeaux, France
4. Dynamic reconfiguration in Wireless Sensor Networks : M. Galos, F. Mieyeville, D. Navarro17th IEEE International Conference on Electronics, Circuits, and Systems ICECS 11-15 Dec 2010 - Athènes, Grèce
5. Dynamic reconfiguration for software and hardwareheterogeneous real-time WSN : F. Mieyeville, M. Galos, D. Navarro, the 6th International conference on sensor technologies and applications Sensorcomm 19 - 24 august 2012 - Rome, Italie
6. Heterogeneous Wireless Sensor NetworkSimulation : D. Navarro, M. Galos, F. Mieyeville, W. Du, the 6th International conference on sensor technologies and applications Sensorcomm 19 - 24 august 2012 - Rome, Italie
7. Towards a Design Framework for Heterogeneous Wireless Sensor Networks : D. Navarro, F. Mieyeville, W. Du, M. Galos, I. O Connor ISAS - Yokohama, Japon 17-19 juin 2011
8. Design Framework for Heterogeneous Hardware and Software in Wireless Sensor Networks : D. Navarro, F. Mieyeville, W. Du, M. Galos, I. O'Connor, 6th International conference on systems and networks communications ICSNC october 23-29, 2011 - Barcelona, Spain

# Table des matières

# Table des figures

# Liste des tableaux

# 1 Introduction

## 1.1 Présentation des Réseaux de Capteurs sans Fil

### 1.1.1 Principes

Les unités sans fil, qui communiquent l'une avec l'autre par radio-fréquence, typiquement à travers des réseaux muli-hop, s'appellent, d'une manière générale, réseaux ad-hoc sans fil (WANET)(12). Les WANETs peuvent être utilisés là où une communication filaire n'est pas faisable. Par exemple, pour offrir la possibilité de communication durant des événements spéciaux (comme les conférences, expositions, concerts, etc.), ou dans les environnements hostiles, les WANETs sont particulièrement efficaces. Les réseaux de capteurs sans fil (Wireless Sensor Networks ou WSNs pour les anglosaxons) sont une classe spéciale de WANET. Les WSNs sont promis à faire partie intégrante de notre vie, autant que les téléphones portables ou les ordinateurs aujourd'hui.

Les réseaux de capteurs sans fil ont attiré beaucoup d'intérêt pour le développement d'un nouveau type de systèmes embarqués. Il s'agit d'un réseau ad-hoc, avec un grand nombre de noeuds qui sont capables de récolter des informations analogiques dans leur environnement immédiat. Ils ont été créés pour faire face aux coûts élevés associés aux câbles, ainsi que pour pouvoir être déployés dans des régions où l'accès est difficile.

Ils sont en train de devenir une solution très utilisée dans l'industrie dans des domaines tels que l'avionique (13), biomédical (14) et automobile (15).

Ces noeuds sont considérés comme des entités intelligentes qui sont dispersés dans une région, pour pouvoir servir à une tâche globale orientée vers une application (12). Chaque entité (noeud) est de petite taille et de faible coût, pour qu'il puisse être déployé en grande quantité. Chaque noeud est un dispositif capable de s'interfacer avec des capteurs matériels, d'effectuer des calculs sur les données captées et communiquer ses informations sans fil (typiquement via une interface radiofréquence) à un autre dispositif (16). Ces réseaux peuvent avoir des centaines ou des milliers de noeuds. Ils peuvent être distribués dans une région géographique très grande (la taille d'un pays, par exemple) ou sur une surface plus réduite, telle que le corps humain.

Les applications qui ont besoin d'une valeur approximative d'une région, comme le glissement de terrain (17) (18) (19), agriculture (20) (21) (22), le mouvement des courants océaniques (23) (24) (25) (parmi beaucoup d'autres) ont commencé à implémenter ce type de réseau. Les sinistres naturels peuvent être prévus bien en avance (comme les vagues du tsunami) (26), ou séismes et contre-mesures peuvent être mis en oeuvre. Des applications militaires sont envisageables, par exemple la détection d'intrusion (27) (28) (29).

Les buts des WSNs sont les suivants : (i) déterminer la valeur d'un ou plusieurs paramètres physiques à un endroit, (ii) détecter l'occurrence des événements et estimer des paramètres des événements captés, (iii) classifier un objet détecté et (iv) suivre un objet. Par conséquent, les besoins particuliers d'un WSN sont les prochaines : (i) utilisation d'un grand nombre de capteurs, (ii) fixer des capteurs stationnaires, (iii) avoir une faible consommation énergétique, (iv) être capable de s'auto-organiser, (v) faire du calcul distribué sur le signal, (vi) possibilité de requête de calcul singulier.

Les noeuds sont normalement dispersés sur un champ de capteurs (sensor field), comme montré dans la Figure 1.1.

Chacun de ces noeuds a la possibilité de collecter des données et de les diriger à travers le réseau jusqu'à un noeud dénommé "sink" ou "noeud collecteur", là où elles sont accessibles

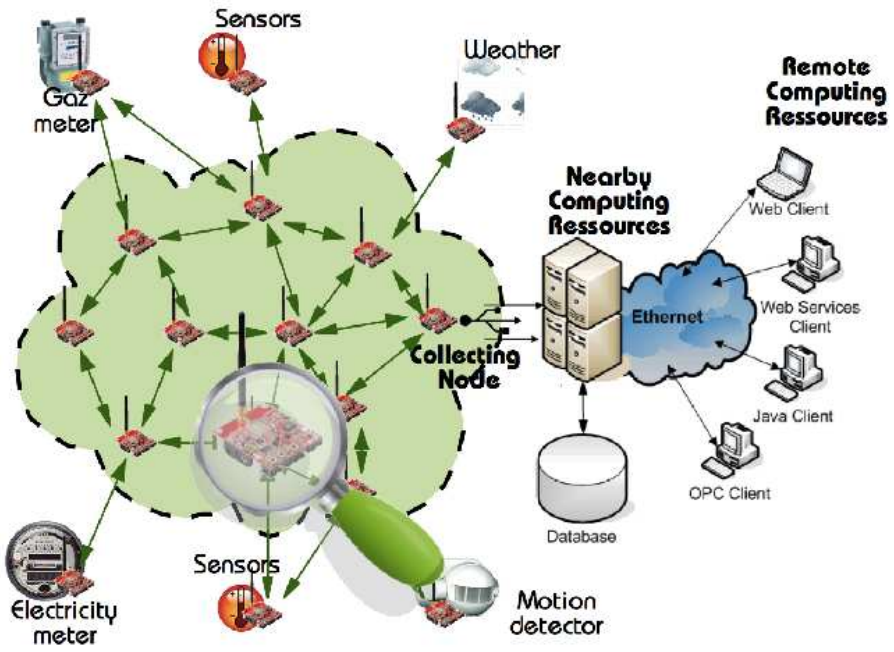


**FIGURE 1.1:** Noeuds de capteurs dispersés sur un champ de capteurs (2).

aux utilisateurs. Ce dernier type de noeud peut communiquer avec une entité de gestion de tâche (souvent un noeud plus puissant, qui gère une région du réseau), qui à son tour s'interface directement avec l'utilisateur. Cette entité peut être par exemple un ordinateur. Ou bien, le noeud collecteur peut s'interfacer lui-même avec l'utilisateur au travers d'autres types de réseau.

Dans les WSNs, les noeuds ont la double fonctionnalité d'être à la fois à l'origine des données et chargés de les router vers le noeud collecteur et enfin à l'utilisateur. Donc la communication entre eux se fait pour deux raisons :

- **Fonction de source** Les noeuds avec information sur un événement effectuent la communication pour transmettre leur paquet au noeud sink.
- **Fonction de routeur** Les mêmes noeuds sont susceptibles de prendre part à la communication même s'ils n'ont pas de données concernant un événement, mais pour faire avancer les paquets à travers le réseau.

Dans les prochaines sous-sections on va présenter comment ces entités communiquent, puis

détailler leur caractéristiques matérielles.

Il se peut que le logiciel qui s'exécute sur les noeuds d'un WSN ait besoin d'être changé. Ceci peut être le cas si les noeuds doivent être calibrés pour accommoder des nouvelles tâches, des changements physiques, ou des changements comportementales au niveau de ces noeuds. Le changement des fonctionnalités qui s'exécutent au niveau de noeud implique une mise-à-jour de son logiciel, sans accès physique à celui-là (reprogrammation ou reconfiguration dynamique). Cette mise-à-jour doit tenir compte de certaines caractéristiques physiques des noeuds, leur particularités si bien au niveau matériel qu'au niveau logiciel. Elle doit répondre aux prochaines questions : (i) Quels noeuds sont visés ? (ii) Combien consume-t-il un noeud pour la faire ? (iii) Combien va-t-il ce noeud être en dehors de ces tâches habituels ?

Le fonctionnement de cette mise-à-jour dans le contexte globale d'un WSN se démontre avec des simulations. La simulation permet d'estimer d'avance les performances du réseau, sa demande énergétique, ainsi que d'estimer et de comparer différentes manières de relier les noeuds sans fil. Elle permet de voir les obstacles qui peuvent arriver dans ce contexte, avant que le noeuds soient déployés physiquement.

La reconfiguration dynamique et la simulation WSN sont les deux objectifs majeurs de cette thèse.

On va se focaliser d'abord sur la reconfiguration dynamique. Pour pouvoir mieux comprendre comment la mise-à-jour du logiciel se fait et où elle se situe dans le point de vue logique au niveau du noeud, on va présenter comment le logiciel lui-même est structuré.

### 1.1.2 Aperçu général : La pile de communication dans les WSNs

Pour pouvoir réaliser les applications existantes, ainsi que celles potentielles, il y a un besoin des protocoles sophistiqués et efficients (2). Pour pouvoir offrir des observations sur des phénomènes qui sont fiables et efficientes et pouvoir initier les actions appropriées, les caractéristiques des ceux-ci doivent être détectées/estimées à partir de l'information collective de noeuds. De plus, au lieu d'envoyer des données brutes à un point central pour les fusionner,

des centres “locaaux” peuvent être construits où des calculs peuvent être faits, calculs qui concernent la région desservie. C’est pour ces propriétés que ce type de réseaux posent des défis particuliers dans le développement des protocoles de communication.

Chaque entité qui détient de l’information qui doit être communiquée fait usage d’un protocole de communication. Un protocole est un ensemble de règles à respecter lorsque le message est envoyé. Il doit répondre à plusieurs questions, par exemple : qui est le noeud qui envoie le premier message, est-ce que sa réception doit être confirmée, que se passe-t-il lorsque la communication s’achève ou entre deux messages.

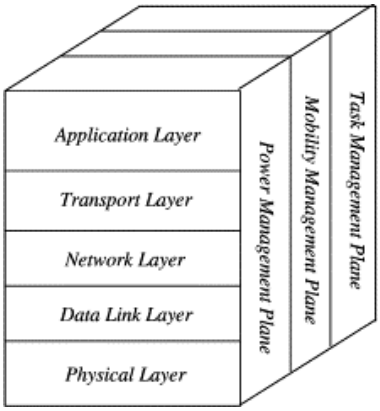


**FIGURE 1.2:** La pile du protocole pour un WSN (3).

Durant ces années, les protocoles ont évolué et chaque nouvelle partie répondait à des questions plus complexes. Par exemple, comment transférer un message d’un côté du réseau à l’autre, ou comment l’envoyer dans un autre réseau voisin. Ce genre de fonctionnalités s’appuie sur une base qui implique le transfert physique radio. Un noeud envoie le message à tous ses voisins et une autre partie du protocole décide si le message est destiné au noeud courant ou non. Ensuite, une autre partie du protocole décide à qui envoyer l’information pour qu’elle soit transférée d’un noeud à l’autre jusqu’à la destination.

Ces différentes parties du protocole s’appellent couches, parce qu’elles s’appuient l’une sur l’autre. Leur structure peut être consultée dans la Figure 1.2.

Les concepts concernant les WSNs sont dérivés de réseaux filaires. C'est pourquoi, d'un côté, sur l'horizontale, on a la structure classique "OSI".

L'interface entre les couches nécessite de la puissance de calcul, ainsi que de ressources de mémoire. D'un autre côté, dans les WSNs, les noeuds ont des faibles caractéristiques techniques (taille petite de la mémoire, fréquence réduite du microcontrôlleur). C'est pour cela que ici, les couches s'imbriquent au point où elles deviennent celles présentées en verticale dans la Figure 1.2. Ceci permet de réduire les interfaces entre les couches, demandant moins de ressources pour pouvoir être implémenté. À savoir, la manière dont les couches sont implémentées a aussi un impact sur l'énergie utilisée par le microcontrôlleur. Les couches bien différenciées vont utiliser plus de mémoire, vont prendre plus de temps à s'exécuter. Ceci veut dire que le microcontrôlleur va avoir besoin de plus longtemps pour elles.

On parle des concepts relatives à la position et synchronisation des nouds ainsi que la topologie du réseau. Pour finir, la troisième couche verticale porte sur les concepts de gestion de consommation, la mobilité des noeuds et la gestion des tâches qui s'exécutent là-dessus.

Les propriétés intrinsèques des noeuds individuels imposent des contraintes additionnelles sur les protocoles de communication, en ce qui concerne la consommation d'énergie. Les applications des WSNs, ainsi que les protocoles, sont adaptées à offrir une efficacité énergétique élevée. La position physique des noeuds sera peut-être inconnue. Ceci permet le déploiement aléatoire, dans les terrains inaccessibles. D'un côté, le déploiement aléatoire requiert des développements de protocoles d'auto-organisation pour la pile de communication. D'un autre côté, une communication trop longue ou avec une grande portée réduit sévèrement l'énergie dont le noeud dispose.

À part pour la position des noeuds, la densité du réseau est une autre chose exploitée dans les protocoles WSNs. Dans un réseau plus dense, les noeuds consomment moins d'énergie pour envoyer l'information à leur voisins, en jouant sur la puissance de transmission. Dû à la courte portée de transmission, un large nombre de noeuds peut être déployé, les uns à côté des autres et être positionnés dans la proximité immédiate. Par la suite, la communication "multi-hop" est

exploitée dans la communication entre les noeuds, puisqu'elle mène à une consommation moins élevée que dans le cas de communication "single-hop". Donc, pour envoyer une information du noeud A au noeud D, au lieu de l'envoyer avec une forte puissance de transmission directement de A à D, on passe par des noeuds intermédiaires B et C, par exemple.

De plus, le dense déploiement couplé avec les propriétés physiques du phénomène introduit des corrélations dans le domaine spatial et temporel. Par la suite, des protocoles corrélés spatio-temporel ont émergé, pour une efficacité améliorée(2).

Les paradigmes de communication WSN sont différents de ceux qui sont associés aux réseaux traditionnels, démarrant le besoin pour des nouveaux protocoles de communication. Dans ce contexte, le protocole IEEE 802.15.4 présente des caractéristiques potentiellement intéressantes, comme l'efficacité énergétique, la garantie de latence et la scalabilité. C'est-à-dire que le paquet est garanti à être livré dans une certaine période de temps.

La couche physique est appelée PHY et sera traitée dans la prochaine section.

### 1.1.3 WSNs : La couche physique ou le matériel

La couche matérielle (aussi dénommée PHY ou physique) est une couche fondamentale, à la base des autres couches logiques (supérieures). Elle traite les caractéristiques du matériel présent sur le noeud et la manière dont elles accèdent au canal pour transmettre / recevoir l'information. Dans les prochaines sous-sections, chaque sous-ensemble du noeud sera traité distinctement.

#### 1.1.3.1 Structure interne d'un noeud

À cause de contraintes spécifiques au domaine (faible coût, faible consommation) ont imposé à leur tour des contraintes sur le matériel qui peut être présent sur un noeud de WSN. Après des innovations telles que les technologies MEMS (microsystème électromécanique), le paradigme utilisé par les processus de mesure a commencé à changer et à

faciliter leur utilisation dans les WSNs. Ils sont devenus de moins en moins coûteux et donc susceptibles d'être employés dans le domaine de WSNs.

Dans ce type de réseaux, la ressource la plus précieuse est l'énergie. Les applications finales vont souvent faire des observations sur des phénomènes pendant une durée étendue dans le temps. Par exemple, une application typique assume des périodes entre deux et trois ans. Les applications ont besoin que le noeud fonctionne et soit autonome pendant ce laps de temps. Par conséquent, le noeud ne doit pas avoir besoin d'un échange de batteries, ni d'autres formes d'intervention locale (les noeuds peuvent être inaccessibles : dispersés par avion, ou dans le corps humain par exemple). Les exigences de performance ont été identifiées dans l'ordre :

- possibilité de configuration à distance
- avoir une faible surcharge (le coût en termes de ressources utilisées compense le coût d'installation)
- faire attention aux ressources (surtout de minimiser le temps de communication radio)

La Figure 1.3 montre la structure interne d'un noeud WSNs.

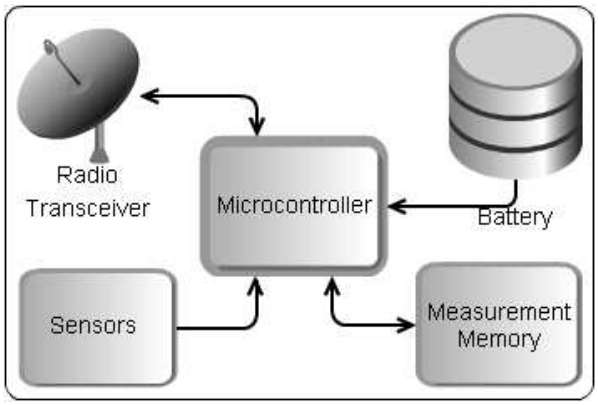


**FIGURE 1.3:** La structure interne d'un noeud WSNs.

### 1.1.4 Produits actuels

#### 1.1.4.1 Le microcontrôleur

En ce qui concerne les noeuds contiennent souvent un microcontrôleur avec peu de ressources. C'est-à-dire une architecture sur 8 ou 16 bits, avec moins de 16kOctets de mémoire RAM et moins de 192kOctets de mémoire Flash. En même temps, cette architecture est très bien adaptée aux besoins pratiques. généralement, les WSN n'ont pas besoin de calculs complexes, ni de résultats rapides. Il n'existe pas de MMU (memory management unit), pour aider avec la mémoire virtuelle et assurer la protection entre les différents fils d'exécution. Souvent, les données captées (analogiques) changent lentement par rapport à la vitesse du processeur du microcontrôleur. Une architecture plus puissante aurait un plus fort besoin d'énergie qui ne serait pas forcément justifié.

Il existe de dizaines de solutions matérielles pour les réseaux de capteurs sans fil. Dans le Tableau 1.2 on remarque la fréquence des microcontrôleurs AVR (30), fabriqués par Atmel, MSP430 (31) de Texas Instruments et PIC (32), fabriqués par Microchip. Ensuite, dans la Figure 1.4, on remarque que les MCU qui dominent ont un chemin de données de 8-bits, avec ceux de 16-bits qui suivent. L'objectif de cette thèse va s'appuyer donc sur ces architectures. Chaque application qui cible une architecture embarquée a de fortes contraintes et l'implémentation du logiciel présente donc une grande difficulté.

Si on compare les trois architectures sur un benchmark à 3 volts, 1Mhz, on trouve les caractéristiques suivantes :

| Architecture Matérielle | Régime actif | Sleep |
|---|---|---|
| Atmel AVR | < 2mA | 8uA |
| Microchip PIC | < 0.7mA | 1.2uA |
| Texas Instruments MSP430 | 0.7mA | < 1.2uA |

**Table 1.1:** Résumé des caractéristiques électrique des microcontrôleurs les plus utilisés dans les WSNs.

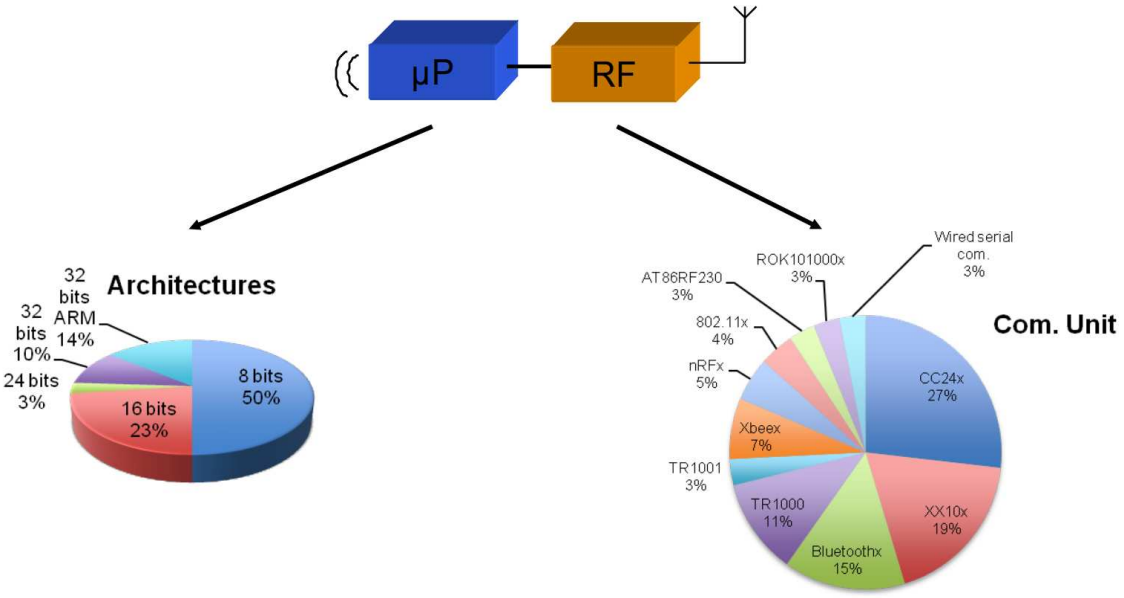


**FIGURE 1.4:** La fréquence de différentes type des microcontrôleurs et transceivers radiofréquence dans les noeuds WSNs. (4).

#### 1.1.4.2 Le transceiver radiofréquence

LA couche physique dénommée “PHY” offre une radio de type “spread-spectrum” qui opère à 2.4 Ghz (802.15.4) avec un débit maximal de 250 kbps. Il existe aussi une autre spécification PHY, à 915 Mhz et 868 Mhz pour des débits inférieurs.

L’interface radiofréquence est la partie qui consomme la plus grande quantité d’énergie (33). Pour envoyer 1 octet, le noeud utilise une quantité d’énergie équivalente à 10000 cycles d’horloge du microcontrôleur.

Du côté transceiver radiofréquence on remarque une prédominance des transceivers de Texas Instruments (CC1000, CC2420). C’est la raison pour laquelle cette thèse s’appuie sur eux. Ceci sera développé dans les prochaines sections. Au fur est à mesure que la technologie avance et les coûts diminuent, des nouveaux transceivers radiofréquence tels que CC2520 ont été développés. Pour certains d’entre eux, une fois configurés, ceux-ci sont capables de prendre des décisions qui portent sur la communication, sans avoir besoin de communiquer avec le microcontrôleur. Ceci veut dire un gain d’énergie au niveau du transfert SPI entre les deux

composants. La plupart des solutions ont une consommation d'environ 20mA pour la réception et un peu moins de 20mA pour l'émission (dépendant de la puissance d'émission).

Avec les paramètres électriques déjà précisés, avec un bas rapport cyclique, on peut envisager une durée de fonctionnement jusqu'à quelques années. Cela implique que le noeud va passer la plupart de ce temps dans le mode sommeil "sleep".

Dû au fait que les paramètres électriques des microcontrôleurs varient peu, ainsi que leur prix, les choix entre eux est difficile. Ceci explique la diversité de noeuds existants.

| Platforme | CPU | Max Clock | Ram/Flash/EEprom | Int. Radio | OS | Organisation |
|---|---|---|---|---|---|---|
| Mica2 | Atmel ATMega128L | 8 | 4K/128K/512B | CC1000 | TinyOS | Crossbow |
| MicaZ | Atmel ATMega128L | 8 | 4K/128K/512B | CC2420 | TinyOS | Crossbow |
| AquisGrain | Atmel ATMega128L | 8 | 4K/128K/512B | CC2420 | - | Philips |
| Fleck | Atmel ATMega128L | 8 | 4K/128K/512B | CC2420 | TinyOS | CSIRO |
| WeC | Atmel AT90L8535 | 4 | 512 /8K/32K | RFM TR1000 | TinyOS | UC Berkeley |
| Rene2 | Atmel Atmega164 | 8 | 1K/16K/32K | RFM TR1000 | TinyOS | UC Berkeley |
| Sun | Atmel AT91FR40162S | 8 | 256K/2M | CC2420 | Squawk | Sun Labs |
| iBadge | Atmel ATMega103L | 6 | 4K/128K | Ericsson BT | Palos | UCLA |
| Smart-its | Atmel ATMega103L | 4 | 4K/128K | Ericsson BT | Smart-its | Teco |
| Cit SN | PIC16F877 | 20 | 368/8K | Nordic nRF903 | TinyOS | CIT |
| Smart-its | PIC18F252 | 8 | 3K/48K/64K | Radiometrix | Smart-its | Teco |
| uPart0140ilmt | rfPIC16F675 | 4 | 64/1K | onchip | Smart-it | Teco |
| U3 | PIC18F452 | 0.031-8 | 1K/32K/256 | CDC-TR-02B | Pavenet | U Tokyo |
| uAMPS | Intel SA1100 | 59-206 | 1M/4M | LMX3162 | uOS | MIT |
| AWAIRS 1 | Intel SA1100 | 59-206 | 1/4M | RDSSS9M | MicroC/OS | Rocwell |
| Telos | TI MSP430F149 | 8 | 2K/60K/512K | CC2420 | TinyOS | UC Berkeley |
| Pluto | TI MSP430F149 | 8 | 4K/60K/512K | CC2420 | TinyOS | Harvard |
| BSN node | TI MSP430F149 | 8 | 2K/60K/512K | CC2420 | TinyOS | Crossbow |
| eyesIFXv2 | TI MSP430F1611 | 8 | 10K/48K | TDA5250 | TinyOS | TU Berlin |
| Ant | TI MSP430F1232 | 8 | 256/8K | nRF24AP1 | SOS | Dynastream Innovation |

**TABLE 1.2:** Comparaison entre différents architectures matérielles pour WSNs (11)

### 1.1.5 WSNs : les couches logiques supérieures

Concernant les couches supérieures, elles sont appelées logiques, parce qu'elles ne tiennent pas compte de la façon dont le matériel accède au médium de communication. La Figure 1.6 les démontre. La première d'entre elles, la couche de données (aussi dénommée MAC) s'occupe de l'interaction des plusieurs radios. Elle offre du support pour plusieurs topologies, comme celles présentées dans la Figure 1.5.

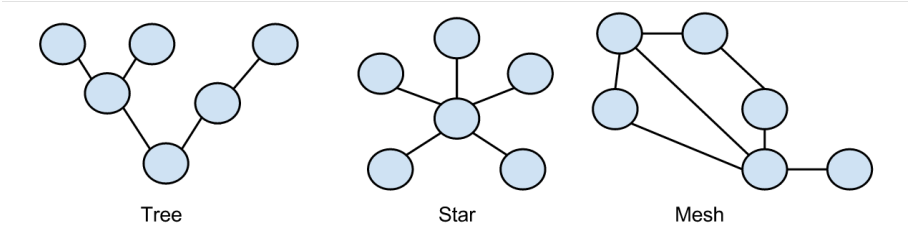


**FIGURE 1.5:** Les différentes topologies 802.15.4

La couche réseau offre la possibilité de communication entre deux points du même réseau à travers plusieurs noeuds, jusqu'à ce que le message arrive bien à la destination.

La couche transport a la même caractéristique que la couche précédente, donc router l'information, mais à travers plusieurs réseaux au lieu de plusieurs noeuds.

Enfin, la couche application comprend le comportement spécifique, ou le but final du réseau.

Pour bien pouvoir assurer l'interopérabilité de dispositifs IEEE 802.15.4, ainsi que pour offrir une facilité pour développer les applications plus facilement, l'alliance ZigBee a été créée.

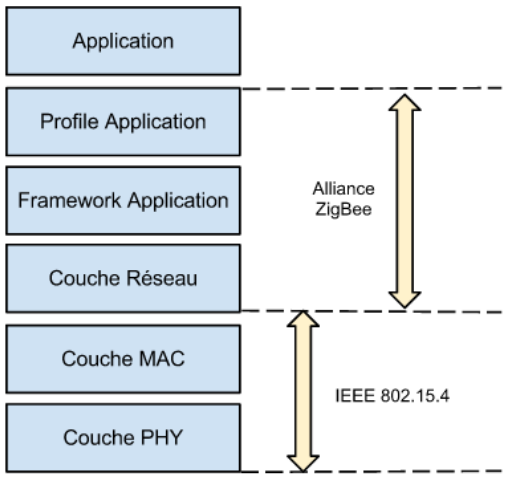


**FIGURE 1.6:** La pile ZigBee et 802.15.4

L'alliance ZigBee, créée en août 2002, est une association d'industriels qui travaillent ensemble pour offrir des produits de basse puissance, connectés sans fil, pour le suivi et le contrôle d'applications. Beaucoup d'institutions ont développé des plate-formes de capteurs sans fil sur

les solutions ZigBee et IEEE 802.15.4. L'alliance ZigBee inclue à peu prêt 200 compagnies dans 24 pays. L'objectif est d'offrir les couches supérieures de la pile du protocole, à partir de la couche réseau et jusqu'à la couche application, incluant les profils d'applications. La technologie ZigBee est utilisée dans des applications concernant la surveillance et le contrôle, qui n'ont pas besoin de haut débit de données, mais qui doivent avoir une faible consommation, un faible coût et une simplicité d'opération. On peut prendre note que ZigBee est construit sur les couches du standard IEEE 802.15.4 (Figure 1.6), pour définir des types d'applications qui peuvent être utilisés.

L'origine du ZigBee vient du fait qu'en définissant simplement une couche PHY et MAC, on ne peut pas garantir que différents dispositifs vont pouvoir communiquer les uns avec les autres. Comme déjà dit, ZigBee commence par le standard 802.15.4 et définit des profils pour les applications, qui permettent aux produits des différentes compagnies de communiquer ensemble. Zigbee supporte les topologies "star" et "mesh".

## 1.2 Reconfiguration dynamique

Il y a deux types des données qui circulent dans un réseau WSN : les valeurs des capteurs et les mises à jour. Beaucoup de travail a été orienté vers les mises-à-jour du code résident sur le noeud. On parle des noeuds qui peuvent se reprogrammer tout seuls, sans intervention humaine physique. Ils vont recevoir une mise à jour et avec l'aide d'un programme spécial nommé "bootloader" ils vont réécrire leur mémoire flash.

La reconfiguration représente donc la mise à jour du logiciel qui est exécutée sur un noeud. On l'appelle dynamique parce qu'on parle de cas où elle est faite après que le noeud soit déployé. Les raisons pour ces mises à jour sont diverses, comme la calibration d'un algorithme un changement d'opérations à exécuter.

D'une perspective d'hétérogénéité logicielle, on veut que la reconfiguration offre du support pour des différents systèmes d'exploitation, pour les raisons argumentées dans la section "Hétérogénéité".

D'habitude, la reconfiguration fait cela en choisissant les composants du système d'exploitation appropriés, sans modifier l'application. Il est désirable d'avoir un système d'exploitation capable de charger des nouvelles applications, mais chaque application peut demander des modules différents. Pour maintenir un système optimisé, le kernel doit toujours avoir une configuration minimale, en enlevant les modules non-utilisés et demandant les nouveaux modules nécessaires. Tout cela en suivant les changements dans le comportement des applications.

Dans le contexte de reconfiguration dynamique, il est souhaité de pouvoir communiquer entre les fonctionnalités. D'habitude, le système d'exploitation utilise soit un mécanisme de relocalisation et un système de tuyaux (avec adresses fixes), soit une mémoire partagée. Dans le contexte de relocalisation, il y a besoin de modifier l'adresse de fonctions qui sont appelées lors d'ajouter la nouvelle fonctionnalité. Par ailleurs, l'adresse en RAM des variables doit être modifiée. Ce processus est compliqué et ce n'est pas difficile d'imaginer pourquoi le nombre de systèmes d'exploitation qui peuvent le faire est petit.

Sugihara et Gupta (34) proposent la taxonomie présentée dans la Figure 1.7 pour la programmation des WSNs. Ils identifient plusieurs niveaux hiérarchiques en jeu lors de la programmation des noeuds d'un WSN.

On va s'appuyer sur la première catégorie de modes de programmation, notamment l'approche "low-level" (Platform-Centric), qui se concentre sur le noeud lui-même.

En ce qui concerne les mises à jour sous forme de code machine, elles entrent dans la première catégorie, au niveau de noeud.

Un deuxième type de programmation au niveau de noeud existe, qui comprend les machines virtuelles. Les machines virtuelles sont développées pour une architecture matérielle cible et interprètent une forme intermédiaire de données, dénommée bytecode.

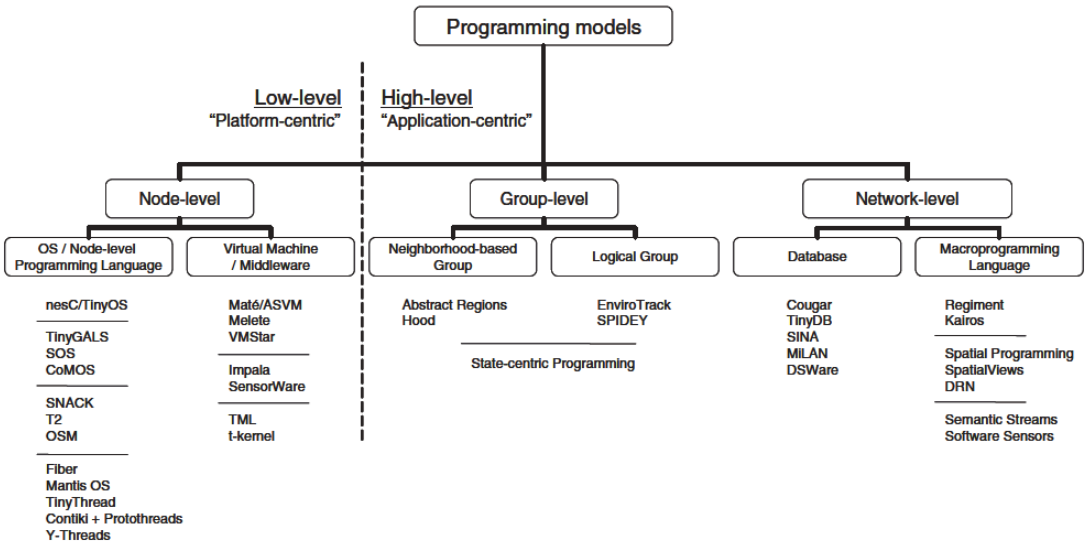


**FIGURE 1.7:** Une taxonomie pour la programmation des WSNs.

# 1.3 Modèle de programmation

Tout d'abord, je vais traiter traiter les mises à jour sous forme de code machine. Ceux-ci sont destinés à être exécutés sur un noeud sur lequel il existe, dans les plupart des cas, un Système d'Exploitation ou une Machine Virtuelle.

## 1.3.1 Systèmes d'exploitation pour WSNs

Les systèmes d'exploitation pour les noeuds WSNs ont été créés pour (i) gérer l'accès des applications à la couche physique et (ii) pour gérer les applications logicielles qui s'exécutent sur le microcontrôlleur du noeud.

Ils font partie de la couche application, étant donc une couche logique.

### 1.3.1.1 Introduction et principes

Les noeuds d'un réseau WSN ont plusieurs parties constitutives, du point de vue logiciel. D'abord, il y a le pilote du module radio. Après, il y a le protocole de communication

avec les noeuds voisins et (éventuellement) un système d'exploitation. Le système d'exploitation peut (et le fait souvent) incorporer certaines parties comme le pilote pour le module radio, une partie de la pile de communication, etc.

Bien sûr il peut y avoir d'autres parties, mais en grandes lignes celles présentées en haut sont toujours présentes.

Classiquement (par exemple, sur un ordinateur), le système d'exploitation se comporte comme un arbitre de ressources en allouant de la mémoire et des périphériques aux utilisateurs d'une manière contrôlée (35). Comme cela, par exemple, chaque programme différent peut à son tour être exécuté par le processeur.

*D'un point de vue architectural*, les systèmes d'exploitation sont divisés en deux catégories monolithiques et modulaires. La première catégorie n'a pas une structure bien définie, alors que les modulaires sont divisés en plusieurs modules bien différentiés. Les machines virtuelles sont une alternative qui interprètent des données intermédiaires, au lieu d'exécuter directement le code machine. Ce processus offre une bonne portabilité, mais de pauvres performances du système.

*Du point de vue de la programmation*, pour les systèmes d'exploitation monolithiques, la fonctionnalité entière est reprogrammée. Pour les modulaires, seulement la nouvelle partie est ajoutée. Du côté machines virtuelles, une seule mise à jour concernant plusieurs architectures matérielles peut être distribuée dans le réseau.

*L'ordonnancement* dicte l'ordre dont les tâches sont exécutées sur le CPU. On parle ici de systèmes d'exploitation préemptifs, ou pas, avec un ordonnancement simple ou temps réel. Les systèmes préemptifs peuvent suspendre l'exécution d'un processus pour donner la possibilité à un autre de (re)commencer.

*L'allocation et protection de la mémoire* peut être présente ou non pas. Au tout départ, les applications dans les noeuds étaient simples et la mémoire était allouée d'une manière statique. Ceci ne soulevait pas de problèmes de protection, car les besoins entiers du système étaient connus à l'avance et donc il n'y avait pas de problèmes d'écrasement de la mémoire. Dès qu'elles

ont commencé à se diversifier, l'allocation dynamique de mémoire a permis d'allouer autant de mémoire que la tâche en question en avait besoin. Pour un système multi-tâche, on a souvent un mécanisme qui empêche une tâche d'écrire dans la zone mémoire d'une autre, donc une protection de la mémoire. Dû au fait que les MCU sur les noeuds n'ont pas de MMU, l'application s'exécute soit en mode superviseur (privilèges totales, souvent dans le cas des systèmes d'exploitation monolithiques), soit en mode utilisateur (avec une couche logicielle qui supervise le comportement). Cette fonction de l'unité MMU est donc soit implémentée en logiciel, soit absente.

*Le support pour le protocole de communication* est un autre aspect important dans ce contexte. On parle ici de la communication inter-procès dans le système, ainsi que dans le réseau. Le système d'exploitation offre une interface de programmation (API), qui est souvent la même, que le système d'exploitation s'exécute sur tel ou tel autre matériel.

*Le partage des ressources* concerne l'accès multiplexé au CPU des différents fils d'exécution, l'accès à la mémoire, etc.

Pour un noeud, si on veut ajouter un nouvel algorithme par une mise à jour, c'est la responsabilité du système d'exploitation de bien intégrer celui-ci avec les autres déjà existants.

Certaines autres problématiques peuvent être soulevées, par exemple, le support pour le temps réel. Certaines applications demandent une réponse à un événement dans un temps critique, donc elles ont besoin d'un système d'exploitation temps réel. Dans ce contexte, la charge sur le CPU demandée par le système d'exploitation est importante.

Un des premiers systèmes d'exploitation pour WSN est TinyOS. Conçu par UC Berkeley/Crossbow (36), il est, comme on peut voir dans le Tableau 1.2, utilisé par beaucoup de noeuds différents. Cependant, les applications ont commencé à devenir de plus en plus complexes, à demander des réponses rapides, à exécuter plusieurs tâches en même temps. Des systèmes d'exploitation préemptifs en temps réel tels que NanoRK, RETOS, MantisOS et SOS ont été développés. Les applications temps réel garantissent une réponse dans une fenêtre de temps établie.

#### 1.3.1.2 Monolithique vs Modulaire

Les systèmes d'exploitation sont, comme déjà dit, divisés en deux catégories : Monolithiques et Modulaires. Les Monolithiques sont liés à l'application au moment de la compilation et une mise à jour ultérieure demande une nouvelle image flash entière. Coûteux du point de vue énergétique, ils ne sont pas préférés si l'application doit être réactualisée souvent.

À cause de ressources limitées du matériel du noeud, un système d'exploitation ne peut pas offrir toutes les fonctionnalités requises par les diverses applications. C'est probable que le noeud sera reconfiguré prochainement avec des nouveaux paramètres et algorithmes. Une approche modulaire serait un avantage. Il faut que le système d'exploitation soit capable d'ajouter des nouvelles fonctions et d'enlever les fonctions existantes.

Les systèmes d'exploitation modulaires sont toujours liés à l'application au moment de la compilation mais ils ont aussi la possibilité d'ajouter de nouvelles fonctionnalités d'une manière dynamique. Ce qui signifie que seulement une partie du code compilé peut être envoyée, ainsi économisant beaucoup d'énergie.

Si une fonctionnalité n'est plus demandée, elle peut être supprimée et la mémoire flash sera libérée. Ou bien, elle peut toujours rester écrite et être exécutée seulemen quand le noeud le décide ou quand il reçoit une commande concernant cela.

Un des autres avantages des systèmes d'exploitation modulaires est l'abstraction au niveau de module. On peut faire attention seulement à une seule partie du code et on peut être sûr que ceci ne va pas demander le changement de l'ensemble. S'il s'agit d'une fonctionnalité qui (pour une raison ou une autre) est considérée cassée, le kernel peut juste l'ignorer et continuer avec les autres tâches.

Par contre, dans un kernel monolithique, il y a une très forte chance qu'une erreur comme cela peut conduire à un échec entier de l'architecture logicielle. En revanche, les kernels monolithiques sont beaucoup plus rapides, n'ayant pas d'aussi nombreux niveaux d'indirection (appels de fonctions pour lier les différentes couches du logiciel). Avec ces derniers kernels, le

système d'exploitation et les applications vont être exécutées en mode superviseur (le plus prioritaire). Les autres ne partagent pas forcément ce comportement. De toute façon, n'ayant pas une MMU, l'exécution de toutes les instructions est permise ainsi pour les applications que pour le kernel. Dans ce cas, l'analyse de fautes est beaucoup plus facile et est limitée au niveau du module.

### 1.3.2 Aperçu général : Machines Virtuelles

Une machine virtuelle est un logiciel qui interprète du code bas niveau écrit pour elle et qui exécute les opérations associées, pour l'architecture sur laquelle elle fonctionne.
Elle va prendre donc le bytecode (c'est-à-dire les instructions pour la machine virtuelle) et elle va le décoder en instructions compatibles avec l'architecture en cause.

La comparaison entre le code machine pur et bytecode a soulevé la possibilité que l'interprétation du bytecode puisse être une technique beaucoup plus économique du point de vue énergétique, grâce au caractère abstrait du bytecode.

Il existe des machines virtuelles aussi pour le WSN. Leur existence est d'habitude orientée vers une architecture matérielle souvent utilisée (par exemple, AVR). Leur implémentations sont justifiées par la popularité du langage Java et l'usage intensif de la machine virtuelle Java.

L'un des avantages les plus forts des machines virtuelles est que leur taille est petite. C'est pourquoi il y a plus d'espace qui est disponible pour l'application finale. Aussi, elles peuvent coexister avec un système d'exploitation (ce qui est souvent le cas). Le système d'exploitation s'occupe de gérer les ressources matérielles alors que la machine virtuelle est chargée d'interpréter les nouvelles fonctionnalités et les mises à jour.

Beaucoup de recherches ont été faites à ce sujet et plusieurs machines virtuelles pour les microcontrôleurs ont été créées. En général, il y a un compilateur croisé (un compilateur qui vise une architecture embarquée, plutôt que l'architecture PC) de langage java qui reste sur l'ordinateur et qui produit le bytecode. Le développeur code donc en langage Java et lorsque le résultat intermédiaire de la compilation est prêt, il sera adapté pour la machine virtuelle

considérée. Ensuite, il est envoyé par radiofréquence vers les noeuds récepteurs. La machine virtuelle qui reste sur le noeud va prendre le bytecode et l'interpréter pour exécuter l'algorithme.

### 1.3.3 Systèmes d'exploitation typiques pour les WSNs

Dans cette section, un aperçu de systèmes d'exploitation les plus connus pour WSNs sera présenté.

#### 1.3.3.1 TinyOS (1)

TinyOS a été créé par l'université Berkeley en Californie. Faisant partie de programme NEST de DARPA, la première version est sortie en 1999. Les applications pour TinyOS sont écrites en langage nesC. Celui-ci est une variante du langage de programmation C, étant optimisée pour les applications WSNs. Les programmes pour TinyOS sont constitués par composants logiciels, dont certains présentent des abstractions matérielles. Les composants sont connectés entre eux en faisant usage d'interfaces. TinyOS offre le support pour des fonctionnalités communes, comme les communications par paquets, routage, parmi d'autres.

TinyOS est un système dit "non-bloquant". Il utilise une seule pile. Alors, chaque opération entrée-sortie qui dure plus que quelques microsecondes est asynchrone et il y a une fonction de gestion associée, qui va être appelée. Pour donner la chance au compilateur d'optimiser le code, TinyOS utilise les caractéristiques de nesC pour s'interfacer avec les fonctions appelées de manière statique. Même si TinyOS est non bloquant et qu'une haute concurrence est garantie avec une seule pile d'exécution, cela force les développeurs à écrire du code complexe en utilisant plusieurs gestionnaires d'évènements.

Pour des opérations plus complexes, TinyOS fait usage de tâches. Un composant peut poster une tâche et le système d'exploitation va l'exécuter plus tard. Les tâches sont non-préemptifs et sont exécutées à la manière FIFO (First In First Out). Lors de l'arrivée d'une nouvelle tâche, celle-ci sera placée dans le fil d'attente en fonction de sa priorité (plus elle est grande, plus le placement est proche de la sortie de la FIFO). Dans le cas où la file d'attente est pleine, la tâche

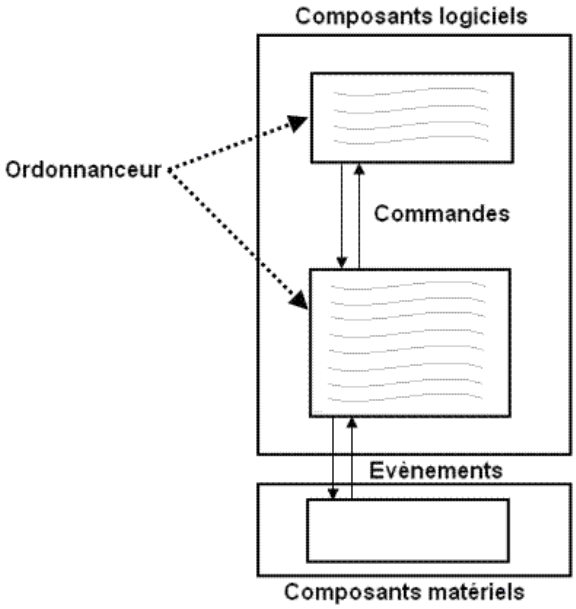


**FIGURE 1.8:** Intéractions en TinyOS (5)

dont la priorité est la plus faible est enlevée de la FIFO. TinyOS est un standard de-facto pour les réseaux de capteurs sans fil. En conjoint avec Mica2, il a été utilisé comme une plateforme pour développer des nouveaux algorithmes de routage et nouveaux algorithmes pour la sécurité du réseau.

Concernant la reconfiguration, TinyOS présente un désavantage, car la taille de la mise à jour est beaucoup plus grande que dans le cas des systèmes d'exploitation modulaires.

On ne peut pas faire des estimations de la taille de ce système d'exploitation. La raison pour ceci est que, étant monolithique, le kernel fait partie de l'application, il n'y a pas une partie de dimension fixe entre des applications différentes.

#### 1.3.3.2 Mantis OS

Mantis OS (37) et un système d'exploitation léger faisant attention à la consommation énergétique et en même temps permettant des fils préemptés. L'architecture du kernel de Mantis OS en ressemble aux ordonnanceurs UNIX classiques. Les services offerts

sont un sous-ensemble des fils POSIX. Il y a des commutations entre les fils d'exécution d'une manière round-robin, en faisant attention à leur priorité. Le système supporte des sémaphores binaires (mutex) et sémaphores avec comptage.

Il existe deux régions différentes de RAM : l'espace pour les variables locales (alloué au moment de la compilation) et celui de variables globales qui est vu comme une heap classique. Lorsqu'un nouveau fil d'exécution est créé, l'espace pour sa pile est alloué en heap par le kernel. Dès que le fil est terminé, l'espace est récupéré. Le contexte du fil courrant est sauvé sur la pile lorsque le fil est suspendu.

Les sémaphores sont des structures de cinq octets qui contiennent un verrou sur les ressources (ou bien un compteur), avec des pointeurs vers le début et la fin de la queue.

L'ordonnanceur reçoit une interruption d'horloge du matériel et fait le changement de contexte. L'interruption d'horloge est la seule traitée par le kernel. Toutes les autres sont envoyées vers le pilote de périphérique spécifique.

Pour MantisOS il existe une pile de réseau. Celle-ci offre du support pour la couche trois et au-dessus, de la pile OSI (la couche de réseau, routage, de transport et d'application). Il y a le support pour le protocole MAC qui est fait par la couche de communication. Cette couche donne une interface unifiée pour les pilotes de périphérique de communication, d'une manière qui fait abstraction du matériel.

Ce système d'exploitation offre le support pour différents capteurs, stockage externe et communication asynchrone (radio et sériel). Chaque périphérique a besoin d'implémenter quatre fonctions et un mutex. Il implémente un tableau statique avec tous les périphériques. Ils peuvent être dans l'état démarré, arrêté ou suspendu.

Du point de vue reconfiguration, MantisOS demande une connaissance du kernel pour pouvoir commencer le développement. De plus, il est, tout comme TinyOS, monolithique ce qui veut dire une grande taille du fichier de mise à jour.

L'usage de RAM pour Mantis OS est de 500 octets de RAM et il occupe 14K octets de flash.

#### 1.3.3.3 NanoRK

Nano-RK (38) est un système d'exploitation multitâche préemptif qui se base sur la réservation en temps réel (RTOS) développé à Carnegie Mellon University. Dans le contexte de réseaux de capteurs sans fil, il offre le support pour les transferts multi-hop. Nano-RK peut actuellement être déployé sur la plateforme FireFly ainsi que les motes MicaZ. Il comprend un kernel léger intégrant des ressources du noyau (RK) avec une riche versatilité et l'allocation du temps.

Nano-RK offre le support pour les applications multitâches préemptifs de priorité fixe. Les tâches peuvent spécifier leurs besoins en ressources et le système d'exploitation fournit en temps utile la garantie et l'accès contrôlé à des cycles de CPU et de paquets réseau. Ensemble, ces ressources constituent des réserves d'énergie virtuelle qui permet à l'OS d'appliquer au système des budgets d'énergie pour chaque tâche. Par exemple, une tâche peut être permise de s'exécuter 10ms chaque 150ms (réservation de CPU), ou un noeud peut être permis de transmettre seulement 10 paquets de réseau chaque minute (réservation réseau).

Nano-RK est un projet open source, écrit en C. Parmi ses points forts on trouve :

- traitement des fautes des tâches
- assurance d'intégrité de la pile
- traitement surcharge de ressources
- traitement de redémarrage soudain
- détection de faible tension
- Horloge watchdog logiciel

Nano-RK est un système d'exploitation monolithique ce qui ne le fait pas un choix facile du point de vue de reconfiguration.

Ce système d'exploitation a besoin de moins de 2K octets de RAM et 16K octets Flash.

#### 1.3.3.4 RetOS

RETOS (39) est un système d'exploitation multi-tâche, préemptif. Il assure la robustesse du système aux fautes avec deux techniques : le mode double fonctionnement et le contrôle du code d'application. L'opération en mode double sépare logiquement le kernel et l'exécution de la région utilisateur. La vérification évalue la validité du code compilé par analyse statique et le comportement d'exécution du code de l'application par contrôle dynamique.

Avec RetOS, le fonctionnement bi-mode est mis en oeuvre avec des piles différentes. Les applications en mode utilisateur utilisent la pile utilisateur et la pile du kernel est utilisée pour les appels système et pour gérer les interruptions. Le fonctionnement en mode double peut exiger des coûts temporels sérieux sur le kernel en implémentant cette idée. Pour éviter l'usage fort de la mémoire, RetOS peut utiliser une seule pile pour le kernel. La commutation des fils est effectuée juste avant de revenir en mode utilisateur, qui se fait au moment où tout le travail présent sur la pile du kernel est terminé. La pile unique pour le kernel ne permet pas d'interrompre les autres fils d'exécution lorsque le kernel est exécuté.

Afin de compenser le manque de fonctionnalité de la mémoire virtuelle en raison de l'absence de MMU, RetOS fournit une technique logicielle appelée "la vérification du code". Celle-ci consiste en des vérifications statiques et dynamiques. Vérifier le code d'application empêche les applications utilisateur d'accéder à la mémoire en dehors des limites allouées et les manipulations directes du matériel. Pour atteindre ce but, la technique inspecte le champ de destination des instructions machine. Le champ source d'instructions peut également être consulté pour empêcher l'application de la lecture du noyau ou d'autres sections protégées. La vérification statique du code vérifie directement ou immédiatement les instructions d'adressage et les sauts relatifs au cours de la compilation. Le contrôle dynamique du code vérifie l'utilisation correcte des instructions d'adressage indirecte (pendant l'exécution). Le contrôle dynamique est également requis pour l'instruction RET (anglais : Return) puisque l'adresse de retour peut être affectée par le dépassement de la mémoire tampon.

La pile unique du kernel réduit la taille de la pile exigée par chaque fil d'exécution et l'analyse de la taille de la pile assigne une pile de taille appropriée à chaque fil, automatiquement. Les systèmes de réservation multifilaires ont besoin d'une pile pour chaque fil. La longueur nécessaire pour cette pile est la somme de ressources requises par les fonctions de fil, les appels système, les gestionnaires d'interruption et la sauvegarde du contexte matériel. Le mécanisme sépare la pile de chaque fil de celui du kernel. La pile du noyau est partagé entre tous les fils. L'accès à la pile du noyau est réglementé de telle sorte que le système n'aille pas arbitrairement altérer le flux d'exécution (y compris le fil de préemption) pendant le mode kernel. Avec la préemption des fils, leurs contextes matériels sont enregistrés dans chaque bloc de contrôle du fil.

RetOS est un système d'exploitation monolithique. Une mise à jour complète du noeud doit être faite pour une reconfiguration dynamique.

Pour fonctionner, RetOS exige environ 800octets RAM et 24Koctets Flash.

#### 1.3.3.5 SOS

SOS (40) se compose d'un noyau commun et des modules d'applications dynamiques, qui peuvent être chargés ou déchargés à l'exécution. Ces modules s'occupent d'envoyer des messages et de communiquer avec le kernel via un système de table des sauts. Ils peuvent également partager des points d'entrée pour des autres modules. SOS, tout comme TinyOS, n'offre pas de support extensif pour la protection de la mémoire, mais le système la protège néanmoins contre les fautes communes.

La fonction et ses points d'entrée sont marquées avec des types en utilisant des caractères comprimés, ce qui permet au système de détecter les erreurs qui pourraient le bloquer. Par exemple, lorsqu'un module attend qu'une nouvelle version d'interface soit chargée sur un noeud qui ne fournit que l'ancienne.

SOS utilise la mémoire dynamique à la fois dans le kernel et les modules d'application, facilitant la programmation et permettant un haut usage temporel de la mémoire. Il utilise l'ordon-

nancement à priorité pour déplacer le traitement hors de son contexte d'interruption et fournir de meilleures performances pour des tâches à temps critique.

A part pour les techniques traditionnelles utilisées dans la conception du système, le kernel SOS a des modules liés dynamiquement. La priorité de la planification de tâches est flexible et il y a un sous-système de mémoire dynamique. Ces services du kernel aident avec les changements après le déploiement. Aussi, ils fournissent une interface API aux programmeurs pour les libérer de la gestion des services ou de ré-implémenter les abstractions courantes. La plupart des applications de réseaux de capteurs se produisent dans les modules qui se trouvent au-dessus du kernel. Le minimum d'une application de réseaux de capteurs en utilisant SOS se compose d'un simple module de détection et d'un module de routage au-dessus du kernel.

Un script de l'éditeur de liens est utilisé pour placer la fonction de gestion d'un module à un décalage connu dans le fichier binaire lors de la compilation. Ceci permet de le relier facilement lors de l'insertion du module. Au cours de l'insertion, une structure des données indexées est créée. Elle est utilisée pour stocker l'adresse absolue de la fonction gestionnaire, un pointeur qui décrit l'état du module et son identité. Enfin, le kernel de SOS appelle le gestionnaire en lui envoyant un message d'initialisation du module.

SOS permet de consommer moins pour une reconfiguration dynamique grâce à la possibilité de charger de nouveaux modules dynamiquement.

Du point de vue de la mémoire, SOS utilise 1.1k octets RAM et 20k octets Flash.

### 1.3.4 Systèmes d'exploitation : Résumé

La Table 1.3 présente un résumé des choses traitées dans les dernières sous-sections.

### 1.3.5 Exemples des Machines Virtuelles pour WSNs

On va maintenant regarder quelques exemples de machines virtuelles pour WSNs. L'idée de machine virtuelle a été développée pour des architectures PC à la base. Avec le temps

| Nom | Mémoire Flash | Mémoire RAM | Particularité |
|---|---|---|---|
| TinyOS | lié à l'application | " | premier OS pour WSNs |
| MantisOS | 14KO | 500 octets | fils préemptés, basse consommation |
| NanoRK | 16KO | 2KO | préemptif temps réel |
| RETOS | 24KO | 800 octets | préemptif, résilient |
| SOS | 20KO | 1.1KO | préemptif, modules dynamiques |

**TABLE 1.3:** Résumé de systèmes d'exploitation WSNs les plus connus.

elles ont évolué et leur taille s'est réduit pour pouvoir être déployées sur les architectures embarquées.

Comme déjà dit, les instructions pour les machines virtuelles sont écrites dans un langage intermédiaire qui s'appelle bytecode. Ce bytecode a une nature plus abstraite que le code machine et aussi une taille plus petite, ce qui le fait une solution plus attractive pour la reconfiguration dynamique.

Elles tombent en deux catégories : "Application-specific VM" et "General-purpose VM".

La première catégorie a la possibilité que le bytecode soit modifié avec chaque application. Les instructions les plus utilisées peuvent avoir un bytecode plus facile à décoder (par exemple, un seul octet au lieu de deux). Les opérations codées dans le bytecode sont adaptées à l'application en cause, et donc perdent de leur nature abstraite en faveur de la minimisation de leur taille. Par exemple, l'instruction "s-classe" du Maté est codée comme "01iiixxx" où "i" sont les bits correspondants à l'instruction et x les bits pour l'argument. Dû au fait que seulement trois bits sont utilisés pour l'adressage de l'argument ainsi que pour l'instruction, on ne peut avoir que 8 instructions de ce type, sur 8 arguments possibles.

Les Machines Virtuelles "General-purpose" sont largement plus grandes en taille et on besoin de plus de mémoire flash, plus de RAM, et leur bytecode est aussi plus grand et prend plus du temps à être interprété. L'avantage associé avec cette catégorie est que les applications visées sont beaucoup plus variées et vastes. Le bytecode n'est plus lié à une application particulière comme dans la catégorie précédente. En revanche, dû à leur bytecode plus grand, le coût de mise-à-jour est plus élevé, ainsi que le coût d'exécution. Par exemple, Darjeeling VM réclame

un coût d'exécution de 30-78x plus élevé que sa représentation en code machine.

#### 1.3.5.1 Maté VM

Maté (41) est un interpréteur de bytecode qui s'exécute sur TinyOS. Il s'agit d'un seul composant TinyOS qui se trouve au "sommet" logique du système de plusieurs composants, y compris les capteurs, la pile réseau et stockage non-volatile. Le code pour Maté est divisé en paquets de 24 instructions, dont chacune a la longueur d'un seul octet. Les programmes plus grandes peuvent être composés de plusieurs "capsules" selon la terminologie employée par les auteurs. En outre de la fonctionnalité, les paquets contiennent l'identification et d'information sur la version. Maté a deux piles : une pile pour les opérandes et une pile pour l'adresse de retour. La plupart des instructions fonctionnent uniquement sur la pile des opérandes. On a aussi quelques instructions qui contrôlent l'exécution du programme. Il y a trois contextes d'exécution qui peuvent fonctionner en même temps à la granularité d'instruction. Les paquets Maté peuvent se transmettre dans le réseau avec une seule instruction. Maté a à la fois un algorithme de routage ad-hoc (l'envoi d'instructions) intégré, ainsi que des mécanismes pour l'écriture de nouveaux algorithmes (l'instruction sendr).

Il existe trois types d'opérandes : valeurs, lectures de capteurs et les messages. Certaines instructions ne peuvent fonctionner que sur certains types. Par exemple, l'instruction "putled" attend une valeur sur le dessus de la pile. Cependant, de nombreuses instructions sont polymorphes. Par exemple, l'instruction d'addition peut être utilisé pour ajouter n'importe quelle combinaison des types, avec des résultats différents.

Comme déjà dit, les programmes Maté sont décomposés en capsules de 24 instructions. Cette limite permet à une capsule de rester dans un seul paquet TinyOS. En rendant la réception des capsules atomiques, Maté n'a pas besoin de mémoriser les capsules partielles, ce qui conserve la RAM. Chaque capsule contient le code type et la version de l'information. Maté utilise quatre types de capsules de code : messages, capsules pour recevoir un message, capsules minuterie et capsules sous-routine. Les capsules sous-routine permettent aux programmes d'être

plus complexes que ce qui peut être mis en oeuvre dans une capsule unique. Les applications appellent et font leur retour de sous-programmes en utilisant l'appel et les instructions de retour. Il y a des noms pour jusqu'à $2^{15}$ sous-programmes ; mais pour garder les exigences de RAM petites, la version courante mise en oeuvre n'a que quatre noms. Sa taille exige 800octets de RAM et 16Koctets de Flash.

#### 1.3.5.2 Darjeeling VM

Pour Darjeeling (42), chaque appel de méthode produit une nouvelle pile "cadre" (registre d'activation) pour l'exécution de la méthode. Les piles de cadre Java contient une section pour les variables locales, diverses autres informations internes et une pile pour les opérateurs. Les informations internes comprennent une adresse de retour, un pointeur de pile et les variables contextuelles diverses (comme un pointeur vers la méthode qui est actuellement exécuté). Traditionnellement les trames de pile sont allouées dans un seul espace pré-défini. La section de variables locales d'un cadre est localisée au début de la trame. Ceci permet de passer des informations entre les différents cadres si on les superpose (la fin d'une trame sur le début de la prochaine)

Darjeeling utilise deux piles pour les opérateurs au lieu d'une seule. Une pile détient les types de référence (pointeurs), l'autre est utilisée pour les types non-référence. Les piles sont allouées dans le même espace de mémoire, dont la taille peut être facilement obtenue par l'analyse du bytecode. Deux pointeurs de pile sont utilisés : l'un est initialisé à la limite inférieure de l'espace de pile, l'autre à la limite supérieure. La première pousse en haut, l'autre en bas. Dès qu'il s'agit d'un symbole, le collecteur peut simplement traverser la pile référence pour le trouver. Ceci va réduire la complexité de cette étape à O(n) et éliminer les faux positifs. Pendant le compactage, chaque fois qu'un objet est déplacé, le collecteur peut traverser toutes les piles de référence et mettre à jour les pointeurs correspondants.

Un des avantages de la programmation Java est qu'il fournit une interface intuitive, modelée par la concomitance de préemption multi-fil. Ceci lui permet de fonctionner sur de modèles

de concomitance basés sur des événements, fils d'exécution ou proto-fils. Aussi, même sur des systèmes qui n'offrent pas la fonction de concomitance. En effet, le fonctionnement de la machine virtuelle est une boucle qui oscille entre les appels des fonctions "dj_vm_schedule()" et "dj_exec_run()".

Darjeeling a un bytecode personnalisé qui est optimisé pour l'arithmétique 16 bits. C'est le travail de l'outil "infuser", de transformer le bytecode Java en celui de Darjeeling. Il n'existe pas toujours une correspondance mot à mot entre les instructions et la transformation peut être sensible au contexte. Dans ces cas, les instructions Darjeeling à générer dépendent du type d'entrée des instructions en Java. Pour permettre ces transformations de contexte une analyse doit être effectuée pour déterminer les types d'entrée de chaque instruction.

#### 1.3.5.3 VM*

VM* est une machine virtuelle pour la plateforme Mica. Elle utilise des concepts de JIT (compilation "Just in time") pour compiler les méthodes choisies en code natif. Il reste toujours une partie qui est interprétée mais globalement, le code est exécuté plus vite que les machines virtuelles classiques.

L'éditeur de liens incrémental ajoute une grande flexibilité à VM*, car il fournit un modèle à fine granularité et en même temps générale, ce qui offre le support pour l'extensibilité. Bien qu'il soit principalement destiné à permettre une évolution transparente aux logiciels de système avec les applications, presque toutes les fonctionnalités peuvent être ajoutées progressivement. L'éditeur de liens est également utilisé pour modifier des méthodes pendant l'enregistrement des classes. L'éditeur de liens incrémental n'est nécessaire que si l'application est susceptible à évolution, dans quel cas la distribution de code nécessaire et routines "bootloader" sont incluses dans le logiciel système. Sinon, une application VM autonome composite peut être programmée sur le noeud.

Lors de l'écriture du code dans la mémoire, les fonctions sont fournies avec un petit intervalle de garde. Si une mise à jour provoque une fonction de réduire ou augmenter en taille, elle

peut le faire sans écraser la fonction suivante. Si un dépassement est impossible à éviter, le code est déplacé vers une région plus grande, toujours en lui donnant un intervalle de garde. Comme ça, seules les références aux fonctions déplacées doivent être changées.

VM* fournit actuellement deux approches de programmation. La première est un modèle dans lequel l'application peut enregistrer les intérêts pour un certain nombre d'événements. L'application se bloque ensuite sur un appel sélectionné. Lorsque l'un des événements enregistrés se produit, des masques sont utilisés pour déterminer quel gestionnaire d'événements exécuter pour le retour de l'appel. Cette technique-ci est similaire à l'appel "select" en UNIX. La seconde est un modèle "action listener" dans lequel les gestionnaires d'événements autochtones invoquent des appels pour traiter spécifiquement la demande.

### 1.3.6 Conclusion sur les machines virtuelles

Les machines virtuelles interprètent une représentation intermédiaire des données qui s'appelle bytecode. La taille du bytecode est inférieure à la représentation en format code machine, c'est pourquoi une mise-à-jour codée en bytecode est considérée comme étant inférieure (du point de vue de la taille du fichier de reconfiguration) à une mise-à-jour codé en code machine. Les machines virtuelles tombent dans dans catégories : "Application-specific virtual machine" (ASVM) ou "General-purpose virtual machine" (GPVM).

Les ASVM ont la taille du bytecode largement inférieure et adaptée à l'application en question. Les GPVM sont plus lourdes, ont besoin de plus de mémoire flash et RAM pour s'exécuter et réclament une plus large taille au niveau du bytecode.

Grâce à son caractère abstrait, le bytecode permet d'incorporer plusieurs instructions pour l'architecture cible dans une seule instruction pour la machine virtuelle en question. Ceci correspond à une minimisation de la taille effective d'une application comparée à sa représentation en code machine. Cette solution est donc attractive pour minimiser la taille des mises-à-jour dans les WSNs, et donc de minimiser l'énergie requise pour réaliser ceci.

### 1.3.7 Diff

Une méthode de faire la reconfiguration dynamique partielle est de prendre le code machine qui était initialement sur le noeud et le comparer avec la nouvelle version. Ceci peut être approprié dans le cas de reconfiguration avec des Systèmes d'Exploitation, ainsi que dans le cas des Machines Virtuelles.

Ensuite, générer un fichier qui comprend seulement la différence entre les deux versions. C'est la solution proposée par Koshy et al. (43). Ce fichier est envoyé au noeud et il utilise un "script", un programme qui remplace les régions affectées par le nouveau code, comme décrit en Figure 1.9. On précise que l'écriture de la mémoire flash est coûteuse du point de vue énergétique. Ensuite, pour ne pas avoir besoin de déplacer de grands morceaux de code de flash lorsqu'on a un ajout, il y a des espaces libres après chaque fonction. Comme ça, si la fonction grandit d'une certaine taille, le cas dont on a parlé plus haut est couvert.

L'efficacité de la solution est difficile à quantifier, puisqu'elle dépend de la différence entre la partie présente sur le noeud et la nouvelle. S'il s'agit d'architectures matérielles hétérogènes, cette solution n'offre pas d'avantages comparée à la reconfiguration complète.

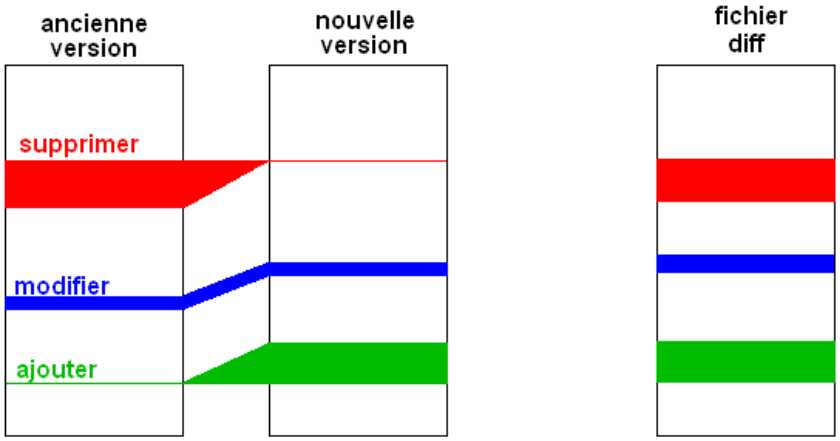


**FIGURE 1.9:** La méthode Diff.

La reconfiguration avec la méthode diff est une solution importante pour les réseaux WSNs homogènes. Le gain en énergie est difficile à estimer, car il dépend du nombre d'octets différents

entre les versions du logiciel.

Cette méthode ne répond pas aux besoins d'hétérogénéité matérielle ni logicielle. Dans ce cas, une mise-à-jour entière doit être distribuée dans le réseau afin d'effectuer une mise-à-jour.

#### 1.3.7.1 Hermes

Panta et al. proposent une solution qui s'appelle Hermes (44). Hermes fait usage de déplacements des données, en allouant les variables dans les mêmes emplacements de la mémoire entre versions et puis adopte deux approches différentes pour gérer les déplacements des données dans la mémoire.

Pour les noeuds TelosB (architecture Von-Neumann), Hermes alloue les variables qui normalement sont censées être dans la section .bss, dans la mémoire flash. Une telle approche conduit à une efficacité réduite, puisque les opérations demandent plus de temps à s'exécuter que celles de RAM, dû au temps d'accès plus élevé.

En revanche, cette stratagème permet de réduire encore plus les différences entre les différentes versions du logiciel.

Ceci fait de cette solution une approche inadaptée pour les applications qui demandent beaucoup de mémoire ou de longue durée.

En ce qui concerne les noeuds MicaZ avec une architecture Harvard, elle laisse une région de garde (10 Octets) entre les sections ".data" et ".bss". Ça conduit à la fragmentation et un usage non optimal de la RAM. Aussi, elle alloue les variables de type ".data" jusqu'à intervalle de garde.

#### 1.3.7.2 R2

Dong et al. (45) proposent une solution qui fait usage de déplacements de la data et aussi de données, en utilisant du code relogeable. Les auteurs argumentent que R2 assure un haut niveau de similitude (en gérant une diversité d'instructions tels que CALL, JMP et MOV, etc.) tout en préservant un haut niveau d'efficacité. Cette solution utilise le fichier

ELF généré par la compilation de l'application. Tout comme Hermes, une variable peut être choisie pour être relogée, dans la section ".bss". R2 est plus optimale comme solution, parce que le fichier Diff a besoin de moins d'octets. Ceci est argumenté par les auteurs, qui disent que R2 gère plus d'instructions de référence par rapport à Hermes. Ils ont découvert que certaines instructions (hormis celle de CALL) peuvent aussi être adressés de références de symboles (tels que BR ou MOV, les routines d'interruption).

#### 1.3.7.3 Zephyr

Panta et al (46) ont choisi d'implémenter un script qui aide à charger une image Diff sur un noeud d'une autre manière. Ils ont conclu que pour que les données transférées par radio soient petites, il faut que le fichier Diff soit le plus peitt possible. Or, pour produire cela, on a besoin de faire des modifications au niveau de l'application pour pouvoir avoir une similitude maximale entre les deux versions de logiciel. Dans la Figure 1.10, les appels aux fonctions fun1, fun2 et funn sont remplacés par des sauts aux emplacements fixes loc1, loc2,...,locn. Ce segment de la mémoire de programme, qui commence par l'emplacement loc1 se comporte comme un tableau d'indirection. En utilisant ce tableau, on fait les vrais appelles aux fonctions.

Lorsque l'appel à la fonction visée est de retour, le tableau d'indirection dirige le flux d'exécution à la ligne d'après l'appel à la fonction (loc-x, où x=1,...,n). Comme avantage, lorsque l'application entière est modifiée ; ou les fonctions qui la composent sont déplacées, il n'y a pas besoin de modifier les adresses d'appel de fonction. Cette approche assure que des segments de code, hormis le tableau d'indirection, préservent une similitude maximale entre versions. Ceci est possible puisque les fonctions sont dirigées vers des emplacements fixes, même si les fonctions ont bougé dans le code machine.

### 1.3.8 Machines Virtuelles et hétérogénéité

L'hétérogénéité matérielle a été un des arguments les plus forts pour développer les machines virtuelles. Ici, au lieu d'envoyer une nouvelle fonctionnalité en code machine

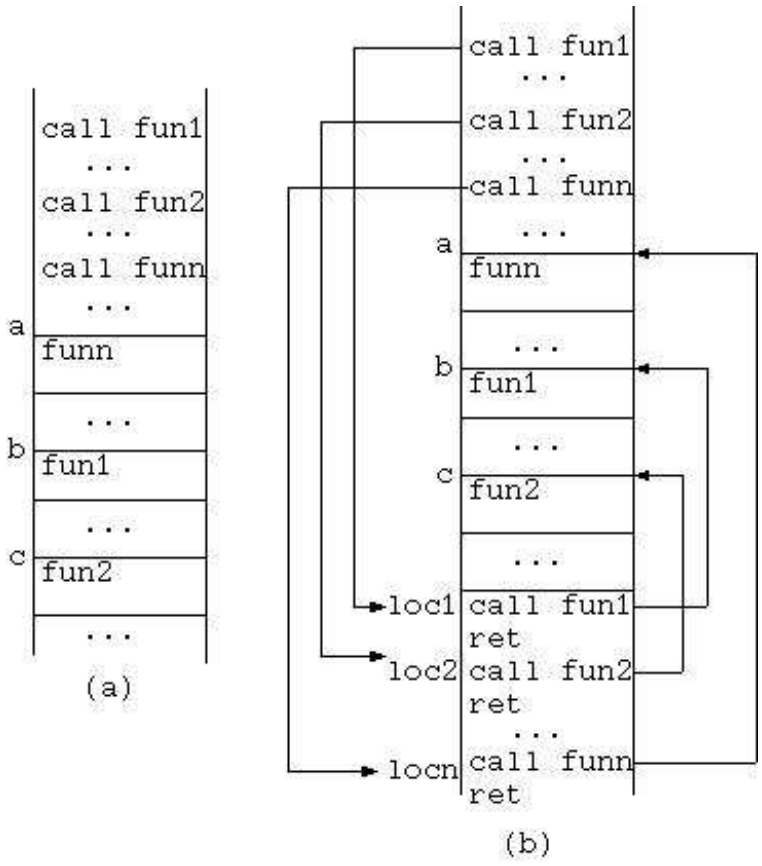


**FIGURE 1.10:** Indirection d'appel de fonction dans Zephyr.

(dépendant de l'architecture), on peut l'envoyer en bytecode. Bien sûr, il s'agit d'un bytecode réduit, adapté aux petites architectures. Chaque instruction pour la machine virtuelle peut englober plusieurs instructions pour le processeur final. Il faut préciser que même si ce bytecode est considéré comme bas-niveau, il est plus abstrait donc plus haut niveau que le code machine, mais largement moins abstrait qu'un langage haut-niveau comme C ou Java. Toutes les machines virtuelles présentées ici sont écrites pour le langage Java. Il existe différentes approches pour chacune.

### 1.3.9 Hétérogénéité

La norme IEEE 802.15.4 a permis l'hétérogénéité radiofréquence dans les WSNs en élaborant des spécifications qui se prêtent à partir de la couche physique jusqu'à la couche MAC. L'hétérogénéité est la possibilité que les noeuds dans un réseau soit différents. Il peut s'agir de l'hétérogénéité logicielle (différents systèmes d'exploitation) ou matérielle (différentes architectures matérielles). Aussi, plus souvent, on peut être dans la situation d'utiliser plusieurs architectures matérielles dès qu'on déploie le réseau.

Bien évidemment, selon le Tableau 1.2, on peut voir une hétérogénéité concernant les solutions existantes, malgré le fait qu'au tout départ ces réseaux ont été conçus d'une manière homogène, c'est-à-dire qu'ils comprenaient les mêmes noeuds. Rapidement, des réseaux hétérogènes sont apparus. Des nouveaux composants discrets sont devenus de moins en moins chers et améliorés de point de vue énergétique. Des nouveaux noeuds, plus performants ont été déployés pour donc coexister avec les solutions déjà existantes. L'hétérogénéité matérielle a été atteinte.

Certaines architectures matérielles sont plus puissantes que d'autres, il est envisageable qu'il ait des différents systèmes d'exploitation qui s'exécutent sur elles. Ou bien, on peut avoir le cas où il existe qu'un seul OS mais avec différentes versions sur une partie du réseau. Avec ça, on peut parler d'une hétérogénéité complète matériel-logiciel.

L'hétérogénéité des systèmes d'exploitation sera développée plus en détail dans les prochaines sous-sections.

### 1.3.10 Conclusion sur les modèles de programmation

Comme l'état de l'art présenté le démontre, les modèles de programmation se divisent en deux : des approches en code machine ou des approches liées à l'usage d'une machine virtuelle. Les approches en code machine peuvent être liées à un système d'exploitation monolithique ou modulaire. Elles peuvent être faites à travers un fichier de différences dénommé "fichier delta", avec la méthode diff. La reconfiguration peut aussi être faite à l'aide d'une machine virtuelle, qui peut être liée à l'application (application-specific virtual machine) ou avoir un caractère plus général (general purpose virtual machine).

## 1.4 Comparatif Machines Virtuelles vs Compilateurs

Le compromis le plus dominant dans le cas de Machines Virtuelles est entre la flexibilité et le coût de mise à jour. Avec une machine virtuelle, on est limité à un nombre fini d'opérations possibles, implémentées dans le bytecode. Mais en même temps, le coût de les envoyer par interface radio est moins important grâce à la nature du bytecode, plus abstrait que le code machine. Néanmoins, il reste encore beaucoup d'améliorations possibles. Un argument est que les applications WSNs contiennent généralement une forme de traitement du signal. Ces algorithmes sont généralement intensifs côté processeur. Le bytecode est inefficace pour encapsuler les encapsuler, par sa nature bas niveau. Deuxièmement, il y a un manque général d'interopérabilité entre ces machines virtuelles et les systèmes d'exploitation existants. Ils ont tendance à être soit lié à un système d'exploitation spécifique ou à ré-écrire entièrement la philosophie.

Pour donner des exemples, Maté ne peut pas être utilisé dans l'absence de TinyOS. VM-STAR met en oeuvre soit un modèle de sélection ou d'un écouteur d'action. Le modèle de sélection est celui dans lequel une application peut enregistrer ses intérêts dans un certain nombre d'événements. L'application se bloque ensuite sur un appel dénommé "select" pour chaque événement pour auquel l'application est intéressée. Cette philosophie imite le principe d'un système d'exploitation.

Les fonctionnalités échangées entre les noeuds d'un WSN ne devraient pas interférer avec le logiciel système en cours d'exécution sur le noeud (machine virtuelle ou le système d'exploitation). Seulement dans le pire des cas où une fonctionnalité a des contraintes de temps réel, elle pourrait informer le logiciel système sur ces points. Nous voulons abstraire l'architecture d'un noeud et le considérer dans un contexte hétérogène. En faisant cela, nous voulons offrir du support pour les systèmes d'exploitation existants tout en étant capable d'envoyer des fonctionnalités vers un noeud d'une manière plus optimale que le bytecode.

## 1.5 Contexte du travail

Comme vu précédemment, les progrès récents sur la reconfiguration d'un réseau WSN tourne autour d'usage des machines virtuelles. Typiquement adaptées aux architectures 8-bits ou 16-bits, elles emploient un sous-ensemble du bytecode Java. Aussi, elles ajoutent leur propre extension. Leur approche est pertinente puisqu'elles offrent le support pour l'hétérogénéité matérielle et, souvent, la reconfiguration dynamique. Mais, comme elles sont des machines virtuelles, il y a forcément une interprétation du bytecode. Ceci veut dire un temps plus long d'exécution. De plus, elles sont liées au système d'exploitation qui s'exécute sur le noeud.

Il est souhaitable de garder la possibilité de reconfiguration dynamique (dans un contexte hétérogène HW/SW), mais en enlevant cette partie d'interprétation, chère du point de vue énergétique. En même temps, il faut avoir accès au plus de RAM possible.

On a établi que le chemin à suivre c'est de construire un nouveau langage haut-niveau, capable de rivaliser avec la petite taille du bytecode. On l'a appelé MinTax(anglais : "Minimal Syntax"). On voulait réduire même davantage le coût de la mise à jour du logiciel du noeud. Ce langage sera compilé in-situ, pendant la période que le noeud est déployé.

On a focalisé notre effort sur ce langage en lui donnant la possibilité d'accéder au matériel directement, contrôler la lecture des entrées analogiques, ainsi que les sorties PWM. On lui a donné la syntaxe pour les conditions, boucles, boucles conditionnelles et il offre du support pour les imbrications des clauses. Les appels des fonctions sont supportées (avec ou sans paramètres) et elles peuvent retourner des valeurs.

## 1.6 Le compilateur : un aperçu général

Une définition d'un compilateur peut être : un programme s'exécutant sur une architecture matérielle qui prend une entrée écrite en un langage haut-niveau et produit du code machine pour cette architecture. Ensuite, l'architecture peut l'exécuter. Les compilateurs ont connu une évolution lente au début, car la partie matérielle n'était pas aussi puissante qu'aujourd'hui. Ces programmes n'ont pas été implémentés jusqu'au moment où les avantages à réutiliser les logiciels sur différents processeurs ont dépassé l'effort d'écrire un compilateur. La capacité limitée des premiers ordinateurs donnait aussi beaucoup de problèmes d'implémentation.

A la fin des années 1950, les langages indépendants du matériel ont été proposés. Par conséquent, plusieurs compilateurs ont été développés. Le premier a été écrit par Grace Hopper en 1952, pour le langage A-0. Ensuite, l'équipe FORTRAN à IBM a créé le langage avec le même nom. Dans plusieurs domaines d'application, l'idée a été adoptée. À cause de l'extension des fonctionnalités des nouvelles langages de programmation et l'augmentation de la complexité des architectures d'ordinateurs, les compilateurs sont devenus de plus en plus complexes.

Les premiers ont été écrits en langage d'assemblage. Le premier compilateur "self hosting" (écrit en même langage qu'il peut compiler) a été créé pour LISP par Tim Hart et Mike Levin au MIT en 1962. À partir de 1970, les gens ont commencé à toujours implémenter le compilateur en même langage qu'il compile (malgré le fait que C et Pascal étaient des choix populaires comme langages d'implémentation). Construire un compilateur pose un problème de démarrage (anglais : "bootstrapping"). Pour qu'un compilateur puisse faire son travail, il faut d'abord qu'il existe, qu'il soit créé. Il faut donc que lui aussi soit compilé par un autre compilateur en utilisant un autre langage (comme le compilateur LISP de Hart et Levin par exemple).

Plusieurs autres langages ont été développés pendant des années. Figure 1.11 montre leur histoire jusqu'à 2004.

En ce qui concerne le langage pour un compilateur donné, il existe beaucoup de paradigmes différents, mais le plus souvent, ils sont divisés en deux catégories : déclarative ou impérative.

Un langage déclaratif est une manière de programmer dans laquelle on décrit ce que le programme doit faire et non pas comment le faire. C'est-à-dire qu'on ne stipule pas des clauses d'un algorithme (47). Programmer en utilisant des clauses déclaratives c'est écrire des programmes très similaires à la logique formelle. De plus, les calculs faits par la machine qui exécute le programme comprennent des déductions dans le même espace logique. Cette manière de programmer, de plus en plus populaire, parce que elle peut beaucoup simplifier l'écriture des programmes parallèles.

Les langages déclaratifs incluent ceux des expressions régulières, celles de programmation fonctionnelle et programmation logique par exemple. Pour être plus précis, la programmation logique fait usage de la logique mathématique et des clauses-théorèmes. Le programmateur doit assurer la vérité de programmes écrits sous forme logique et c'est le générateur du modèle qui est responsable de les résoudre d'une manière efficace.

Toujours dans le contexte de langages déclaratifs, la programmation fonctionnelle opte pour calculer le résultat en évaluant des fonctions mathématiques. Donc cela repose sur l'usage de

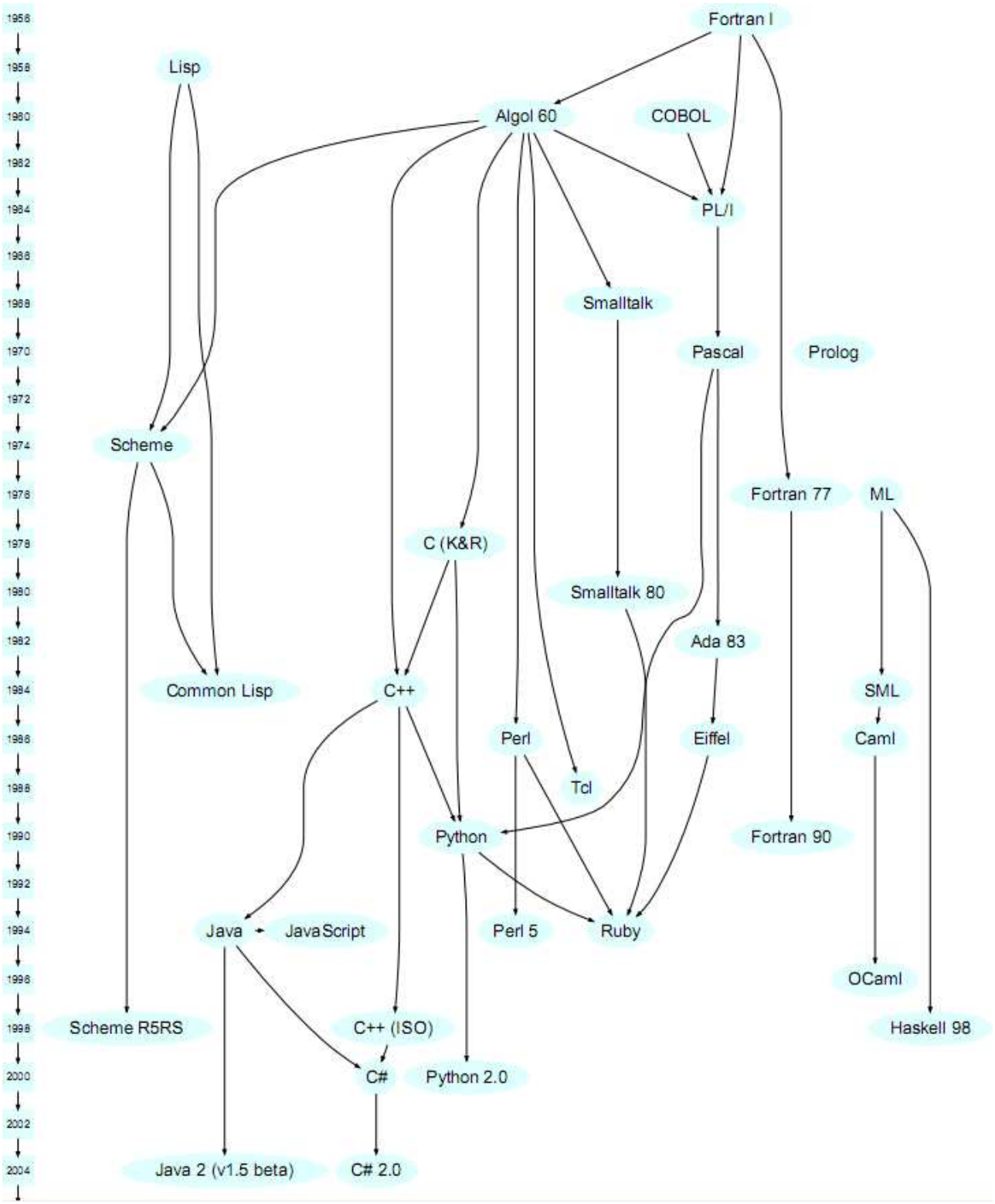


**FIGURE 1.11:** Evolution des langages de programmation (6).

fonctions et peut éviter l'usage de variables d'état ou d'objet immuable (un objet "constant") qui ne peut pas être modifié après sa création).

Si on regarde le concept de programmation impérative, on trouve des choses plus familières. Cette façon de programmer implique l'utilisation de clauses qui ont comme effet un changement d'état. Le programme passe par différents états, tout en exécutant un algorithme. L'algorithme dit à la machine les séquences de commandes à faire.

La programmation procédurale est une subdivision de la programmation impérative, où les commandes sont incluses dans des fonctions (aussi connues comme "procédures").

Si le changement d'état d'algorithme est localisé au niveau de la fonction, ou limités par rapport à leur arguments et leurs valeurs de retour s'appelle "programmation structurée".

En effet, grâce à la façon simple de faire la maintenance sur du code structuré, ainsi que pour implémenter un algorithme, ce paradigme a été adopté dès le début de la programmation. Descendant de l'époque d'assemblage, pour donner d'instructions simples à des processeurs 4-8 bits, c'est bien évident qu'une approche impérative est meilleure. Ce fait est aussi démontré dans le processeur lui-même, où on a des micro-instructions impératives, pour chaque instruction exécutée. Par exemple, l'instruction pour un processeur fictif "ADD A,B" doit d'abord prendre l'opérande B dans le registre accumulateur d'ALU, puis faire l'addition et mettre le résultat en A. Aussi, A peut être une adresse de mémoire et dans ce cas il faut accéder à l'adresse et puis écrire le résultat. En général les instructions qui font un accès à la mémoire prennent plus de temps (voire plus de cycles d'horloge).

### 1.6.1 Grammaire du langage

Chaque langage humain a des règles qui disent la manière dont la phrase est structurée. Ainsi, on a le sujet, l'action (le verbe) et d'autres parties qui peuvent être présentes ou pas.

Lorsque l'on parle d'un langage de programmation, la grammaire d'un langage est un formalisme qui donne les règles syntaxiques pour celui-ci. Il existe des grammaires LR, LL et

indépendantes de contexte. Un interpréteur LR (anglais "Left to right") lit l'entrée de gauche à droit (comme l'aperçu visuel) et produit une dérivation droite. Le terme "interpréteur LR(k)" est aussi utilisé, où k est le nombre de symboles qui ne sont pas encore lus. D'habitude k est égal à 1 et le terme interpréteur LR est souvent compris comme ça. La syntaxe de beaucoup de languages de programmation peut être définie par une grammaire LR(1), ou proche de cela. Par conséquent, les grammaires LR sont souvent utilisées par les compilateurs pour l'analyse syntaxique de l'entrée. Un interpréteur LR est dit de faire l'interprétation bottom-up puisqu'il essaye de déduire les productions de niveau terminal en construisant le contexte avec les feuilles d'arbre abstrait. Plus sur ce point va être dit dans les sections suivantes.

Par contre, un interpréteur LL (anglais "Left to Left") marche d'une manière top-down. Il fait toujours la lecture des entrées de gauche à droite et il construit une dérivation gauche.

### 1.6.2 Notation polonaise inverse

Puisque les règles d'un langage sont décortiquées en sous-règles, la manière dont un interpréteur (ou compilateur) voit le fichier d'entrée s'appelle "notation polonaise inverse". Ceci est dû au caractère récursif des règles, qui fait que les sous-règles sont réduites à des non-terminaux afin que la règle mère puisse être réduite.

La notation polonaise inverse (48) est une notation mathématique où chaque opérateur suit ses opérandes. Elle s'appelle aussi notation Postfix et a comme avantage le fait que les parenthèses peuvent être supprimées. Cette notation est utilisée pour réduire l'accès à la mémoire et utiliser la pile pour évaluer les expressions dans un langage formel. Les calculs peuvent être faits dès que l'opérateur est trouvé. Comme ça, les opérations sont calculées une à la fois.

Cette notation est facile à comprendre, parce qu'elle ressemble aux calculs qu'on peut faire manuellement. On écrit déjà les numéros, afin d'effectuer les opérations arithmétiques. Il existe aussi une pile automatique, qui permet le stockage des résultats intermédiaires pour plus tard. En effet, c'est cette caractéristique qui permet l'évaluation des expressions de complexité variable. L'état de la machine dans le contexte de l'expression polonaise est toujours une pile de valeurs

qui attend une opération, c'est impossible de mettre un opérateur sur la pile. Dès qu'un est trouvé, les opérateurs sont pris de la pile d'attente et l'opération est effectuée.

Un exemple pour la notation polonaise inverse pourrait être : "2 3 + 1 -" ce qui correspond à l'expression arithmétique "2 + 3 - 1".

### 1.6.3 Structure interne d'un compilateur

Les différentes parties d'un compilateur classique sont présentées dans la Figure 1.12. Il s'agit de deux phases : premièrement, il y a la phase d'analyse qui est encore subdivisée en Analyse Lexicale, Analyse Syntaxique et Analyse Sémantique. Après quoi, la prochaine phase est celle de synthèse, où il y a une optimisation intermédiaire du code (facultative). La synthèse finit avec la génération du code, qui est le résultat de la compilation. Ces différents stages seront abordés en détail prochainement.

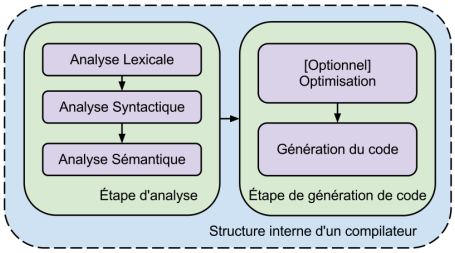


**Figure 1.12:** Structure classique d'un compilateur.

Au coeur du compilateur se trouvent l'analyseur lexical et l'analyseur sémantique. On va se concentrer premièrement sur l'analyseur lexical. Celui-ci est responsable d'échantillonner l'entrée en unités. A leur tour, celles-ci s'appellent atomes. Un atome peut être un seul caractère ASCII ou bien le nom d'une fonction, sur deux octets par exemple. L'analyseur est guidé par un ensemble de règles qui disent ce qui peut poursuivre. C'est lui qui fait une première vérification de l'entrée. S'il y a une erreur dans la syntaxe, le processus de compilation s'arrête sans produire du code objet. Plus d'informations sur les erreurs d'entrée sont données dans la section "L'analyseur sémantique".

L'Analyse Sémantique est responsable de l'étude du sens des constructions du programme d'entrée. Par analogie, une langue parlée par les humains a un verbe et un ordre de mettre les mots dans la phrase, elle est garante que cet ordre soit respecté. Sinon, la compilation va s'arrêter et une erreur va être signalée. Durant cette phase, l'analyseur sémantique va faire des inférences sur l'entrée selon un fichier de grammaire où se trouvent des règles. Ces règles seront premièrement détaillées à l'aide de leur sous-règles afin qu'elles puissent être réduites (achevées). La table de symboles ainsi que les structures de métadonnées sont écrites en même temps.

## 1.7 Conclusion

Dans cette section on a présenté l'état de l'art concernant la reconfiguration dynamique au niveau de noeud dans les WSNs. Il existe deux grandes approches pour cela, qui font usage du code machine ou du bytecode pour les Machines Virtuelles. Le code machine peut être transmis à un système d'exploitation résident sur le noeud, entièrement (systèmes d'exploitation monolithiques) ou de manière modulaire (systèmes d'exploitation modulaires). Les scripts Diff permettent de reconfigurer un noeud en lui envoyant seulement la différence entre la nouvelle version du logiciel et celle qui s'exécute déjà sur le noeud.

Ces approches n'offrent pas le support pour une hétérogénéité matérielle qui pourrait être présente dans le réseau. C'est pourquoi les Machines Virtuelles ont été développées.

Dans un deuxième temps, on a présenté les particularités associées avec les langages de programmation et la structure classique d'un compilateur. Ceci sera utilisé comme introduction pour la prochaine section.

Cette prochaine section va traiter la possibilité d'utiliser un langage de programmation haut-niveau, et à travers d'un compilateur, générer le code objet directement sur les noeuds. Tout comme avec les machine virtuelles, l'hétérogénéité matérielle sera assurée grâce au caractère

abstrait du langage. D’un autre côté, puisque le code serait généré par un compilateur, ceci prendra la forme de code machine. Contrairement à l’approche des machine virtuelles, ce code ne sera pas interprété et donc exécuté en format native pour l’architecture cible visée, ayant besoin de moins de temps pour l’exécution.

# 2 MinTax

## 2.1 Cahier de charges pour la reconfiguration dynamique

Pour pouvoir répondre aux problématiques liées à la reconfiguration dynamique on a identifié plusieurs critères de performance. Premièrement, le temps dont l'interface radio a besoin pour faire le transfert d'une fonctionnalité. Le fait que la mise-à-jour soit modulaire ou monolithique joue sur le temps d'écriture de la mémoire flash et pourrait mettre le noeud inutilisable pendant cette période. Il est désiré que ce temps soit minimisé. Le coût additionnel pour l'exécution de la fonctionnalité doit être pris en compte aussi. Un code interprété aura besoin de plus de cycles d'horloge qu'un code en format natif à l'architecture matérielle visée.

Enfin, une mise-à-jour peut viser (ou non pas) plusieurs architectures matérielles et logicielles. Si la mise-à-jour contient du code objet pour l'architecture visée, elle est liée à un jeu d'instructions particulier et ne peut être utilisée que pour une architecture matérielle. Si elle a un caractère abstrait, une seule peut être employée pour plusieurs cibles matérielles ou logicielles.

On a comparé l'état d'art à travers les critères de performance identifiés. Cette comparaison est démontrée dans la Figure 2.1.

Concernant les mise-à-jour par solution Delta, bien que celles-la sont économes du point de vue de la transmission radio, elles ne visent qu'une architecture matérielle ou logicielle. Les systèmes d'exploitation modulaires ou monolithiques n'offrent souvent pas (avec quelques

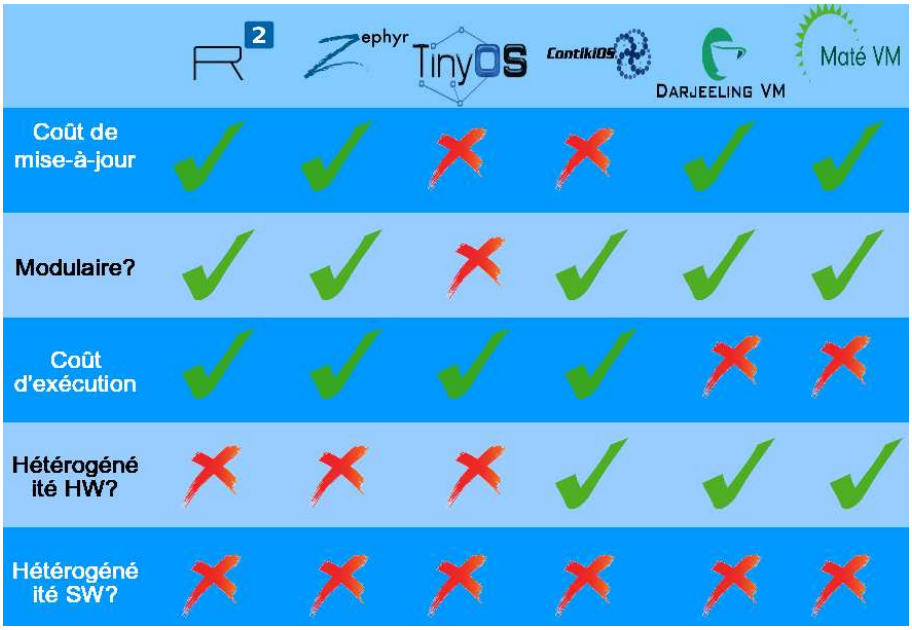


**Figure 2.1:** Comparaison entre différents types de reconfiguration dynamique.

exceptions) du support pour une hétérogénéité matérielle.

La question d'une hétérogénéité logicielle (plusieurs systèmes d'exploitation) ne se pose même pas, les mises-à-jour visant des systèmes d'exploitation bien particulières.

Les machines virtuelles ont été créées pour répondre à la problématique d'hétérogénéité matérielle. Elles utilisent une représentation intermédiaire des données qui s'appèle ń bytecode ż et qui est indépendante de l'architecture matérielle. Cependant, ce bytecode est particulier pour chaque Machine Virtuelle, donc il n'adresse pas le problème d'hétérogénéité logicielle. De plus, le bytecode doit être décodé à chaque exécution de la fonctionnalité, ce qui augmente le coût d'exécution de celle-ci.

Les machines virtuelles sont souvent liées à des systèmes d'exploitation particuliers (par exemple Maté à TinyOS ou VMSTAR à OSSTAR). Regardant l'analyse présentée dans les derniers paragraphes, on s'aperçoit qu'il y a peu de solutions qui adressent l'hétérogénéité matérielle et aucune qui offre du support pour l'hétérogénéité logicielle. C'est pour cela qu'on a développé une approche propre.

Pour pouvoir minimiser la consommation énergétique durant la reconfiguration d'un noeud, trois points qu'une solution éventuelle doit supporter ont été identifiés.

- un temps d'envoi par radiofréquence minimal
- support pour l'hétérogénéité logicielle
- support pour l'hétérogénéité matérielle

Avec les derniers deux points, on pourrait s'assurer que le code ne soit pas envoyé plusieurs fois, pour chaque architecture logicielle/matérielle différente qui puisse exister dans le réseau. Une solution est donc de créer un nouveau langage suffisamment haut-niveau pour envelopper beaucoup d'opérations dans une syntaxe qui n'est pas trop lourde. Un langage particulièrement adapté à la reconfiguration dynamique. C'est-à-dire une syntaxe qui n'occupe pas trop de temps dans la transmission radio. On a décidé que la manière classique de programmation d'un noeud, soit avec du code machine (avec sa manque du support pour l'hétérogénéité matérielle) soit avec du bytecode (avec son long temps d'interprétation et la manque d'hétérogénéité logicielle) ne sont pas optimales pour les besoins d'un WSN.

Cette syntaxe doit réponde aux besoins identifiés. Ensuite, on a envisagé un compilateur adapté à chaque architecture HW avec du support pour plusieurs systèmes d'exploitation, donc une solution multi-SW. Ce compilateur pourrait compiler la syntaxe et la transformer en code objet pour le matériel ciblé.

Dans les prochaines sections, plus de détails sont donns en ce qui concerne les besoins pour le tandem langage-compilateur et les limitations afférentes.

## 2.2 Principes : Compilateur MinTax adapté aux WSNs

Les caractéristiques matérielles des noeuds ont été pris en compte et ont été considérés comme exigences du langage MinTax. Parce que le matériel est de faible coût (et parfois même jetable après usage), sa puissance de calcul est basse. Un aperçu de ceci peut être trouvé dans la figure 2.2. Comme on peut le voir, la taille de la flash dépasse rarement la limite de 128 kOctets. Bien évidemment, avec le progrès technologique, ces paramètres vont augmenter.

Souvent, dans les WSN, il n'y a pas trop de calcul sur des types de donnés complexes, donc il n'y a pas forcément besoin de les supporter.

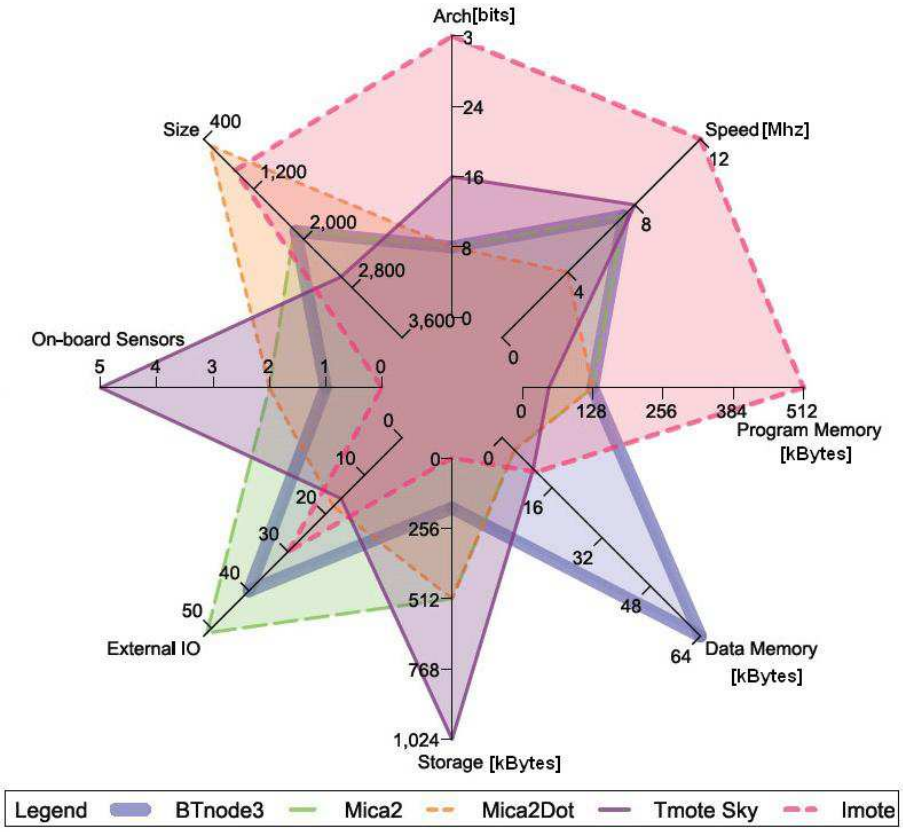


**Figure 2.2:** Comparaison entre différents types des noeuds (7).

On a identifié quelques points que le compilateur doit absolument supporter.

On parle ici des aspects classiques des fonctions, par exemple. La fonction est l'unité élémentaire de MinTax. Chaque fonction peut appeler d'autres fonctions, ainsi qu'être appelée elle-même. Plus de détails sur ce sujet sont donnés dans section la section "Fonctions en MinTax".

### 2.2.1 Origine de MinTax

La syntaxe de MinTax est inspiré par C et Perl. Les noms des fonctions sont minimisés (un ou deux caractères), pour ne pas occuper d'espace dans le paquet radio. Le but final est de minimiser la transmission radio concernant la mise à jour du logiciel qui s'exécute sur le noeud. Il n'y a pas de traitement des erreurs dans la syntaxe ou sémantique du code. On considère que s'il est envoyé par radio, il peut générer un code natif correct. Ici, c'est souvent le cas qu'il n'y a pas un utilisateur qui peut voir une erreur et la corriger. Il peut s'agir des noeud auquel on ne peut pas accéder directement, à quelques dizaines de sauts (anglais : "hops") distance. Même si une erreur était identifiée, le coût du dialogue d'aller-retour pour la corriger entre ce noeud et l'utilisateur serait trop cher.

Cependant, le code est cross-compilé sur un ordinateur et vérifié afin d'être distribué dans le réseau par un développeur de logiciel. Le compilateur de MinTax peut être compilé et son exécution peut être démarré sur un ordinateur, en faisant usage d'un utilitaire de configuration.

### 2.2.2 Principes MinTax : Clauses supportées

Il y a des boucles d'itérations qui sont supportées. Le "for" et "while" du langage C sont présents. Les imbrications de ces clauses sont supportées. Plus des détails dans le chapitres "Clauses itératives et conditionnelles" et "Imbrication en MinTax".

Les sauts conditionnels sont supportées en MinTax. Il s'agit des clauses "if" et "switch-case" du C ; discutées en section "Sauts conditionnels".

L'accès direct au matériel en utilisant MinTax est supporté. On peut lire directement une patte analogique dans une variable, ainsi que demander une sortie MLI d'une valeur sur un

autre. Les sections "Conversion analogue-numérique en MinTax" et "Pulse Width Modulation en MinTax" vont détailler ce sujet davantage.

Le compilateur utilise la mémoire RAM comme tampon pour des opérations intermédiaires de compilation. Comme il est intéressant d'avoir accès à cette mémoire après la compilation, on a recours à l'allocation dynamique. Le résultat de la compilation sera écrit dans la mémoire à une adresse connue auparavant. Ce dernier paragraphe est détaillé en sections "La fonction malloc du Compilateur MinTax" et "La carte de mémoire flash".

### 2.2.3 Principes : Coexistence avec d'autres fonctionnalités

À savoir, les communications entre plusieurs fonctionnalités présentes sur les noeuds des WSNs. Celles-ci sont la principale cause pour laquelle la compilation in-situ n'a pas été abordée auparavant dans les WSNs. Un compilateur in-situ doit (i) être développé pour une architecture particulière et tout (ou en grande proportion) réécrit pour une autre ; et (ii) il doit pouvoir donner la possibilité aux fonctions compilées de communiquer avec le logiciel déjà présent sur le noeud.

Aujourd'hui il existe des systèmes d'exploitation qui offrent la possibilité de tuyaux entre les différents processus, aussi que des fonctionnalités de temps réel et multitâche préemptif. Sinon, une implémentation locale de malloc() sera utilisée et la communication entre les processus sera faite avec de la mémoire partagée.

Il y a deux possibilités pour générer du code par un compilateur. Il y a le mode "Position independent code" ("Code indépendant de la position") et code relocalisable. Pour tenir compte de plusieurs choix pour le système d'exploitation possible, on a décidé de générer du code du type indépendant de la position. C'est-à-dire que le code fait usage des adresses absolues et n'a pas besoin d'un éditeur de liens pour le lier à la fonctionnalité qui s'exécute sur le noeud.

### 2.2.4 Principes : Syntaxe

Avec MinTax, les clauses commencent toujours avec un mot-clé. Les mots-clé sont écrits avec des lettres majuscules et ont la longueur d'un caractère. La capitalisation délimite le mot-clé de son paramètre, que ce soit une variable ou une constante. Par conséquent, on peut enlever les espaces, parenthèses, et d'autres caractères qui ajoutent à "l'overhead" et qui font augmenter la taille du programme. Lorsque l'on pense à un langage de programmation, on pense à des blocs d'instructions. Ces instructions peuvent être autonomes ou dans des fonctions. Les fonctionnalités en WSNs ont tendance à s'exécuter plusieurs fois, on peut aussi parler de parties du code qui sont exécutées dans des boucles. Par ailleurs, il y a des instructions qui font un type de calcul sur des arguments ou des variables. En conclusion, il vaut mieux mettre ces instructions dans des fonctions. Ces blocs dans les fonctions sont séparés par des délimiteurs (par exemple, ' ;' par défaut, comme en C). Les fonctions peuvent avoir des valeurs de retour. Les implémentations des fonctions ont des marqueurs de début et fin.

En tenant compte de tout ce qu'on a déjà dit, on voudrait aussi pouvoir tester le compilateur sur un PC pour voir ses points faibles et limitations. Donc, on a ajouté cette possibilité ainsi que l'option que la sortie soit en code machine ou instructions d'assemblage. Cela permet aussi une analyse du programme pour les éventuelles erreurs de syntaxe ou sémantique.

Un paramètre de configuration dit si la sortie sera dirigé vers l'interface UART ou si elle sera écrite dans la mémoire flash.

# 2.3 Le Compilateur pour MinTax

## 2.3.1 Aperçu général du compilateur de MinTax

Les plateformes supportées par notre compilateur sont présentées en Table 2.4. Comme on peut voir dans le tableau, le support pour l'hétérogénéité matérielle est présent. Des réseaux composés par différents noeuds peuvent être programmés en utilisant notre solution. Les besoins de taille pour la RAM et la flash sont présentés dans le Tableau 2.5. Dans le même tableau on présente quelle couche radio est utilisée pour chaque plateforme matérielle. Comme il y a plusieurs architectures matérielles présentes, le compilateur a des variations selon la cible visée. Ceci est présenté dans la Figure 2.3. Ceux-ci seront abordés prochainement. Pour l'instant la partie commune entre eux est présentée.

Chaque architecture a deux modes différents de sortie : "Debug" et "Release". Le mode "Release" produit du code machine qui sera écrit dans la mémoire flash du microcontroleur. Le mode "Debug" sort le résultat de la compilation par l'interface UART (si le compilateur s'exécute sur un microcontrôleur) ou la sortie standard (PC). Dans ce dernier cas, la sortie est en fait un fichier d'assemblage mélangé avec des commentaires de type C. Ceci aide à pouvoir lire ce fichier. Il faut préciser que les besoins de flash vont augmenter si le mode "Debug" est actif, car toutes les séquences de caractères doivent être conservées en flash.

| WSN Node | MCU | Architecture / Radio | Osc. Freq | Flash | RAM |
|---|---|---|---|---|---|
| Mica2 | ATMega128L | 8bits/433Mhz | 7.3728Mhz | 128kB | 4kB |
| AVRRaven | ATMega1284p | 8bits/2.4Ghz | 4Mhz | 128kB | 16kB |
| Zolertia Z1 | MSP430f2617 | 16bits/2.4Ghz | 16Mhz | 92kB | 8kB |
| - | MSP430f5438 | 16bits/- | 16Mhz | 256kB | 16kB |

**Table 2.1:** Plateformes supportées par le compilateur MinTax.

| WSN Node | Flash usage | RAM usage | Radio Stack |
|---|---|---|---|
| Mica2 | 30kB | 175 bytes | TI |
| AVRRaven | 30kB | 175 bytes | Contiki |
| Zolertia Z1 | 27kB | 175 bytes | uRacoli |

**TABLE 2.2:** Besoin de mémoire pour le compilateur MinTax et pour la couche radio du noeud.

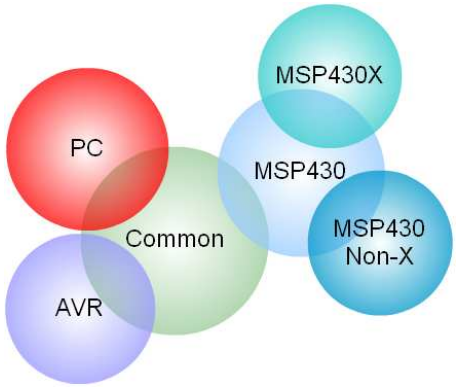


**FIGURE 2.3:** Compiler's supported hardware architectures

Comme le système d'exploitation SOS, le résultat de la compilation de MinTax est de type PIC (anglais : Position Independent Code). C'est-à-dire que le code machine est construit de telle manière qu'il peut être exécuté quelle que soit sa position en mémoire. PIC est couramment utilisé pour des bibliothèques partagées, parce que ce sera possible de les lier à l'application au moment d'exécution. Il peut y avoir beaucoup d'applications qui en utilisent, du coup cela aide à réduire la taille de l'application finale.

Il faut préciser que dans le cas de microcontrôlleurs PIC, les sauts sont contraints dans la limite de 4kOctets. S'il s'agit de programmes plus gros de 4kOctets, et il y a un saut dehors cette région, l'adresse sera tronquée. Ceci s'applique également aux appels de fonctions. Pour résoudre le problème lié à cela (ainsi que pour des autres raisons présentées plus tard), on utilise un tableau de correspondance pour les fonctions.

Le compilateur de MinTax a été écrit lui-même dans un autre langage, bien évidemment. Celui-ci était C, et une fois écrit, il a fallu le compiler pour pouvoir l'exécuter sur les noeuds supportés.

Pour l'analyseur lexical, la syntaxe a été définie en utilisant des expressions régulières. Le

langage utilisé est re2c (49) (Regular Expression to C). Re2c prend le langage défini par des expressions régulières et génère un fichier C qui est en fait l'Analyseur de Syntaxe (Lexicale). La syntaxe est celle qui fait la liaison entre les terminaux de la grammaire (présents dans la prochaine étape d'analyse, celle sémantique) et les différentes notations de MinTax. Par exemple, la règle lexicale "BEGIN : := " déclare comment l'implémentation d'une fonction doit commencer.

L'Analyseur Sémantique est, lui aussi, écrit dans un métalangage sous forme de règles de grammaire hors-contexte. Une règle hors-contexte peut être définie sous la forme : A→B, où "A" est un symbole non-terminal et "B" est soit vide, soit lui aussi non-terminal, soit un terminal. On donne un exemple, pris de l'implémentation du compilateur MinTax, pour définir la sémantique d'une fonction :

*funcDeclaration : := IDENTIFIER formalParameters BEGIN localVarDecl programBlock END functionImplicitReturn SEMICOLON.*

Ici, les terminaux sont en lettre majuscule et les non-terminaux ne sont pas, ils ont leur propre définition ailleurs dans le fichier de grammaire.

L'analyseur sémantique va créer une interprétation intermédiaire du code, qui s'appelle arbre abstrait. Avec cette forme intermédiaire, le compilateur peut avoir un aperçu de l'ordre des opérations à générer. Par exemple, effectuer la multiplication avant l'addition dans l'expression "4+3*5 ;", malgré le fait que l'addition est lu premièrement.

Lorsque tous les fichiers nécessaires ont été générés, il faut juste compiler la solution entière pour avoir le compilateur de MinTax pour la cible visée. Si on parle d'une plateforme embarquée, on utilise un compilateur croisé. Pour l'architecture AVR celui-ci est avr-gcc et pour MSP430 c'est IAR Workbench.

Le résumé de ceci peut être trouvé dans la Figure 2.4.

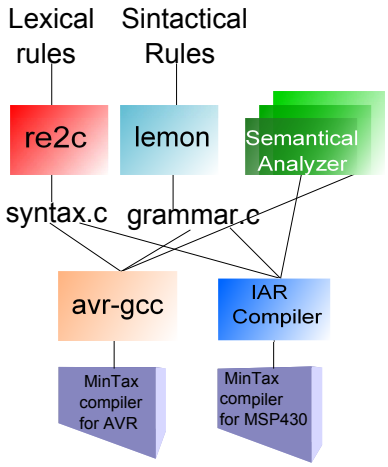


**Figure 2.4:** La compilation du compilateur MinTax.

## 2.4 La fonction malloc du Compilateur MinTax

La figure 2.5 montre la façon dont la fonction “malloc()” classique du C alloue la mémoire. On voit bien qu’il y a des informations de contrôle distribuées parmi la mémoire allouée. Ces informations retiennent la dimension du bloc de mémoire alloué, ainsi que le prochain bloc libre. Ceci est un fort obstacle pour notre compilateur, car l’itération à travers de la liste des atomes se fait en sachant la dimension d’un élément et le début de la liste, à travers d’un pointeur, faisant des opérations arithmétiques sur celui-ci. Or, on ne peut pas être sûrs qu’en incrémentant un pointeur on trouve bien un élément du type du pointeur ou une information de contrôle.

Aussi, dans la figure, il y a des trous dans la mémoire, selon l’allocation et la désallocation. C’est-à-dire qu’elle n’est pas occupée d’une manière optimale. Ceci est visible dans la ligne 1 (colonnes e, f, g) et ligne 3 (colonnes b-e). On appelle ça la fragmentation de la mémoire. À cause de cela, si on veut allouer 15 emplacements libres, malloc va retourner une erreur, alors qu’il reste suffisamment de place. La fonction malloc classique ne peut pas allouer sur une certaine limite sans écrire un bloc de contrôle après.

Ces deux contraintes nous ont motivé à réimplémenter la fonction malloc. Elle a été beaucoup simplifiée et l'information de contrôle a été enlevée. On n'en a pas besoin puisqu'on sait que la mémoire alloué dynamiquement ne sera pas desalloué d'une manière aléatoire, mais à l'ordre inverse du processus d'allocation. Ainsi, cette fonction malloc() alloue d'une façon contiguë. Ceci assure qu'il n'y a pas de fragmentation de la mémoire.

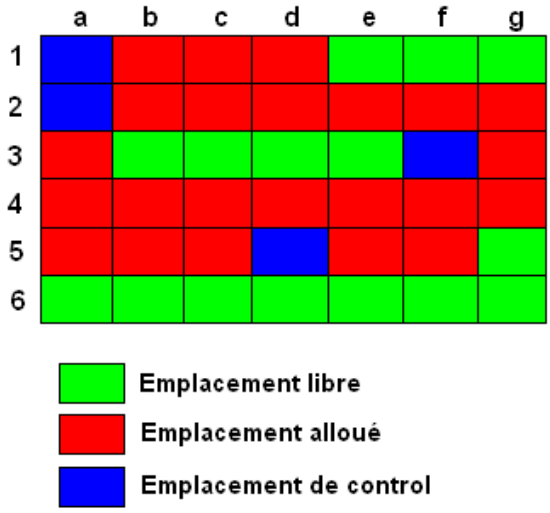


**FIGURE 2.5:** Carte de mémoire pour la fonction malloc classique.

Le compilateur remplit, pendant le processus de compilation, certaines structures de données qui offrent des informations sur les données d'entrée. Ceci est donc une espèce d'information sur l'information, qui s'appelle "métadata" ou "métadonnée". La plupart de ces métadonnées sont sous forme de tableau de pointeurs vers tel ou tel type d'instruction, selon le fichier d'entrée.

Puisqu'il n'y a plus l'information de contrôle de la fonction malloc() (et comme déjà dit on tient seulement un pointeur vers chaque instruction haut-niveau en MinTax), et comme on a besoin de faire d'arithmétique de pointeurs, il faut que les métadonées sur une plage d'adresses soient homogènes. Pour être plus clair, lorsqu'on fait une opération sur instructionFor[5], on a besoin qu'il n'y ait pas des autres types d'instructions allouées entre instructionFor[0] et

instructionFor[5], sinon une lecture mauvaise sera faite. C'est pourquoi toutes les métadonnées, pour chaque instruction (ainsi que quelques tableaux de métadonnées sur les métadonnées) seront allouées avant que la compilation commence.

Intrinsèquement, ceci veut dire qu'un nombre fixe de métadonnées pour chaque instruction haut-niveau (chaque "switch", "while", instructions arithmétiques, etc.) est alloué et que le fichier d'entrée ne peut pas dépasser les limitations de ce nombre d'instructions. Ceci est l'inconvénient associé à ce genre de méthodologie, qui est en outre dictée par la nouvelle implémentation de malloc.

Une structure séparée, implémentée sous forme de pile, doit être allouée pour l'analyseur sémantique, qui contient des informations sur l'état actuel, la règle-mère et qui a une profondeur (configurable) de 100 éléments. Ces aspects seront développés dans la section "L'Analyseur Sémantique".

Une macro de compilation permet d'imprimer le nombre d'octets alloués, pour être sûr que la mémoire allouée ne dépasse pas la limite de mémoire physique disponible.

## 2.5 Spécificités matérielles

Parce que l'architecture MSP430 est une architecture 16-bits, les instructions générées par le compilateur (soit le compilateur IAR Workbench) qui accèdent à la RAM ont un régime particulier. Par exemple, si on a besoin d'extraire un champ dans une structure, et que ce champ a la longueur de 8bits, alors que la structure a une longueur impaire, IAR va arrêter l'exécution de la session de debug. L'erreur signalée est "Memory access on odd address"". L'accès à la mémoire est 2 octets, donc 16 bits à la fois, même pour les champs d'un octet (pour les structures). Le chemin de données du processeur est bien 16 bits, mais l'accès au niveau d'octet est supporté.

On ne peut pas expliquer pourquoi, dans les cas de structures, l'accès à un octet dans une structure avec nombre impair d'octets n'est pas possible, et le compilateur IAR veut toujours accéder à 2 octets. Donc, pour éviter cet obstacle, les structures de métadonnées ayant un nombre impair d'octets sont alignés à une forme paire, par l'ajout d'un octet de remplissage.

## 2.6 L'allocation de mémoire

Dans la mémoire flash, la situation est à peu près comme dans la Figure 2.6. La position du compilateur et du système d'exploitation n'est pas fixe, mais toujours au début de la mémoire. L'application suit. Il reste une région d'espace libre avant le bootloader, mais celui-ci peut être absent. Le bootloader est responsable de démarrage du système, et l'exectutions de programmes spéciaux avant l'exécution de l'application sur le noeud.

Lorsque un nouveau fichier d'entrée pour le compilateur est reçu, il va être mis dans la région après le bootloader, donc dans la mémoire flash, même si il ne sera utilisé que pour produire du code objet. On a décidé de faire cela au lieu d'utiliser la mémoire RAM pour en laisser le maxium au processus de la compilation. Cette région de flash pourra être réutilisée dès que le compilateur aura fini son travail.

On remarque qu'il n'y a pas d'adresses indiquées sur la figure. C'est grâce aux macro-instructions du préprocesseur qu'on peut configurer les limites des régions.

Une fois l'allocation de mémoire finie, le processus de compilation peut commencer, tout d'abord avec l'analyse lexicale.

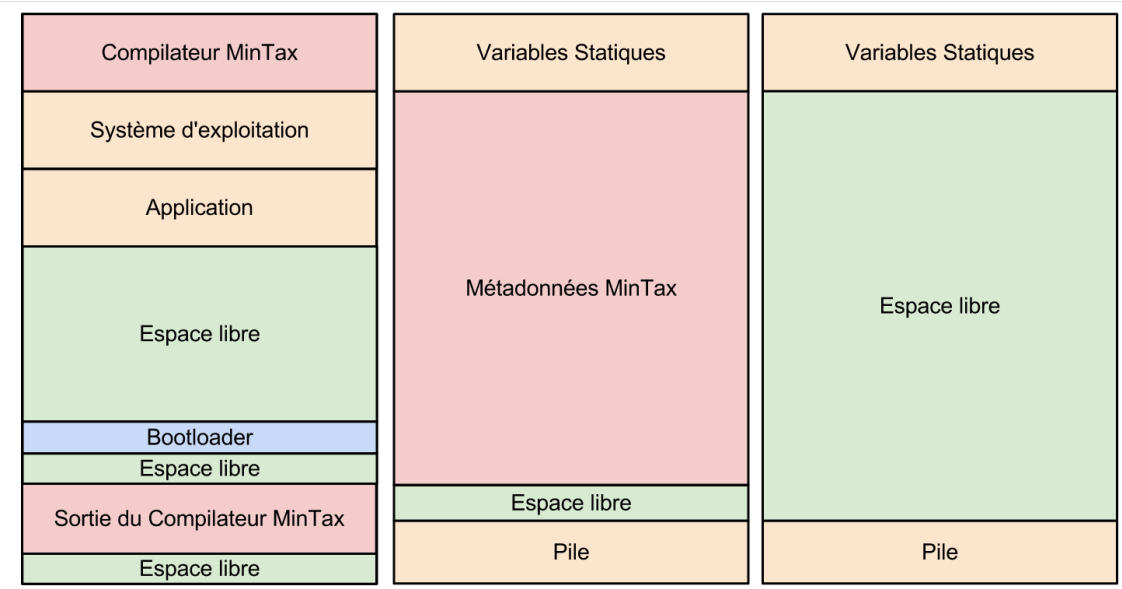


**Figure 2.6:** Carte de mémoire pour le compilateur MinTax. A gauche, la carte pour la mémoire flash. Au centre, la mémoire RAM durant le processus de compilation. A droite, la mémoire RAM après la compilation.

## 2.7 L'Analyseur lexical

La syntaxe de MinTax est écrite dans un fichier d'entrée pour le programme re2c, à côté des autres fonctions qui sont appelées pendant que l'analyse lexicale est en cours d'exécution. Par exemple, la fonction de remplissage de la FIFO d'entrée pour cette analyse. Elle modifie un tableau de pointeurs lors du parcours de celle-ci. Ce tableau tient des pointeurs vers des informations tels que le début du fichier MinTax dans la mémoire tampon de la réception radiofréquence, la position courante dans la FIFO d'analyse, la limite pour un nouveau remplissage ... Aussi, on peut avoir le cas où un atome est construit de plusieurs caractères, selon les règles de syntaxe (par exemple, le nom des fonctions ou de variables). Dans ce cas, si la différence entre le pointeur de fin dans la mémoire tampon de réception et le pointeur vers le caractère courant dans la FIFO est inférieur à la longueur possible d'atome, un nouveau remplissage est automatiquement demandé. Ce cas est présenté dans la Figure 2.7.

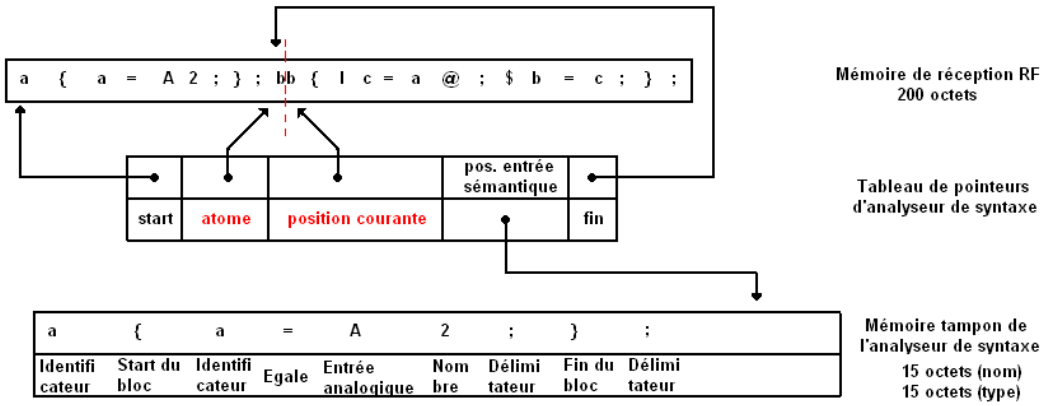


**FIGURE 2.7:** L'analyse de syntaxe

Dans les cas normaux, où le pointeur vers le dernier atome et le pointeur vers la position courante coïncident, après 9 octets lus (taille configurable), l'analyseur de syntaxe est démarré avec les données lues dans sa FIFO tampon. Dans le cas décrit auparavant, une nouvelle lecture d'un octet de la mémoire radiofréquence vers la FIFO est faite. Ceci continue jusqu'à la fin de l'atome. Pour pouvoir gérer cela, la FIFO a une longueur (configurable) de 15 octets.

Le résultat de l'analyseur lexical c'est un tableau en RAM (avec le nom d'atome, ainsi que sa classe et le numéro de la ligne). Il y a plusieurs classes d'atomes :

- nombres
- opérateurs (comme '+' ou '<=')
- délimiteurs (comme ' ;' ou ',')
- ponctuation (le caractère '.')
- identificateurs (les noms des variables, fonctions)
- mots clé (comme '{' ou '}')

Dans un deuxième temps, l'analyseur de syntaxe modifie automatiquement la portée des données selon la fonction dans laquelle il se trouve. Les entrées dans la table de symbole de la prochaine étape y seront écrites avec cette valeur. Ceci est important car on peut avoir le même

nom de variable locale dans plusieurs fonctions différentes, et on a besoin de savoir dans un tel contexte, de quelle variable on parle lors d’une opération qui l’implique.

Pour des raisons de consommation de la flash, on a décidé de ne pas écrire l’analyseur mais de le générer. Il existe des logiciels pour cela, appelés “générateurs d’analyseurs lexicales”. En effet, le code généré est en C, mais il utilise beaucoup des expressions "goto", qui se traduisent en moins d’instructions d’assemblage que l’appel des fonctions, par exemple. Ceci donne la plus petite taille possible sans demander de compromis. Le générateur d’analyseur lexical s’appelle re2c (anglais "regular expression to C"). En effet, il utilise un langage d’expressions régulières comme règles d’entrée. Ceci est un très fort avantage car des ajouts à la syntaxe seront faciles et rapides.

MinTax a une couche de configuration selon laquelle la syntaxe peut être configurée facilement, si besoin est.

Normalement, s’il n’y a pas des erreurs lexicales, détectées par l’analyseur lexical, c’est l’analyseur sémantique qui sera mis en exécution.

Un cas spécial est la notation hexadécimale. Ces nombres sont précédés d’un ’x’. Donc, les variables, fonctions ou constantes qui contiennent “x” dans leurs noms ne sont pas autorisées. Ceci est une des premières sources d’erreurs possibles lors du développement du code MinTax.

De plus, puisque tous les mots clés en MinTax (sauf “$”, le symbole d’un port d’entée / sortie digitale) sont écrit en lettre majuscule, et qu’il n’y a pas plus que 26 lettres dans l’alphabet, on a dû faire quelques subterfuges. Par exemple, pour offrir le support pour les nombres entiers 16-bits avec signe, en syntaxe C “int”, on a donc associé le mot clé “I” en MinTax. Mais aussi, il fallait le support pour de clauses conditionnelles “if” du C. Puisque le mot clé “I” était déjà pris, on l’a assigné à “E” - soit “evaluate expression”.

On a décidé que pour minimiser la consommation de la RAM, toutes ces informations (sauf le nom du symbole, le champ d’application et les lignes où il est utilisé) vont utiliser 1 octet chacun. Les exceptions nommées ont une longueur de 2 octets. Bien sûr, il y a des contraintes sur la taille du fichier d’entrée, à cause de tout ce qui a déjà été abordé :

– le nombre maximal de fonctions est de 7
– le nombre maximal d'instructions dans les fonctions est de 25
– le nom d'un symbole est 2 octets (plus encore un '\0' interne)
– le nombre de paramètres effectifs pour une fonction est de 4)

Ce tableau de limitations est configurable selon l'architecture matérielle et, surtout, le montant de RAM disponible. Ces valeurs sont recommandées pour 16kOctets de RAM.

Le résultat du tableau des symboles est mis en RAM. Ces informations seront utilisées ensuite par l'alloueur de variables et le générateur de code.

Pour chaque atome identifié, il y a l'analyseur sémantique qui l'interprète. Il appelle des fonctions locales pour construire l'arbre abstrait afférent au fichier d'entrée.

# 2.8 L'Analyseur Sémantique

Pour cette étape, on a utilisé Lemon (50). Lemon est un générateur d'interpréteurs qui fait usage de clauses pour générer une grammaire (un fichier C). On l'a donc utilisé pour générer la grammaire pour notre langage MinTax. Les clauses font partie de l'entrée pour Lemon, qui prend cette grammaire hors-contexte et génère les routines pour interpréter les constructions en MinTax. Les clauses (ou bien, règles), donnent la forme que le fichier MinTax doit suivre (par exemple, quoi attendre après une variable, ou le nom d'une fonction).

Lemon génère un interpréteur qui est récursif et qui fait donc usage d'une pile pour se souvenir où il est avec l'interprétation dans une règle. Cette pile est configurable, mais pour notre compilateur, sa profondeur est fixée à 100 éléments.

Ces éléments s'agissent de terminales et non-terminales qui composent des règles.

Voici un exemple pour démontrer celà :

*declaration_variable : :=UCHAR list_variables SEMICOLON.*

*list_variables : :=IDENTIFICATEUR varlist_reste.*

*list_variables : :=IDENTIFICATEUR NOMBRE varlist_reste.*

*varlist_reste : := COMMA IDENTIFICATEUR varlist_reste.*

*varlist_reste : := COMMA IDENTIFICATEUR NOMBRE varlist_reste.*

*varlist_reste : :=.*

Ici, la règle attend un atome (terminal) de type UCHAR, ensuite la liste de variables et un point virgule (SEMICOLON). La liste de variables est une règle en soi-même, donc c'est un non-terminal. Sa définition est aussi composée par un non-terminal, varlist_reste, qui a une définition récursive. Comme toute définition récursive ; une condition de stop est requise. Cette condition est soit un terminal soit l'atome vide, comme vu dans la toute dernière règle. La profondeur de cette pile a des implications sur la profondeur de ces définitions, qu'elles soient récursives ou non.

Dans l'exemple précédent de grammaire, on peut avoir des initialisations pour les variables. Ceci est possible en ajoutant un nombre juste après le nom de la variable.Cette partie peut aussi manquer, dans le cas où la variable n'est pas initialisée. Pour savoir quelle clause il évalue, l'interpréteur tient une variable d'état et chaque règle a son état associé.

Le problème est que le résultat de Lemon était initialement prévu pour être exécuté sur un ordinateur, non pas sur les noeuds de WSNs. En effet, les différents états sont normalement compris dans des variables tableaux qui sont mis dans la mémoire RAM. Ceci devient un gros problème avec des grammaires qui font usage de règles complexes, comme avec MinTax. La RAM est déjà trop petite pour tenir des tableaux des variables, et en plus elle est requise pour la compilation. Comme que ces tableaux sont statiques et qu'ils changent peu, on a décidé de les mettre dans la mémoire flash. Ceci est possible en ajoutant le mot-clé "PROGMEM" après les noms de variables, à l'endroit où ils sont déclarés, dans le fichier généré "sin.c". Ceci est fait automatiquement par un script *sed* qui s'exécute avant la génération de l'analyseur sémantique.

Puisque que ce fichier est généré, il faut un genre de post-traitement, qui fait la tâche requise. Ceci est donc géré par un script pour "sed". Il fait aussi la substitution du type des variables. Il va remplacer "int" par "uint16_t", par exemple. Ceci est nécessaire parce que les données n'ont pas la même longueur sur le PC (avec un CPU à 32 ou 64 bits) et sur l'architecture du noeud. Ce processus est présenté dans la Figure 2.8.

Pour chaque règle de grammaire, il y a deux actions possibles : "shift" (déplacer) ou "reduce" (réduire). Pour l'action de shift, une partie d'une règle est décomposée de nouveau dans des sous-règles par l'analyseur. Dans l'exemple de grammaire précédent, une action de shift est exécutée lorsque l'analyseur atteint le non-terminal "list_variables" dans la règle "declaration_variable". Ensuite, l'analyseur se concentre sur la définition de la sous-règle (du non-terminal). Ce processus continue d'une manière descendante, récursive. Lorsque une sous-règle qui contient un terminal est atteinte, l'évaluation de la règle mère peut enfin poursuivre. Ceci implique le processus de réduction, car la sous-règle est réduite à la règle mère, plus générale.

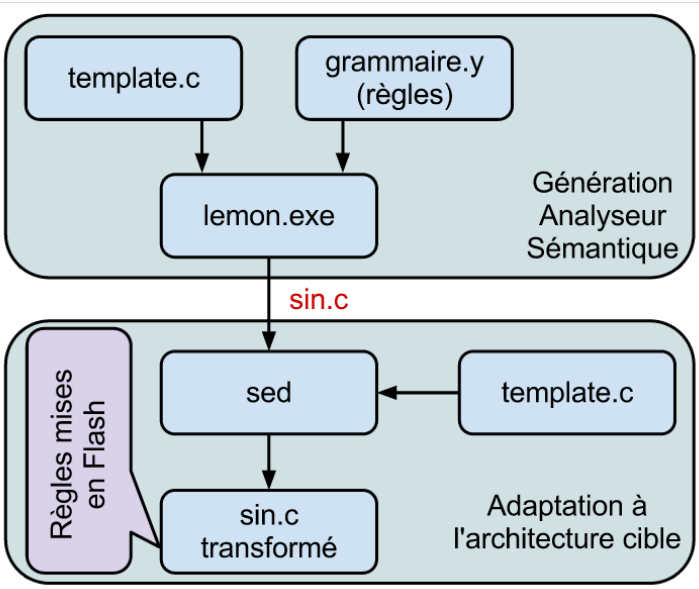


**FIGURE 2.8:** Le processus pour générer la grammaire pour MinTax

Lorsqu'une règle est réduite, le fil d'exécution du programme appelle des fonctions internes du compilateur MinTax, et qui écrivent les structures de métadonnées pertinentes. Pour une clause itérative "for" par exemple, une structure de métadonnée est écrite avec les arguments pour le for (la valeur de départ, de fin, le pas, la variable à être itérée). Certaines informations pourraient manquer. Les synopsis pour les différentes instructions possibles en MinTax donnent plus d'informations là-dessus. Elles sont présentes dans les sous-sections qui traitent les instructions supportées par MinTax.

Ensuite, une fois la structure pour l'instruction en cause écrite, une référence à l'entrée de ce tableau est à son tour écrite dans un tableau qui comprend toutes les instructions du fichier d'entrée courant.

La Figure 2.9 explique l'enchaînement du processus de compilation.

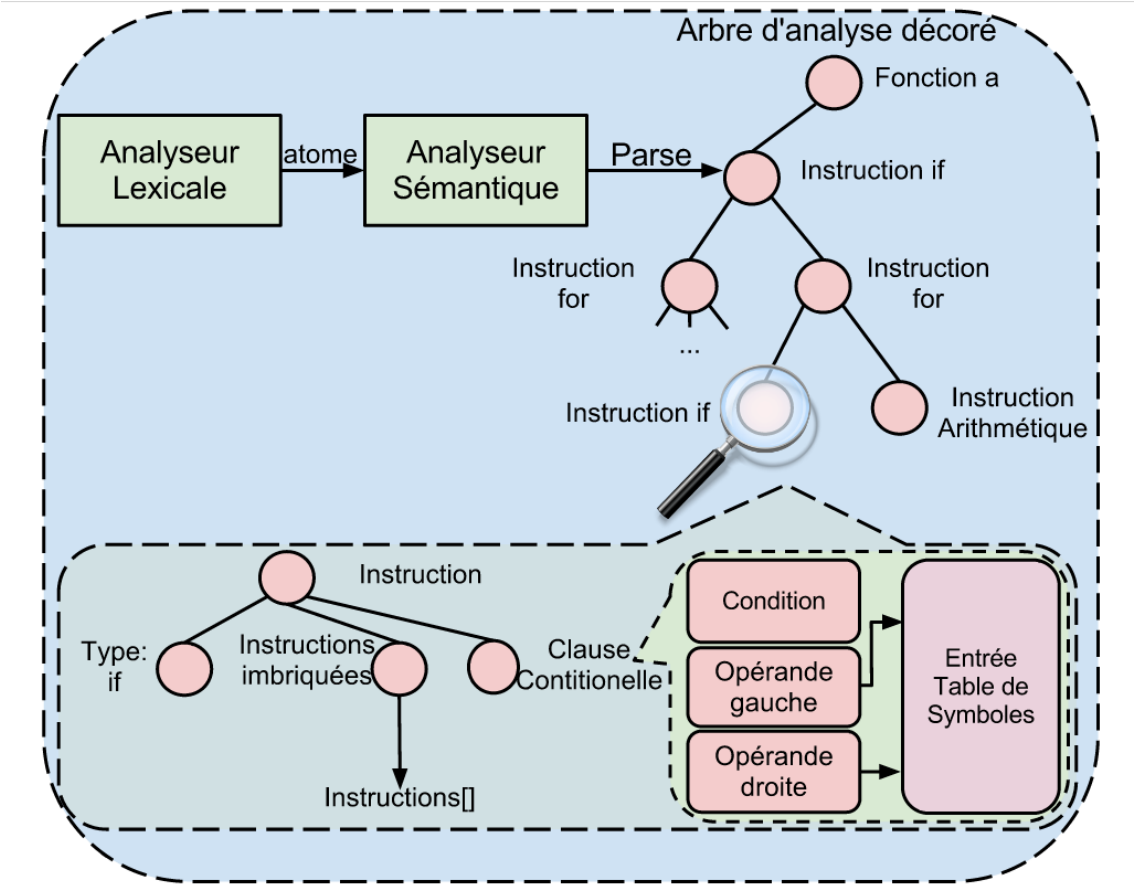


**FIGURE 2.9:** Décoration de l'arbre abstrait.

### 2.8.1 Table de Symboles

La table de symboles est la partie responsable de faire la différence entre les différentes variables fonctions et constants. Selon le contexte, on peut identifier s'il s'agit d'un appel de fonction, ou une soustraction des variables. Un exemple est présenté dans la Figure 2.10. Pour les variables, le type est aussi retenu. Chaque référence à cette variable sera ajouté au tableau dans le champ "Line/End". Ensuite, des informations comme la valeur d'initialisation pour les variables ou le nom des parents(pour des éléments complexes comme les structures) sera rempli. Le champ "Add. Info" donne des informations supplémentaires sur la nature du symbole. Le champ d'application pour le symbole est stocké en "Scope". On peut parler des variables globales et variables locales aux fonctions. Différentes fonctions peuvent avoir le même nom de variable locale. Aussi, pour les membres d'une structure, le nom du parent sera écrit ici.

```
Name: dd    Type: UCHAR     Parent/Init: !    Add. Info: None                    Scope: 0, Line/End: 1
Name: dd    Type: None      Parent/Init: !    Add. Info: Function Implementation Scope: dd, Line/End: 2 , 3
Name: ee    Type: None      Parent/Init: !    Add. Info: Function Implementation Scope: ee, Line/End: 5
Name: x     Type: UCHAR     Parent/Init: !    Add. Info: Formal Parameter        Scope: ee, Line/End: 5 , 6, 7
Name: y     Type: UCHAR     Parent/Init: !    Add. Info: Formal Parameter        Scope: ee, Line/End: 5
Name: ff    Type: None      Parent/Init: !    Add. Info: Function Implementation Scope: ff, Line/End: 8
```

**Figure 2.10:** Exemple de sortie du tableau des symboles

## 2.9 L'Alloueur de variables

S'il s'agit d'incrémentation d'une variable, par exemple, on a besoin d'au moins 3 instructions :

- une pour prendre les données et les mettre dans un registre
- une pour faire l'incrément
- une pour écrire la nouvelle valeur en retour

Les opérations sur les registres du processeur n'auraient pas besoin d'accès à la mémoire et demanderaient moins d'instructions pour être exécutées. Au contraire, si la variable est allouée dans la mémoire, elle doit être lu et réécrite après une opération qui modifie sa valeur. On a pris la décision de lier les variables des registres et par la suite, réduire la taille des instructions générées. L'allouer de variables est celui qui va faire cette correspondance entre les variables et les registres (ou bien, si besoin est, la mémoire heap). Il commence à allouer à partir du premier registre un nombre configurable de variables globales. S'il reste de la place, les variables locales aux fonctions suivent. Pour chaque fonction, le début sera toujours alloué au même registre, c'est-à-dire que parmi différentes fonctions, leurs variables locales seront dans les mêmes registres. Dans le cas que deux fonctions avec les mêmes registres s'appelent, il y a un prologue d'instructions PUSH et un epilogue d'instructions POP qui assurent que l'informations ne s'ecrasent pas.

S'il ne reste pas de registre disponible, l'allouer va mettre la variable dans la RAM, plus précisément dans la heap. Ceci est illustré dans la Figure 2.11. Aussi, s'il s'agit d'une variable partagée parmi plusieurs fonctionnalités, qui s'exécutent sur plusieurs fils d'exécution, le changement de contexte complique les choses. Avec une région de mémoire partagée, ce problème est résolu.

En MinTax on peut avoir un nombre qui suit immédiatement après la déclaration d'un nom de variable. Ceci est en fait la valeur d'initialisation de la variable. Toutes les variables sont initialisées à zéro si elles ne sont pas déclarés comme étant initialisées à une autre valeur. Cette étape est démarrée une fois finie l'allocation de variables.

Il existe un type par défaut pour les variables, qui est configurable à la compilation du compilateur de MinTax. Ceci est implicitement uint16_t (entier sans signe, 16-bits). Si le compilateur est en train de compiler un programme en MinTax et il trouve une variable qui n'est pas dans la liste de symboles, il l'ajoute tout de suite, afin de produire l'arbre abstrait utilisé pour la génération du code machine. Ceci est un mécanisme similaire à celui de Perl, et est en effet une caractéristique empruntée à ce langage de programmation.

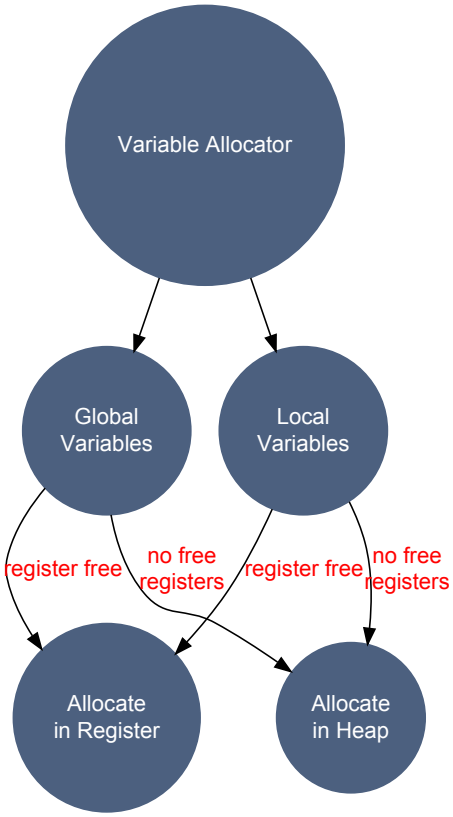


**FIGURE 2.11:** L'allouer de variables.

L'allouer de variables démarre avant le générateur de code et il construit une table variables-registres dans la RAM. Il existe plusieurs types des variables, alors on aura des variables liées aux plusieurs registres. Le type caractère occupe 1 octet, le type entier et pointeur 2 octets et 4 octets seront alloués pour le type fractionnaire. Les variables de plus d'un octet sont allouées en format "little-endian".

## 2.10 Le Générateur de code

Le processus de génération de code dépose son résultat dans une mémoire tampon afin qu'il soit écrit en flash. Chaque fois que cette mémoire est pleine, une nouvelle écriture en flash sera demandée. Une autre sera effectuée lorsque le compilateur aura fini, pour être sûr qu'elle est vide et que toutes les instructions sont bien écrites dans la flash.

La mémoire tampon de sortie a une longueur fixe de 256 Octets. Cette taille est configurable. Si on prend l'exemple présenté dans la Figure 2.13, on peut voir que pour chaque clause "if" il y a un saut vers la prochaine instruction à exécuter si la condition est fausse. Ce saut est une instruction de JMP (anglais : jump), qui est faite vers une adresse qui n'est pas connue a priori. C'est pourquoi cette adresse est considérée 0 et une fois la prochaine instruction connue (et sur ce, l'adresse de saut), cette instruction est modifié dans le buffer de sortie (mémoire de sortie) avec les nouvelles données. Le problème surgit lorsque la mémoire de sortie est pleine et donc une écriture de la flash avec le code généré est demandée, avant que la modification ait l'opportunité d'être modifiée. Dans ce cas, si le saut n'a pas été résolu, l'instruction de JMP serait écrite avec l'adresse 0. Ceci veut dire qu'au lieu d'un saut, un reset "soft" est réalisé.

Plusieurs scénarios ont été évalués pour essayer de résoudre ce problème. Premier scénario est de ne rien faire. Au pire, selon la taille de la mémoire tampon, le saut doit se contenter d'être réalisé dans 256 octets, souvent moins (parce qu'une instruction de JMP peut apparaître au

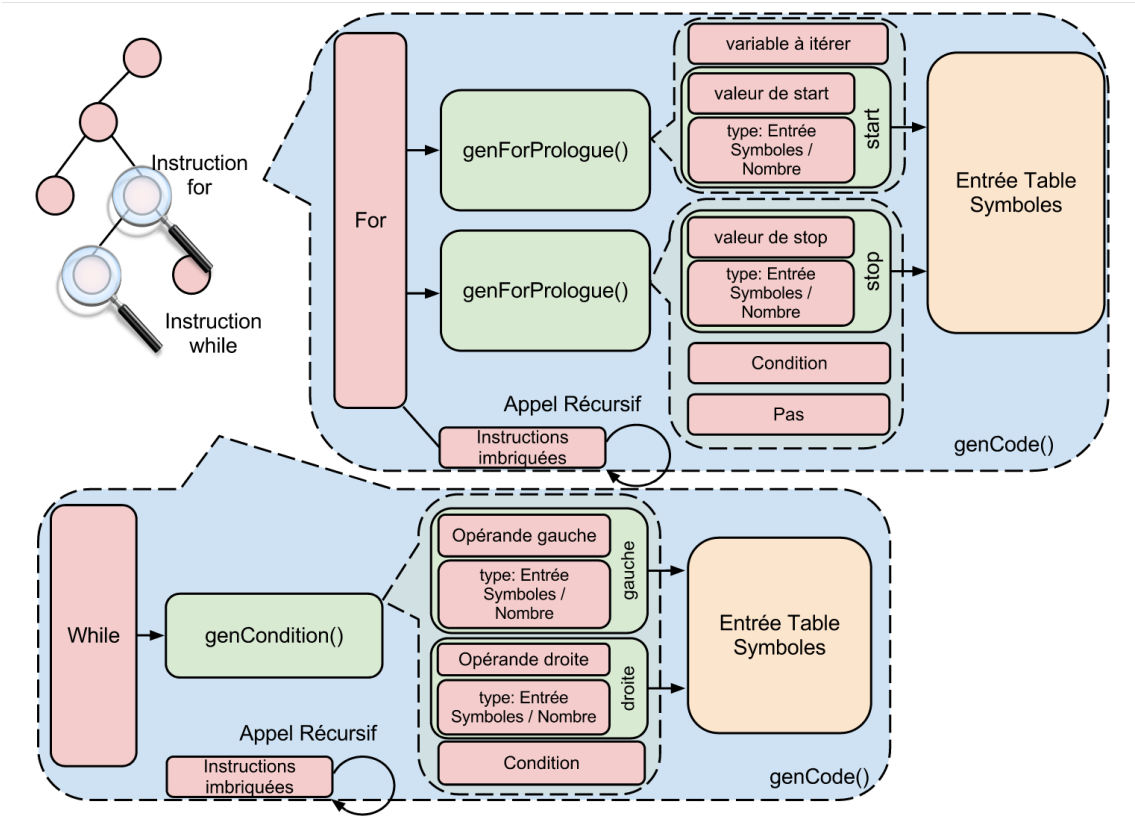


**FIGURE 2.12:** Le procès de génération de code

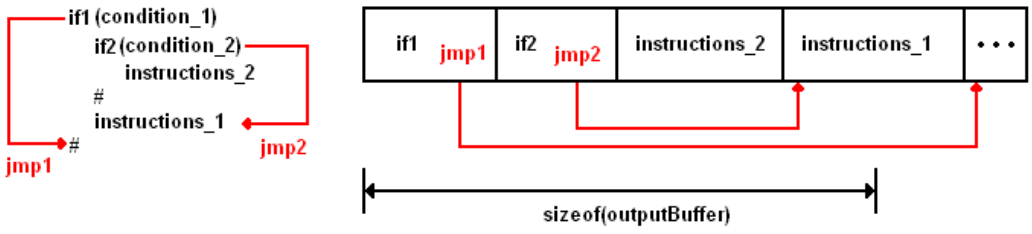


**FIGURE 2.13:** Example de débordement de la mémoire tampon de sortie.

milieu de la mémoire tampon). Pour donner un ordre de grandeur, pour les AVRs, où la plupart d'instructions ont une longueur de 2 Octets, cela veut dire moins de 128 instructions.

Le deuxième scénario est de prendre une mémoire tampon de quelques kilooctets pour comprendre toutes les instructions de saut. Le problème ici est premièrement qu'il n'y a pas assez de RAM à y dédier. Un fait pareil voudrait dire qu'il y aura moins de RAM pour la compilation et donc plus de contraintes sur la complexité du code d'entrée. Deuxièmement, il reste toujours le problème d'adresses vides qui peuvent exister vers la fin de la mémoire de tampon.

Le troisième scénario est celui qui a été implémenté. Il s'agit d'utiliser une mémoire tampon d'une longueur de 512 octets. Normalement, s'il n'y a pas d'instruction de saut non résolue dans celle-ci, lors du remplissage des premiers 256 octets, son contenu est écrit dans la flash. S'il y a des JMPs non résolus, elle n'est pas encore écrite, mais elle continue à être remplie avec des nouvelles instructions jusqu'à ce que toutes les références soient résolues ou la taille maximale atteinte.

Il reste toujours à répondre à une question : "Comment trouver les instructions de JMP dans la mémoire tampon de sortie ?". Il se peut qu'il y ait d'autres instructions qui ressemblent au code machine de cette instruction, par exemple l'argument 16-bits d'une autre instruction sur 4 Octets, telle qu'un "call". C'est pourquoi nous avons choisi de tenir un tableau de pointeurs vers l'adresse de ce saut, comme décrit en Figure 2.14.

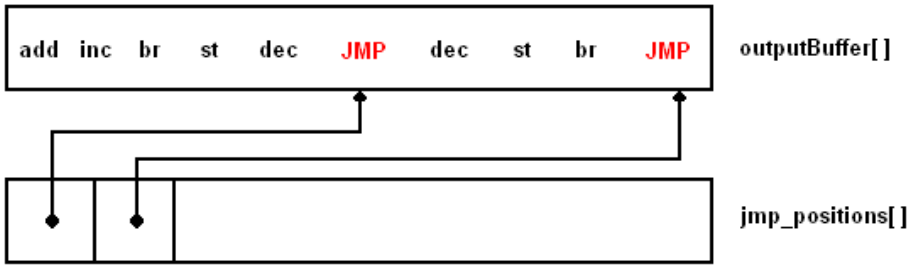


**FIGURE 2.14:** Pointeurs vers les sauts.

## 2.11 Imbrication en MinTax

Les instructions peuvent être imbriquées. Il s'agit d'avoir un bloc conditionnel dans un autre bloc conditionnel, par exemple. Mais les blocs peuvent être des catégories différentes. L'imbrication en MinTax est alors supportée, mais cela consomme de la pile. Selon la taille du programme, des imbrications plus ou moins fortes seront possibles. Cependant, un niveau maximal de 5 est recommandé. Ceci est lié à l'implémentation interne du compilateur. Il existe toujours le risque que la pile s'écrase avec les autres informations plus en bas dans la RAM, dans le cas où ce niveau est dépassé.

## 2.12 Opérations Arithmétiques

On a expérimenté avec plusieurs manières d'implémenter la grammaire pour les opérations arithmétiques. La représentation sous forme polonaise (48) empêcherait une lecture facile du code MinTax par l'utilisateur, malgré sa facilité d'implémentation.

Pour minimiser la syntaxe des opérations arithmétiques, on a enlevé la nécessité d'usage d'égalité ('='). Bien sûr que on peut quand même l'utiliser si on le souhaite. On parle ici des opérations arithmétiques où une variable sera calculée et le résultat mis dans la même variable. Par exemple, "a+5 ;" va incrémenter 'a' de 5. L'usage du caractère point-virgule est obligatoire entre les expressions, pour les délimiter. Pour différentes clauses, il faut avoir un délimiteur pour signaler la fin de la clause (par exemple pour un "if"), qui est par défaut '#', mais le langage permet de le changer en faisant usage du fichier de configuration.

Il y a des priorités pour les expressions arithmétiques. La multiplication et division sont donc évaluées avant l'addition ou la soustraction. Ces opérations peuvent être présentes dans le champ de retour d'une fonction. Par contre, ce n'est pas possible de les mettre dans la liste de

paramètres effectifs d'une fonction, en l'appelant.

S'il s'agit de plusieurs opérations arithmétiques en cascade, elles peuvent être enchaînées une après l'autre dans la même construction, comme "a*5-b+1 ;".

Une opération arithmétique relativement complexe en MinTax peut être "a-2*3+5 ;". La manière dont l'arbre correspondant à cette expression est construit est dans la Figure 2.15. En plus, le résultat (soit l'arbre décoré, après l'interprétation de tous les caractères jusqu'au point-virgule) est décrit dans la Figure 2.16. Dans cela, on a un tableau de fonctions (functions_T[ ]),avec "Fonction A", "Fonction B", etc. Chaque fonction pointe vers ses instructions, dans le tableau des instructions (instr_T[ ]). C'est ici que le type d'instruction est stipulé, ainsi que l'adresse en mémoire de l'instruction elle-même. Les instructions peuvent être de différents types, par exemple "if","while" ou arithmétiques.

De manière implicite, les opérations entre des nombres (par exemple, pour évaluer une condition) sont considérées comme effectués entre des types signés 16-bits.

### 2.12.1 Tableaux en MinTax

Les Tableaux de variables sont supportés par le langage. Lors de la déclaration d'un tableau, un type implicite y est assigné. Ceci est dû au fait qu'on ne voulait pas utiliser plus qu'une lettre pour la déclaration du tableau.

La Figure 2.17 illustre celà.

Le synopsis pour déclarer et faire des opérations sur des tableaux :

**déclaration :** *A<nom>.<longueur> ;*

**usage :** *<nom>.<index dans le tableau> <expression> ;*

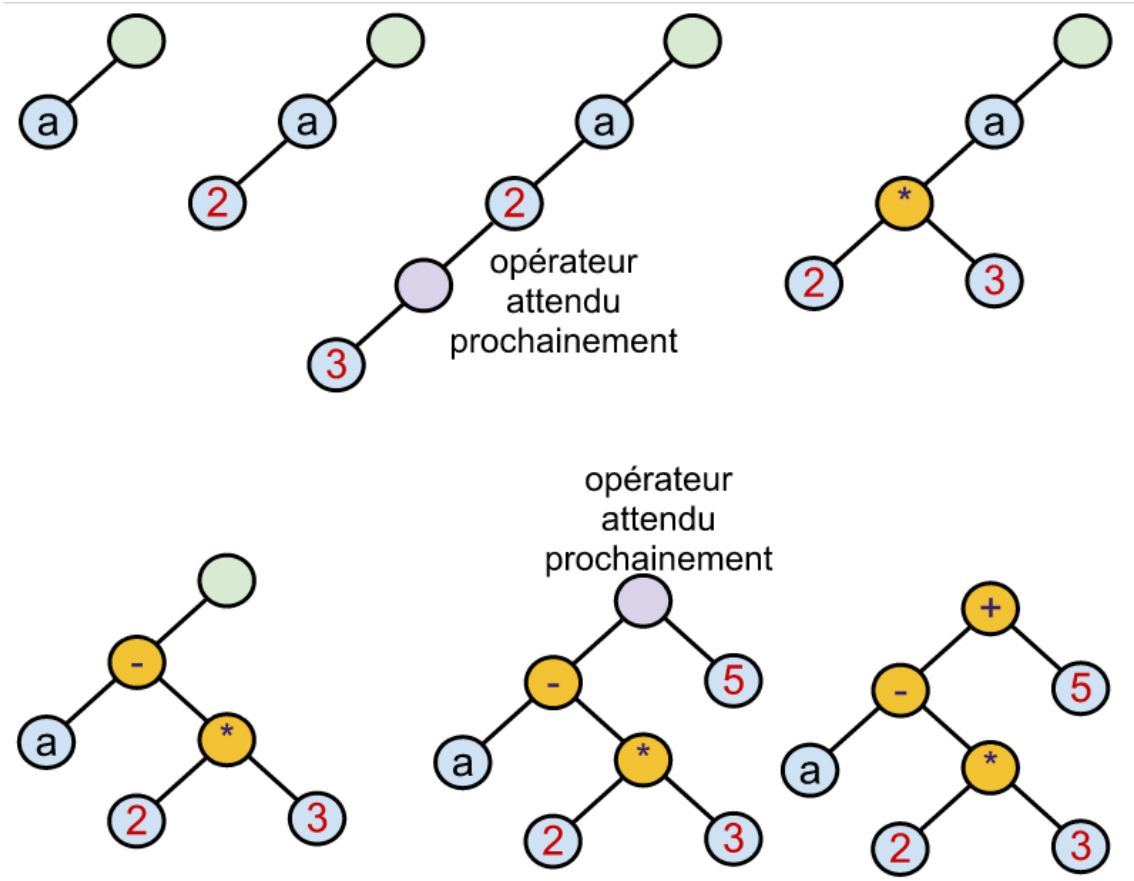


**FIGURE 2.15:** Arbre correspondant à l'expression "a-2*3+5 ;" en MinTax.

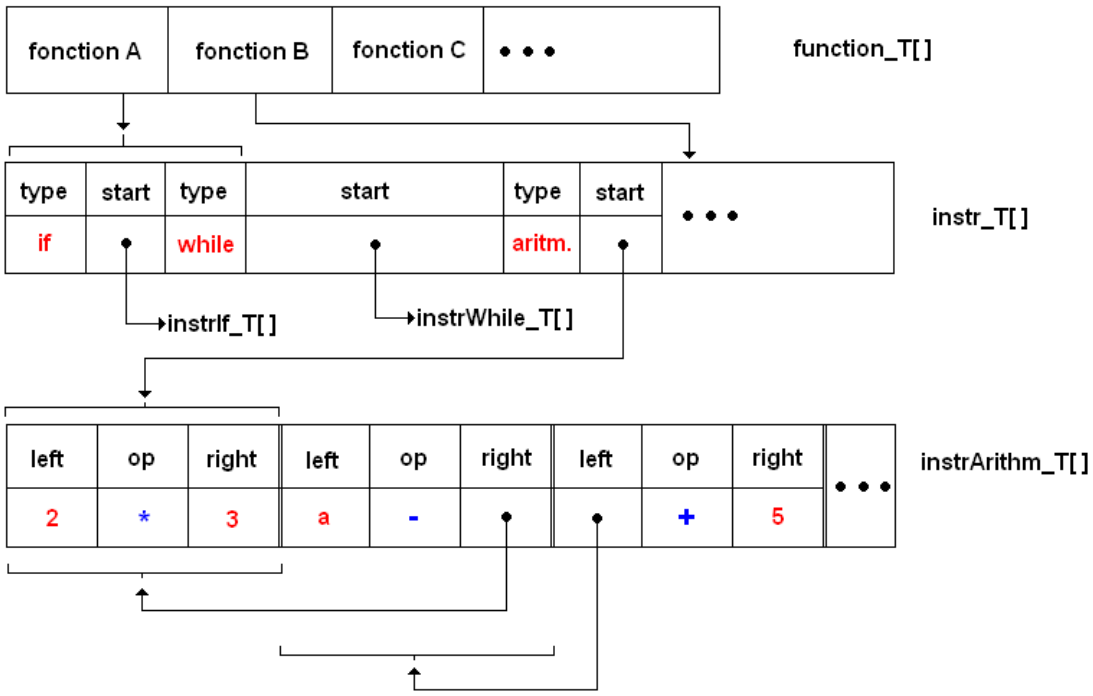


**FIGURE 2.16:** Représentation de l'arbre abstrait décoré pour l'expression "a-2*3+5 ;"

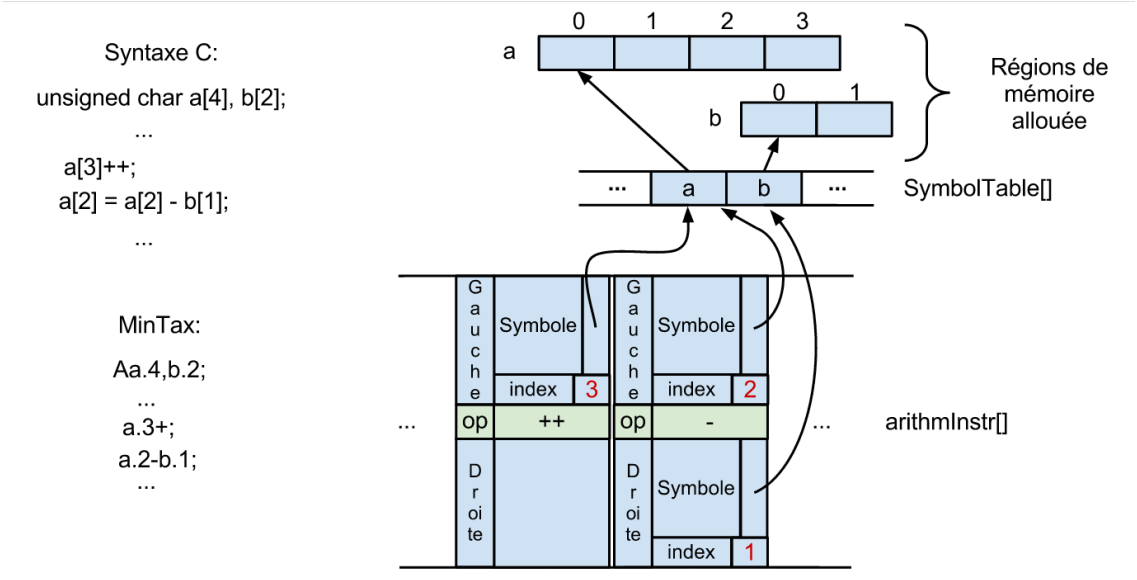


**FIGURE 2.17:** Représentation de l'arbre abstrait décoré avec des opérations sur des tableaux (présentés à la gauche de la figure).

# 2.13 Clauses itératives et conditionnelles

Les clauses itératives sont celles qui vont parcourir un bloc d'instructions plusieurs fois, en ayant (d'habitude) une condition d'arrêt. Ce sont les équivalents de "for" et "while" en C. Les clauses conditionnelles sont les clauses 'if' en C. Ils ont le synopsis suivant :

**for :**

*F<variable><décroissance(-) ou incrément<+> ; <début>,<fin> ; <nombre de pas ou rien> <instructions dedans> <#>*

**while :**

*W<variable><condition><nombre ou 'T' pour toujours vrai> ; <nombre de pas ou rien> <instructions dedans> <#>*

**if :**

*E<variable><condition><numéro> <#>*

**conditions possibles :** *<,>,<=,>=, ?(négation)*

Ces clauses vont se terminer toujours par '#'. Pour les clauses itératives, le pas d'itération peut manquer et un sera une valeur configurable, considérée par défaut. Pour les clauses conditionnelles, le type de variable sera évalué avant de générer le code pour la comparaison.

En ce qui concerne les clauses conditionnelles, la génération du code est présentée dans la Figure 2.18.

On prend comme exemple le morceau de code présenté dans la Table 2.3. L'arbre abstrait décoré est présenté dans la Figure 2.19.

Pour les conditions, il faut connaître l'adresse de la prochaine instruction. Si la condition est vraie, l'instruction qui suit dans la mémoire flash sera exécutée. Sinon, après une condition fausse, l'adresse de la prochaine instruction doit être connue et un saut à celle-ci sera effectué. En tous cas, l'instruction afférente de saut sera générée, et selon la valeur de variable, elle sera effectuée ou pas.

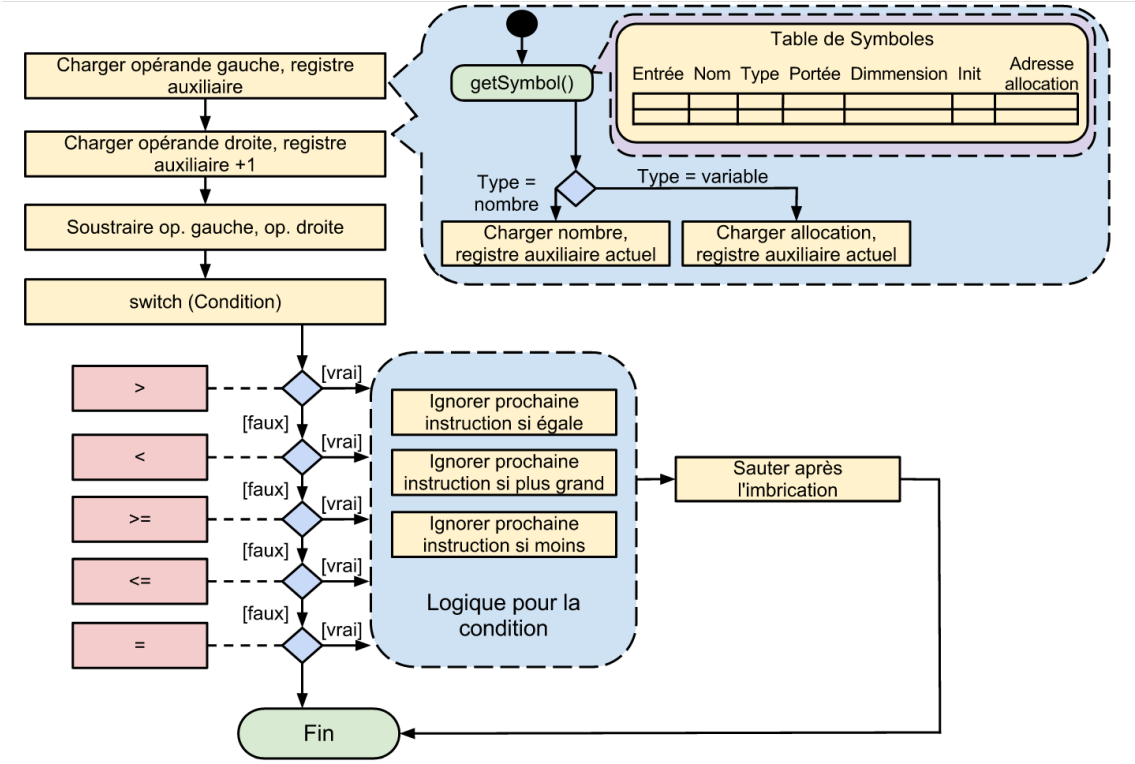


**FIGURE 2.18:** Génération du code pour les clauses conditionnelles en MinTax.

```
Fi
10,1-;
Ez>7;
z+;
b-;
#
a+1;
#
```

**TABLE 2.3:** Morceau de program contenant un For et un If en MinTax

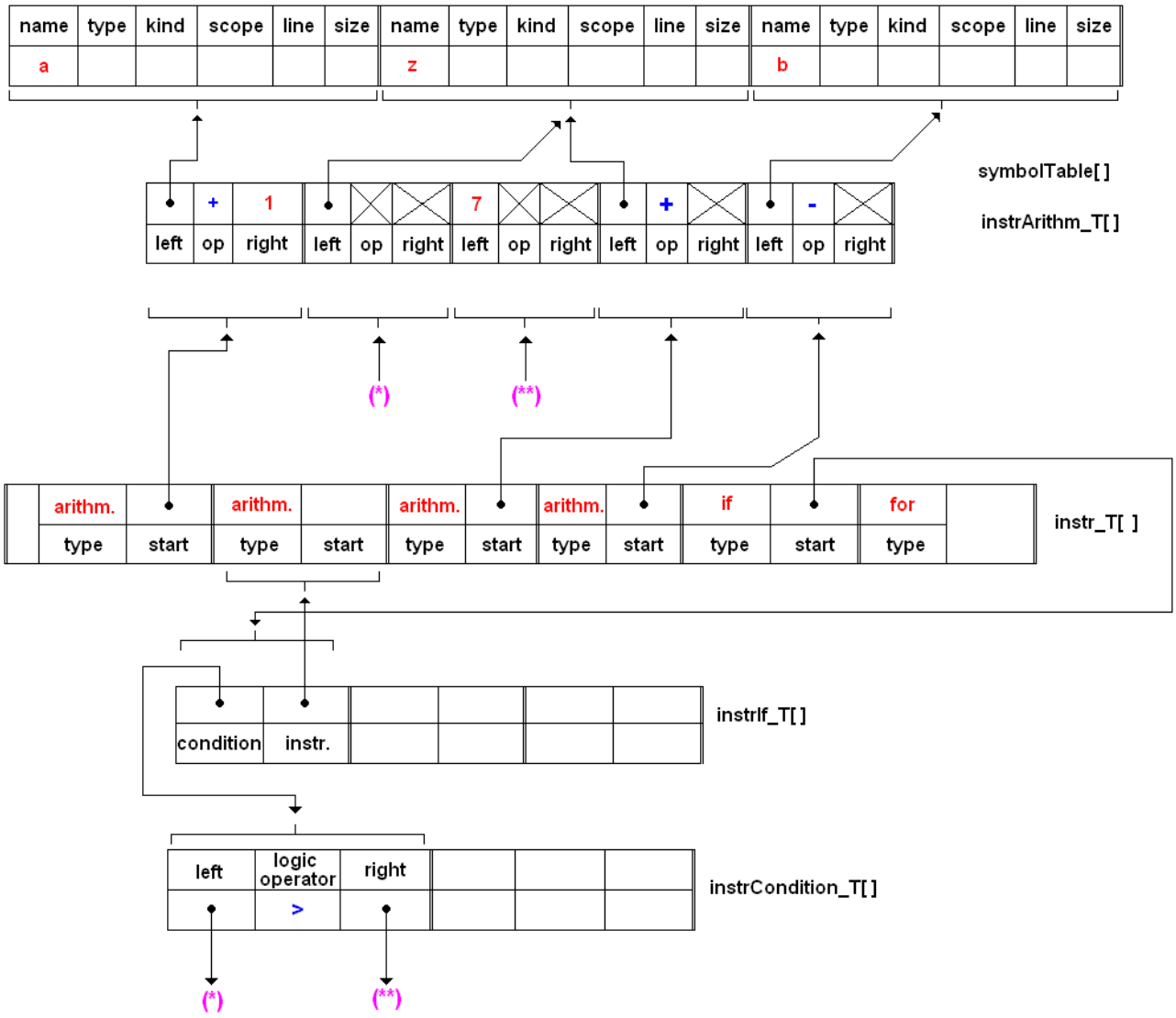


**FIGURE 2.19:** Représentation de l'arbre abstrait décoré pour le programme présenté dans la Table 2.3

Le problème est qu'il y a des instructions qui suivent (dans le If - condition vraie) et que la taille du code généré pour ces instructions n'est pas connu a priori. Même si cette instruction de saut (condition fausse) est générée, l'adresse de saut reste inconnue jusqu'au moment où cette information manquante est résolue. Le jump est donc construit avec l'adresse 0. Une fois les instructions dans le If interprétés, le jump est réévalué et l'adresse exacte est remplie.

Pour la clause itérative For, il n'y a pas forcément besoin de préciser une variable qui doit être itérée. C'est le cas où la variable n'apparaît pas dans le corps de For. L'utilisateur n'a qu'à préciser la clause de For et l'intervalle à itérer (et éventuellement le pas). La boucle sera exécutée plusieurs fois, avec le pas implicite ou celui précisé. Cette extension permet d'économiser 1 octet dans le fichier final de MinTax, octet normalement utilisé pour le nom de la variable à itérer.

## 2.14 Conversion analogique-numérique en MinTax

Les conversions analogiques-numériques sont possibles et supportées par MinTax. Le résultat sera déposé dans une variable. Pour l'instant, seul le support pour le type des données sur 2 octets est implémenté. La résolution du résultat est de 12 bits, et le résultat sera représenté en little-endian. Ceci est une restriction imposée par l'alloueur de variables. L'application peut spécifier la broche d'entrée analogique qui sera lue.

Une configuration par défaut est faite avant la conversion. L'entrée AVCC est pris comme référence, le mode sélectionné est une seule conversion. La division d'horloge est mis à 64.

Voilà le synopsis pour cette clause : *<variable>=A<numéro d'entrée><;>*

## 2.15 Modulation de Largeur d'Impulsion (MLI) en MinTax

Le contrôle des sorties MLI sont possibles avec MinTax. Pour l'architecture AVR, la sortie MLI utilise le timer 1, avec la configuration par défaut (en faisant usage des macro-instructions PWM_DEFAULT_REG_A et PWM_DEFAULT_REG_B).

Les règles suivantes sont à respecter :

- la sortie MLI sera toujours précisée auparavant(pour AVR, pin B2)
- mettre la sortie sur 0 en cas de coïncidence entre la valeur du minuteur et la valeur définie (décrit par PWM_DEFAULT_REG_A)
- la valeur maximale est 0x00FF(décrit par PWM_DEFAULT_REG_A)
- mettre la sortie à jour à la valeur maximale.(décrit par PWM_DEFAULT_REG_A)
- le diviseur d'horloge est 1024(décrit par PWM_DEFAULT_REG_B)

Cela donne le synopsis suivant pour le MLI :

*P<Valeur pour le registre de coïncidence (moins de 256)>,<Configuration pour Registre A ou PWM_DEFAULT_REG_A><rien ou : ,Configuration pour Registre B>< ;>.*

## 2.16 Accès direct aux entrées et sorties

Le contrôle des sorties numériques, ainsi que la lecture des entrées est possible en MinTax. Pour faire une écriture, il faut spécifier le nom de la sortie ainsi que le numéro de la patte. Il y a une abstraction pour le nom, car MinTax est conçu pour des réseaux hétérogènes, où chaque architecture matérielle peut avoir ses propres noms de sorties. Par conséquence, le compilateur utilise une table de correspondance entre le nom (unique entre les architectures) et la sortie physique pour l'architecture en question.

Le langage offre le support pour mettre des opérations arithmétiques dans une clause de sortie numérique. C'est-à-dire que les opérations seront évaluées avant d'écrire la valeur du résultat à la sortie. On peut aussi demander un complément d'un seul bit dans une sortie numérique.

Pour la lecture numérique, on a besoin d'une variable. Celle-ci va toujours être écrite avec une valeur de 8-bits, tant comme la dimension d'une entrée numérique. Le synopsis est le suivant :

**écriture de sortie :** *$ <nom de sortie>=expression (opération arithmétique ou variable ou nombre)< ;>*

**complémentation d'un bit de sortie :** *$ <nom de sortie>%<numéro du pin>< ;>*

**lecture digitale :** *variable=$ <nom d'entée>< ;>*

## 2.17 Sauts conditionnels

Les sauts conditionnels sont les équivalents des constructions "switch-case" en C. Il y aura une variable spécifiée qui sera comparée avec certaines valeurs et en fonction du résultat de la comparaison, le programme va suivre un chemin ou un autre.

Le synopsis pour ces instructions est le suivant :

*B<nom de la variable> ;*

*<valeur1> :<expressions> ;*

*<valeur2> :<expressions> ;*

....

*D<expressions>*

*#*

## 2.18 Fonctions en MinTax

Les fonctions en MinTax commencent toujours par leur nom, suivi de la liste des paramètres effectifs. Il faut préciser que la dimension de ces paramètres ne peut pas dépasser 4 octets. MinTax ne supporte pas des clauses de disparité. C'est-à-dire que chaque expression devra être dans une fonction. Les fonctions se finissent par un mot clé configurable, souvent '}' comme en C. Après celui, la liste des données de retour suit. Si la fonction ne retourne rien, cette région peut être vide. Un caractère délimiteur (implicite ' ;') finit l'implémentation de la fonction et la sépare de la prochaine.

MinTax offre aussi le support pour demander le retour explicite, dans l'implémentation de la fonction, avant même qu'elle ne soit pas terminée. Ainsi, plusieurs points de sortie sont possibles.

## 2.19 Plateforme Logicielle et Matérielle

Comme déjà précisé, l'analyseur lexical du compilateur est écrit en re2c (Regular expression to C). Le générateur d'analyseur lexical utilisé est Lemon. Pour la compilation du compilateur MinTax, on a utilisé avr-gcc 4.3.2 (WinAVR 20090313) pour AVR et IAR Workbench pour MSP430.

On a commencé le développement du compilateur pour MinTax sur la plateforme MICA2 de CrossBow. Celle-ci contient un microcontrôleur AVR ATMega128, la mémoire flash disponible est de 128 kilo octets, et il y a 4 kO de RAM.

La prochaine architecture est la famille MSP430 de Texas Instruments, une architecture Von Neumann. Le microcontrôleur choisi est MSP430F5438, avec 256 kilo octets de flash et 16 kilo octets de RAM. On a implémenté notre compilateur pour cette architecture, ainsi que pour la

MSP430f2617. Un aperçu général sur les plateformes matérielles supportées, ainsi que sur les détails de mémoire requis, peut être trouvé dans le Tableau 2.4 et 2.5.

| WSN Node | MCU | Architecture / Radio | Freq. Osc. | Flash | RAM |
|---|---|---|---|---|---|
| Mica2 | ATMega128L | 8bits/433Mhz | 7.3728Mhz | 128kB | 4kB |
| AVRRaven | ATMega1284p | 8bits/2.4Ghz | 4Mhz | 128kB | 16kB |
| Zolertia Z1 | MSP430f2617 | 16bits/2.4Ghz | 16Mhz | 92kB | 8kB |
| - | MSP430f5438 | 16bits/- | 16Mhz | 256kB | 16kB |

**TABLE 2.4:** Les noeuds supportés par le compilateur MinTax.

| WSN Node | Flash usage | RAM usage | Radio Stack |
|---|---|---|---|
| Mica2 | 30kB | 175 bytes | Chipcon |
| AVRRaven | 30kB | 175 bytes | Contiki |
| Zolertia Z1 | 27kB | 175 bytes | uRacoli |

**TABLE 2.5:** Le besoins par noeud pour le compilateur MinTax et la couche radiofréquence utilisé.

## 2.20 Hétérogénéité complète Materielle-Logicielle

Comme on peut le voir dans le Tableau 2.4, MinTax offre le support pour une hétérogénéité matérielle au niveau du noeud. Puisque le code en MinTax ne contient rien qui concerne le logiciel qui s'exécute sur le noeud, une solution logicielle est également possible.

Dans les sections suivantes on argumente que MinTax est la première solution de reconfiguration dynamique qui offre un support complet pour les mises à jour, indépendamment de l'architecture matérielle du noeud ou du système d'exploitation qui s'exécute.

### 2.20.1 Le compilateur MinTax pour AVR

Les AVRs sont des microcontrôleurs 8-bits qui font usage d'une architecture Harvard modifiée. Les variables et les instructions du programme sont mises dans des régions différents

de la mémoire physique. Pour la lecture, la machine peut accéder aux entités dans la mémoire flash comme si elles étaient dans la RAM, mais seulement en utilisant des instructions spéciales. Avec ça, l'emplacement de certains chaînes de caractères dans la RAM peut être évité dans la phase d'initialisation, et elles peuvent être lues directement à partir de la Flash quand on le veut.

Pour compiler notre solution pour AVR, on a utilisé WinAVR v20100110. L'architecture AVR fait usage de 32 registres généraux, ce qui en fait une candidate idéale pour notre schéma d'allocation de la mémoire, dû au nombre élévé de ces registres. Une illustration visuelle pour montrer comment notre allouer fonctionne peut être vue dans la Figure 2.20. L'alloueur peut réserver jusqu'à 21 registres pour des variables (R0 à R20). Celles-ci seront allouées dans ce registres. Les variables globales seront allouées premièrement, et ensuite, celles qui sont locales. Les registres R26 et R27 sont utilisés pour calculer les résultats des expressions booléennes. Les registres de R28 à R31 sont utilisés pour passer des arguments aux fonctions, et pour prélever leur valeur de retour.

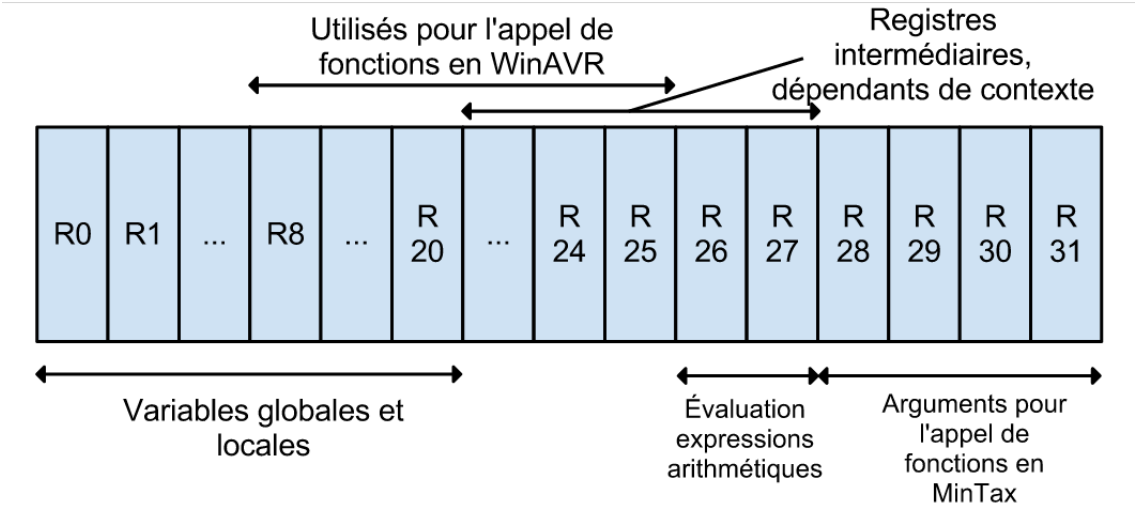


**Figure 2.20:** L'emploi de registres sur AVR

Selon l'opération effectuée, le compilateur peut utiliser des registres de R21 à R27 comme registres auxiliaires, pour calculer des valeurs intermédiaires, ou pour copier des valeurs dans

la partie inférieure du fichier de registres. Ceci est dû à l'architecture AVR, qui permet de charger la valeur d'une constante seulement dans la moitié supérieure du fichier de registres (avec l'instruction LDI (anglais : Load Immediate)).

Pour appeler des fonctions externes qui n'ont pas été compilées avec le compilateur MinTax (mais avec WinAVR), le processus est un peu plus difficile. WinAVR attribue chaque paramètre effectifs aux registres de R25 à R8, avec le premier paramètre commençant à R25. On a construit une couche d'interface (anglais "wrapper") pour connecter le compilateur avec la convention d'appel de fonctions et de retour de valeurs de WinAVR. Celle-ci est restreinte aux spécifications du langage MinTax, qui supporte au plus 4 octets comme longueur de paramètres d'appel ou de retour.

Les limites de notre compilateur pour AVR sont récapitulées dans le Tableau 2.6. Ces valeurs sont configurables par l'utilisateur, mais il y a des estimations qui doivent être faites d'abord, pour s'assurer que la pile n'est pas écrasée.

Un tableau de correspondance est utilisé pour lier les noms des ports (en MinTax) aux adresses physiques. Pour une sortie MLI, s'il n'y a pas de port spécifié, le compilateur va utiliser un port par défaut. En outre, s'il n'y a pas un mode choisi pour ce genre de sortie, elle sera "Phase Correct", à 8-bits. Cela va mettre la patte de sortie à 0 lorsque la valeur du compteur atteindra une valeur prédéfinie lors du comptage de l'état haut, et le mettre à 1 le temps de compter l'état bas.

En ce qui concerne les conversions analogique-numérique (ADC), l'utilisateur peut spécifier la patte qui sera échantillonnée. Une fois de plus, le compilateur utilise le tableau de correspondance pour trouver l'adresse du port ADC pour l'architecture cible. Pour l'AVR, celle-ci correspond au PORTF.

Dans le cas d'opérations arithmétiques, on notera que les AVRs sur lesquels notre solution est déployée n'ont pas d'unité de multiplication. Par la suite, les opérations de multiplication, ainsi que les divisions sont gérées logiciellement. Le code machine généré a une largeur de

2 octets pour chaque mot (avec quelques exceptions de 4 octets). Il faut qu'il soit formaté en format "little-endian" avant qu'il soit écrit dans la mémoire flash.

Dans le mode "Release", il y a une macro de compilateur qui dit où le résultat de la compilation doit être placé. Un autre informe où est l'adresse de début pour allouer la métadonnée nécessaire. Dans le mode "Debug", les données sont sérialisés par l'interface UART à 9600bps, 8N1.

### 2.20.2 Le compilateur MinTax pour MSP430

Le MSP430 est une famille de microcontrôleurs 16-bits qui utilisent une architecture Von Neumann. Celle-ci a un simple espace d'adressage pour les deux mémoires "data" et "program". Le code machine peut être placé et exécuté à partir de n'importe quelle région de ces deux mémoires. On précise qu'exécuter le code à partir de la RAM est bien moins coûteux dans le point de vue énergétique, par rapport à la Flash (140uA/Mhz à 3V, vs 320uA/Mhz à 3V). Parce que les applications finales compilées avec notre solution sont censées être petites et simples, le code généré peut être copié dans la RAM et exécuté à partir de là.

Pour construire notre compilateur pour cette architecture, on a utilisé IAR Workbench v5.10. Le chemin de données de MSP430 a 16 registres. Mais seulement 13 sont disponibles à notre compilateur. Les premiers trois sont réservés au PC ("Program Counter"), le pointeur de pile ("Stack Pointer") et le registre d'état (où les flags (drapaux) résident). La Figure 2.21 donne l'emploi des registres pour cette architecture.

| Name | Maximum value |
|---|---|
| Maximum lexical tokens | 305 |
| Maximum lexical token length | 2 bytes(+1, the \0 character) |
| Maximum initializer length | 4 bytes (+1, the \0 character) |
| Maximum symbols | 50 |
| Maximum references per symbol | 10 |

**Table 2.6:** Limitations pour le compilateur de MinTax

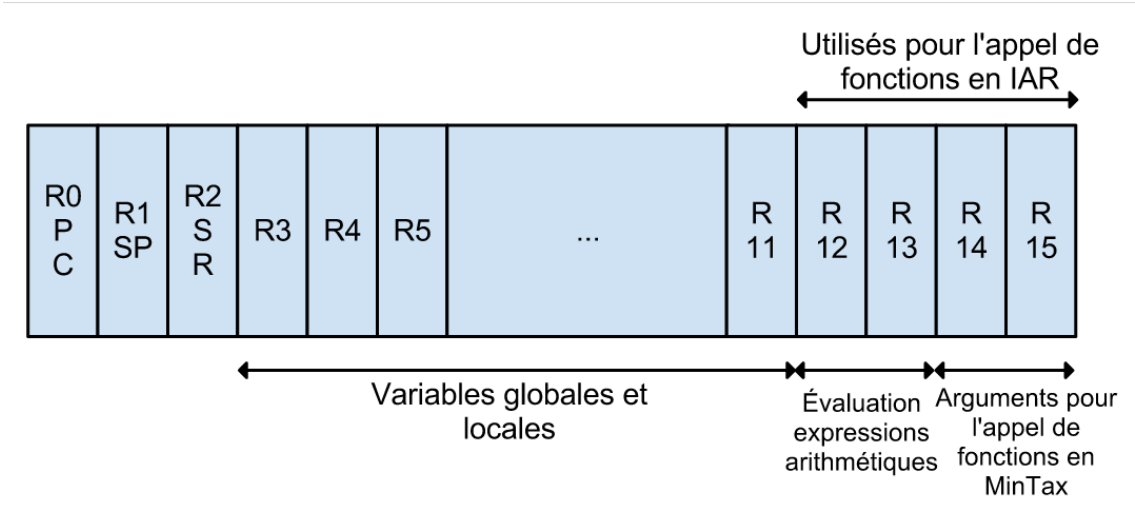


**FIGURE 2.21:** L'emploi des registres sur MSP430

### 2.20.3 Hétérogénéité logicielle : Plusieurs systèmes d'exploitation

Sachant que certaines architectures sont plus puissantes que d'autres, il est logique d'extrapoler le concept au niveau logiciel. On peut donc avoir des systèmes d'exploitation qui offrent plus de caractéristiques, ou qui utilisent moins de mémoire flash/ram.

Puisque dans un mise à jour MinTax il n'y a pas d'autre information que la fonctionnalité à compiler, MinTax peut offrir la possibilité non seulement d'hétérogénéité matérielle mais aussi logicielle. C'est-à-dire plusieurs systèmes d'exploitation dans un réseau. Ceci mène à ce qu'on appelle "hétérogénéité complète HW-SW".

Pour démontrer ce support, on a compilé le compilateur de MinTax avec les systèmes d'exploitation et sur les noued présentés dans le Tableau 2.7.

| Name | Node | Flash (OS) | RAM (OS) |
|---|---|---|---|
| ContikiOS | AvrRaven | 4kB | 236 bytes |
| FunkOS | Z1 | 2kB | 100 bytes |
| FreeRTOS | AvrRaven | 5kB | 236 bytes |

**TABLE 2.7:** Les OS auxquels a été lié le code objet compilé à partir de MinTax

# 2.21 Résultats

Cette partie traite tout d'abord les performances du compilateur MinTax, sur plusieurs algorithmes de test pour ensuite passer à des exemples mesurés avec l'oscilloscope.

## 2.21.1 Quantification de performances du Compilateur MinTax

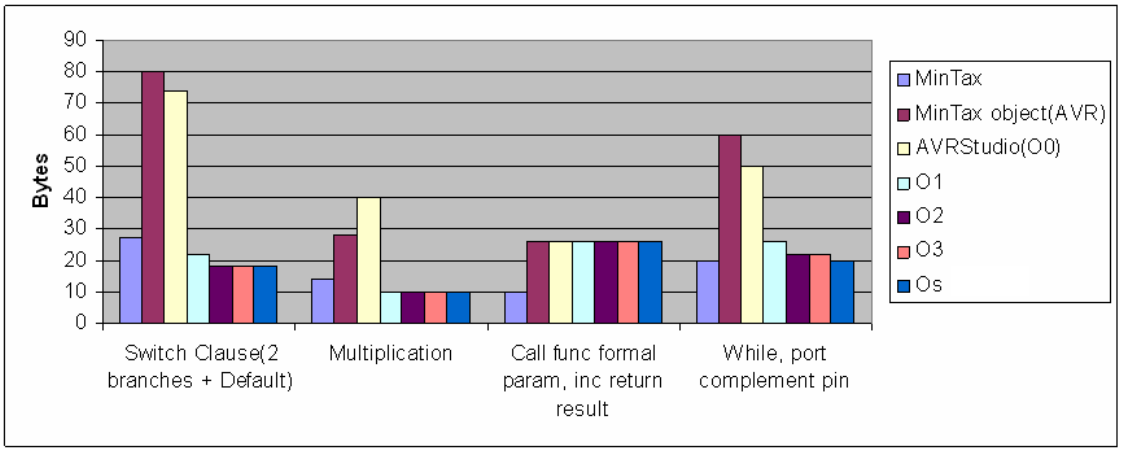


**Figure 2.22:** Comparaison de la sortie du compilateur MinTax avec AVRStudio, sur plusieurs algorithmes et plusieurs optimisations, AVR.

## 2.21.2 Mesures d'oscilloscope

Dans un premier temps, on a envoyé un programme de Blink d'un noeud AvrRaven qui s'exécute sous ContikiOS à un Zolertia Z1 qui s'exécute sous FunkOS, comme présenté dans la Figure 2.30. La trace d'oscilloscope associée est présentée dans la Figure 2.31.

```
a{          //fonction a
WT          //boucle infinie
$a%1;       //complementer Pin A1
#           //fin boucle
};          //fin fonction
```

**Table 2.8:** Program de test : Blink.

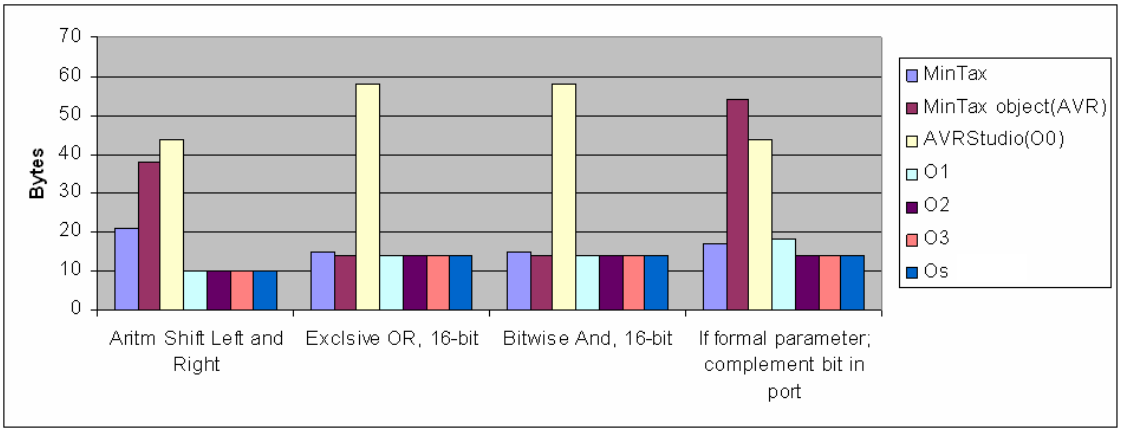


**FIGURE 2.23:** Comparaison de la sortie du compilateur MinTax avec AVRStudio, sur plusieurs algorithmes et plusieurs optimisations, AVR.

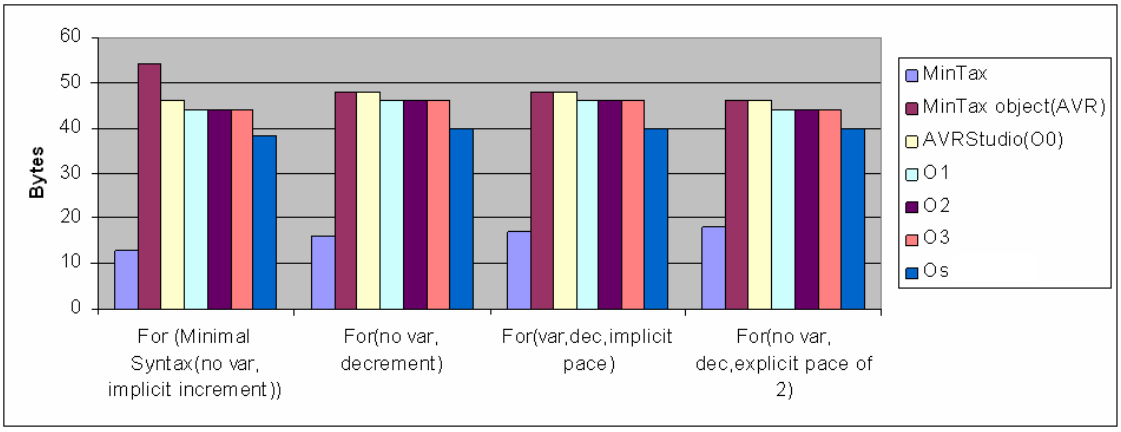


**FIGURE 2.24:** Comparaison de la sortie du compilateur MinTax avec AVRStudio, sur plusieurs algorithmes et plusieurs optimisations, AVR.

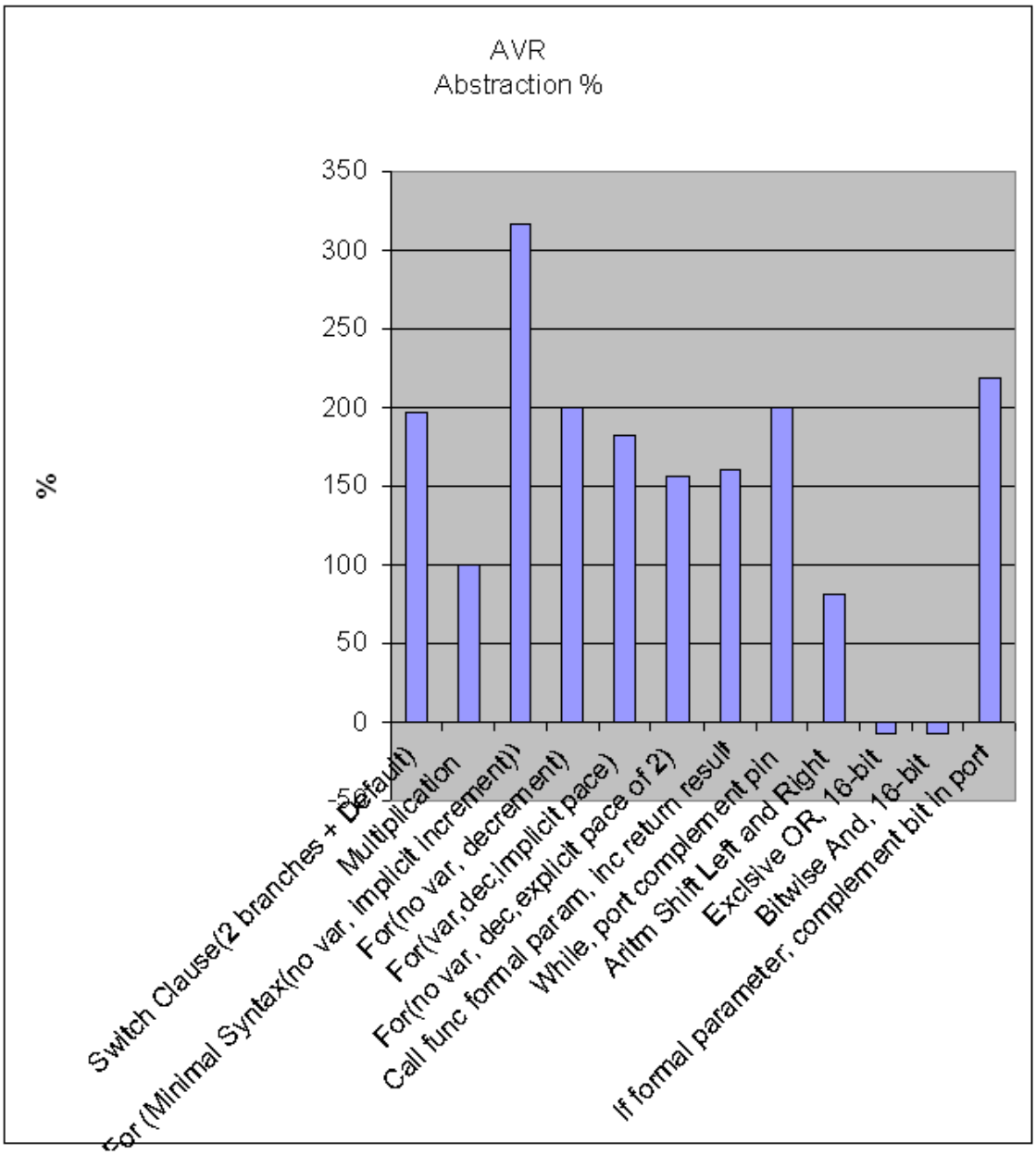


**FIGURE 2.25:** Rapport en pourcentage entre le code généré par le Compilateur MinTax et le fichier d'entrée, AVR.

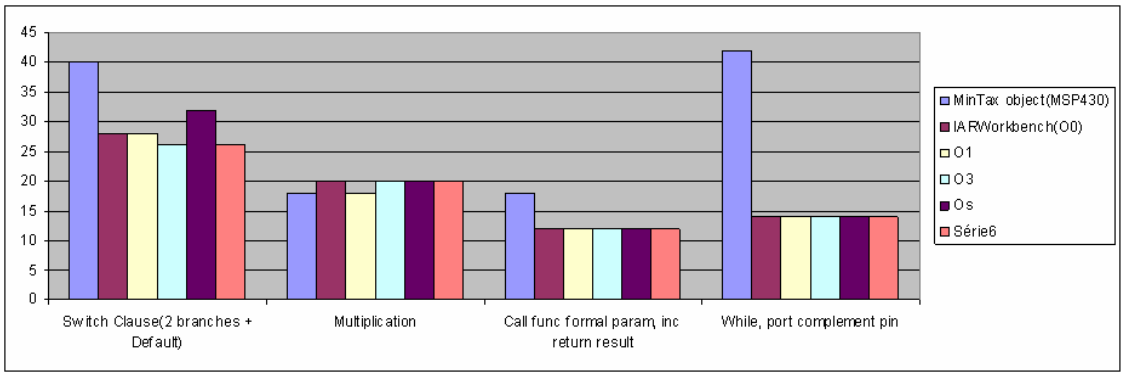


**FIGURE 2.26:** Comparaison de la sortie du compilateur MinTax avec IARWorkbench, sur plusieurs algorithmes et plusieurs optimisations, MSP430.

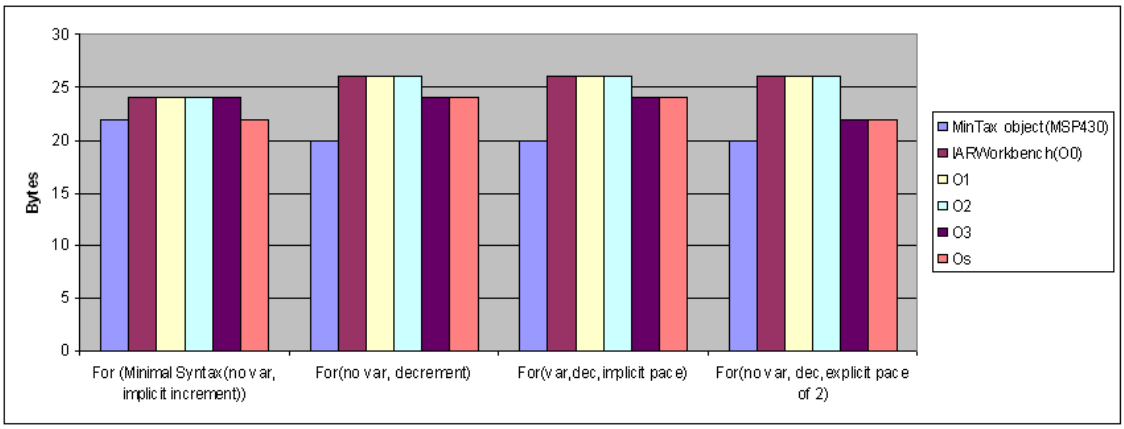


**FIGURE 2.27:** Comparaison de la sortie du compilateur MinTax avec IARWorkbench, sur plusieurs algorithmes et plusieurs optimisations, MSP430.

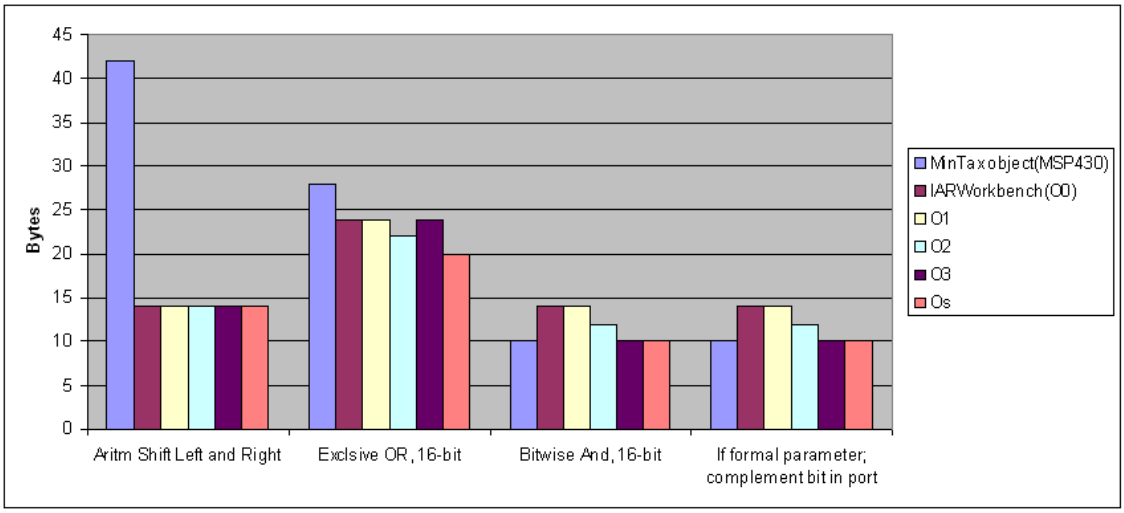


**FIGURE 2.28:** Comparaison de la sortie du compilateur MinTax avec IARWorkbench, sur plusieurs algorithmes et plusieurs optimisations, MSP430.

Dans un deuxième temps, on a construit un programme en MinTax qui lit une valeur analogique d'un pin et l'envoie par radio si la valeur est supérieure à une certaine limite. Ceci est présenté dans la Figure 2.32. Ensuite, le programme entre dans une boucle de 11 itérations afin qu'il recommence tout le processus. Le programme est présenté dans la Table 2.9. Le but est de savoir si c'est plus intéressant de l'envoyer en forme brute, ou bien si ça serait mieux qu'on l'ait envoyé sous forme compressée. Puisque la taille du fichier MinTax n'est pas grande, on a opté pour une compression simple, comme Huffmann.

On a envoyé ce programme sous forme compressée, d'abord à partir d'un noeud Zolertia Z1 qui s'exécute sous FunkOS à un AvrRaven sous FreeRTOS. L'AvrRaven le reçoit, le compile et se reprogramme avec la fonctionnalité. Ensuite, un profil générique de tâche est fait, dans lequel un pointeur vers le nouveau code est créé. Après, le système d'exploitation envoie la fonctionnalité reçue pour simuler une propagation du code dans le réseau. Une fois ce processus terminé, l'exécution de tâches présentes sur le noeud peut commencer. La taille du code généré par la compilation est de 96 octets, et la taille du code MinTax pour le produire s'étale sur 60 octets.

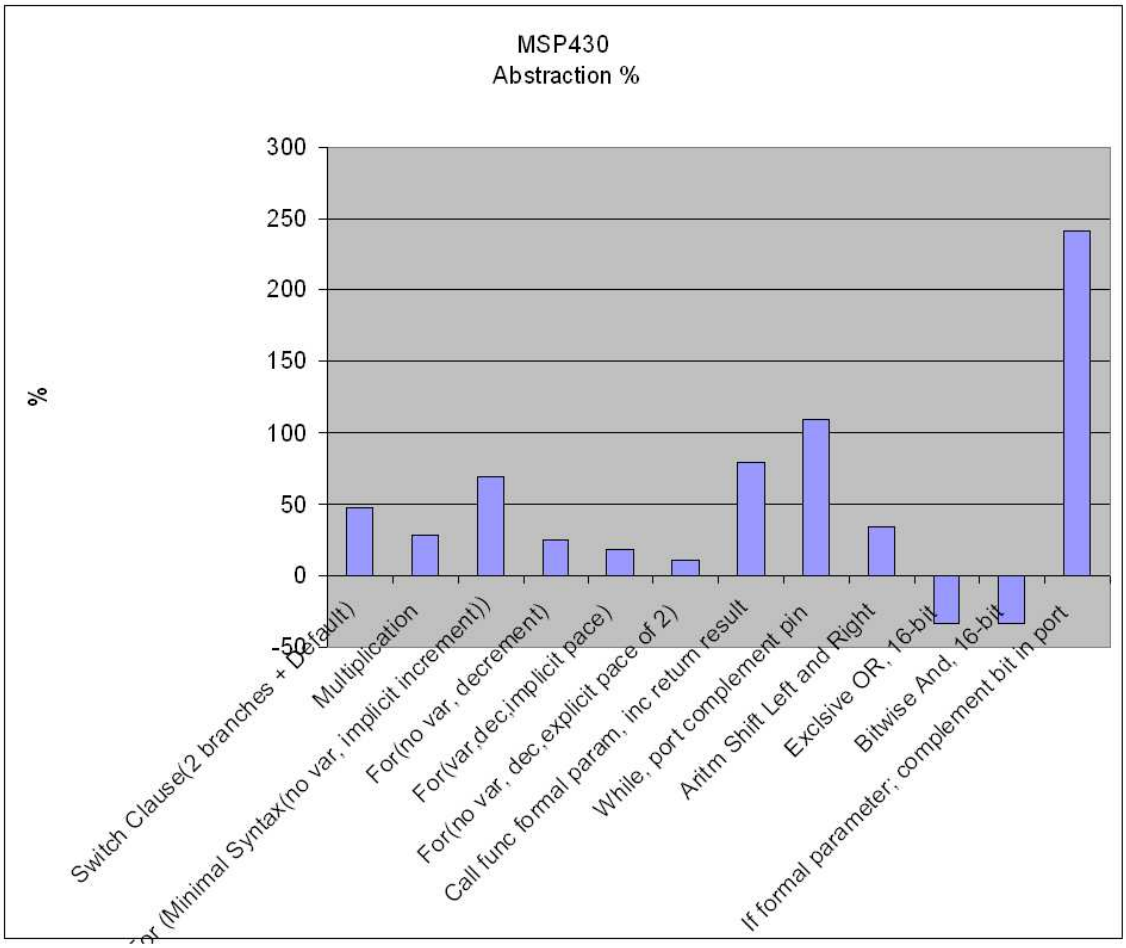


**FIGURE 2.29:** Rapport en pourcentage entre le code généré par le Compilateur MinTax et le fichier d'entrée, MSP430.

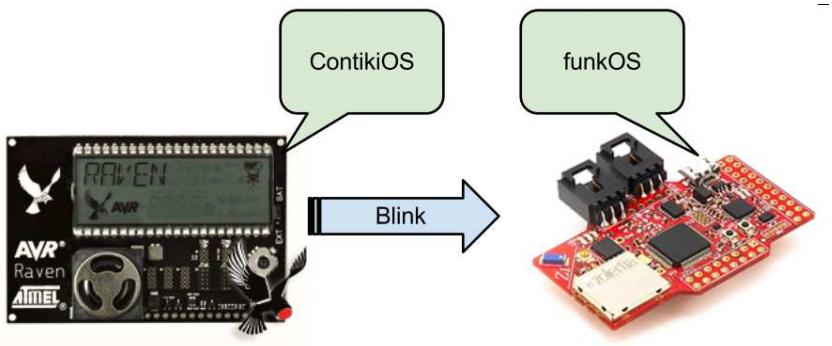


**FIGURE 2.30:** Envoi d'exemple blink d'AvrRaven à Z1.

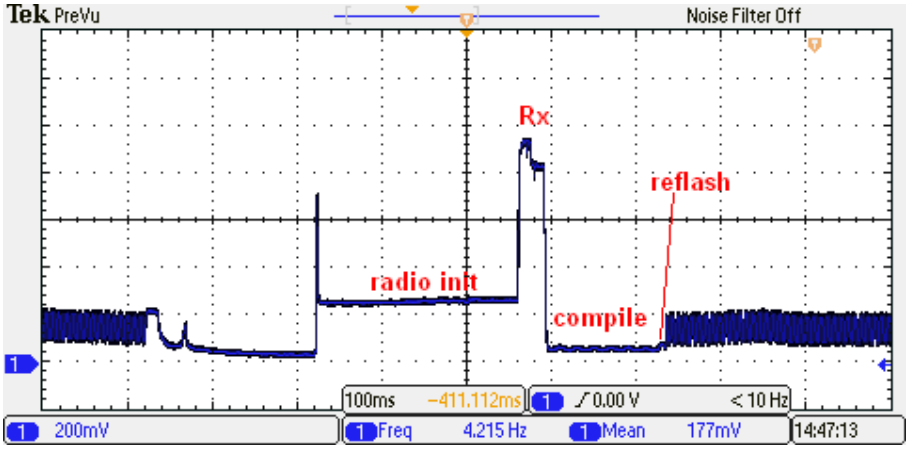


**FIGURE 2.31:** Réception, compilation et Blink sur Z1 (FunkOS).

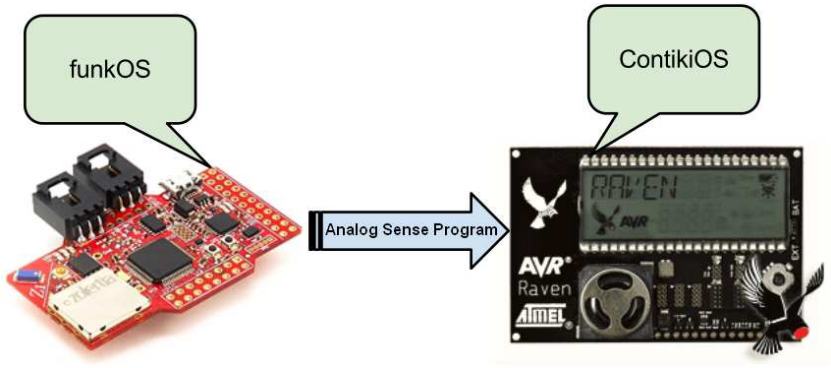


**FIGURE 2.32:** Envoi d'exemple lecture analogique à partir du Z1 au AvrRaven.

```
bCa{   //fonction b, char parameter a
Ea<11;//si a<11
s@a;   //envoi a
#      //fin de  if
};     //fin de fonction

aa{    //fonction aa
Wa<11;//tant que variable a<11
a+;    //incrementer a
#      //fin de while
};     //fin de fonction


a{    //fonction a
WT    //boucle infinie
b=$a;//lecture digitale
b@b; //appeler b, param. a
b+;  //incrementer variable b
$a=b;//ecriture valeurb au port a
aa@; //appeller aa pour delai
#    //fin boucle infinie
};   //fin fonction
```

**TABLE 2.9:** Program de test : aquisition de données, envoi si supérieur à une limite, delai.

#### 2.21.2.1 Forme brute

La Figure 2.33 montre la reconfiguration sous forme brute, avec les prochains états :

1. la réception des données
2. la compilation in-situ
3. le code machine généré est écrit dans la mémoire flash et lié au kernel FunkOS
4. le broadcast du code MinTax aux autres noeuds, pour simuler la reconfiguration d'un réseau. Ce pas est fait à la fin du processus pour vérifier l'intégrité du fichier reçu. Ainsi, il n'y aura pas de data corrompue, et un fichier inutilisable ne sera pas envoyé à travers le réseau. Cette approche offre une robustesse supplémentaire au processus de reconfiguration.

Les différents paramètres avec leur consommation énergétique associée, sont présentés dans le Table 2.14.

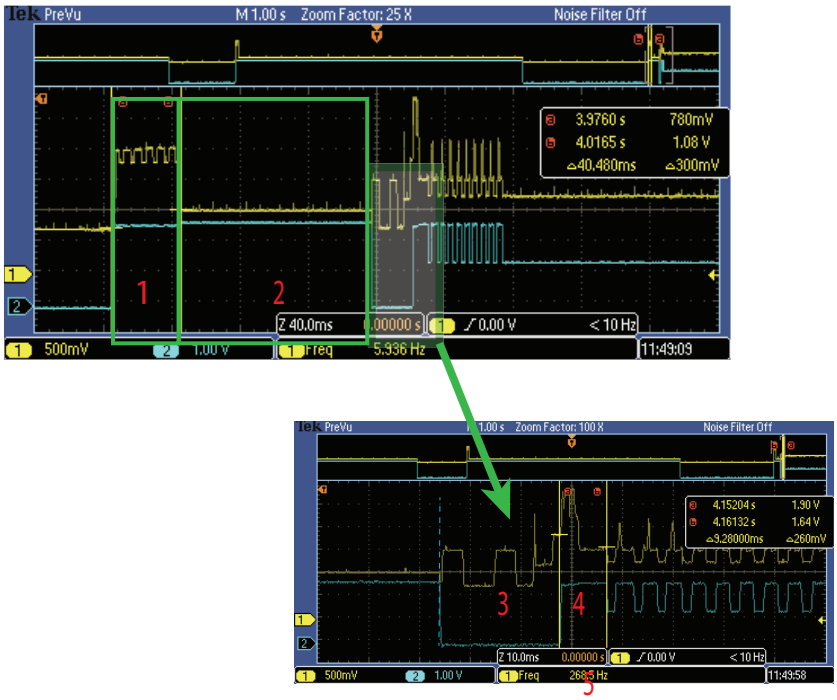


**Figure 2.33:** Les mesures physiques du processus de reconfiguration, forme brute.

| État | Durée (ms) | Puissance (mW) | Énergie ($\mu$J) | Énergie totale($\mu$J) |
|---|---|---|---|---|
| RX | 40.8 | 127,15 | 5187 | 8506 |
| Compilation | 111.28 | 2.1 | 233.68 | |
| Reflash | 23.3 | 80,66 | 1879 | |
| TX | 9.28 | 130,01 | 1207 | |

**TABLE 2.10:** La consommation énergétique détaillée du processus de reprogrammation, forme brute.

La compilation entière a pris 184.66ms, en utilisant consommant moins de 9mJ. La durée de transmission ainsi que le budget est bien inférieur dans la phase de réception. Si on utilise un protocole IEEE 802.15.4 pour transmettre le code MinTax initial de Z1 à AvrRaven, le renvoi est accompli sans aucune couche protocolaire. Cela n'a pas été réalisé ainsi, car on voulait évaluer le coût d'"overhead" du protocole sur notre solution. Comme résultat, l'overhead nécessaire pour une communication IEEE 802.15.4 compte pour jusqu'à 75% de l'énergie nécessaire pour la transmission.

Bien évidemment, ce coût a un fort impact sur le processus entier. D'où, si MinTax permet un gain de 33% sur la taille du code transmis, ceci représente seulement un gain de 5ms sur les 40.8ms de la communication radiofréquence globale. Pour ainsi dire, MinTax devrait être utilisée avec un protocole RF léger, comme celui de Contiki.

De plus, il est important de noter que des exemples plus complexes pourraient mener à des gains plus importants. Sinon, les performances globales de MinTax représentent une amélioration substantielle en comparaison avec la littérature. Par exemple, Deluge (51) a besoin de 58.77mJ sur 50 secondes. Si d'un côté, Deluge offre le support pour la dissémination qu'on n'offre pas, MinTax est utilisable avec n'importe quel algorithme de dissémination ou middleware.

#### 2.21.2.2 Forme compressée

Pour la forme compressée, on a obtenu une taille de 49 octets avec la compression Huffman, donc un gain de 12%. Si on considère la conclusion de l'envoi sous forme brute, c'est évident que la compression n'aurait pas un grand impacte dans un contexte WSN utilisant

le protocole IEEE 802.15.4. Néanmoins, on a effectué les mesures, pour évaluer le coût de la décompression sur le processus entier. Les résultats de l'expérience sont présentés dans le Tableau 2.11.

Les différents états peuvent être vus dans la Figure 2.34, et sont les suivants :

1. la réception des données
2. décompression Huffman
3. la compilation in-situ
4. le code machine généré est écrit dans la mémoire flash et lié au kernel FunkOS
5. le broadcast du code MinTax aux autres noeuds, pour simuler la reconfiguration d'un réseau.

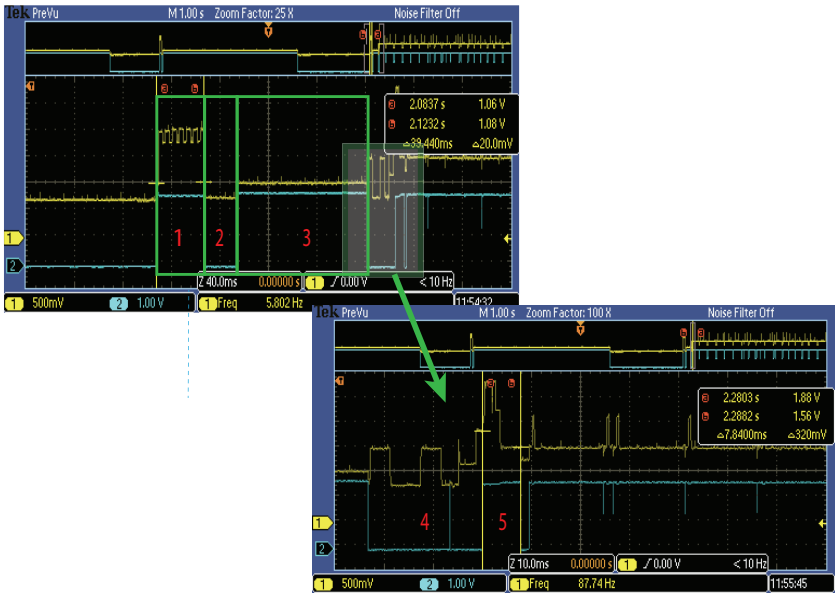


**FIGURE 2.34:** Les mesures physiques du processus de reconfiguration, forme compressée.

Du point de vue global, la reconfiguration coûte presque la même chose qu'à la dernière étape. La décompression représente 7.4% de la consommation globale, mais a un impact plus dur sur la latence, avec 25.84ms. Ceci représente la même durée que l'écriture de la flash, soit 12.6% de la durée totale (204.76ms).

| État | Durée (ms) | Puissance (mW) | Énergie ($\mu$J) | Total ($\mu$J) |
|---|---|---|---|---|
| RX | 39,44 | 126,43 | 5,057 | 18,129 |
| Décompression | 25,84 | 52,46 | 1,356 | |
| Compilation | 108,24 | 82,03 | 8,879 | |
| Reflash | 23,44 | 78,91 | 1,850 | |
| TX | 7,84 | 134,96 | 1,058 | |

**TABLE 2.11:** La consommation énergétique détaillée du processus de reprogrammation, forme compressée.

La perte de performances comparée au cas où il n'y ait pas de compression, est aux alentours de 15%. Néanmoins, le temps gagné pour la réception et renvoi du code peut justifier l'utilisation de la compression.

Dans un réseau dense, si le canal est occupé lors du renvoi (à cause de la durée étendue du transfert), c'est probable que les pertes énergétiques soient plus élevées, car le noeud consomme en attendant la fenêtre du renvoi.

#### 2.21.2.3 ObjectTracker

Dans un troisième temps, on a implémenté l'algorithme "ObjectTracker" qui calcule la position d'un objet lumineux dans un WSN à l'aide des capteurs de lumière. Les noeuds connaissent leur position dans un plan XY et calculent celle de l'objet lumineux selon les instructions présentées dans le Table 2.12. Le code associé en MinTax est présenté dans le Tableau 2.13.

Le processus commence par l'envoi du code en MinTax par un noeud disséminateur. Ceci emploie une transmission "point-to-multipoint" pour envoyer le code MinTax. L'algorithme ObjectTracker contient les positions du noeuds, donc le noeud disséminateur envoie *i* programmes en MinTax, ou *i* est le nombre de noeuds qui participent à l'algorithme. La routine d'envoi fait usage de l'adressage et ignore le programme si, après avoir regardé l'adresse, la MinTax n'y est pas destinée.

```
PROCESS_THREAD(use_regions_process, ev, data)
{
  PROCESS_BEGIN();
  while(1)
  {
    value = pir_sensor.value();
    region_put(reading_key, value);
    region_put(reg_x_key, value * loc_x());
    region_put(reg_y_key, value * loc_y());
    if(value > threshold)
    {
        max = region_max(reading_key);
        if(max == value)
        {
            sum = region_sum(reading_key);
            sum_x = region_sum(reg_x_key);
            sum_y = region_sum(reg_y_key);
            centroid_x = sum_x / sum;
            centroid_y = sum_y / sum;
            send(centroid_x, centroid_y);
        }
    }
    etimer_set(&t, PERIODIC_DELAY);
    PROCESS_WAIT_UNTIL(etimer_expired(&t));
    }
PROCESS_END();
}
```

**TABLE 2.12:** L'algorithme ObjectTracker, implémentation sur Contiki.

```
1. a{
2. GLa.3c.3d.3;
3. i=0;
4. j=1;
5. k=2;
6. WT;
7.      bN;
8.      a.i=b;
9.      c.i=b*j;
10.     d.i=b*k;
11.     Eb>5;
12.     mMa;
13.         Em=b;
14.             eSa;
15.             fSc;
16.             gSd;
17.             h=f/e;
18.             l=g/e;
19.             r@h,l,i;
20.         #
21.     #
22.     d@100;
23.#
24.};\0
```

**TABLE 2.13:** L'algorithme ObjectTracker, implémentation avec MinTax.

Ensuite, les noeuds destinataires compilent les programmes reçus et l'exécution de l'algorithme peut commencer.

Dans le Tableau 2.13, la fonction "a" contient le code pour la fonctionnalité. Il y a trois vecteurs, correspondant à trois noeuds qui sont déclarés comme étant de type global (avec le mot-clé "G") et ayant le type entier, sur deux octets ("L"). Ensuite, on a l'initialisatios concernant les positions des neouds et une boucle infinie ("WT", soit "while - true").

Il y a une lecture d'un port analogique établi par défaut, et la valeur est mise dans la variable "b". Puisque b n'était pas déclarée avant ce point, le compilateur de MinTax saura bien qu'il s'agit d'une nouvelle variable et va automatiquement allouer son emplacement en mémoire. Les prochaines instructions mettent la valeur analogique lue sur la position équivalente à l'index du noeud dans le vecteur afférent. Puisque les vecteurs ont été déclarés globaux, l'exécution du programme se bloque immédiatement après l'écriture de la valeur dans le vecteur. Il y a un protocole de synchronisation qui démarre et qui a comme but d'informer les autres noeuds de la valeur propre, ainsi que de prendre leurs valeurs. Une fois qu'un nombre de valeurs équivalent au nombre de noeuds a été reçu, l'exécution du code généré peut continuer.

Ce qui suit (la ligne 11) est une clause conditionnelle pour évaluer si cette valeur est supérieure à une limite inférieure (dans ce cas, 5). Après quoi, l'élément maximal du vecteur *a* est calculé et mis dans la variable *m*.

La ligne 13 évalue si la lecture analogique propre a donné la valeur maximale. C'est-à-dire une clause conditionnelle qui vérifie si l'intensité de la lumière est la plus grande parmi les noeuds participant à l'algorithme. Si non, les instructions dans la clause conditionnelle ne sont pas exécutées et le contrôle du programme continue à appeler la fonction de délai d'une seconde ("d@100"). Si oui, il y a trois sommes qui sont calculées (lignes 14, 15 et 16). La position dans le plan X et Y de la source lumineuse est calculée dans le lignes 17 et 18. Enfin, la fonction résultat et appelée pour transmettre la position calculée, ainsi que l'index du noeud qui l'a fait.

Les caractères "#" représentent les fins des clauses conditionnelles. Après la fonction de délai et la fin de la boucle *while*, la fonction *a* s'achève et le programme MinTax se termine

| Paramètre | Quantité | Unité |
|---|---|---|
| Taille de MinTax | 113 | octets |
| Temps de Réception | 298 | ms |
| Temps de compilation | 477 | ms |
| Temps de la l'écriture du résultat | 227 | ms |
| Nombre d'instructions d'assemblage générées | 202 | - |
| Taille du code d'assemblage généré | 570 | octets |

**TABLE 2.14:** Les différents paramètres associés à la compilation d'algorithme ObjectTracker.

avec le caractère NULL.

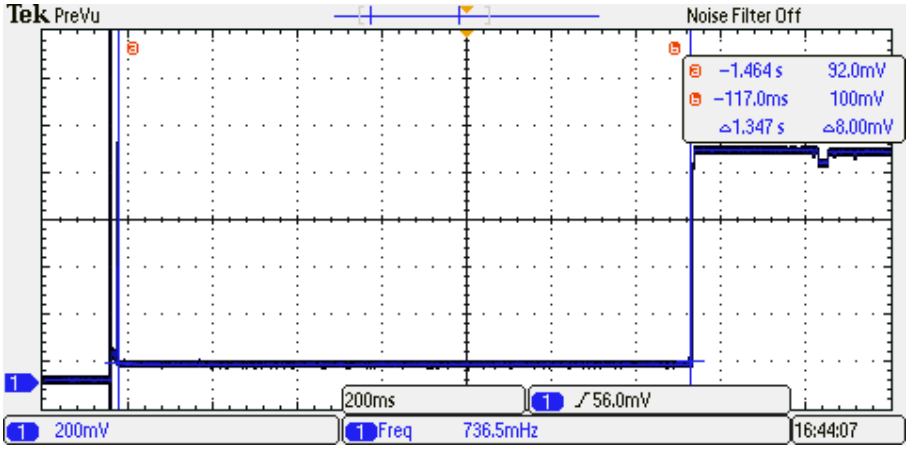


**FIGURE 2.35:** L'initialisation du noeud pour la réception du code en MinTax pour ObjectTracker.

Cependant, la transmission compte pour seulement 5ms, avec 34ms pour la stabilisation de l'oscillateur.

Puisque les moments où la transmission et la réception commencent ne sont pas synchronisés, le noeud qui reçoit passe un peu plus de temps à prendre les données transmises. Ceci dure 298ms 2.36.

### 2.21.3 Comparaison différentes solutions

Pour pouvoir démontrer l'avantage de MinTax sur les solutions existantes, on a utilisé la machine virtuelle VMSTAR, proposée par Dunkels et al. (52). VMSTAR est une machine

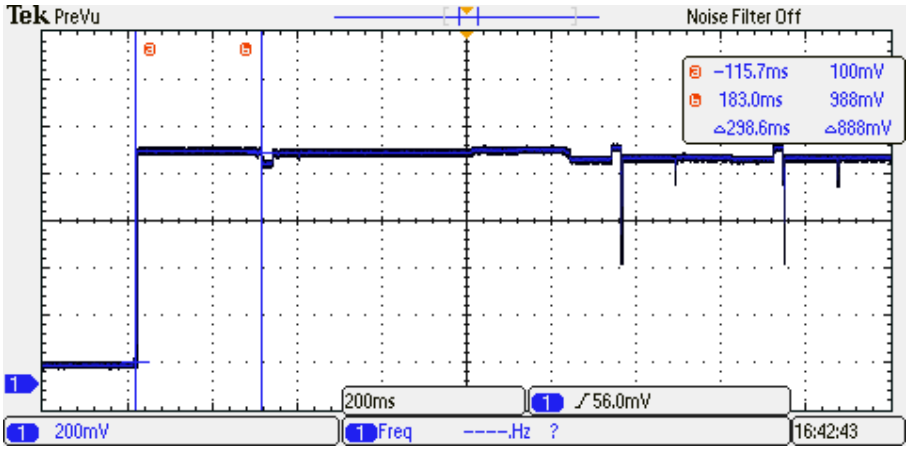


**Figure 2.36:** La réception du code en MinTax pour ObjectTracker.

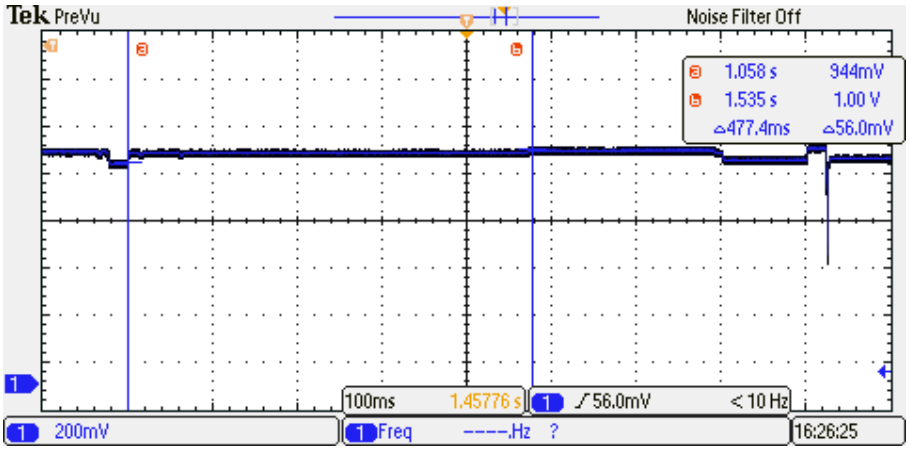


**Figure 2.37:** La compilation du code en MinTax pour ObjectTracker

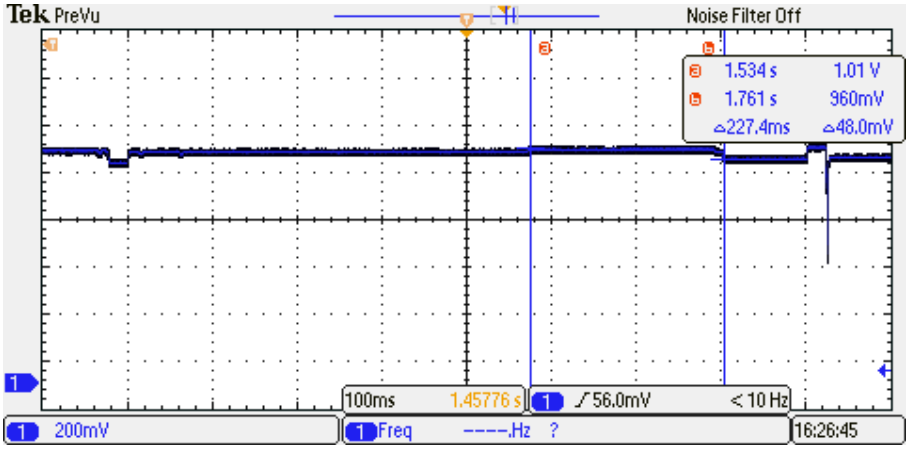


**Figure 2.38:** La réécriture de la flash avec le codé généré.

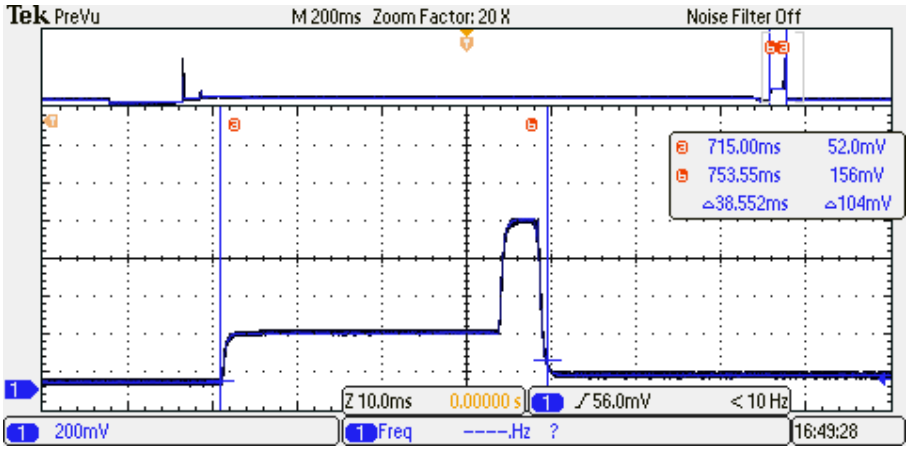


**FIGURE 2.39:** La transmission du code ObjectTracker.

| Solution | Programme | Taille code (octets) | Taille data (octets) | Taille fichier (octets) | Hétérogénéité matérielle | Hétérogénéité logicielle | Énergie RX RF (mJ) |
|---|---|---|---|---|---|---|---|
| VMSTAR | Blink | 130 | 14 | 361 | Oui | Non | 5.9 |
| | Object Tracker | 344 | 22 | 758 | Oui | Non | 12 |
| MinTax | Blink | 13 | 0 | 13 | Oui | Oui | 1.2 |
| | Object Tracker | 54 | 0 | 54 | Oui | Oui | 7.59 |

**TABLE 2.15:** Différents paramètres concernant les applications Blink et Object Tracker déployés sur VMSTAR et MinTax.

virtuelle qui fait usage du concept JIT (just in time compilation) qui lui permet un caractère hybride du code. Certaines portions sont interprétées et certaines portions sont exécutées en mode natif (code machine) après avoir été compilées. Cette approche hybride est meilleure du point de vue énergétique aux machines virtuelles classiques, comme argumenté dans la publication. C'est la raison pour laquelle on l'a choisie comme base de comparaison. Dans le cadre de reconfiguration pour VMSTAR, les auteurs proposent un nouvel format intitulé CELF, beaucoup plus petit que le format ELF. Ceci veut dire un temps de transmission et réception plus court.

Le format CELF contient toujours des adresses relatives de mémoire où le fichier doit être mis dans la mémoire flash. Ensuite, un éditeur de liens traduit ces adresses relatives en adresses

| Solution | Application | État | Durée (ms) | Énergie (mJ) | Total (mJ) |
|---|---|---|---|---|---|
| VMSTAR, Dunkels et al. | Blink | Écriture EEPROM<br>Link et relocate<br>Écriture Flash | 0.164<br>252<br>70 | 1.1<br>1.2<br>0.62 | 2.9 |
| | Object Tracker | Écriture EEPROM<br>Link et relocate<br>Écriture Flash | 0.282<br>600<br>106 | 1.9<br>2.9<br>5.5 | 5.5 |
| MinTax | Blink | Écriture EEPROM<br>Compilation<br>Écriture Flash | 0<br>110<br>21 | 0<br>0.53<br>0.18 | 0.71 |
| | Object Tracker | Écriture EEPROM<br>Compilation<br>Écriture Flash | 0<br>477<br>227 | 0<br>2.3<br>1.62 | 3.92 |

**TABLE 2.16:** La consommation énergétique détaillée pour Blink et ObjectTracker, utilisant VMSTAR et MinTax.

absolues afin que l'écriture de la mémoire flash puisse commencer. Ces adresses représentent une information supplémentaire qui augmente la taille du paquet de reconfiguration, ce qui veut dire une quantité d'énergie plus élevée pour le transmettre. Ceci est corroboré par les mesures présentées dans le Tableau 2.15.

Les résultats d'oscilloscope résumées dans le Tableau 2.16 montrent aussi que la solution MinTax est supérieure en terme d'énergie au processus de linkage présent dans VMSTAR. L'énergie requise pour ceci s'élève à 0.71 mJ pour MinTax et 2.9 mJ pour VMSTAR pour l'exemple Blink. Pour ObjectTracker, les résultats montrent un nécessaire de 3.92 mJ pour MinTax et 5.5 mJ pour VMSTAR.

Cela correspond à un gain total de MinTax sur VMSTAR de 78% pour l'exemple Blink et 34.2% pour ObjectTracker. Puisque le code implémenté en MinTax ne sait rien de la plate-forme logicielle de destination, un autre avantage supporté est la présence de l'hétérogénéité logicielle. En plus, d'après nos connaissances, MinTax est la seule solution qui supporte une hétérogénéité complète logicielle-matérielle pour la reconfiguration dynamique dans les WSNs.

# 2.22 Conclusion et perspectives

## 2.22.1 Sommaire du chapitre

Dans ce chapitre, un nouveau langage de programmation intitulé MinTax et son compilateur ont été présentés. Ce langage a été développé pour aider la mise à jour du logiciel qui s'exécute sur les noeuds des WSNs. Le compilateur associé permet de générer du code machine pour l'architecture cible (MSP430 ou AVR), offrant ainsi une hétérogénéité matérielle, donc une indépendance des caractéristiques matérielles des noeuds par rapport à la fonctionnalité (mise à jour). Grâce au fait que la mise à jour ne contient aucune information concernant le logiciel qui s'exécute sur le noeud avant la reconfiguration, une hétérogénéité logicielle est possible. Pour démontrer cela, notre solution a été déployée sur ContikiOS, FunkOS et FreeRTOS.

MinTax a été inspiré de C et Perl, et c'est un langage de programmation impératif. C'est-à-dire, il y a une suite d'instructions qui changent le comportement du programme implémenté. Il est aussi un langage structuré, faisant usage d'instructions conditionnelles et itératives pour construire le code (et non pas des sauts dans le code qui le rendraient difficile à lire).

Il offre un typage dynamique des variables comme discipline de typage.

Ceci veut dire que la variable ne doit pas forcement être déclarée à priori avant d'être écrite.

Les clauses classique de C, "if", "for", "while", "switch-case", les appels de fonctions, les clauses arithmétiques sont supportées. L'accès direct aux entrées/sorties numériques, les entrées analogiques ainsi que les sorties modulées en largeur d'impulsion sont supportés.

Pour pouvoir quantifier le gain que notre solution amène par rapport aux solutions existantes, on l'a comparé à VMSTAR, une machine virtuelle avec des performances largement supérieures aux autres solution de reconfiguration dynamique. Pour deux algorithmes testés, MinTax offre un gain de 78% (Blink) et 34%(ObjectTracker) par rapport à VMSTAR.

### 2.22.2 Perspectives

Pour l'avenir, le compilateur de MinTax pourrait offrir le support pour les microcontrôlleurs PIC.

Le code généré pourrait être optimisé pour enlever le nombre de instructions paires PUSH-POP successives.

Pour l'instant, ceci est dû aux étages d'imbrication et les étapes intermédiaires qui concernent la génération du code.

D'autres systèmes d'exploitation pourraient aussi être supportés avec les prochaines itérations du compilateur MinTax.

Une première version d'un traducteur de code C vers MinTax pourrait facilement être développée, car la syntaxe MinTax est très similaire au C. Une version beta existe déjà mais elle n'a pas été testée avec la dernière version de la grammaire MinTax.

Pour le long terme, on peut envisager une extrapolation aux niveaux supérieurs de notre solution. On parle ici de la génération du code MinTax utilisant des autres langages plus haut niveau pour faire la programmation au niveau de groupe ou au niveau du réseau, voire des réseaux.

# 3 BLISS et IDEA1TLM

## 3.1 Aperçu général

Pour tester les performances du compilateur MinTax, pour chaque exemple que l'on a démontré dans la section précédente, on a eu besoin d'un cas d'étude (testbed). Ceci veut dire avoir les noeuds physiques et les programmer, ainsi que faire de la communication radio.

On a voulu intégrer le travail de MinTax dans la plateforme d'exploration et simulation de WSNs IDEA1 (53) (54).

IDEA1 est un simulateur de WSNs inspiré de SCNSL (55), écrit en C++ et SystemC et validé par des mesures. Il peut faire des analyses énergétiques d'un réseau. SystemC est largement utilisé dans la communauté micro-électronique et offre le support pour la modélisation comportementale à partir de cycle d'horloge dans le microcontrôleur, au niveau du noeud et jusqu'au niveau de réseau.

IDEA1 offre le support pour les microcontrôleurs PIC de Microchip, ainsi que pour les AVR d'Atmel et des émetteurs-récepteurs radio tels que CC2420 (Texas Instruments) et MRF24J40 (Microchip). Il peut simuler à la fois les protocoles IEEE 802.15.4 "slotted" et "non-slotted CSMA-CA".

Dans l'implémentation initiale d'IDEA1, chaque noeud avait sa propre machine à états finis. Des extensions ont été faites pour modéliser le logiciel qui s'exécute sur le noeud lui-même.

Différentes tailles du logiciel en format binaire qui s'exécute sur le noeud correspondent à des temps d'exécutions différents qui peuvent être représentées avec une machine à états fini. De plus, l'interface entre le microcontrôleur et les capteurs, ainsi que avec lémetteur-récepteur radio doivent être modélisés en TLM (anglais "Transaction-Level Modelling") pour la meilleure estimation de la consommation énergétique. Avec TLM, le temps de communication entre le microntrôleur et l'émetteur-récepteur radio, ainsi que le capteur est modélisé d'une manière plus fine, permettant une estimation plus précise.

## 3.2 Modèle Matériel du Noeud

Par défaut, IDEA1 offre le support pour les architectures AVR et PIC. Avec nos extensions, dans un premier temps, on a créé un modèle générique pour le noeud (c'est-à-dire une réimplémentation d'IDEA1). Puis on a ajouté les particularités pour chaque architecture et on a aussi choisi d'offrir le support pour MSP430. Le modèle matériel du noed est celui présenté dans la Figure 3.1.

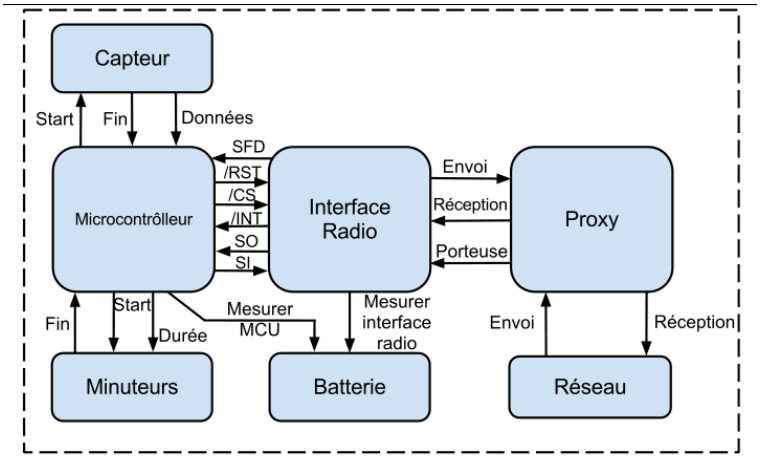


**Figure 3.1:** Modèle SystemC du noeud.

Le noeud est relié à un Proxy qui à son tour est connecté au modèle du réseau. Le modèle

du réseau permet de simuler l'environnement physique dans lequel les signaux radio sont émis, ainsi que pour gérer les paramètres tels que l'atténuation, l'interférence radio, et le temps de la transmission.

Pour ce modèle matériel, la consommation de courant est présentée dans le Tableau 3.1. Les valeurs sont issues des datasheets du fabricant pour chaque composant considéré.

| Parameter | Quantity | Units |
|---|---|---|
| AVR active@1Mhz@3V | 0.7 | mA |
| AVR sleep | 0.001 | mA |
| PIC active@1Mhz@3V | 0.3 | mA |
| PIC sleep | 0.0002 | mA |
| MSP430 active@1Mhz@3V | 0.6 | mA |
| MSP430 sleep | 0.0005 | mA |
| CC2420 TX@-25dBm | 8.5 | mA |
| CC2420 TX@-15dBm | 9.9 | mA |
| CC2420 TX@-10dBm | 11 | mA |
| CC2420 TX@-5dBm | 14 | mA |
| CC2420 TX@0dBm | 17.4 | mA |
| CC2420 RX | 19.7 | mA |
| MRF24J40 TX@-25dBm | 22.4 | mA |
| MRF24J40 TX@-15dBm | 22.5 | mA |
| MRF24J40 TX@-10dBm | 22.9 | mA |
| MRF24J40 TX@-5dBm | 22.95 | mA |
| MRF24J40 TX@0dBm | 23 | mA |
| SPI Pins | 100 | uA/pin |

**TABLE 3.1:** Paramètres concernant la consommation énergétique du modèle matériel du noeud.

Le microcontrôleur s'interface avec émetteur-récepteur radio par une interface SPI et avec le capteur en utilisant de signaux de contrôle et un chemin de données. Le bloc de la batterie est responsable de tenir compte de la consommation énergétique du noeud, lors d'une transition d'état du microcontrôleur ou de l'émetteur-récepteur radio, ainsi que de la consommation du capteur.

Étant donné le temps pour qu'un seul bit soit transféré sur le bus SPI (en tenant compte de la fréquence du fonctionnement du microntrôleur, la vitesse de la communication SPI) et le protocole de la communication SPI, on peut calculer exactement combien de temps dure un transfert. Le modèle pour le matériel prend en compte des choses telles que les commandes d'initialisation pour l'émetteur-récepteur radio, l'écriture dans la mémoire FIFO de réception et d'envoi, ainsi que la reprise de ces actions en utilisant le bloc "Timer".

Ensuite, en connaissant toutes ces informations, la tension d'alimentation et le courant consommé par chaque bloc dans chaque état possible, on peut avoir une estimation fine, raffinée de l'empreinte énergétique du noeud.

Pour choisir entre les architectures supportées, il suffit de changer un fichier de configuration dans un seul endroit et toutes les fonctions et états associés à l'architecture seront automatiquement choisis pour elle. Des autres variables comme la vitesse de la communication SPI, la fréquence du microntrôleur, la tension d'alimentation et des messages de debug pour la communication SPI, la vitesse de transfert et le temps pour la calibration radio, peuvent aussi être configurés. Pour le protocole 802.15.4, on peut configurer la durée du "backoff" (en jouant sur le "beacon exponent" et le nombre des actions de ce type).

## 3.3 Modèle Logiciel du Noeud

Si une fonctionnalité est définie par SystemC étant la description fonctionnelle d'un système ("Untimed Functional Description" en anglais), le logiciel qui la compose est la description fonctionnelle temporisée ("Timed Functional Description"). Dans ce cas, les temps d'exécution de chaque processus sont importants, mais on fait abstraction de la manière dont ce processus communiquent entre eux. Ils le font par ce qui est compris dans le modèle "Bus Cycle Accurate" (aussi dénommé "BCA", précis au niveau du cycle de bus) et "Register Transfer Level" (ou "RTL" tout court, précis au niveau du cycle d'horloge). Ceci est présenté dans la Figure 3.2.

Les niveaux RTL, CABA et BCA donnent ce qu'on appelle TLM ("Transaction Level Modelling"), donc un modèle précis comportemental du noeud.

Une approche commune serait d'utiliser TLM pour que l'unité de compilation SystemC prenne en compte le logiciel qui s'exécute sur le noeud, en modélisant les différents signaux présents dans l'unité de calcul (CPU). C'est-à-dire de modéliser les micro-instructions associés

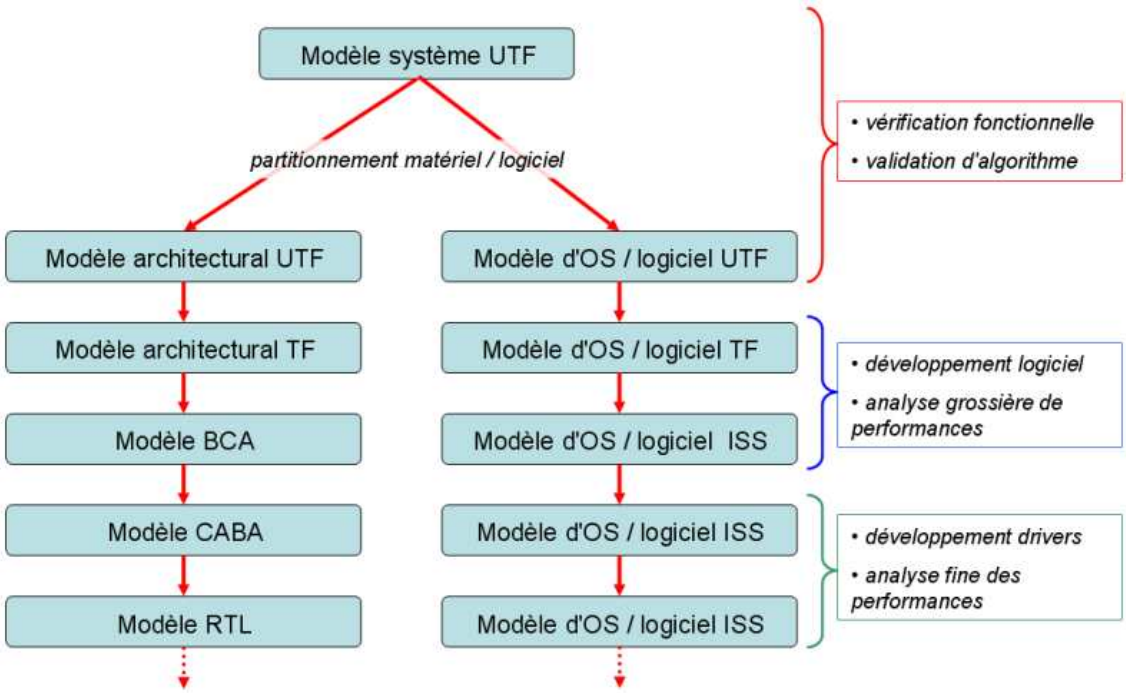


**FIGURE 3.2:** Développement d'un système : rafinement des modèles matériels / logiciels (8).

à chaque instruction que le CPU exécute (le fait de faire amener l'instruction à partir de la flash, la décoder, l'exécuter, l'accès éventuel à la mémoire et l'écriture du résultat). On a décidé pour une autre façon. On a implémenté une approche en deux phases, composé par une représentation du logiciel par une machine d'état fini, couplé avec la métadonnée générée par un simulateur du set d'instructions ("Instruction Set Simulator", ou "ISS"). Les deux phases seront clarifiées prochainement.

Dans ce cas, puisque le noeud n'est pas lié aux signaux internes et aux micro-instructions, il a un caractère générique. L'ISS qu'on a développé et couplé avec la machine d'état fini dont on parlait tout à l'heure (pour avoir donc le modèle logiciel du noeud), s'appelle BLISS.

La méthodologie qu'on veut utiliser pour le simulateur de réseau peut être consultée dans la Figure 3.3. L'ancienne version d'IDEA1 ne la respecte pas, c'est pourquoi on a développé la nouvelle version, IDEA1TLM.

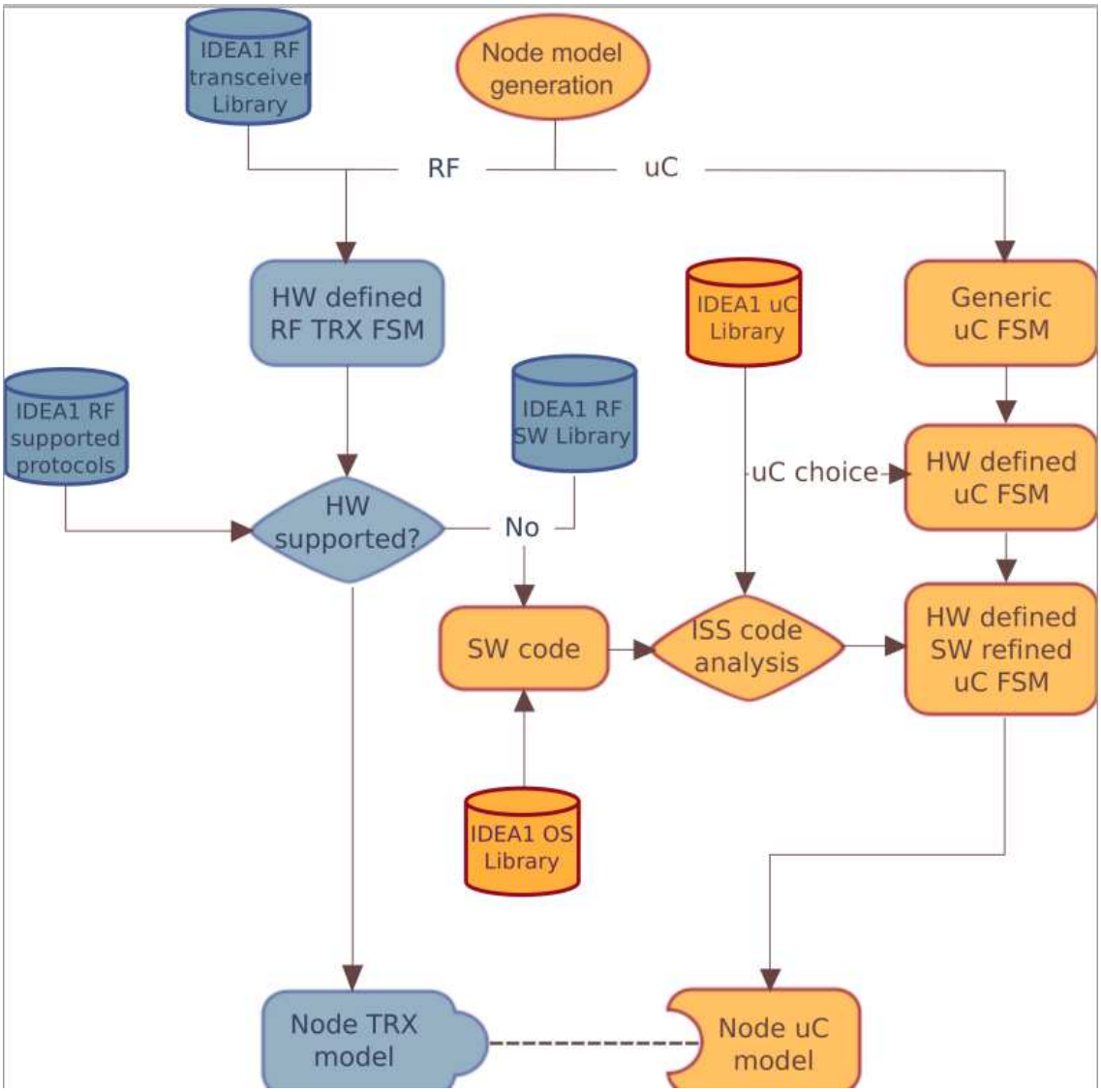


**FIGURE 3.3:** Méthodologie IDEA1TLM.

# 3.4 IDEA1TLM

## 3.4.1 Aperçu global

Ce qu'on a voulu construire c'était un modèle générique du noeud, capable de bien pouvoir donner une forme au comportement de celui-ci tout en tenant compte des particularités associées avec les différents paramètres des composants y présents. On a imaginé une architecture logicielle globale pour plusieurs types des émetteurs-récepteurs radiofréquence, une autre pour les différentes microcontrôlleurs et pour la batterie. On a imaginé greffer ce modèle seulement avec des informations pertinentes concernant la consommation énergétique du chaque composant.

Parce que chaque émetteur-récepteur radiofréquence a des registres internes différents qui doivent être initialisés pour pouvoir communiquer, ceci serait impossible au niveau d'émetteur-récepteur radiofréquence, mais triviale au niveau de microcontrôlleur. Triviale, parce que la structure interne du microcontrolleur n'est pas modélisée. Au lieu de cela, il y a que l'interface SPI qui l'est. Les deux émetteurs-récepteurs radiofréquence pour lesquels on offre support dans IDEA1 TLM sont CC2420 et MRF24J40. On ne peut pas comprendre le comportement des deux dans un module générique, car elles traitent la transmission / réception radio d'une manière différente. CC2420 a besoin des commandes par transferts SPI du microntrôleur pour savoir quand transmettre, quand se mettre en veille, etc. Dans le cas de MRF24J40, il suffit de configurer les registres afférents une fois et il y aura moins de communication SPI avec le microntrôleur, beaucoup de décisions étant prises par l'émetteur-récepteur radio lui-même.

Une contrainte sérieuse de SystemC est le manque de support pour l'héritage des classes. Ceci est dû partiellement aux instructions de préprocesseur (macros pour déclarer le module, le constructeur, etc.). D'un autre côté, SystemC n'était tout simplement pas prévu avec du support pour ce mécanisme (en ce qui concerne la liste de sensibilité, etc.).

Par conséquent, l'implémentation d'une partie commune pour CC2420 et MRF24J40 (le fils concernant la réception ou transmission, plus précisément l'accès au canal à travers du proxy) est impossible. Les deux sont implémentés comme deux modules différents.

Dû au fait que le microntrôleur d'un node normal et celui d'un coordinateur se comportent différemment dans le point de vue logiciel, eux aussi sont deux modules distincts. Bien évidemment c'est le cas où le temps pour le slot d'envoi est calculé par le microntrôleur. Le coordinateur est chargé d'obtenir les données des noeuds et dans le mode "Beacon" c'est lui qui émet un paquet spécial avec ce nom. Selon le type d'émetteur-récepteur radiofréquence présent sur le noeud, la procédure d'initialisation de celle-ci est différente, ainsi que la manière dont le microntrôleur gère la transmission ou la réception.

Le module de consommation a des interfaces avec tous les modules, pour pouvoir déterminer la consommation de chacun.

La Figure 3.5 offre un aperçu général de l'architecture logicielle d'IDEA1 TLM. Ceci sera repris plus en détail dans les sections suivantes. Les couches matérielles sont représentées par des boites rouges, celles logicielles sont en bleu ou bien vert pour les couches de plus haut niveau.

Lorsqu'un noeud transmet ses données, il peut être dans deux situations. Soit il transmet une Trame Longue (Long Frame), soit une Trame Courte (Short Frame). Une trame tombe dans l'un de deux cas selon la longueur de la data, supérieure ou inférieure à 18 octets. Un intervalle minimal de temps est requis avant qu'une nouvelle transmission puisse recommencer. Ceci offre la possibilité au récepteur de lire sa FIFO de réception. Ce processus est démontré dans la Figure 3.5. Pour la trame longue, un intervalle LIFS (Long Interframe Spacing) est requis, et pour une trame courte, un SIFS (Short Interframe Spacing).

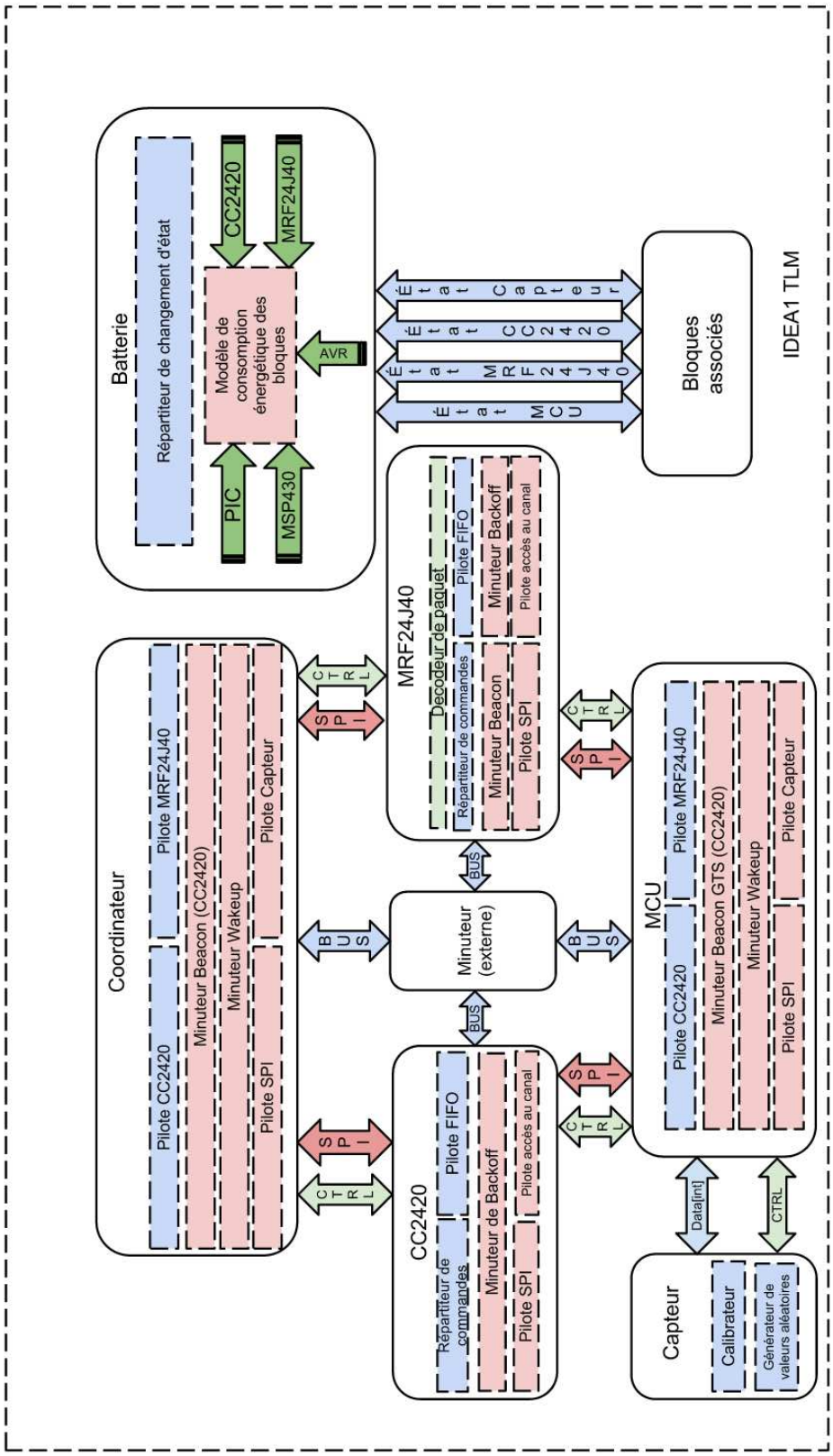


**FIGURE 3.4:** L'architecture logicielle d'IDEA1 TLM.

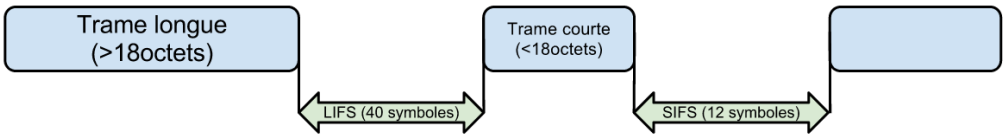


**FIGURE 3.5:** Les intervalles inter-trame et leur longueur.

### 3.4.2 Modèle du Microcontrôleur

#### 3.4.2.1 Généralités

Il y a deux modules qui comprennent la fonctionnalité du microcontrôlleur, “MCU” et “Coordinateur” (genericmcu.c et coordinator.c). Dû au fait que le Microcontrôlleur d’un noeud se comporte différemment si le noeud est coordinateur ou pas, on a décidé sur cette approche pour bien les différencier.

Quoiqu’elle soit l’architecture du microntrôleur visée, il n’y a pas de différence d’implémentation au niveau de module matériel de microntrôleur ou Coordinateur. En revanche, ce modèle doit gérer des émetteurs-récepteurs radiofréquence avec des différentes signaux d’interface, donc les ports diffèrent selon la partie radio. Puisqu’on veut pouvoir avoir des configurations hétérogènes microntrôleur-émetteur-récepteur radiofréquence dans le réseau, le modèle de microntrôleur doit comprendre tous les pins requis par toutes les radios, même si certains uns ne seront pas utilisés. Du côté de la simulation, SystemC ne peut pas marcher sans avoir tous les pins reliés à des signaux, même s’ils ne sont pas utilisés. Par conséquent, on a implémenté des interfaces “stub” qui ne sont donc connectées à rien.

#### 3.4.2.2 Le microntrôleur gère la communication radiofréquence

Dans le cas où le microntrôleur se charge de piloter l’émetteur-récepteur radio, il doit avoir un dialogue durant la période de veille avec celui-ci, à travers l’interface SPI. Le microntrôleur va recevoir les données de l’interface radio, va les décoder, et évaluer le type de paquet. S’il

s'agit d'un paquet beacon, il va attendre un délai équivalent pour pouvoir commencer la communication.

Les modules Noeud et Coordinateur ont cinq parties composantes au niveau matériel. Celles-ci sont les ports matériels, le pilote d'interface SPI, de capteur, le minuteur de veille et le minuteur pour l'intervalle de beacon, dans le cas de CC2420.

- L'interface SPI utilise les signaux SPI standard (MOSI, MISO, SCK et RESET). Le temps d'envoi pour chaque bit individuel peut être configuré. En ce qui concerne la communication avec les émetteurs-récepteurs radiofréquence, on fait appel à une fonction d'envoi qui a le même nom quoiqu'il s'agit du CC2420 ou MRF24J40. La dernière a besoin que l'adresse d'écriture soit envoyée avant chaque octet de la donnée. L'interface teste une variable pour connaître le type de l'émetteur-récepteur connecté et décide si elle doit envoyer l'adresse d'abord.
- Le compteur de veille retire périodiquement le microntrôleur du mode sleep. Il sera remis en sleep une fois fini l'acquisition et l'envoi des données du capteur.
- Le beacon est un paquet spécial dans le protocole 802.15.4, ayant le rôle de synchroniser la transmission des données entre les noeuds. Les noeuds se calent sur ce paquet considéré comme balise et font une demande de transmission au coordinateur immédiatement après avoir reçu le paquet de beacon. Cet intervalle de temps a le nom de "CAP" (Contention Free Period). La durée de la CAP est configurable, mais reste un multiple de 60 symboles. Chaque symbole a une durée variable de $16us * 2^{SO}$ (où SO est le "Superframe Order").
- Pour pouvoir faire la demande, les noeuds font un CSMA-CA (Carrier Sense Multiple Access - Collision Avoidance) : ils se mettent en mode réception et s'ils ne reçoivent rien pendant la période de 8 symboles, ils considèrent que le canal est libre. Ensuite, ils émettent leur demande.
- Le coordinateur la prend en compte et avec le prochain beacon, leur alloue un intervalle (ou bien un "slot" en anglais) pour qu'ils puissent émettre dans l'intervalle dénommé "CFP" (Contention Free Period). Dans la période de CFP, un slot porte le nom de GTS

| Champ | Longueur (octets) | Description / bits | Sous-description |
|---|---|---|---|
| Longueur du packet | 1 | - | |
| Frame Check Sequence | 2 | Type de paquet | - |
| Adressage | 4 à 20 | Source, destination | |
| Superframe Specification | 2 | 15 : Association permit<br>14 : Coordinator bit<br>13 : Reserved<br>12 : Battery save extension<br>11 à 8 : Final CAP slot<br>7 à 4 : Superframe order<br>3 à 0 : Beacon order | - |
| GTS | variable | GTS[i] Specification | 7 : GTS[i] Perimit<br>6 à 3 : Reserved<br>2 à 0 : GTS[i] Descriptor cnt |
| | | GTS directions | - |
| | | GTS[i+1] next list element. | - |
| Pending Address | variable | Pending address spec. | 7 et 3 : Reserved<br>6 à 4 : Ext. Addr. pending<br>2 à 0 : Short addr. pending |
| Beacon Payload | variable | - | - |
| Frame Check Sequence | 2 | Cyclic redundancy check | - |

**TABLE 3.2:** La structure d'une trame Beacon.

(Guaranteed Time Slot). À chaque noeud est alloué un slot différent. Il existe jusqu'à 7 GTS, chacun contenant 8 sub-slots. Dans le paquet de beacon, il existe de champ spécial qui indique si le GTS(i) existe et quel est son dernier sub-slot. Si un noeud n'a pas la permission d'émettre dans un GTS, il va essayer d'émettre dans la CFP d'après le prochain beacon. S'il y a trop de noeuds qui en essayent et donc le canal est occupé, le noeud va incrémenter un compteur et va demander de nouveau accès après le prochain beacon. Ceci se répète jusqu'à ce que le noeud accède à un GTS, ou émet dans la CFP ou bien, le compteur atteint une certaine valeur. Dans ce dernier cas, le paquet est considéré perdu. Une fois la période CFP finie, on peut avoir un intervalle inactif avant le prochain beacon. La période de temps qui coule entre deux beacons s'appelle "Super Frame" (une "super" trame).

La Figure 3.7 montre le format d'une Super Trame. La durée de la période active (CAP + CFP) et période inactive peut être ajustée avec les paramètres BO (Beacon Order) et SO (SuperFrame Order).

Le Coordinateur construit le paquet beacon selon les spécifications présentées dans la Ta-

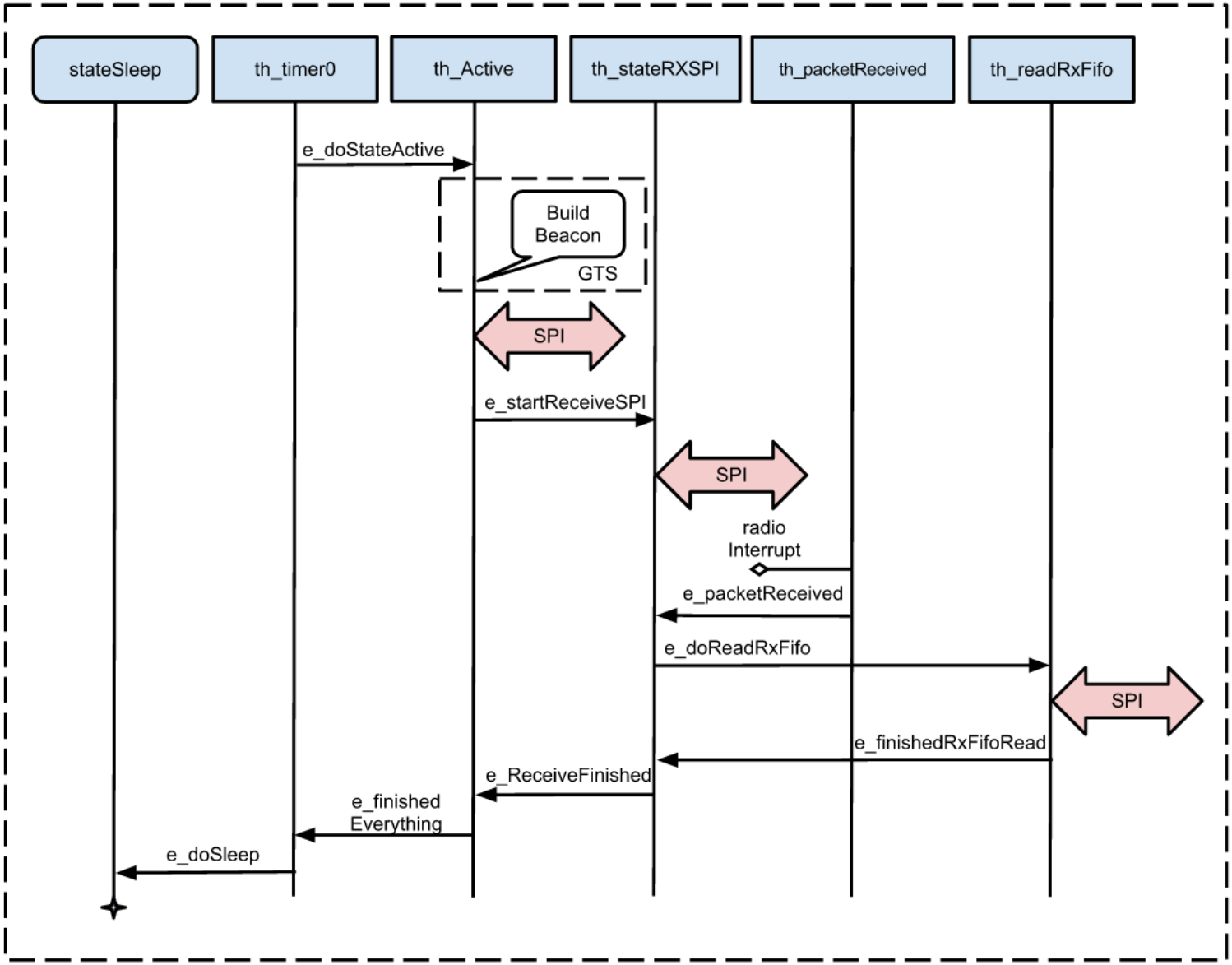


**FIGURE 3.6:** Le module Coordinateur et son architecture interne.

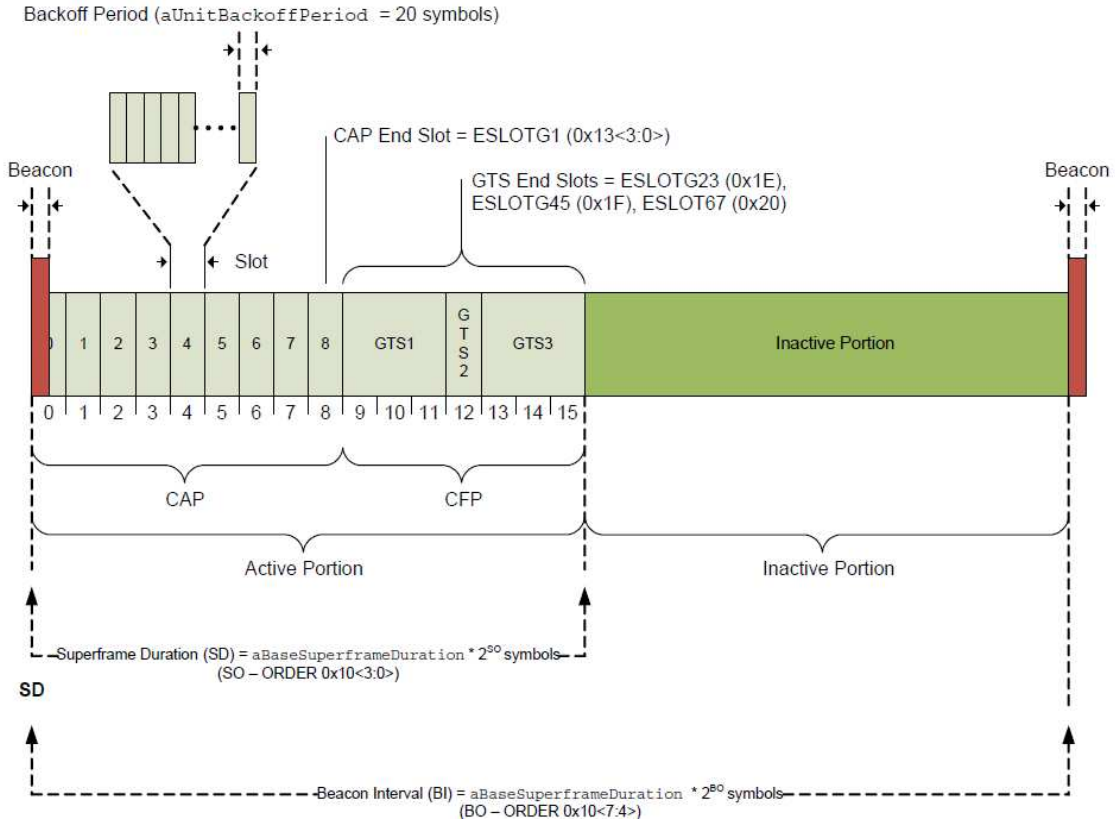


**FIGURE 3.7:** La Super Trame 802.15.4. Image prise de la documentation du composant MRF24J40.

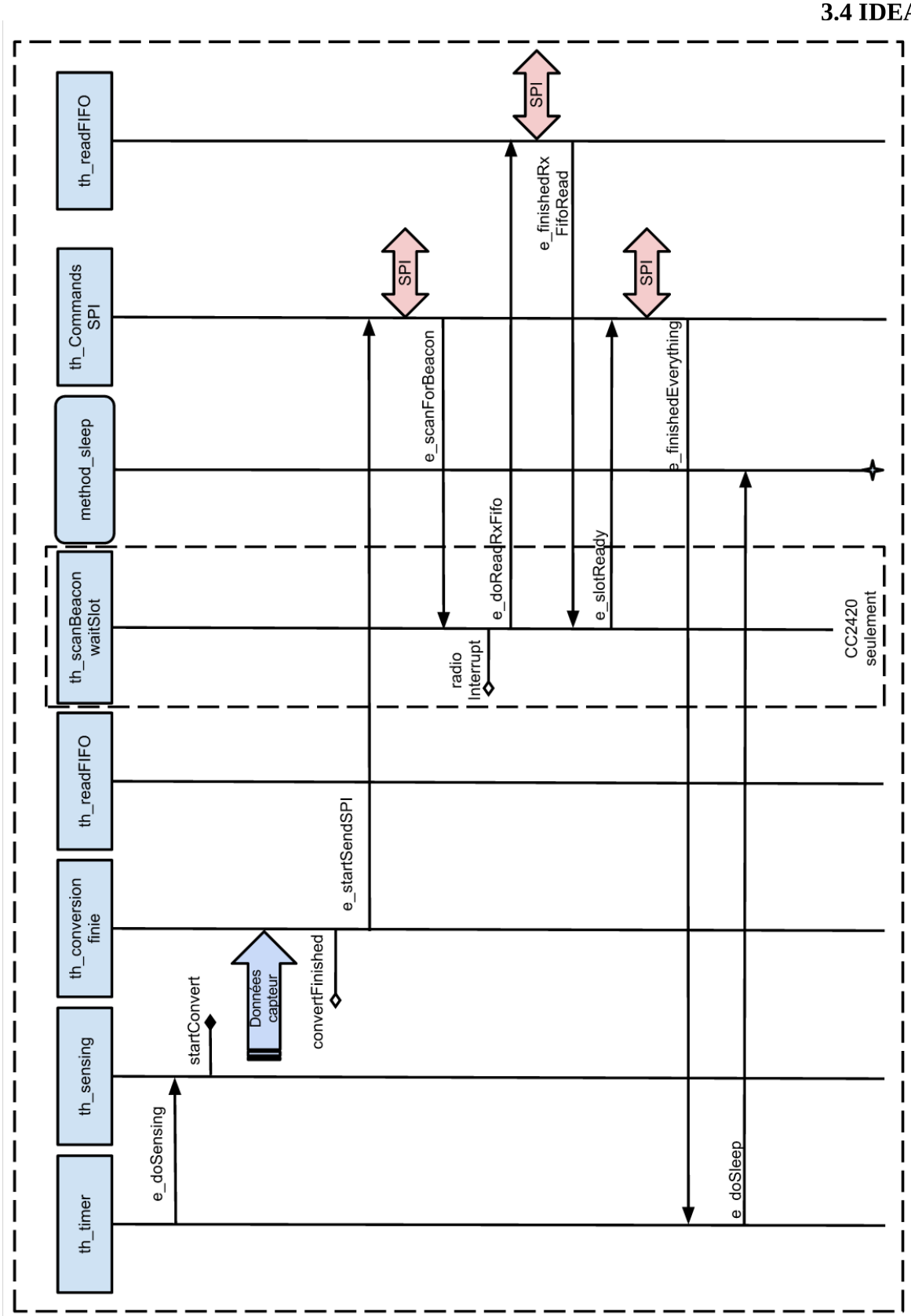


**FIGURE 3.8:** Le module microntrôleur et son architecture interne.

ble 3.2 et demande à l'émetteur-récepteur radiofréquence de les transmettre selon la Figure 3.7. Il envoie ce paquet lorsque le timer afférent expire. Les noeuds doivent achever leur transmission avant la fin de la Super Trame, puisqu'à l'expiration du timer de beacon, le Coordinateur va en émettre un nouveau. Normalement, si les noeuds n'ont pas réussi à transmettre leur données entièrement (dû, par exemple à la grande taille du paquet et la courte durée d'un slot), ils vont essayer le faire dans les prochaines Super Trames.

Pour l'instant, IDEA1 TLM supporte seulement les transmissions complètes sur plusieurs Super Trames. C'est-à-dire qu'une fois que le noeud transmet son information, il le fait entièrement. S'il n'a pas accès au canal, il va essayer de la transmettre dès la réception d'un nouveau beacon, bien évidemment. Mais pour une payload de grande taille, IDEA1 TLM attend que l'utilisateur joue sur la valeur de BO et SO pour que la longueur d'un slot soit suffisamment longue pour qu'un paquet entier puisse être transmis.

Les architectures internes des modules Microcontrôlleur et Coordinateur sont décrites dans les Figures 3.8 et 3.6.

#### 3.4.2.3 L'émetteur-récepteur radio gère la communication radiofréquence

Il est possible d'avoir un émetteur-récepteur radio qui, une fois configuré, se charge de la communication radio sans pilotage additionnel de la part du microntrôleur. C'est-à-dire qu'une grande partie du processus décrit dans la sous-section précédente est maintenant implémenté dans l'interface radio. Ceci aide à minimiser les communications SPI pour un transfert de données. L'interface radio va, en outre, décoder les trames beacon, va automatiquement calculer le délai requis pour le slot assigné et commencer l'envoi, sans aide du microntrôleur.

### 3.4.3 Modèle de l'émetteur-récepteur radio CC2420

L'architecture interne de l'émetteur-récepteur CC2420 est présentée dans la Figure 3.9. Ce modèle prend en compte le pilote SPI, le timer de Backoff et le pilote d'accès au canal. Dès que le CC2420 est initialisé, il fait un Backoff de durée aléatoire. Ceci empêche (voire

réduit) la possibilité de collisions dans les transmissions unslotted ou dans la période CAP de la transmission slotted.

Le Backoff est en effet une durée comprise entre *macMinBE* et $2^{BE}$ – 1 de 20 symboles chacune. “BE” est l’exposant beacon (Beacon Exponent) et il a la valeur 3 à l’initialisation. Pour chaque backoff qui se fait sans succès de transmission, cette valeur est incrémentée jusqu’à 5. Ensuite, elle est mise à zéro.

Le modèle CC2420 se pose sur une couche matérielle qui modélise l’interface SPI. Ceci est responsable pour sérialiser les informations et accepter celles qui sont sérialisées par le microntrôleur. Il existe deux fonctions responsables pour la cette communication, *th_receiveSPI* pour la reception et *th_sendSPIByte* pour l’envoi. Après chaque commande reçue, th_receiveSPI envoie au microntrôleur la valeur du registre d’état.

Dès qu’une information est reçue par SPI, la routine de réception informe un thread responsable de la répartition des événements, dénommé “handleReceivedSPI”. Il s’occupe d’envoyer des événements vers les fils correspondants.

La première commande envoyée à l’émetteur-récepteur radiofréquence et celle de calibration de l’oscillateur. Après avoir attendu la durée nécessaire pour la stabilisation, le registre d’état est mis à jour avec le bit correspondant. Pendant ce temps, le microntrôleur fait la lecture du registre d’état en lui envoyant la commande “NOP” (No opération). Ceci a comme effet que le CC2420 envoie son état. Dès que la stabilisation du cristal est faite, le microntrôleur peut procéder à d’autres commandes, la construction de paquets, la transmission, la réception, etc.

Le timer de Backoff et le Pilote d’accès au canal sont implémentés dans les routines de transmission et réception et seront traités dans les prochaines sous-sections.

#### 3.4.3.1 Transmission

Dans les prochains paragraphes la description de la transmission avec CC2420 va être continuée. Il faut préciser qu’elle commence par l’envoi de l’événement *e_tx* par le répartiteur SPI. Le répartiteur SPI est une entité du module de l’émetteur-récepteur radio

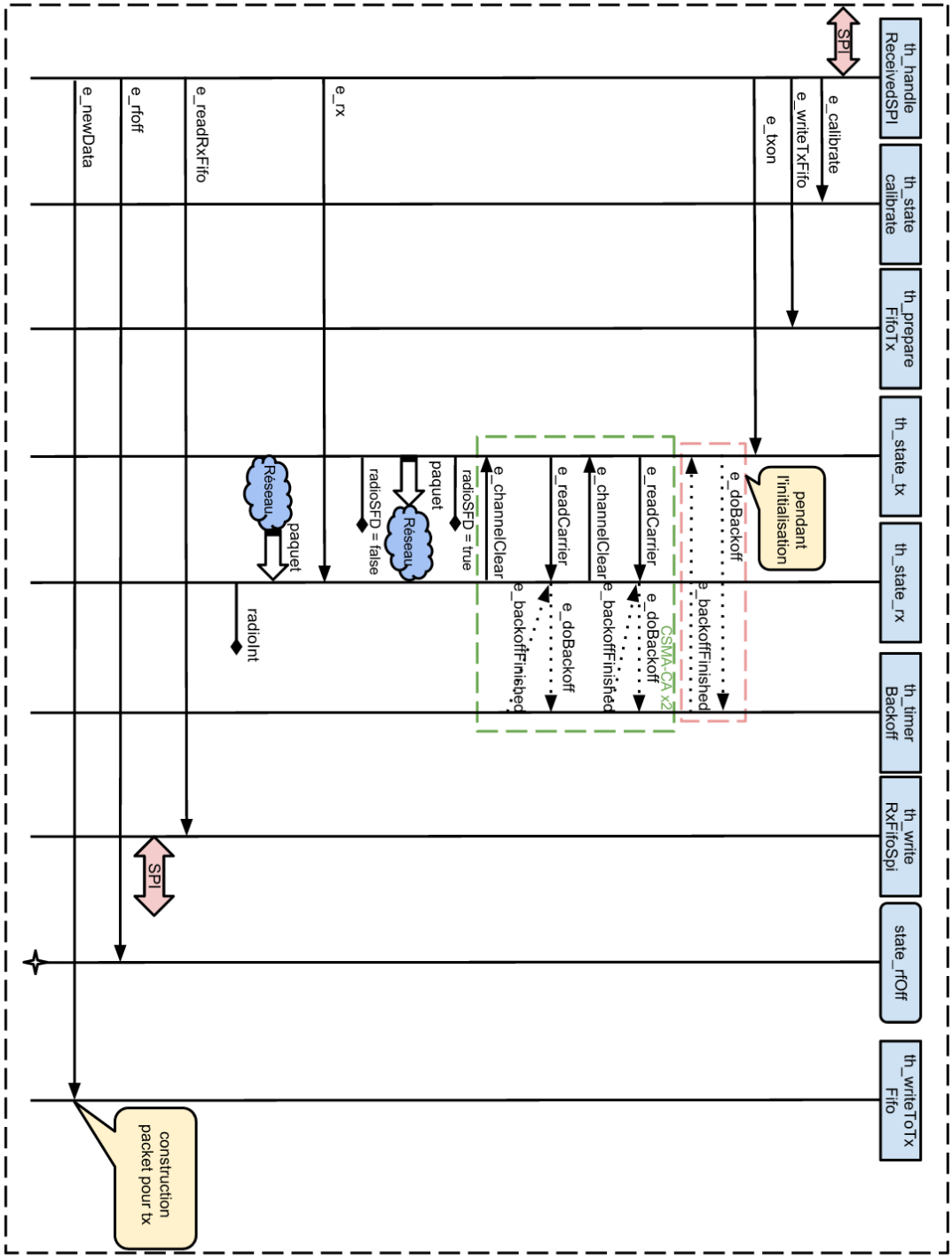


**FIGURE 3.9:** Le module cc2420 et son architecture interne.

responsable du décodage des commandes SPI, reçues par le bus de données. Le microntrôleur communique avec le CC2420 pour mettre des informations dans la FIFO d'émission. Il existe deux types de données envoyées par SPI entre les deux : adresses et données. Tout d'abord, le microntrôleur envoie l'adresse d'où il veut écrire ou lire. Certaines adresses sont des registres qui contrôlent, par exemple le démarrage de l'oscillateur.

Le microntrôleur envoie une commande d'accès à la FIFO de transmission. Ensuite, il envoie le premier octet de la trame IEEE 802.15.4 qui est en effet la longueur du paquet.Il n'envoie plus d'adresse jusqu'à ce que la longueur de données soit transféré. Le microntrôleur envoie l'adresse d'accès au registre "TXON" qui est en effet une commande de démarrage de transmission.

En ce qui concerne la construction du paquet, à chaque nouvelle donnée reçue par SPI, le modèle la met dans le champ qui correspond à la position dans le paquet d'envoi. Il fait ça à chaque octet reçu plutôt qu'une seule fois, avec une conversion de type. La conversion de type est nécessaire dû à l'implémentation de la structure du paquet dans SCNSL. Il remplit donc la métadata du paquet, ensuite le champ "Valeur". Il écrit 8 octets dans ce champ, après quoi il remplit le champ "Temps". Si la longueur de la data est supérieure à 8 octets, le reste il écrit dans la prochaine paire valeur-temps.

Si le CC2420 veut émettre et qu'un slot n'est pas alloué, il va essayer d'emmètre dans la période CAP (Contention Access Period). Il doit faire une écoute du canal durant 8 symboles, puis encore une écoute de 8 symboles. S'il n'y a pas d'activité pendant ces deux périodes, le canal est considéré libre et la transmission peut commencer.

Si le canal est occupé, le noeud fait un Backoff, incrémente son compteur de nombre de backoffs, et essaye plus tard.

Si le Coordinateur a alloué un slot pour le noeud (le mode slotted), il n'y a plus d'écoute du canal et la transmission peut commencer tout de suite.

CC2420 met le pin "radioSFD" au niveau haut et informe le module batterie d'un changement d'état de la consommation, en lui envoyant le nouveau état "T" (transmission). Ensuite, le paquet est transmis au proxy qui l'avance dans le réseau.

Il faut préciser que dû au fait que IDEA1 TLM utilise les modèles Proxy et Réseau de SCNSL, on est obligé d'utiliser la structure du paquet dans leur implémentation (avec les valeurs "Valeur" et "Temps" dont on a parlé tout à l'heure). Pour résumer, la partie utile du paquet est censé être un tableau de deux valeurs "double", qui se représentent sur 8 octets. Par conséquent, la payload ne peut être qu'un multiple de 16 octets. De plus, avec une variable uint64_t (aussi avec une longueur de 8 octets) on ne peut pas faire une conversion de type directe à double. On est requis à passer par des pointeurs, sinon on risque de perdre des informations dû au caractère limité du type de data. Selon la longueur du paquet, le IDEA1 TLM simule un temps d'attente pour la propagation dans le réseau, afin qu'il informe le module batterie d'état inactif de l'émetteur-récepteur radiofréquence ("I").

#### 3.4.3.2 Réception

Le fil d'exécution pour la réception commence dès qu'il reçoit un événement "e_rx" par le répartiteur de commandes dans le modèle. Ensuite, il y a une lecture du canal durant 8 périodes de symbole. Si le canal est actif cela veut dire qu'il y a un noeud voisin qui émet. L'information est prise du canal, et le modèle attend que le canal soit de nouveau libre. C'est-à-dire que le noeud émetteur achève sa transmission. Ensuite, la patte de "radioINT" est mise au niveau haut pour une certaine durée, afin qu'il soit remis à zéro.

Pour le mode debug, il existe une fonction qui peut imprimer le paquet reçu, ainsi que les valeurs double de le payload en notation hexadécimale.

### 3.4.4 Modèle de l'émetteur-récepteur MRF24J40

L'architecture du MRF24J40 est présentée dans la Figure 3.10. Pour communiquer avec le microntrôleur, il utilise le Pilote d'interface SPI. Les autres parties émulées sont le timer de Beacon, le timer de Backoff et le Pilote d'accès au canal.

La réception SPI est gérée par *th_receiveSPI* et l'envoi par *th_sendSPIByte*. L'accès aux registres du MRF24J40 peut être fait, selon le registre, avec soit un adressage long soit un court.

Le type d'adressage est dicté par le premier bit envoyé pour l'adresse (0 = court, 1=long). Plus de détails sur la réception peuvent être trouvés dans la Figure 3.11. Pour chaque valeur envoyée, l'adresse d'écriture ou lecture doit la précéder. Les valeurs, à part pour les adresses longues, sont toujours sur 8 bits.

Le modèle pour cet émetteur-récepteur radiofréquence contient les registres associés pour la commande du chip.

Pour pouvoir gérer les informations qui sont envoyées à MRF24J40 par SPI, on a implémenté un répartiteur dans le thread *handleReceivedSPI*. Une fois que l'adresse et la valeur sont décodées, cette fonction modifie la valeur du registre associé et éventuellement, envoie un événement au fil d'exécution associé à la commande. C'est ainsi que le chip sait quand commencer la routine de reset, ou de stabiliser son oscillateur.

Le modèle pour MRF24J40 a plusieurs FIFOs, une pour la réception (RX FIFO, 144 octets), une pour la transmission normale (TX Normal, 128 octets), une pour le beacon (TX Beacon) et une (seule) pour l'envoi GTS (TX GTS).

#### 3.4.4.1 Transmission

Tout d'abord, à la phase d'élaboration, l'utilisateur doit choisir s'il s'agit d'un réseau qui offre du support pour la transmission "slotted" ou "unslotted". Ceci aura un impact sur quelle FIFO séra utilisée pour l'envoi.

Pour que la tranmission puisse commencer, le paquet doit être écrit dans une des mémoires FIFO de sortie. Selon le type de paquet, le modèle utilise la TX GTS FIFO (pour la transmission GTS, slotted), la TX Beacon (pour un paquet Beacon, slotted) ou bien, la TX Normal FIFO (pour une transmission normale, unslotted).

Si on parle d'une transmission CSMA-CA "unslotted", le comportement du modèle est identique à celui du CC2420. Il y a donc deux mesures du canal (chacun de 8 périodes de symbole) et s'il est libre, la transmission commence immédiatement. Sinon, il y a un Backoff qui est fait (en utilisant le Minuteur de Backoff).

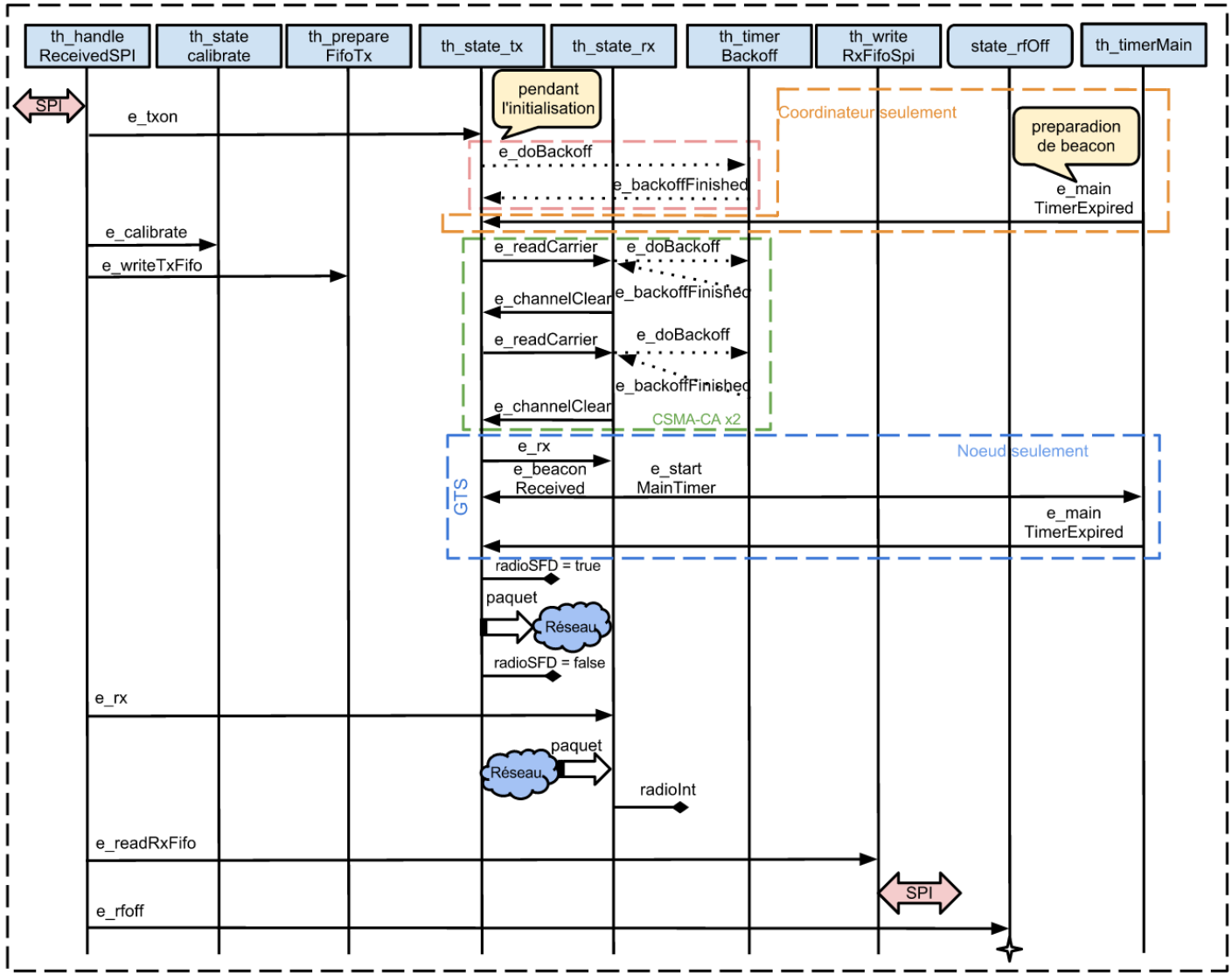


**Figure 3.10:** Le module MRF24J40 et son architecture interne.

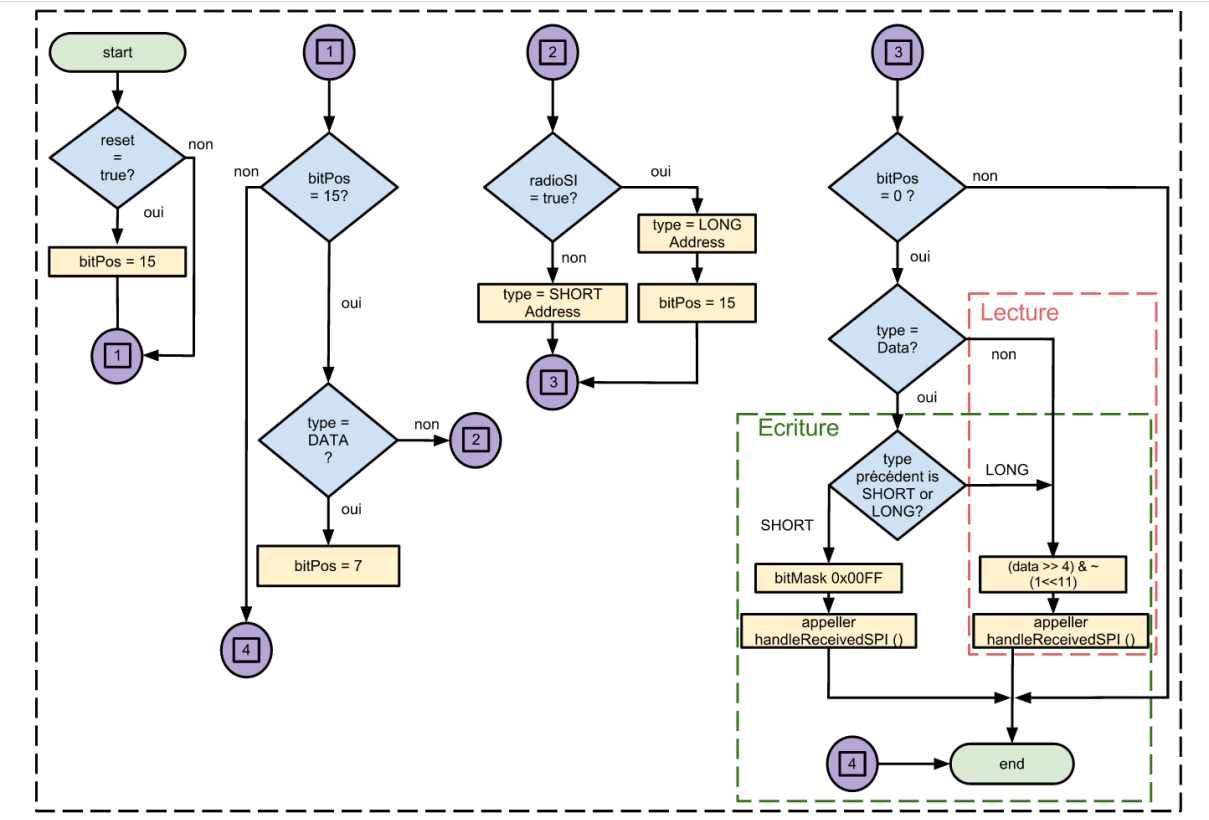


**FIGURE 3.11:** Le modèle d'interface SPI du module MRF24J40.

Pour la transmission “slotted” (dans la période CFP de la trame 802.15.4), un événement *e_rx* est émis pour chercher un paquet de Beacon. Une fois reçu, le fil attend que le slot alloué soit prêt pour pouvoir commencer la transmission. Il est informé de cela par le fil *th_scan4BeaconWaitSlot* qui lui envoie l’événement *e_slotReady*. Ensuite, la transmission peut commencer, en utilisant le Pilote d’accès au canal. La Batterie est informée de changement vers l’état “T”. Après un délai variable selon les données envoyées, le module batterie est informée de nouveau de transition vers l’état inactif (“I”). Si un slot ne peut pas être alloué pour le noeud, la transmission “unslotted” est choisie. Le noeud va faire une écoute de canal, deux fois en total, pour 8 périodes de symbole à la fois (CSMA-CA), pour se rassurer que le canal soit libre. Si c’est bien le cas, la transmission peut commencer. Sinon, un compteur est incrémenté et un nouveau essai est fait. Cet algorithme se répète jusqu’à ce que le compteur atteigne un certain nombre ou le canal est libre. Le fil se bloque pour une durée de 12 symboles (SIFS) si le paquet avait la longueur de jusqu’à 18 octets ou 40 symboles (LIFS) si plus.

Le fonctionnement du Pilote d’accès au canal, ainsi que du timer de Backoff sont déjà décrits pour CC2420 (dans son diagramme associé) et leur implémentation est similaire. Ils ne seront pas traités dans cette section. En revanche, le modèle pour MRF24J40 comprend le timer de Beacon que CC2420 n’a pas.

#### 3.4.4.2 Réception

La réception affecte seulement la RX FIFO. Elle est déclenchée par un des événements *e_rx* ou *e_readCarrier* (pour CSMA-CA). Le modèle informe le module batterie qu’il y avait un changement d’état vers réception, en lui envoyant le caractère “R”. Le canal est évalué pendant 8 périodes de symbole et s’il y a des données, cela veut dire qu’un voisin émet. Les données du canal sont copiées dans un paquet.

Si le noeud n’est pas un Coordinateur, le champ *frameControl* de la métadonnée du paquet est évalué, pour savoir s’il s’agit d’un paquet de Beacon ou non.

Si oui, le payload du paquet est interprété et les registres afférents sont écrits. Ces registres sont ceux qui disent si le Coordinateur a alloué au noeud un slot, le dernier slot dans la période “CAP”, ainsi que le dernier sous-slot pour chaque slot de GTS dans la période CFP. Ensuite, le Minuteur de Beacon est démarré pour effectuer un délai équivalent à la durée stipulée dans le paquet de beacon, afin que la Transmission des valeurs au Coordinateur puisse commencer.

La Réception s’achève en informant le module batterie du mis en mode inactive (état “I”). Bien évidemment, cet état consomme la même chose que la réception, mais on a décidé le représenter pour garder l’homogénéité avec l’implémentation de CC2420 ainsi qu’avec les prochains émetteurs-récepteurs radio qui seront supportés par IDEA1 TLM.

#### 3.4.4.3 Minuteur de Beacon

Pendant la routine d’initialisation de l’émetteur-récepteur radiofréquence, le microntrôleur écrit un registre (*RXMCR*) pour l’informer si le noeud est un Coordinateur ou pas, conforme au standard 802.15.4.

Si oui, ce minuteur sera utilisé pour calculer le moment où le prochain paquet Beacon sera envoyé. Sinon, il sera employé pour effectuer un délai jusqu’au moment où le slot de temps alloué par le Coordinateur commence.

Une fois configuré, l’émetteur-récepteur radiofréquence n’a plus besoin d’intervention du microntrôleur en ce qui concerne l’envoi. Ce dernier lui envoie juste des commandes de sommeil ou de mise en veille et les données, toute la gestion de la communication, les calculs associés au slots d’envoi étant faits automatiquement. Il n’y a pas du support offert pour le mode Auto-ACK pour l’instant.

#### 3.4.4.4 Mode Debug

Puisqu’il y a une plus grande partie de calcul du slot approprié qui est faite sur le MRF24J40 au lieu de microntrôleur, ce module dispose d’un mode Debug. Celui-ci est composé par

quatre macro-définitions, *db_spi*, *db_showRegAccess*, et *db_handleSpiAdditionalInfo* qui se trouvent dans le fichier “globalconf.h”

**Db_spi** montre toute la communication SPI faite entre le microntrôleur et le MRF24J40.

**Db_handleSpiAdditionalInfo** montre l’adresse où la lecture ou lecture est faite, en notation hexadécimale.

**Db_showRegAccess** montre le nom du registre qui est accédé pour une lecture / écriture.

### 3.4.5 Modèle Module de consommation

Comme présenté dans la Figure 3.12, l’architecture du module de consommation est générique et greffée avec des spécifications de consommation pour chaque architecture du microntrôleur ou émetteur-récepteur radiofréquence visé.

Lorsque l’objet “batterie” est créé, on lui spécifie le microntrôleur et le module radio. Comme ça, le module batterie connait la consommation instantanée de chaque état. Les parties consommatrices du noeud n’ont qu’à informer le module batterie d’un changement d’état. Ensuite, il y a un répartiteur qui désactive le thread d’état courant. Le thread va calculer la durée de temps qu’il a passé dans cet état et sachant la consommation instantanée de courant, ainsi que la tension d’entrée, il peut calculer l’énergie utilisée. Le répartiteur de commandes dans le modèle envoie un signal d’activation à un nouveau thread, correspondant à un nouveau état. Voilà donc le principe de fonctionnement du module batterie.

Il faut préciser qu’il existe en effet deux répartiteurs, un pour le microntrôleur (*th_dispatcherMcu*) et un pour la radio (*th_dispatcherRadio*).

Les différentes commandes, ou états que le microntrôleur ou la radio informent le module batterie sont déjà décrits dans les dernières sections, ainsi que sur les diagrammes d’architecture pour ces parties.

Pour récapituler, les états pour le microntrôleur :

– **’S’** : Sleep (sommeil)

– **’D’** : Conversion analogique-numérique.

– **'A'** : Active (en train de faire de calculs ou attendant des signaux de la radio)
– **'C'** : Communication (le microntrôleur est actif et communique par SPI avec la radio)

Les états pour la radio :

– **'R'** : Réception
– **'B'** : Backoff, équivalent à la réception du point de vue énergétique
– **'T'** : Transmission
– **'I'** : Inactive, attend les prochaines commandes du microntrôleur. Pour le MRF24J40, celui-ci correspond à l'état de réception.
– **'0'** : Power-off

### 3.4.6 Sortie de la Simulation

Durant la simulation, le stream *stdout* est écrit avec certaines informations concernant les valeurs de capteurs, la communication entre les noeuds, etc. Une fois la simulation achevée, on écrit la consommation énergétique de chaque noeud et du coordinateur est envoyée vers la sortie standard (stdout). Ensuite, deux autres streams sont ouverts vers deux fichiers : *nodeLogs/all.txt* et *nodeLogs/all_coo.txt*. Les valeurs de la consommation des noeuds et coordinateur sont ajoutées au contenu du fichier, séparées par des virgules. Ces valeurs comprennent le nom du noeud, la consommation SPI, Tx, Rx, ainsi que les dépenses énergétiques en mode idle et sleep pour le microntrôleur et la radio.

### 3.4.7 Résultats

On a pris un exemple avec 1 Coordinateur et 8 noeuds qui font une capture de température chaque seconde. Les noeuds vont transmettre la valeur de la température au coordinateur en utilisant le protocole IEEE 802.15.4 slotted GTS.

Au tout départ les noeuds font un backoff initial aléatoire. Ensuite, ils envoient la demande d'envoi initiale et après sont en état de réception et attendent le premier paquet Beacon. Une

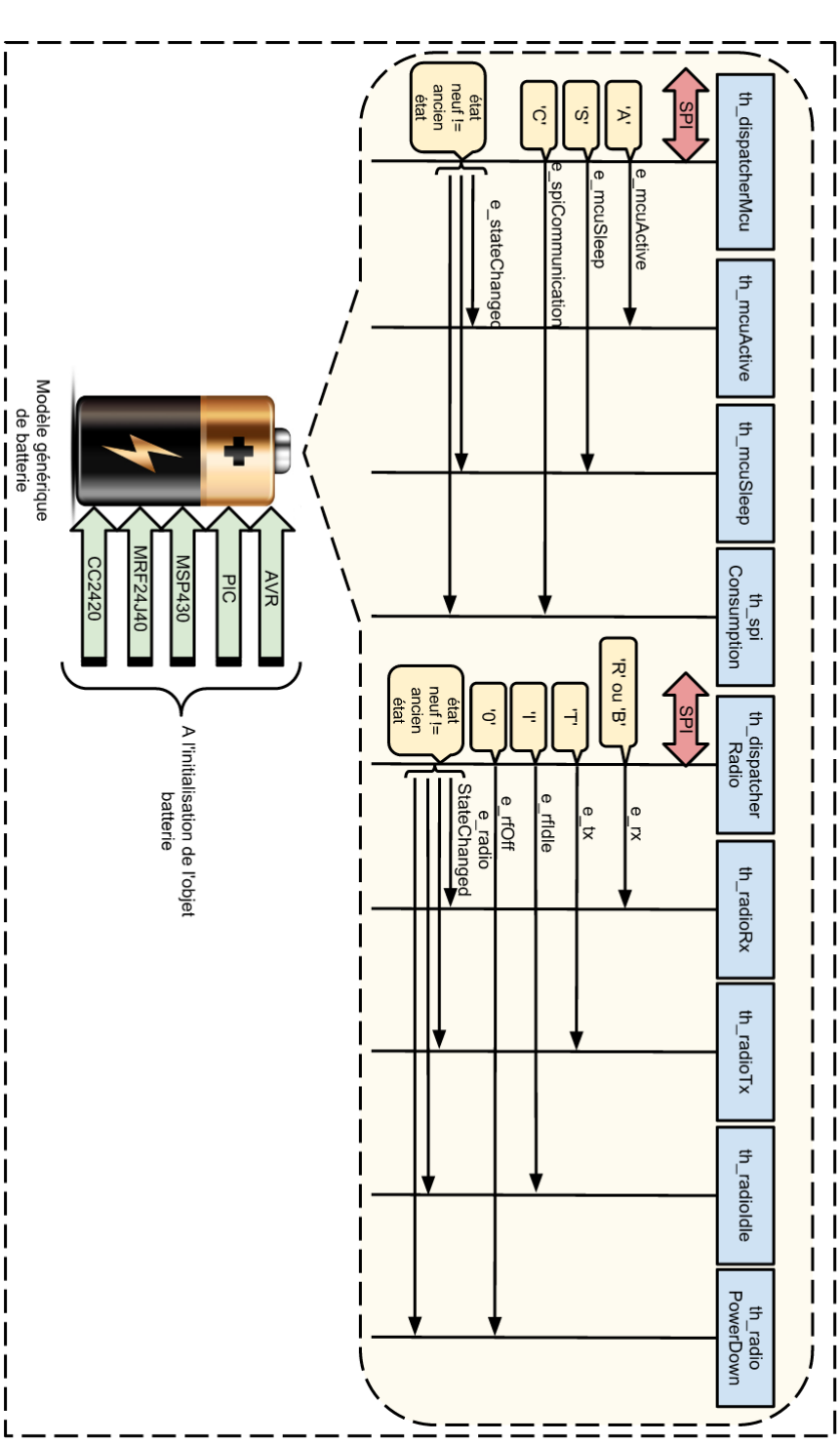


**FIGURE 3.12:** Le module Batterie et son architecture interne.

fois reçu, les slots de temps allouées par le Coordinateur sont extraites et chaque noeud va envoyer ses données au moment préétabli. Les Figures 3.13, 3.14 et 3.15 montrent la consommation énergétique moyenne pour chaque microcontrôlleur, émetteurs-récepteurs radiofréquence et globalement, au niveau de chaque noeud du réseau. Pour produire ces résultats, on a répété la simulation 20 fois. La consommation globale est plus faible dans le cas des noeuds avec un microcontrôleur PIC et un émetteur-récepteur radio MRF24J40 que dans celui des noeuds avec AVR et CC2420. Pour un temps de simulation donné de 10 secondes, la consommation de le premier groupe s'élève à 0.9 mJ, alors que pour le deuxième elle dépasse légèrement 1 mJ.

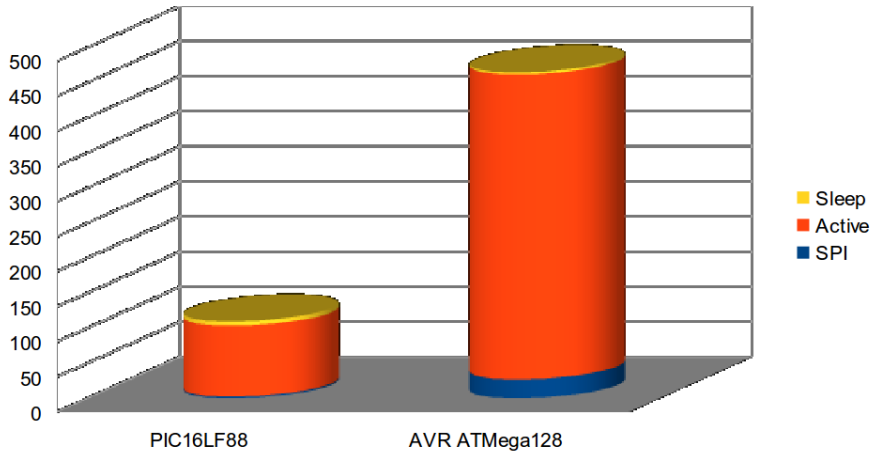


**FIGURE 3.13:** Énergie moyenne utilisée (uJ) par les microcontrôleurs.

Dans la prochaine section on va présenter BLISS, un simulateur de set d'instructions pour affiner le modèle du noeud.

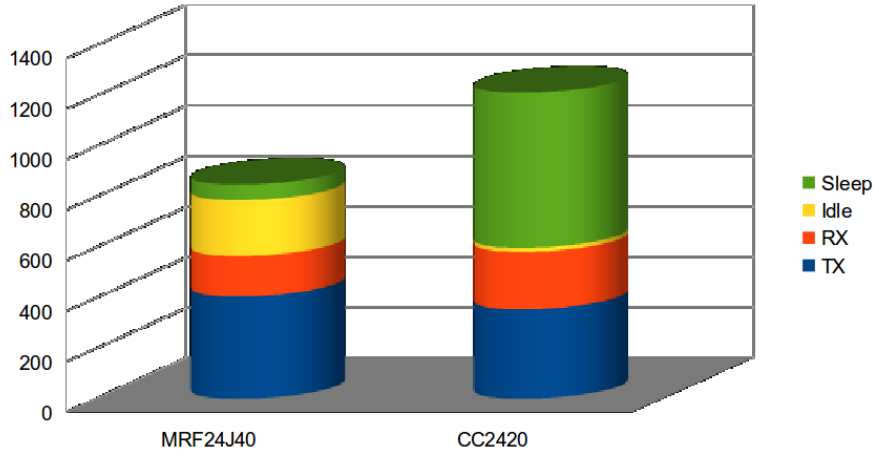


**FIGURE 3.14:** Énergie moyenne (uJ) utilisée par les émetteurs-récepteurs radio.

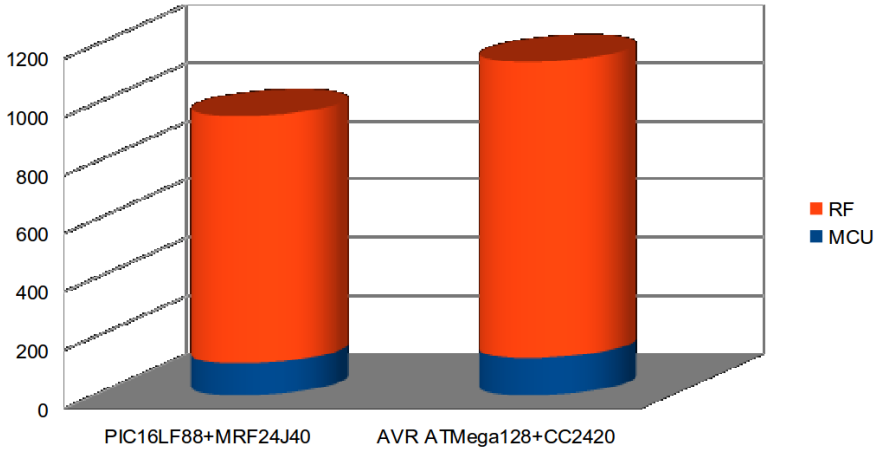


**FIGURE 3.15:** Énergie moyenne totale (uJ) utilisée par les noeuds.

# 3.5 BLISS

## 3.5.1 Aperçu général

Bien évidemment, les ISS ne sont pas nouveaux dans l'état de l'art et d'autres exemples incluent Avrora (56) ou MspSim (57), mais souffrent de deux inconvénients majeurs. Le premier c'est qu'ils ciblent seulement une architecture matérielle et le deuxième c'est qu'ils ne sont pas tous écrits dans le même langage de programmation. On a développé BLISS en C++ pour faciliter l'interface avec IDEA1. Il offre du support pour AVR et MSP430.

BLISS peut simuler les opérations (instructions) qu'un microcontrôleur exécute et crée un profil qui consiste d'une paire fonction-nombre de cycles d'horloge, pour chaque fonction présente dans l'implémentation du logiciel. Ensuite, sachant la fréquence du fonctionnement du microntrôleur, son état et le courent associé à chacun, on peut faire une estimation fine du besoin énergétique de chaque fonction. C'est ainsi que le modèle RTL est pris en compte. Ceci correspond à la première étape de l'approche en deux pas mentionnée auparavant.

La deuxième étape consiste d'utiliser ces métadonnées produites par BLISS dans IDEA1, par moyen d'un fichier ".h".

BLISS lit la fonctionnalité logicielle en question à partir de la sortie d'un compilateur avec lequel la fonctionnalité a été compilée. Intrinsèquement, cela veut dire que BLISS dépend du compilateur.

Le format du fichier ELF a été initialement développé et publié par "Unix System Laboratories", comme partie de l'interface binaire d'application (10). Le comité "Tool Interface Standards" a choisi ELF comme format d'objet portable qui pourrait fonctionner sur des architectures Intel 32-bits et sur une diversité de systèmes d'exploitation. La norme ELF est destinée à simplifier le développement de logiciels.

Comme on peut voir dans la figure 3.16, au tout début du fichier ELF on a “l’en-tête ELF” qui a le rôle d’une carte, pour décrire l’organisation du fichier.

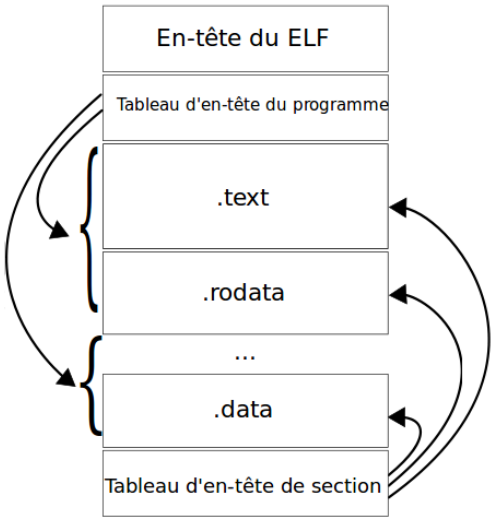


**FIGURE 3.16:** Format du fichier ELF. (9)

L’entête du programme, si présent, informe le système comment créer une image d’un processus. Les fichiers utilisés pour construire une image d’un processus (exécuter un programme), doivent avoir une entête de programme. Les fichiers délocalisables n’en ont pas besoin.

La table d’entête de section contient des informations concernant les différentes sections du fichier (dont les plus importantes sont .bss, .data, .init, .symtab et .text ). Chaque section a une entrée dans cette table, et chaque entrée offre des informations décrivant le nom de la section, sa taille, etc. Plus d’informations sont données dans le Tableau 3.3 et 3.4.

Un aperçu général sur les informations contenues dans les différentes en-têtes est présenté dans les Figures 3.17 et 3.18.

```
#define EI_NIDENT       16

typedef struct {                                typedef struct {
        unsigned char   e_ident[EI_NIDENT];             Elf32_Word      sh_name;
        Elf32_Half      e_type;                         Elf32_Word      sh_type;
        Elf32_Half      e_machine;                      Elf32_Word      sh_flags;
        Elf32_Word      e_version;                      Elf32_Addr      sh_addr;
        Elf32_Addr      e_entry;                        Elf32_Off       sh_offset;
        Elf32_Off       e_phoff;                        Elf32_Word      sh_size;
        Elf32_Off       e_shoff;                        Elf32_Word      sh_link;
        Elf32_Word      e_flags;                        Elf32_Word      sh_info;
        Elf32_Half      e_ehsize;                       Elf32_Word      sh_addralign;
        Elf32_Half      e_phentsize;                    Elf32_Word      sh_entsize;
        Elf32_Half      e_phnum;                } Elf32_Shdr;
        Elf32_Half      e_shentsize;
        Elf32_Half      e_shnum;
        Elf32_Half      e_shstrndx;
} Elf32_Ehdr;
```

**FIGURE 3.17:** L'entête du fichier ELF et l'entête d'une section. (10)

```
typedef struct {
        Elf32_Word      st_name;
        Elf32_Addr      st_value;
        Elf32_Word      st_size;
        unsigned char   st_info;
        unsigned char   st_other;
        Elf32_Half      st_shndx;
} Elf32_Sym;
```

**FIGURE 3.18:** Les entrées dans la table de symboles du fichier ELF.

| | |
|---|---|
| .bss | Cette section contient des informations qui contribuent à l'image de la mémoire du programme. C'est ici que les variables allouées d'une manière statique (au moment de la compilation), sans valeur d'initialisation sont allouées. Le système l'initialise les données présentes ici avec des zéros lors du démarrage du programme. La section .bss n'occupe pas des espaces dans le fichier ELF, il y a qu'une valeur de sa longueur qu'y est écrite. |
| .comment | Des informations relatives à la version sont présentes ici. |
| .data | Ici, les variables initialisées sont placées, avec leur valeur d'initialisation. |
| .debug | Cette section détient des informations pour faire du debug symbolique. Le contenu n'est pas spécifié. |
| .dynamic | Il y a de l'information relative au liens (linkage) du programme. |
| .dynstr | Cette section comprend les différents tableaux de caractères requis pour le linkage dynamique, plus souvent des noms associés aux entrées de la table de symboles. |
| .dynsym | Cette section contient la table de symboles du linkage dynamique. |
| .fini | Ceci contient des instructions exécutables qui contribuent au code de fin du processus. C'est-à-dire, elles seront exécutées lors de la fin du programme. |
| .got | La table des décalages globaux, utilisée pour avoir des adresses absolues dans le cas du code indépendant de position ("Position Independent Code") |
| .hash | Contient une table de hachage des symboles. Il y a une clé unique donnée à chaque symbole, calculé avec un algorithme. Les symboles pourraient après être référencés avec cette clé. |
| .init | Cette section contient des instructions exécutables, qui contribuent au code d'initialisation du processus. C'est-à-dire, lorsqu'un programme commence, le système exécute ce code avant de commencer le code dans la routine principale. |

**TABLE 3.3:** Les sections du fichier ELF.

| | |
|---|---|
| .interp | Cette section contient le chemin de l'interpréteur du programme. |
| .line | Des informations relatives au numéro de ligne dans le fichier source, pour faire du debug symbolique. |
| .note | Diverses informations relatives au producteur de logiciel. Destiné à être vérifiées par d'autres logiciels pour la compatibilité. |
| .rel | Cette section détient des informations de relocalisation, si le code généré n'est pas indépendant de position. |
| .rodata | Ces section contient des informations qui contribuent à un segment de lecture seule dans l'image du processus. |
| .shstrtab | Les noms des différentes sections. |
| .strtab | Les noms associés aux entrées dans le tableaux de symboles. |
| .symtab | Ceci contient un tableau de symboles, selon le format présenté en Figure 3.18. |
| .text | Cette section détient les instructions en code machine, directement exécutables par l'architecture ciblée. |

**TABLE 3.4:** Les sections du fichier ELF (continuation).

Des différentes informations qui constituent le fichier ELF, il n'y a qu'une partie qui est importante pour BLISS. Notamment les sections .bss, .data, .init, .symtab et .text. En ce qui concerne les autres sections, il n'y a que leur longueur qui est importante, pour bien savoir d'où commencer la lecture.

Parmi les caractéristiques notables de BLISS, il y a la possibilité de simuler les interruptions du timer, la simulation des interruptions matérielles ainsi que les points d'arrêt lorsqu'une fonction bien précise est atteinte.

Pour pouvoir commencer, tout d'abord, il y a une lecture à partir du fichier d'entrée ELF vers un tableau qui émule la flash, dans la RAM du PC.

C'est dans cette mémoire flash que le code d'instructions sera mis et c'est de là que le programme et lu.

Pour les deux implémentations, BLISS exécute chaque instruction comme le microntrôleur l'aurait fait, en utilisant des différents tableaux de données qui correspondent. Ensuite, il met à jour les drapeaux (flags) internes et il évalue les timers pour le débordement. S'il y a une interruption, des instructions PUSH mettent l'adresse courante sur la pile (pour pouvoir se souvenir où continuer à exécuter le code une fois l'interruption achevée) et la valeur du registre de statut. À ce point, les deux architectures désactivent le bit qui gère le traitement d'interruption. Ceci est une mesure qui permet éviter les interruptions imbriquées (c'est-à-dire les interruptions qui puissent surgir durant le traitement d'une routine d'interruption précédente). Si l'application demande l'usage d'interruptions imbriquées, le développeur pourrait re-activer ce bit immédiatement dans le début de la routine d'interruption.

Ensuite, le registre PC ( "Program Counter") est mis à jour avec l'adresse de l'interruption qui est déclenchée. Une fois la routine d'interruption finie, il y a un nombre d'instructions POP (équivalent au nombre de PUSH initial) qui est fait pour restaurer la valeur des registre PC et statut. Le code continue à s'exécuter où il était avant que l'interruption soit apparue. S'il y a besoin que le code dans l'interruption communique avec une autre partie de l'application, ceci se fait au travers de variables globales.

La simulation s'achève lors d'une instruction de saut à la même adresse (boucle infinie), générée par le compilateur.

La sortie de BLISS est écrite sous forme de fichier C++ "header", et consiste en une paire nom de fonction - nombre de cycles requis.

Le nombre de cycles dont une fonctionnalité a besoin est lié non seulement à la quantité d'instructions mais aussi leur longueur, et les imbrications éventuelles (la complexité de l'algorithme). Si l'application simulée contient des boucles, il y a forcément un saut plus haut dans le code pour le répéter. Hormis le cas où la boucle est infinie, cette instruction de saut est conditionnée par une expression logique. Selon la valeur de cette condition, le saut est fait ou pas. Le nombre d'instructions simulé par BLISS dépend donc aussi de conditions qui jouent sur les sauts conditionnels dans les boucles.

En ce qui concerne la simulation avec BLISS lui-même, l'approche est différente pour AVR et MSP430, car la première architecture est Harvard et la deuxième est Von Neumann.

On précise que les durées d'instructions sont prises à partir de spécifications du producteur et elles ne tiennent pas compte de pipeline du processeur, qui est à 3 niveaux sur MSP430 et 2 niveaux pour l'AVR. Celle-ci est responsable de décoder la prochaine instruction avant que celle qui est en train de s'exécuter n'ait pas fini.

Les durées réelles sont susceptibles d'être plus courtes, et donc la simulation BLISS donnent des résultats "pire cas". Le méchanisme pipeline n'est pas pris en compte parce que le producteur ne donne aucune indictation sur l'implémetation de celui-ci.

### 3.5.2 BLISS AVR

Pour l'AVR, à cause du fait que la mémoire flash et la RAM ont des espaces adresses différents, les variables sont chargées d'abord dans des registres internes et les opérations sont faites sur les registres. Pour cette variante, BLISS lit la sortie d'avr-gcc. L'information de debug doit être incluse dans la sortie (en utilisant les flags de compilation "-gdwarf-2").

| Action | Nombre de cycles | Longueur |
|---|---|---|
| Retour d'une Interruption (RETI) | 5 | 1 |
| Interruption acceptée | 6 | - |
| Reset de WDT | 4 | - |
| Reset (RST / NMI) | 4 | - |

**TABLE 3.5:** Durée des instructions d'Interruption et Reset, MSP430

### 3.5.3 BLISS MSP430

En ce qui concerne le MSP430, la sortie d'IAR Workbench (v5.10.1) est lue. Le code doit être compilé avec l'option "-yan" pour le linker et le format de sortie mis à "elf-dwarf". Le fichier ELF produit est donc livré à BLISS et celui-là commence à simuler le logiciel.

Les régions de mémoire du MSP430 sont partagées. C'est-à-dire que la RAM et la flash sont contiguës. Dans ce cas, les opérations peuvent être faites entre un registre et une région de mémoire ou entre deux régions de mémoire même.

En ce qui concerne l'architecture MSP430, il y a trois types, ou groupes d'instructions. D'après la terminologie employée par Texas Instruments, les groupes s'appellent Format I, Format II et Format III. Le type Format III sont les instructions de saut, qui prennent tous 2 cycles d'horloge, que le saut soit exécuté ou non (pour le saut conditionnels, ainsi que pour le sauts inconditionnels).

Les durées pour les deux premiers types sont présentées dans les Tableaux 3.6 et 3.7. Dans ces tableaux, il y a plusieurs modes d'adressage possibles : registre direct (Rn), registre indirecte (@Rn), indirecte avec post-incrément (@Rn+), immédiat (#N où #N est un label), constant #N (où #N est une constante entière), ou absolu (&Addr).

| Mode d'adressage | RRA, RRC,SWPB, SXT | PUSH | PUSHX | CALL | CALLX | Longueur | Exemple |
|---|---|---|---|---|---|---|---|
| Rn | 1 | 3 | 2 | 4 | 3 | 1 | SWPB R5 |
| @Rn | 3 | 4 | 3 | 4 | 4 | 1 | RRC @R9 |
| @Rn+ | 3 | 5 | 3 | 5 | 4 | 1 | SWPB @R10+ |
| #N | - | 4 | 3 | 5 | 4 | 2 | CALL #0F00h |
| X(Rn) | 4 | 5 | 4 | 5 | 4 | 2 | CALL 2(R7) |

**TABLE 3.6:** Durée des instructions simulées par BLISS, à un seul opérande (Format II) pour MSP430 et MSP430X ("extented").

| Mode d'adressage source | Mode destination | Cycles | Cycles X | Longueur | Exemple |
|---|---|---|---|---|---|
| Rn | Rm | 1 | 1 | 1 | MOV R5,R8 |
| | PC | 2 | 2 | 1 | BR R9 |
| | x(Rm) | 4 | 3 | 2 | ADD R5,4(R6) |
| | label | 4 | 3 | 2 | XOR R8,label |
| | &label | 4 | 3 | 2 | MOV R5,label |
| @Rn | Rm | 2 | 2 | 1 | AND @R4,R5 |
| | PC | 2 | 3 | 1 | @BR R8 |
| | x(Rm) | 5 | 4 | 2 | XOR @R5,8(R6) |
| | label | 5 | 4 | 2 | MOV @R5,label |
| | &label | 5 | 4 | 2 | XOR @R5,label |
| @Rn+ | Rm | 2 | 2 | 1 | AND @R5+,R6 |
| | PC | 3 | 3 | 1 | @BR R9+ |
| | x(Rm) | 5 | 5 | 2 | XOR @R5,8(R6) |
| | label | 5 | 5 | 2 | MOV @R5+,label |
| | &label | 5 | 5 | 2 | XOR @R9+,&label |
| #N | Rm | 2 | 2 | 2 | MOV #20,R9 |
| | PC | 3 | 3 | 2 | @BR #2AEh |
| | x(Rm) | 5 | 4 | 3 | MOV #0300h,0(SP) |
| | label | 5 | 5 | 3 | ADD #33,label |
| | &label | 5 | 4 | 3 | ADD #33,&label |
| label1 | Rm | 3 | 3 | 2 | AND label1,R6 |
| | PC | 3 | 3 | 2 | BR label1 |
| | label2 | 6 | 6 | 3 | CMP label1,label2 |
| | x(Rm) | 6 | 6 | 3 | MOV label1,0(SP) |
| | &label2 | 6 | 6 | 3 | MOV label1,&label2 |
| &label1 | Rm | 3 | 3 | 2 | AND &label1,R8 |
| | PC | 3 | 3 | 2 | BR &label1 |
| | label2 | 6 | 6 | 3 | MOV &label1,label2 |
| | x(Rm) | 6 | 6 | 3 | MOV &label1,0(SP) |
| | &label2 | 6 | 6 | 3 | MOV &label1,&label2 |

**TABLE 3.7:** La durée des instructions simulées par BLISS, à double opérande (Format I) pour l'architecture MSP430 et MSP430X ("extented").

Grâce au fait que MSP430 est une architecture Von Neumann, les instructions peuvent prendre des opérants qui sont soit des registres soit des régions de mémoire. Une architecture Harvard aurait besoin d'une instruction particulière pour accéder à la mémoire pour la lecture et une autre pour l'écriture. C'est pour ainsi que les instructions pour MSP430 peuvent être résumées facilement dans les tableaux précédentes, alors que celles pour AVR sont présentées dans l'annexe.

## 3.6 Résultats pour BLISS

Pour l'évaluation correcte des cycles simulés, on a comparé les résultats de BLISS pour un certain nombre d'algorithmes connus, avec les mêmes implémentations simulées en utilisant AVRStudio et IAR Workbench. Puisque pour chaque algorithme les implémentations réelles peuvent varier, on a choisi les tests de performance WCET ("Worst Case Execution Time" en anglais) de C-Lab (58) comme point de départ.

Les résultats sont présentés dans la Table 3.8.

| Algorithme | BLISS (AVR) | AvrStudio | BLISS(MSP430) | Iar Workbench |
|---|---|---|---|---|
| Bubble Sort(500) | 3,538,519 | 3,538,519 | 3,297,421 | 3,297,421 |
| InsertSort(10) | 1,449 | 1,449 | 1,221 | 1,221 |
| Fibonacci | 319 | 330 | 366 | 366 |
| Matrix Multiply([2][2]) | 2,724 | 2,724 | 2,219 | 2,219 |
| Matrix Multiply(20) | 526,130 | 526,130 | 378,164 | 378,164 |

**TABLE 3.8:** Résulatts de la simulation BLISS

Dans un premier temps, on a déployé l'algorithme de tri à bulles *BubbleSort* sur BLISS. Celui-ci fait monter les plus grands éléments, d'une manière progressive tout comme les bulles d'air montent dans un liquide. On a utilisé 500 éléments à trier et les résultats sont normalisés sur trois plages de valeurs, dans les intervalles [-30000, 30000], [-30000,0] et [0,30000]. Les

nombres utilisés pour le tri ont été générés de façon aléatoire, utilisant un programme écrit en C qui fait usage de la fonction de bibliothèque *rand()*.

Ensuite, en gardant le même processus concernant la génération de nombres aléatoire, on a testé l'algorithme de tri *InsertSort* sur BLISS. Dix éléments ont été triés, ayant besoin d'environ 1500 cycles (AVR) et environ 1200 (MSP430).

L'algorithme de Fibonacci et la multiplication de matrices (58) ont aussi été testés avec succès.

On précise que dans le cas d'algorithmes de tri (BubbleSort, InsertSort), le temps pour trier les vecteurs dépend effectivement de valeurs triées.

```
#ifndef __functionCycles_H_
#define __functionCycles_H_

#include <stdint.h>

typedef struct{
        char name[50];
        uint64_t cycles;
}TfunctionCycles;

TfunctionCycles SfunctionCycles[]=
{
        {main,60162}
        {a,61},
        {bf,30042},
        {sendMinTax,30029},
        {radio_init,758},
        {trx_io_init,44},
        {trx_reg_write,18},
        {trx_bit_write,32},
        {radio_set_param,83},
        {radio_set_state,45496},
        {trx_bit_read,49},
        {trx_reg_read,20},
        {buffer_init,20},
        {buffer_append_char,33},
        {radio_send_frame,97},
        {trx_frame_write,87}
};

#endif
```

**TABLE 3.9:** Sortie du BLISS pour le programme présenté dans la Table 2.9, AVR, -Os.

Dans un deuxième temps, on a simulé l'algorithme présenté dans la Table 2.9 avec BLISS, pour pouvoir avoir des résultats concernant une application de WSN. Normalement, lors d'un

envoi de données par radio, il y a un dialogue entre le microntrôleur et le transceiver radio, fait par SPI. Le microntrôleur sait qu'il peut poursuivre avec des nouvelles commandes en lisant un drapeau d'activité de cette interface. Or, ce drapeau est géré par le matériel, ce qui veut dire qu'une simulation BLISS se bloquerait dans une boucle infinie, attendant la fin du transfert SPI. C'est pourquoi, au lieu de faire cette lecture, on a tout simplement fait une boucle de délai d'une microseconde.

L'application fait la lecture d'un port, l'envoie par radio et l'écrit dans un autre port. Une fonction de délai est implémente dans la fonction "aa". Celle-ci est optimisé par le compilateur et imbriquée dans le corps de la fonction "a". C'est pour ainsi qu'elle n'est pas présente dans le tableau de correspondance sorti par BLISS.

L'application a été modifiée pour se répéter qu'une seule fois, au lieu d'une boucle infinie et dure à peu prêt 60000 cycles d'horloge.

## 3.7 Résultats BLISS+IDEA1TLM

### 3.7.1 Exemple 1

Pour qu'on puisse voir des données concernant la consommation énergétique d'un noeud, on a choisi de simuler la transmission d'un texte par radio, sous forme compressée Huffmann (59) et non-compressée. Pour le protocole de communication, on a utilisé 802.15.4, en mode non-slotted CSMA-CA.

Pour que la simulation ait une bonne approximation statistique, on a utilisé huit noeuds et un coordinateur et on l'a répétée 20 fois, avec 10 cycles compression-envoi ou envoi simple. Le but était de voir si ça vaut le coût de transmettre un text comme celui-là en forme compressée plutôt que normale. La question est bien justifiée, car la taille du paquet et ainsi réduite, donc

les noeuds passent moins de temps en mode transmission qui est chère du côté puissance. 1 octet transmis par l'interface radio consomme en moyenne, la même énergie que 8000 cycle d'horloge.

D'une façon inverse, on peut poser la question : le temps pour faire la compression/décompression tient le microcontrôleur en mode actif, donc est-ce pertinent pour compenser le gain de la transmission radio ?

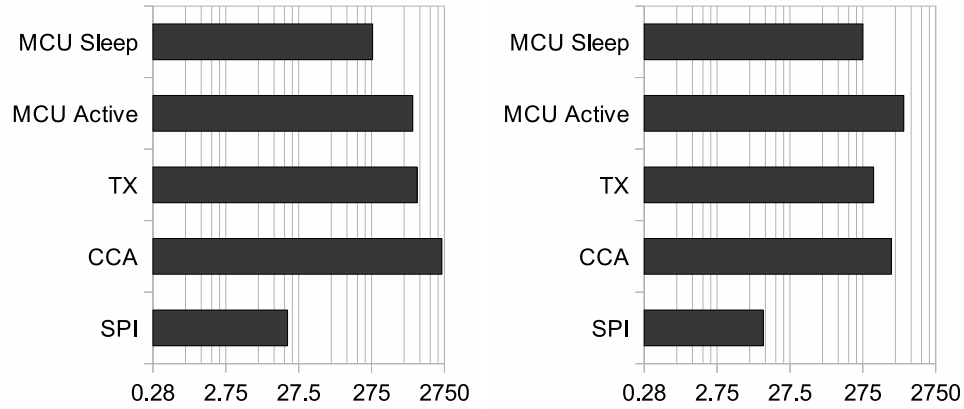


**FIGURE 3.19:** Énergie moyenne utilisée par le noeud (uJ) pour envoyer le texte en forme brute (gauche) et sous forme compressée (droite).

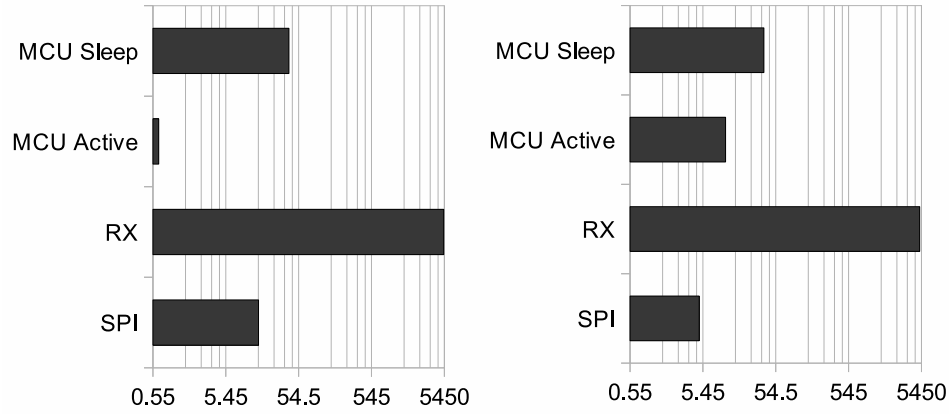


**FIGURE 3.20:** Énergie moyenne utilisée par le Coordinateur (uJ) pour recevoir la forme brute (gauche) et compressée (droite) du texte.

Les résultats sont présentés dans le Tableau 3.10. En utilisant les données issues de BLISS, on sait qu'il nous faut le nombre de cycles d'horloge (pour la compression/décompression). Ensuite, sachant la fréquence de fonctionnement du microntrôleur (8Mhz), on peut calculer le temps réel dont le microntrôleur a besoin pour achever ces opérations. Ceci est équivalent à 50.98 ms pour la compression et 29.14 ms pour la décompression. Ces opérations sont implémentées comme des instructions de *wait(temps)* en SystemC. C'est donc ainsi que les données fournies par BLISS sont incorporées en IDEA1TLM.

| Paramètre | Quantité | Unité |
|---|---|---|
| Compression texte | 407919 | cycles d'horloge |
| Temps de compression (8Mhz) | 50.98 | ms |
| Décompression texte | 233195 | cycles d'horloge |
| Temps de décompression (8Mhz) | 29.14 | ms |
| Taille non-compressée | 769 | bytes |
| Taille compressée | 463 | bytes |
| Moyenne totale énergétique par paquet, compressé | 2719.8 (Noeud) 5214 (Coordinateur) | uJ |
| Moyenne totale énergétique par paquet, non-compressé | 5015.2 (Noeud) 5403.2 (Coordinateur) | uJ |

**TABLE 3.10:** Paramètres pour envoyer et recevoir le texte

Les Figures 3.19 et 3.20 montrent la moyenne de la distribution d'énergie pour l'envoi d'un paquet, côté noeud et coordinateur. En moyenne, le noeud a besoin de 2719 uJ pour l'envoi compressé. Le coordinateur reçoit et décompresse l'abstract en utilisant 5214 uJ. On peut bien voir comme l'algorithme Huffman augmente la consommation du noeud dans l'état actif.

Les résultats montrent donc, que l'envoi sous forme compressée consomme moins d'énergie. On voit bien qu'en effet c'est la CCA (anglais "Clear Channel Assessment", l'évaluation pour voir si le canal radio est libre) qui joue un rôle important dans la quantité totale d'énergie requise pour l'envoi. Ceci est dû au fait que dans le cas non-compressé, le temps d'envoi effectif est plus élevé, donc il existe plus de chance que le canal soit occupé et le noeud doit attendre son tour pour qu'il puisse commencer la transmission.

La simulation montre que la consommation énergétique moyenne pour envoyer un paquet compressé au lieu de l'envoyer en forme brute est 1.84 fois moins élevée.

### 3.7.2 Exemple 2

Pour bien démontrer un algorithme de WSN, on a développé un exemple qui fait la lecture analogique d'une patte, l'envoie par radio si la valeur est supérieure à un seuil, pour enfin se remettre en mode sommeil *sleep*. Ceci est répété chaque seconde.

Pour pouvoir quantifier le nombre de cycles utilisés par chaque fonction avec BLISS, on a dû faire quelques modifications concernant le dialogue avec l'interface radio. Comme au moment d'envoi par SPI, le microntrôleur évalue un drapeau de fin de transfert, ceci bloquerait BLISS à cet endroit puisque ce drapeau est changé par le matériel. C'est pourquoi au lieu de l'évaluer, on a fait un délai de quelques microsecondes. Ensuite, au lieu de trourner en boucle, l'algorithme est exécuté et évalué qu'une seule fois. On considère qu'il sera entièrement exécuté de nouveau avec la prochaine fois.

Le noeud arme deux interruptions, une interruption de timer et une sur la conversion analogue-numérique. Dès que l'interruption de timer arrive, elle déclenche la conversion. Une fois finie, la routine de communication radio est appelée. Lorsque tout est fini, le noeud se met en sleep et attend une nouvelle interruption de timer.

Les différents états du microntrôleur ainsi que leurs cycles associés sont présentés dans la Figure 3.21.

| Paramètre | Quantité | Unités |
|---|---|---|
| SPI | 27.0125 | uA |
| TX (Radio + MCU) | 354.816 | uA |
| RX (Radio only) | 786.73 | uA |
| Radio IDLE | 1.3 | uA |
| Radio Sleep | 30.79 | uA |
| MCU Active | 787.945 | uA |
| MCU Sleep | 3.06 | uA |

**TABLE 3.11:** Paramètres concernant la consommation énergétique du noeud pour l'exemple 2.

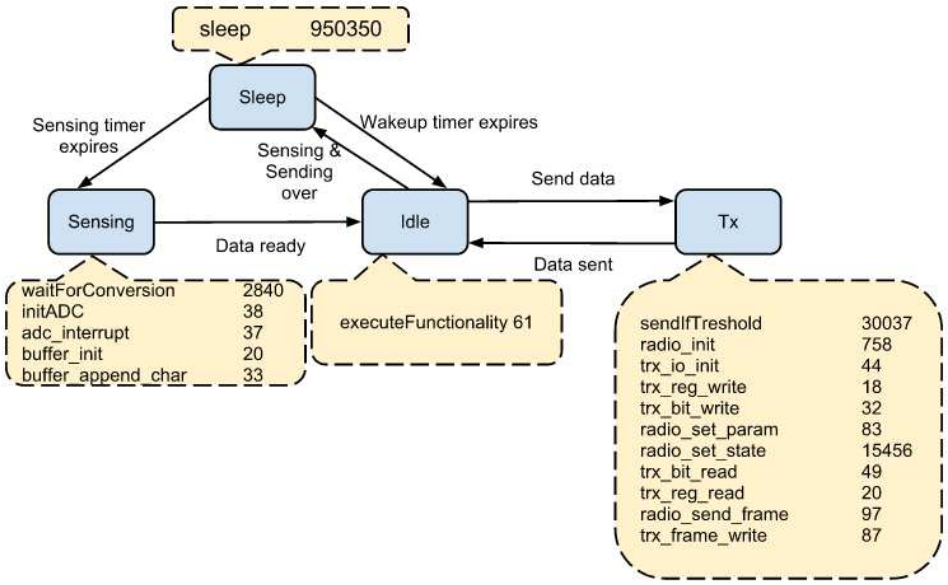


**FIGURE 3.21:** Les différents états du microntrôleur et le nombre de cycles associés.

Les paramètres issus de la simulation sont présentés dans la Table 3.11.

Le tableau présente les valeurs moyennes pour 8 noeuds et un coordinateur équipés d'un microntrôleur AVR et d'un transceiver radio MRF24J40. Elle a été répétée 3 fois. Parce que MRF24J40 se charge de la réception, il n'y a pas un état "RX" dans le microntrôleur.

## 3.8 Conclusion et perspectives

### 3.8.1 Sommaire du chapitre

Étant donné le temps relativement court pour développer la nouvelle version d'IDEA1 (deuxième partie de ma troisième année de doctorat), celle-ci n'a pas eu le temps d'arriver à sa maturité. Néanmoins, on a mis les fondations pour cette nouvelle version qui promet

une modélisation raffinée, encore plus fine que l'ancienne version.

Premièrement, ceci est possible à la modélisation du comportement matériel des différents composants présents sur le noeud. La communication faite entre le microcontrôleur et l'émetteur-récepteur radio est représentée au niveau de cycle de bus de communication SPI. Le module qui implémente le microcontrôleur envoie des commandes au module radio à travers un canal qui modélise cette communication SPI. Le module radio sait interpréter ces commandes et réagir en changeant son état interne, selon un modèle prédéfini de machine d'état fini.

La communication entre plusieurs interfaces radio est modélisée en utilisant la bibliothèque SCNSL. Celle-ci permet de tenir compte de phénomènes d'atténuation, interférence, etc. Plusieurs modèles de microcontrôlleurs (AVR, PIC et MSP430) et émetteurs-récepteurs radio (CC2420 et MRF24J40) ont été implémentés. Aussi, IDEA1TLM offre la possibilité de combiner le premier groupe avec le deuxième menant à des configurations de noeuds hétérogènes matériellement.

Deuxièmement, on a développé un simulateur de set d'instructions dénommé BLISS. Celui-ci permet d'avoir des profils des fonctions du logiciel qui s'exécute sur les noeuds des WSNs. BLISS peut générer des tables de correspondance où chaque fonction a le nombre de cycles d'horloge requis pour son exécution. Grâce au fait que les résultats de BLISS sont pris en compte par IDEA1TLM, on peut avoir différents logiciels qui s'exécutent sur les différents noeuds du réseau. Ceci veut dire que IDEA1TLM offre le support pour une hétérogénéité logicielle au niveau du réseau.

D'après notre connaissance, IDEA1TLM est la seule plate-forme de simulation pour des WSNs hétérogènes matériellement-logiciellement. En plus, elle est aussi la seule qui modélise le comportement des noeuds au niveau TLM.

### 3.8.2 Perspectives

IDEA1TLM est proche d'une version complète. La chose manquante la plus importante est le PDR (Packet Delivery Ratio), une mesure qui comprend le rapport entre les nombre totale de paquets envoyés et le nombre de paquet reçus.

## 3. BLISS ET IDEA1TLM

Pour le long terme, IDEA1TLM pourrait être étendue à d'autres architectures matérielles de microcontrôlleurs et émetteurs-récepteurs radio.

En ce qui concerne BLISS, ce simulateur pourrait être développé pour offrir le support pour d'autres architectures de microcontrôlleurs, notamment PIC. Pour l'instant, BLISS ne tient pas compte de concepts de bande d'assemblage présents dans l'implémentation matérielle du microcontrôlleur.

# 4 Conclusions

## 4.1 Conclusion sur la Reconfiguration Dynamique des WSNs

Un des objectifs de cette thèse porte premièrement sur la reconfiguration (la mise-à-jour) dynamique dans les Réseaux des Capteurs sans Fil comprenant plusieurs architectures matérielles (hétérogénéité matérielle) et systèmes d'exploitation (hétérogénéité logicielle). Ces réseaux sont constitués d'une multitude de systèmes électroniques communicants par radio fréquence, très contraints en énergie. Il est bien connu que la partie de communication radio entre ces noeuds est la plus consommatrice. C'est pourquoi la minimisation du temps effectif de ce transfert est souhaitable pour une consommation d'énergie plus faible. On a implémenté une solution qui consiste à envoyer au noeud un fichier de reconfiguration codé utilisant un langage de haut niveau (MinTax). Ensuite, le noeud sera capable de compiler lui-même ce fichier et générer le code object correspondant à la mise-à-jour pour son architecture, in-situ. Grâce au caractère abstrait du MinTax, ce langage ne comprend pas d'information concernant une architecture matérielle précise. C'est pourquoi le même fichier MinTax peut être utilisé pour configurer plusieurs architectures matérielles différentes. Sur le même raisonnement, plusieurs systèmes d'exploitation peuvent être utilisés avec notre solution.

Les gains obtenus par notre solution MinTax on été quantifiés en les comparant à une machine virtuelle dénommée VMSTAR, une solution qui promet des performances largement supérieures aux autres solutions de reconfiguration dynamique. Pour deux algorithmes testés,

MinTax offre un gain de 78% (Blink) et 34%(ObjectTracker) par rapport à VMSTAR.

Deuxièmement, pour pouvoir quantifier les gains apportés par MinTax dans un contexte de réseau globale, on a cherché à intégrer MinTax dans un simulateur de réseau. Le candidat idéal pour cela était IDEA1, développé au sein de notre équipe. Pour pouvoir intégrer MinTax en IDEA1, in nous a fallu d'abord intégrer l'aspect logiciel en ceci. On a développé une nouvelle version d'IDEA1, basée sur l' ancienne version. Celle-ci s'appelle IDEA1TLM et modélise le comportement du noeud de WSN au niveau très fin. L'aspect logiciel y est intégré à travers un simulateur du jeu d'instructions dénommé BLISS.

## 4.2 Conclusion sur la Simulation des WSNs

Dans un deuxième temps, le travail de thèse est lié à simulateur de réseaux de capteurs IDEA1TLM, développé dans l'équipe Conception de Systèmes Hétérogènes. La difficulté de mise en oeuvre de ces réseaux pousse à les étudier -en phase amont de la conception- par simulation. Il est donc primordial de fournir un simulateur précis et rapide dont les résultats sont très détaillés. Le simulateur IDEA1TLM permet ainsi de prédire quels circuits et quelle configuration sont les plus adéquats à une application sans fil donnée. Dernièrement, le simulateur a été amélioré pour permettre la simulation rapide de systèmes électroniques matériellement différents (hétérogènes) dans un même réseau, ce qui est relativement novateur. IDEA1TLM offre du support pour les architectures matériels AVR, MSP430 et PIC. Pour avoir un aperçu du comportement du logiciel sur le réseau, IDEA1TLM a été encore affiné avec des profils de ce logiciel à travers un simulateur de set d'instructions dénommé BLISS. Celui-ci est actuellement implémenté pour AVR et MSP430.

A part pour la consommation énergétique pour chaque partie (microcontrôleur, unité radio) de chaque noeud, IDEA1TLM est capable d'évaluer la consommation dans chaque état de

fonctionnement de ces unités. De plus, c'est possible de sortir un fichier de trace pour voir le comportement des composants modelés.

D'après notre connaissance, IDEA1TLM est la seule plate-forme de simulation pour des WSNs hétérogènes matériellement-logiciellement. En plus, elle est aussi la seule qui modélise le comportement des noeuds au niveau TLM.

## 4.3 Conclusion Générale

Cette thèse se focalise sur deux problématiques dans le contexte des réseaux de capteurs sans fil : la reprogrammation dynamique et la simulation. La contribution emmenée à la première catégorie se résume en un mot : MinTax. MinTax est une solution qui vise la minimisation des mises-à-jour en réduisant le temps de la communication radio et en même temps faisant attention à des autres aspects tels que la latence, les hétérogénéités matérielles et logicielles qui peuvent être présentes au sein d'un réseau.

Les perspectives générales incluent l'extension du compilateur MinTax pour des autres architectures embarquées (comme PIC par exemple) et d'autres systèmes d'exploitation. Bien que les performances du compilateur ont été testées sur des algorithmes simples et complexes, un déploiement dans un réseau d'une taille plus grande n'a pas été effectué. Ceci a été l'argument pour utiliser IDEA1TLM, qui comprend la deuxième partie de cette thèse. Le but c'était de pouvoir construire un aperçu global du comportement logiciel-matériel dans un contexte WSN à travers IDEA1TLM.

Concernant IDEA1TLM, des nouveaux composants supportés sont à envisager en ce qui concerne le microcontrôleur, l'interface radio et le capteur. Pour le moment, la partie d'IDEA1TLM responsable de simuler le comportement logiciel sur le noeud, BLISS, ne tient pas compte des concepts de ligne d'assemblage présents dans l'unité de calcul du microcontrôleur. L'ajout dynamique au réseau, ainsi que la possibilité de diviser la partie de payload entre plusieurs paquets

ne sont pas encore supportés.

# Références


[1] Philip Alexander Levis. **TinyOS : An Open Operating System for Wireless Sensor Networks (Invited Seminar)**. In *MDM*, page 63, 2006. xxii, 20

[2] I.F. Akyildiz and M.C. Vuran. *Wireless Sensor Networks*. Ian F. Akyildiz Series in Communications and Networking. John Wiley & Sons, 2010. xxvii, 3, 4, 7

[3] Ian Akyildiz and Mehmet Can Vuran. *Wireless Sensor Networks*. John Wiley & Sons, Inc., New York, NY, USA, 2010. xxvii, 5

[4] Carolina Fortuna. **Why is sensor data hard to get ?** COIN-ACTIVE Summer School on Advanced Technologies for Knowledge Intensive Networked Organizations in Aachen, 2010. xxvii, 10

[5] Jason Lester Hill. *System architecture for wireless sensor networks*. PhD thesis, 2003. Adviser-Culler, David E. xxvii, 21

[6] **Programming languages history**. xxvii, 41

[7] **The Sensor Network Museum**. In *http ://www.snm.ethz.ch/pub/uploads/Projects/core.jpg*. xxvii, 50

[8] *SystemC Tutorial*. xxix, 117

[9] Wikipedia. **Elf File Format diagram**. xxx, 144

[10] System V Application Binary Interface. *Executable and Linking Format (ELF) Specification Version 1.2*. TIS Committee, May 1995. xxx, 143, 145

[11] Benny Lo and Guang-Zhong Yang. **Body Sensor Networks**. In *Body Sensor Networks*, pages 416–417, London, UK, 2006. Springer-Verlag. xxxi, 11

[12] Nitaigour P. Mahalik. *Sensor Networks and Configuration : Fundamentals, Standards, Platforms, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 1, 2

[13] Jude Allred, Ahmad Bilal Hasan, Saroch Panichsakul, William Pisano, Peter Gray, Jyh Huang, Richard Han, Dale Lawrence, and Kamran Mohseni. **SensorFlock : an airborne wireless sensor network of micro- air vehicles**. In *SenSys '07 : Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 117–129, New York, NY, USA, 2007. ACM. 2

[14] Ashay Dhamdhere, Vijay Sivaraman, Vidit Mathur, and Shuo Xiao. **Algorithms for Transmission Power Control in Biomedical Wireless Sensor Networks**. In *APSCC '08 : Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference*, pages 1114–1119, Washington, DC, USA, 2008. IEEE Computer Society. 2

[15] Hsin-Mu Tsai, Wantanee Viriyasitavat, Ozan K Tonguz, Cem U Saraydar, Timothy Talty, and Andrew MacDonald. **Feasibility of In-car Wireless Sensor Networks : A Statistical Evaluation**. In *SECON*, pages 101–111, 2007. 2

[16] S. Beeby and N. White. *Energy Harvesting for Autonomous Systems*. Smart Materials, Structures, and Systems. Artech House, 2010. 2

[17] Andreas Terzis. **Slip surface localization in wireless sensor networks for landslide prediction**. In *In IPSN 2006*, pages 109–116. ACM Press, 2006. 2

[18] Alberto Rosi, Marco Mamei, Franco Zambonelli, Antonio Manzalini, and Telecom Italia. **Landslide Monitoring with Sensor Networks : a Case for Autonomic Communication Services Invited Paper**, 2010. 2

[19] Anmol Sheth, Chandramohan A. Thekkath, Prakshep Mehta, Kalyan Tejaswi, Resh Parekh, Trilok N. Singh, and Uday B. Desai. **Senslide : A Distributed Landslide Prediction System**, 2010. 2

[20] Research Challenges Ian, Ian F. Akyildiz, and Erich P. Stuntebeck. **Wireless Underground Sensor Networks :**. *Ad Hoc Networks*, **4** :2006. 2

[21] Antonio-Javier Garcia-Sanchez, Felipe Garcia-Sanchez, and Joan Garcia-Haro. **Wireless sensor network deployment for integrating video-surveillance and data-monitoring in precision agriculture over distributed crops**. *Comput. Electron. Agric.*, **75**(2) :288–303, February 2011. 2

[22] Agnelo R. Silva and Mehmet C. Vuran. **(CPS)2 : integration of center pivot systems with wireless underground sensor networks for autonomous precision agriculture**. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 79–88, New York, NY, USA, 2010. ACM. 2

[23] Kebin Liu, Zheng Yang, Mo Li, Zhongwen Guo, Ying Guo, Feng Hong, Xiaohui Yang, Yuan He, Yuan Feng, and Yunhao Liu. **OceanSense : Monitoring the Sea with Wireless Sensor Networks**. 2

[24] Silvia Nittel, Niki Trigoni, Konstantinos Ferentinos, Francois Neville, Arda Nural, and Neal Pettigrew. **A drift-tolerant model for data management in ocean sensor networks**. In *Proceedings of the 6th ACM international workshop on Data engineering for wireless and mobile access*, MobiDE '07, pages 49–58, New York, NY, USA, 2007. ACM. 2

[25] Nitthita Chirdchoo, Wee-Seng Soh, and Kee Chaing Chua. **MU-Sync : a time synchronization protocol for underwater mobile networks**. In *Proceedings of the third ACM international workshop on Underwater Networks*, WuWNeT '08, pages 35–42, New York, NY, USA, 2008. ACM. 2

[26] Santosh Bhima, Anil Gogada, and Rammurthy Garimella. **A tsunami warning system employing level controlled gossiping in wireless sensor networks**. In *Proceedings of the 4th international conference on Distributed computing and internet technology*, ICDCIT'07, pages 306–313, Berlin, Heidelberg, 2007. Springer-Verlag. 2

[27] Ashfaq Hussain Farooqi and Farrukh Aslam Khan. **A survey of Intrusion Detection Systems for Wireless Sensor Networks**. *Int. J. Ad Hoc Ubiquitous Comput.*, **9**(2) :69–83, February 2012. 2

[28] Ilker Bekmezci. *Wireless Sensor Networks : A Military Monitoring Application*. VDM Verlag, Saarbr&#252 ;cken, Germany, Germany, 2009. 2

[29] Sang Hyuk Lee, Soobin Lee, Heecheol Song, and Hwang Soo Lee. **Wireless sensor network design for tactical military applications : remote large-scale environments**. In *Proceedings of the 28th IEEE conference on Military communications*, MILCOM'09, pages 911–917, Piscataway, NJ, USA, 2009. IEEE Press. 2

[30] **AVR ATMega Microcontroller**. In *http ://www.atmel.com/products/avr*. ATMEL Corporation. 9

[31] **MSP430 16-bit Microcontroller Family**. In *http ://www.ti.com*. Texas Instruments. 9

[32] **Pic Microcontroller**. In *http ://www.microchip.com*. MICROCHIP Corporation. 9

[33] Kurtis Kredo Ii and Prasant Mohapatra. **Medium Access Control in Wireless Sensor Networks**, 2007. 10

[34] Ryo Sugihara and Rajesh K. Gupta. **Programming models for sensor networks : A survey**. *ACM Trans. Sen. Netw.*, **4**(2) :8 :1–8 :29, April 2008. 14

# RÉFÉRENCES


[35] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. **Wireless sensor network survey**. *Comput. Netw.*, **52**(12) :2292–2330, August 2008. 16

[36] **Mica2 Sensor Nodes(UC Berkley/Crossbow)**. In *http ://www.xbow.com/*. UC Berkley/Crossbow. 17

[37] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. **MANTIS OS : an embedded multithreaded operating system for wireless micro sensor platforms**. *Mob. Netw. Appl.*, **10**(4) :563–579, 2005. 21

[38] Anand Eswaran, Anthony Rowe, and Raj Rajkumar. **Nano-RK : An Energy-Aware Resource-Centric RTOS for Sensor Networks**. In *RTSS '05 : Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 256–265, Washington, DC, USA, 2005. IEEE Computer Society. 23

[39] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon. **RETOS : resilient, expandable, and threaded operating system for wireless sensor networks**. In *IPSN '07 : Proceedings of the 6th international conference on Information processing in sensor networks*, pages 148–157, New York, NY, USA, 2007. ACM. 24

[40] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. **A dynamic operating system for sensor nodes**. In *MobiSys '05 : Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM. 25

[41] Philip Levis and David Culler. **Maté : a tiny virtual machine for sensor networks**. In *ASPLOS-X : Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM. 28

[42] Niels Brouwers, Koen Langendoen, and Peter Corke. **Darjeeling, a feature-rich VM for the resource poor**. In *SenSys '09 : Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 169–182, New York, NY, USA, 2009. ACM. 29

[43] Joel Koshy. **Remote incremental linking for energy-efficient reprogramming of sensor networks**. In *In Proceedings of the second European Workshop on Wireless Sensor Networks*, pages 354–365. IEEE Press, 2005. 32

[44] Rajesh Krishna Panta and Saurabh Bagchi. **Hermes : Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks.** In *INFOCOM'09*, pages 639–647, 2009. 33

[45] Wei Dong, Yunhao Liu, Chun Chen, Jiajun Bu, and Chao Huang. **R2 : Incremental Reprogramming using Relocatable Code in Networked Embedded Systems**. In *30th IEEE Conference on Computer Communications (IEEE INFOCOM) Mini-Conference*, 2011. 33

[46] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. **Zephyr : efficient incremental reprogramming of sensor nodes using function call indirections and difference computation**. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 32–32, Berkeley, CA, USA, 2009. USENIX Association. 34

[47] **Declarative Programming**. In *http ://en.wikipedia.org/wiki/Declarative_programming*. 40

[48] **Reverse Polish Notation**. In *http ://en.wikipedia.org/wiki/Reverse_polish_notation*. 43, 75

[49] Peter Bumbulis and Donald D. Cowan. **RE2C : a more versatile scanner generator**. *ACM Lett. Program. Lang. Syst.*, **2**(1-4) :70–84, 1993. 56

[50] **The Lemon Parser-Generator**. In *http ://www.hwaci.com/sw/lemon/*. 65

[51] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. **Efficient incremental code update for sensor networks**. *ACM Trans. Sen. Netw.*, **7** :30 :1–30 :32, February 2011. 100

[52] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. **Run-time dynamic linking for reprogramming wireless sensor networks**. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 15–28, New York, NY, USA, 2006. ACM. 106

[53] Wan Du, Fabien Mieyeville, and D. Navarro. **IDEA1 : A SystemC-based System-level Simulator for Wireless Sensor Networks**. In Wen Chen and Shaozi Li, editors, *IEEE International Conference WCNIS2010*, **2**, pages 618–623. IEEE, June 2010. 113

[54] Wan Du, David Navarro, Fabien Mieyeville, and Ian O'Connor. **IDEA1 : A Validated System-level Simulator for Wireless Sensor Networks**. In *The 4th International Workshop on Wireless Sensor, Actuator and Robot Networks (WiSARN-Fall 2011)*, Valencia, Spain, October 2011. 113

[55] Franco Fummi, Davide Quaglia, and Francesco Stefanni. **A SystemC-based Framework for Modeling and Simulation of Networked Embedded Systems.** In *Forum on Specification, Verification and Design Languages (FDL)*, pages 49–54, 2008. 113

[56] *Avrora : scalable sensor network simulation with precise timing*, April 2005. 143

[57] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik Österlind, and Thiemo Voigt. **MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards**. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007. 143

[58] C-Lab. WCET benchmarks. Available from http ://www.c lab.de/home/en/download.html. 152, 153

[59] Compression Huffman. In *Wikipedia*. 154

**Reconfiguration dynamique et simulation fine modélisée au niveau de transaction dans les Réseaux de Capteurs sans Fil hétérogènes matériellement/logiciellement**

-Résumé en Français-

Cette thèse porte premièrement sur la reconfiguration dynamique et la simulation hétérogène dans les Réseaux des Capteurs sans Fil. Ces réseaux sont constitués d'une multitude de systèmes électroniques communicants par radio-fréquence, très contraints en énergie. La partie de communication radio entre ces noeuds est la plus consommatrice. C'est pourquoi la minimisation du temps effectif est désirée. On a implémenté une solution qui consiste à envoyer au noeud un fichier de reconfiguration codé utilisant un langage de programmation haut niveau (MinTax). Le noeud sera capable de compiler ce fichier et générer le code object associé à son architecture, in-situ. Grâce au caractère abstrait du MinTax, plusieurs architectures matérielles et systèmes d'exploitation sont visés.

Dans un deuxième temps, ce travail de thèse est lié au simulateur de réseaux de capteurs IDEA1TLM. IDEA1TLM permet de prédire quels circuits et configurations sont les plus adéquats à une application sans fil donnée. Ce simulateur a été amélioré pour permettre la simulation rapide des systèmes électroniques matériellement différents dans le même réseau ainsi que le logiciel présent sur les noeuds.

**Mots clés :** Reconfiguration dynamique, Compilation in-situ, MinTax, Hétérogénéité, IDEA1TLM.

**Dynamic reconfiguration and fine-grained simulation modelled at transaction level in hardware/software heterogeneous Wireless Sensor Networks**

-Summary in English-

This PhD thesis concerns the dynamic reconfiguration and simulation of heterogeneous Wireless Sensor Networks. These networks consist of a multitude of electronic units called ?nodes ?, which communicate through a radio interface. The radio interface is the most power-consuming on the node. This is why the minimisation of the radio-time would lead to improved energy efficiency. We have implemented a software solution which consists in sending an update to a node which is coded in a high-level language (MinTax). This file is compiled by the node and machine code is generated for the target hardware architecture. Owing to the abstract nature of MinTax, multiple hardware architectures. as well as operating systems are supported.

As a second part of this PhD, work has been focused on a network simulator called IDEATLM. IDEA1TLM allows us to predict which circuits and configurations are the most appropriate for a given task. This solution has been improved to allow a faster simulation of electronic systems which are different from a hardware standpoint, yet part of the same network, as well as to model the actual software running on them.

**Keywords :** Dynamic reconfiguration, In-situ compilation, MinTax, Heterogeneity, IDEA1TLM.