|  |  |
| --- | --- |
| http://www.mwq.be/servlet/Repository/?IDR=2988&IDQ=20 | Helha – Campus Technique  Master en Electronique  Année académique 2015-2016 |
| **Projet Electronique** | |

Projet Tilt and Shocks Sensor



Julien Delvaux - Michael Manzella - Loic Quinet

Contenu

[1. Introduction 1](#_Toc450328905)

[2. Analyse de l’existant et cahier des charges 2](#_Toc450328906)

[2.1. Analyse de l’existant 2](#_Toc450328907)

[2.2. PICOS18 2](#_Toc450328908)

[2.3. NDEF – NFC Data Exchange Format 2](#_Toc450328909)

[2.3.1. Structure d’un NDEF Record 3](#_Toc450328910)

[2.3.2. Organisation de la mémoire d’un tag NFC et blocs TLV 5](#_Toc450328911)

[2.4. Capteurs 6](#_Toc450328912)

[2.4.1. Température 6](#_Toc450328913)

[2.4.2. Accéléromètre 6](#_Toc450328914)

[2.5. EEPROM 7](#_Toc450328915)

[2.5.1. Librairies développées par Denis Alaime 7](#_Toc450328916)

[2.5.2. Présentation du composant M24LR04E\_R 8](#_Toc450328917)

[2.6. Android 9](#_Toc450328918)

[2.7. Cahier des charges 10](#_Toc450328919)

[3. Développement 11](#_Toc450328920)

[3.1. PICOS18 11](#_Toc450328921)

[3.2. Accéléromètre 13](#_Toc450328922)

[3.3. Capteur température 15](#_Toc450328923)

[3.4. Mémoire EEPROM 16](#_Toc450328924)

[3.4.1. Améliorations 16](#_Toc450328925)

[3.4.2. Explication des librairies 17](#_Toc450328926)

[3.5. Android 19](#_Toc450328927)

[4. Résultats 22](#_Toc450328928)

[4.1. Liste des modifications apportées – nouvelles fonctionnalités 22](#_Toc450328929)

[4.2. PIC 22](#_Toc450328930)

[4.2.1. I²C 22](#_Toc450328931)

[4.2.2. Timer 23](#_Toc450328932)

[4.2.3. Mode Sleep 24](#_Toc450328933)

[4.2.4. Accéléromètre 25](#_Toc450328934)

[4.3. Application Android 25](#_Toc450328935)

[4.4. Consommation 27](#_Toc450328936)

[5. Conclusion 29](#_Toc450328937)

[6. Bibliographie 30](#_Toc450328938)

# Introduction

Ce projet nous a été proposé par Denis Alaime. Il a imaginé un prototype électronique avec un microcontrôleur et des capteurs pour répondre à un besoin actuel dans le transport de colis. Que ce soit des colis fragiles ou des colis médicaux, cette carte est capable de détecter des évènements non autorisés et de les enregistrer. Le client final peut alors scanner le colis afin de savoir comment s’est passé le transport.

Ce travail se décompose en deux parties distinctes. La première concerne le développement d’une application Android. Celle-ci sera le point de contact entre la carte électronique et le client. La seconde concerne le développement du programme du microcontrôleur. Plusieurs points sont à accomplir : réduction de consommation, ajout de fonctionnalités par rapport au prototype reçu, calibration des capteurs …

Ce rapport s’articule autour de trois grands axes. Le premier concerne l’analyse de l’existant, le cahier des charges mais aussi une explication des différents composants avec quelques-unes de leurs caractéristiques sans oublier une introduction au PICOS18. Le second s’intéresse au développement. Il est composé des capteurs, du PICOS et de l’application Android. Le dernier point s’intéresse aux résultats : la consommation du prototype, quelques problèmes rencontrés avec le PIC … Enfin, la conclusion permettra de donner quelques pistes de réflexion sur les améliorations possibles.

Afin de mener à bien ce projet, nous avons utilisé Git pour la gestion des fichiers sources mais aussi Slack pour communiquer entre nous. Ces deux outils permettent d’améliorer notre efficience.

# Analyse de l’existant et cahier des charges

## Analyse de l’existant

Le projet tel que nous l’avons reçu était au stade de preuve de concept. Cela implique que le code était brouillon, non optimisé. De plus, nous n’avons reçu aucune documentation concernant ce qui avait été fait. Il nous a fallu un peu de temps pour nous plonger dans le code, comprendre comment cela fonctionnait. Le challenge en plus de se familiariser avec le PICOS, était d’apprendre à manipuler le PIC.

Au niveau hardware, la carte électronique était déjà réalisée et avait été testée par Denis Alaime. Lors de la prise en main, nous avons eu l’occasion de vérifier que tout fonctionnait correctement.

Certaines choses ont déjà été réalisées au niveau de la programmation du PIC. Le PICOS18 était fonctionnel et le programme réussissait à communiquer avec les capteurs (accéléromètre et température) et la mémoire EEPROM.

Denis Alaime utilisait une application Android développée par STMicroelectronics afin de vérifier qu’il arrivait à écrire correctement dans l’EEPROM. Il n’avait rien créé comme application Android.

## PICOS18

Le PICOS18 est un OS temps réel (*real-time OS*) développé par la société Pragmatech sous licence GPL (open-source). La version actuelle, numérotée 3, est sortie en 2007. Elle utilise le compilateur C18 de Microchip. La norme OSEK permet de définir le fonctionnement interne de l’OS comme la liste des services du noyau et la gestion interne des tâches et des ressources.

Le noyau temps-réel fonctionne autour de trois grands axes :

* Le noyau : il regroupe un ensemble de fonctionnalités appelées services dont un exemple typique est le *scheduler*, responsable de la gestion de celles-ci. Il garantit également la stabilité du système et contrôle les ressources.
* Multitâches : Plusieurs tâches peuvent fonctionner de manière séparée et sont régulées par le noyau. Si celui-ci le permet, il est possible de donner l’impression qu’elles s’exécutent en parallèle. Le noyau est dit « multitâches préemptif » s’il est à même de gérer ce fonctionnement parallèle. Dans le cas contraire, on le dénomme « multitâche coopératif ».
* Temps-réel : Le noyau multitâche alloue un temps égal et une zone mémoire identique pour chaque tâche. Cependant, celles-ci ont rarement la même priorité et elles doivent être appelées le plus rapidement possible. Plutôt que d’essayer d’avoir un temps de réactivité quasi-nul (impossible en pratique), le noyau garantit temps de latence constant : c’est le déterminisme.

## NDEF – NFC Data Exchange Format

Le NDEF est un format d’encapsulation de données communes aux équipements ; la spécification NDEF définit le format d’encapsulation d’un message échangé entre un NFC Forum Device et une cible (NFC Forum Device ou NFC Forum Tag). Ce message peut être de différents types ; leur implémentation est décrite dans les spécifications « Record Type Definition ». Ainsi, pour un message de type texte, il faudra se référer à la spécification « NFC Text Record Type Definition ». Les types peuvent être des URL, média ou des types spécifiques au NFC (les « NFC well-known types ») utilisés par des applications NFC, par exemple le type texte.

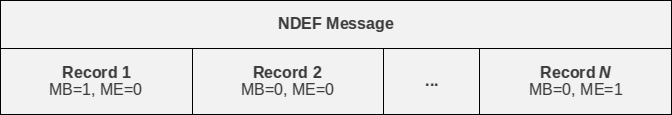
Un message NDEF est composé d’un ou de plusieurs « NDEF record » que l’on peut traduire par « enregistrement NDEF » (voir Figure 1). Les NDEF records servent à encapsuler un ou plusieurs payload définis par une application, de type et de taille arbitraire dans un seul message. Un message NDEF peut aussi contenir des « chuncked records », c’est-à-dire des chaines records, afin de supporter des payloads plus volumineux ou pour partitionner dynamiquement le contenu. Chaque « chuncked payload » contient un « record chunk » initial, suivi de zéro ou plusieurs records chuncks « intermédiaires » et enfin se termine par un record chunck final.

Figure : Encapsulation des NDEF records dans un message NDEF

### Structure d’un NDEF Record

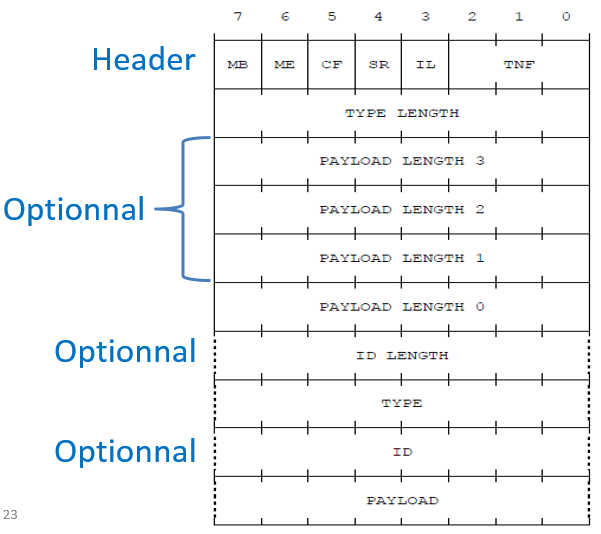


Figure : Structure d'un NDEF record

#### Header

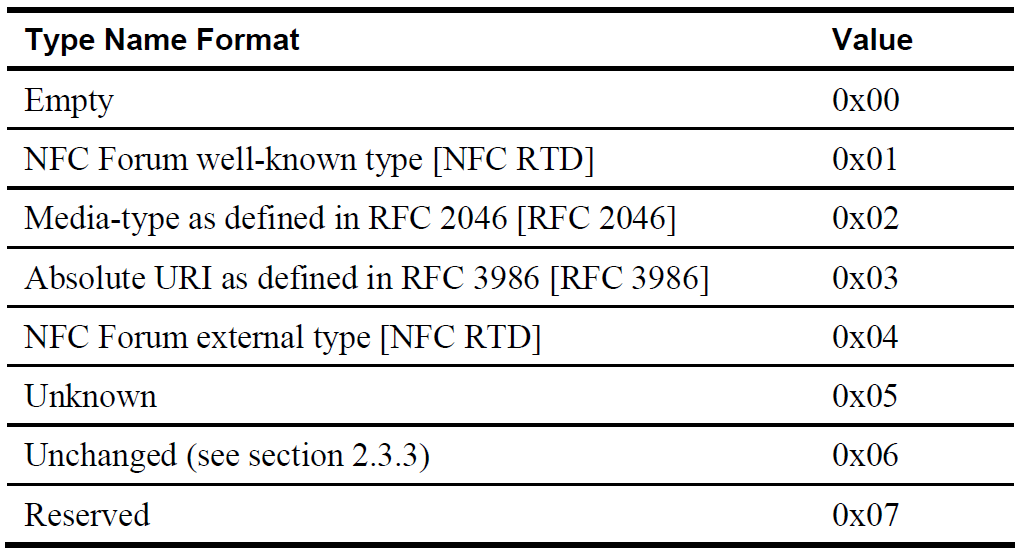
* **MB (Message Begin)**: Indique si le record est le premier du message NDEF si ce bit est à 1.
* **ME (Message End)**: Indique si le record est le dernier du message NDEF si ce bit est à 1. Notons que s’il n’y a qu’un message les bits MB et ME sont à 1.
* **CF (Chunk Flag)**: Indique si le record est le record initial, intermédiaire ou final.
* **SR (Short Record)**: Indique si la taille du payload est codée sur 1 byte si ce bit est à un, sinon sur 4 bytes si ce bit est à 0.
* **IL (ID\_LENGTH field is present)**: Indique si les champs “ID\_LENGTH” et “ID” sont presents.
* **TNF (Type Name Format)**: Code qui indique la structure du type du payload (voir Figure 3)

Figure : Type Name Format

#### TYPE\_LENGTH

Spécifie la taille en octets du type du payload.

#### ID\_LENGTH

Spécifie la taille en octets du champ « ID ».

#### TYPE

C’est un identifiant qui décrit le type du payload. Par exemple pour le type texte du « the NFC Forum well-known type », l’identifiant est le code ASCII du caractère ‘T’. L’encodage (UTF-8, UTF-16) et le code de langage (« en », « fr », …) sont décrits au sein du payload.

Le type URI permet de stocker une URI (Uniform Resource Identifier), tel qu’un URL pour une page web par exemple. Mais l’intérêt du type URI évite de devoir encoder le préfixe de l’URI dans le payload, ce qui économise plusieurs octets (voir Figure 4). Par exemple pour le préfixe « http://www. » le code est 0x01.

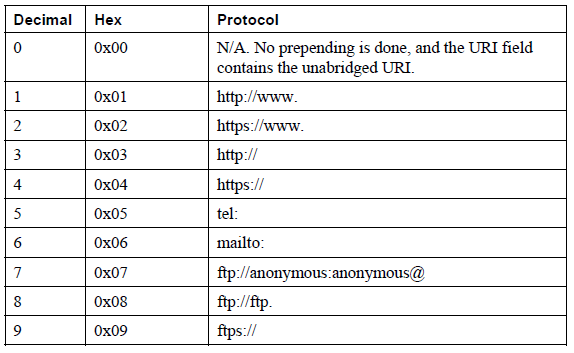
Le « NFC Forum external type » permet de créer son propre type avec un nom de domaine (ex : myCompany.org). Il prend la forme d’un URI (urn:nfc:ext : myCompany.org :t).

Figure : Extrait des codes de préfixe des URI

Le type media est utilisé pour les photos, les musiques, les vidéos ainsi que pour les types défini par les applications. Il permet aussi d’utiliser en quelque sorte le type binaire générique. Le type prend la forme Type/Subtype. Pour notre application nous avons utilisé le type binaire, qui correspond à « application/octet-stream ».

Pour plus d’information aller voir le document « NFC Data Exchange Format (NDEF) » sur le NFC Forum, ou le document pdf à l’adresse <http://www.eet-china.com/ARTICLES/2006AUG/PDF/NFCForum-TS-NDEF.pdf>

### Organisation de la mémoire d’un tag NFC et blocs TLV

La mémoire est divisée en blocs, où chaque bloc contient 4 bytes.

Le premier bloc contient le « capability container », qui décrit les capacités du tag (taille de la zone mémoire et permissions d’écriture/lecture).

Les données sonr organisées par champs TLV (T pour “Tag Field”, L pour “Length Field » et V pour « Value Field »). Un bloc est composé de soit un champ « Tag », un champ « length » et un champ « Value », soit juste un champ « terminating » à la fin de la zone de données. Le champ tag décris le type de contenu (message NDEF, propriétaire, …) et le champ « Length » donne la taille de la mémoire utilisée pour les données dans le bloc TLV. Il est soit sur 1 byte, soit sur 3 bytes auquel cas le byte de poids faible est à 0xFF et la taille est sur les 2 bytes de poids fort.

Pour plus d’information, aller voir dans le NFC Forum le document « Type 4 Tag Operation Specification ».

## Capteurs

## Température

Le capteur de température est le EMC1001 de chez SMSC. Voici quelques une de ses caractéristiques :

* Résolution de 0.25°C et de ± 1.5°C entre 40°C et 85°C
* Deux interruptions utilisables
* Faible consommation

La structure interne de ce capteur est reprise dans la Figure 5. Une diode interne est sensible à la température. Le circuit interne utilise le courant de cette diode pour déterminer la température grâce à un ADC. Celle-ci est alors écrite dans un registre interne. Si des seuils ont été définis (seuil haut et/ou bas) pour la température, le capteur est capable de générer une interruption afin de prévenir d’un dépassement d’un de ces seuils.

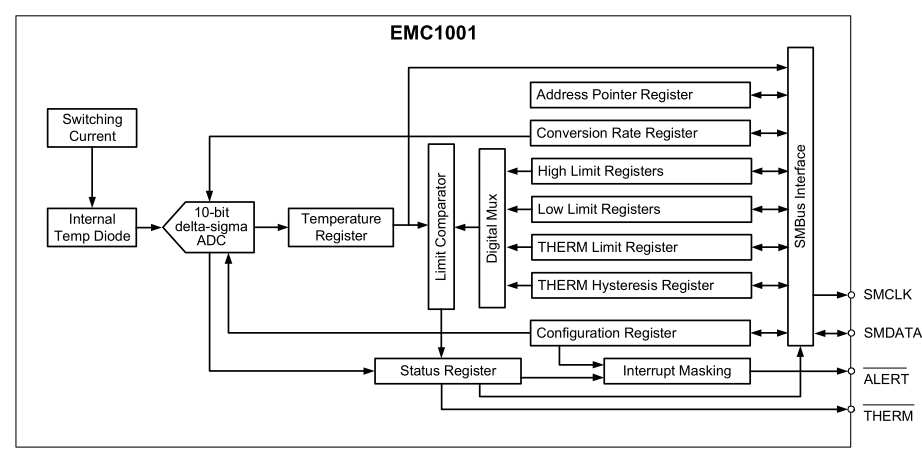


Figure Schéma block interne du capteur de température

## Accéléromètre

L’accéléromètre est le FXLS8471Q de Freescale Semiconductor. C’est un accéléromètre 3 axes. Voici quelques-unes de ses caractéristiques :

* Echelle de ± 2g, ± 4g ou ± 8g
* Fréquence de capture de 1.563 Hz à 800Hz
* ADC interne de résolution 14-bit
* Fonctionnalités programmables :
  + Détection de mouvement et chute libre
  + Détection d’orientation
  + Détection de « tap »
* Mode sleep
* Fonctionnement en I²C ou SPI

Son fonctionnement est décrit par la Figure 6. La fonction d’acquisition est réalisée par les blocs verts. Grâce aux transformations internes de l’accéléromètre, les valeurs acquises sont numérisées. Le bloc *Embedded DSP Functions* regroupe toutes les fonctionnalités possibles avec l’accéléromètre.

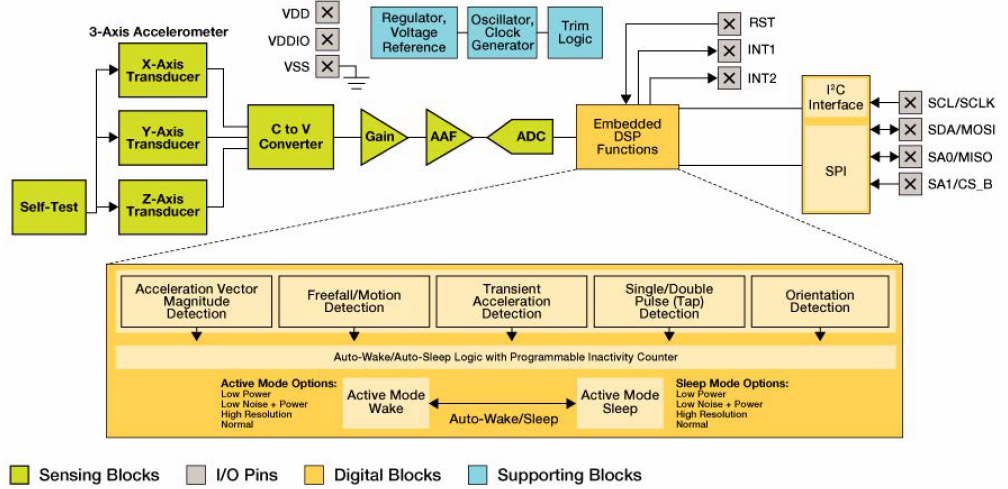


Figure Schéma fonctionnement FXLS8471Q

## EEPROM

### Librairies développées par Denis Alaime

Les librairies déjà existantes en rapport à l’EEPROM n’ont pas été faciles à prendre en main. La compréhension des messages NDEF a aussi été un challenge. D’une part, il a fallu comprendre l’utilisation du driver I²C du PICOS18 afin de communiquer avec l’EEPROM. D’autre part, il a aussi fallu comprendre comment Denis Alaime avait fait pour construire un message NDEF et le stocker dans l’EEPROM. Ce fût la partie la plus compliquée pour l’EEPROM. En effet, de longues recherches ont été nécessaires pour comprendre comment fonctionnent les trames NDEF, ce qui se fait en pratique, l’organisation de la mémoire d’un tag NFC et ensuite comprendre son implémentation dans le cadre de notre application.

La partie utilisation du driver I²C du PICOS, adapté à l’EEPROM a été implémentée dans la librairie « M24LR04E\_R », ainsi que le stockage d’un message NDEF construit au préalable.

La partie construction d’un message NDEF a été implémentée dans les librairies « NDEFMessage » et « NDEFRecord ».

La librairie « NDEFRecord » a été utilisée comme un objet. Elle contient une variable globale *NdefRecord* de type *\_NdefRecord\_t*, utilisée comme une variable de classe. La librairie contient ensuite toute une série de fonctions, dont une fonction *NdefRecordConstructor*, utilisées comme des méthodes de classe. C’est comme si on avait un objet instancié de manière statique, qui contient toutes les données d’un NDEF record stockées au moyen de la structure *NdefRecord*. Tous ses champs sont initialisés avec des valeurs par défaut lors de l’appel de la fonction *NdefRecordConstructor*.

La librairie  « NDEFMessage » s’occupe ainsi de construire le payload et de remplir tous les champs du NDEF record. Notons que Denis Alaime n’avait implémenté que l’écriture d’un seul NDEF record dans un seul NDEF message, ce qui simplifiait grandement les choses.

### Présentation du composant M24LR04E\_R

L’EEPROM a deux interfaces de communication : une en I²C et une en NFC. La mémoire fait 64-Kbit organisé en : 8192 bytes en I²C et en 2048 blocs en NFC. L’EEPROM fait plus de 1 million de cycles d’écriture. La communication I²C peut aller jusque 400KHz, supporte l’écriture de page (une page fait 4 bytes) et la lecture aléatoire et séquentielle. Il est important de savoir que **l’EEPROM possède un timeout en I²C**.

On peut utiliser l’EEPROM en interruption car elle possède une broche RF WIP/BUSY qui passe à l’état 0 dès qu’il y a une communication RF. Il n’est pas possible de communiquer via NFC lorsqu’on communique déjà en I²C avec l’EEPROM.

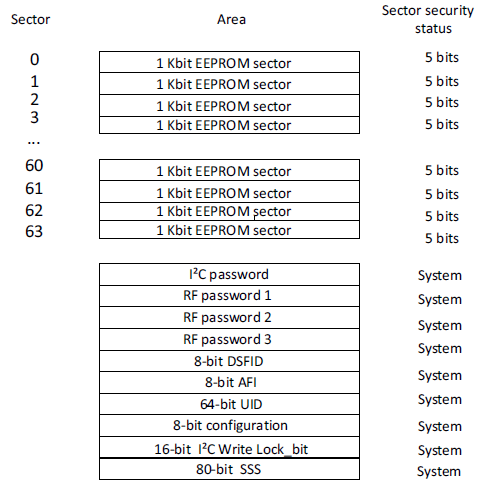
L’EEPROM possède aussi une sécurité en écriture/lecture sur la mémoire accessible en I²C et en RF. La mémoire de l’EEPROM est divisée en 64 secteurs. Chaque secteur contient 1024 bits. En mode RF, un secteur fournit 32 blocs de 4 bytes. Il y a un accès en écriture/lecture sur chaque bloc. En I²C, un secteur fournit 128 bytes qui peuvent être accessibles en lecture et en écriture. Enfin, à la fin de la mémoire il y a les bytes de configuration de l’EEPROM (voir Figure 7).

Figure : Organisation de la mémoire de l'EEPROM M24LR04E\_R

## Android

L’application utilisée était celle développée par STMicroelectronics pour son module de communication NFC M24LR-DISCOVERY (voir **Erreur ! Source du renvoi introuvable.**). Ce kit nous a été fourni pour nos tests et nous familiariser avec le protocole de communication NFC.

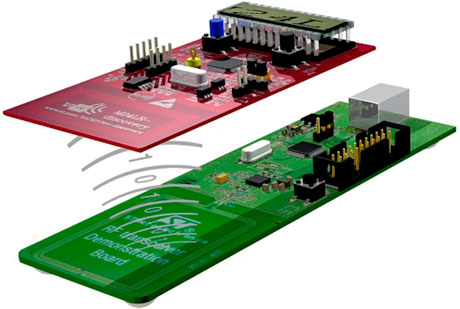


Figure : Kit NFC M24LR-DISCOVERY de chez STMicroelectronics

Cette application permet plusieurs fonctions comme la lecture de message NDEF, la lecture/écriture dans la mémoire sous forme de block ou encore la possibilité d’envoyer ou recevoir des fichiers images par NFC.

|  |  |
| --- | --- |
| C:\Users\michael\Desktop\Annee 2015-2016\Bloc 2\Projet Master\Application Android St electronique\screen shots\picture0.png  (a) | picture3  (b) |

Figure  : Capture d'écran de l'application Android développée par STMicroelectronics (a) écran d’accueil de l’application, (b) écran de lecture de message NDEF

Les messages NDEF lu sont aussi enregistrés dans un fichier texte qui peut être lu avec l’application. Les informations enregistrées dans ce fichier sont la date de lecture du message NDEF, le type (TEXT, URI, …) et le contenu du message.

Le code source de l’application nous a été fourni. Après l’avoir analysé et déterminé les fonctions que l’on voulait implémenter à notre application Android, nous avons commencé à le développer.

## Cahier des charges

Nos objectifs pour ce projet étaient les suivants :

1. Garder le PICOS18.
2. Faire clignoter la LED à intervalles réguliers.
3. Ajouter des fonctionnalités au prototype.
4. Optimiser la consommation afin de faire durer la batterie le plus longtemps possible.
5. Développer l’application Android customisée pour ce projet.

Pour ce faire, nous avons divisé le groupe en deux :

* Michael Manzella a travaillé au développement de l’application Android.
* Loic Quinet et Julien Delvaux ont travaillé avec la carte électronique : PIC et capteurs/mémoire.

# Développement

## PICOS18

Le microcontrôleur utilise le PICOS18 pour fonctionner. Dans notre cas, nous avons une tâche principale (*TASK\_Main*) et deux tâches secondaires nécessaires par l’OS : une pour l’I2C et une autre pour l’UART (utile pour le débug). Ces deux dernières sont fournies par le site du PICOS sous la forme de « driver ». Il suffit simplement de les inclure dans le projet et de modifier certains paramètres comme le baudrate par exemple.

Le fonctionnement de la tâche principale est illustré à la Figure 10. L’idée est de mettre le PIC en mode *SLEEP* dès que possible afin de minimiser sa consommation autant que possible. Uniquement lorsqu’il y a une interruption, le microcontrôleur passe en mode *RUN* afin de traiter celle-ci. Il y a trois sources d’interruptions possibles :

1. RTCC : la *Real-Time Clock and Calendar[[1]](#footnote-1)* va permettre de faire clignoter la LED (toutes les 2 secondes), de lire la configuration au niveau de l’EEPROM (toutes les 20secondes si la variable isRF\_WIP\_BUSY a été modifiée), de faire une acquisition de température (toutes les minutes) et réécrire la date et heure dans l’EEPROM (toutes les 10 minutes).
2. EEPROM : Dès qu’une personne approche son GSM, une variable (isRF\_WIP\_BUSY) est modifiée. Cela va permettre de recharger la configuration que l’utilisateur aurait changé. Une modification des modifications est avertie par l’écriture dans le byte RF\_Busy de l’EEPROM.
3. Accéléromètre : Lorsqu’un retournement ou une accélération a été détecté, le microcontrôleur va écrire ce message dans l’EEPROM.

Lorsqu’une interruption se produit, le microcontrôleur fait un *SetEvent* qui va être traité dans la tâche principale. C’est une des particularités du PICOS.

Lorsque la RTCC déclenche le clignotement de la LED (toutes les deux secondes), le microcontrôleur maintient la LED allumée pendant 50ms. Celui-ci utilise le timer1 ce qui permet de le repasser en mode SLEEP afin de minimiser sa consommation.



Figure Fonctionnement de la tâche principale

## Accéléromètre

La programmation de l’accéléromètre n’a pas été facile à réaliser. Les fichiers sources donnés par Denis Alaime contenaient un peu de configuration, comment récupérer les valeurs mais n’étaient pas très lisible et facile de s’y plonger en venant de l’extérieur.

J’ai choisi de revoir la manière dont était faite cette librairie afin de (me) faciliter le travail par la suite. L’idée étant d’avoir des fonctions et des arguments qui soient suffisamment explicites que même quelqu’un n’ayant pas lu la datasheet puisse comprendre l’idée générale.

Il y a trois grands types de fonctions dans cette librairie :

1. Fonctions pour faire la communication I2C avec l’accéléromètre : regroupe les fonctions nécessaires par l’OS pour la communication I2C et des fonctions pour l’écriture/lecture de registre.
2. Fonctions privées à la librairie : appelées par les fonctions publiques.
3. Fonctions publiques : fonctions auxquelles le développeur peut faire appel dans son code.

Bien souvent, les fonctions publiques font appel aux fonctions privées (rarement à la première catégorie) avant de faire appel aux fonctions I2C. Une fois que la première catégorie a été réalisée, ce qui a été un gros challenge, le développement du reste n’a pas posé de réel soucis.

L’initialisation se passe de la façon suivante et est illustrée par la Figure 11 :

1. Test de connexion avec le module
2. Configuration et calibration de l’accéléromètre

Lorsqu’un évènement non autorisé par l’utilisateur (dans notre cas : retournement et accélération trop importante) est détecté par l’accéléromètre, une interruption a lieu sur le microcontrôleur. Celui-ci passe alors par une fonction dans la librairie de l’accéléromètre afin de connaître la source de cette interruption. Si l’interruption était justifiée, un message est écrit dans l’EEPROM. Ce fonctionnement est illustré par la Figure 12.



Figure Initialisation de l'accéléromètre



Figure Interruption accéléromètre

## Capteur température

Afin d’optimiser la consommation du capteur, celui-ci doit être placé en standby. Mais, dans le cas où l’on souhaite utiliser les interruptions, le capteur doit être obligatoirement en mode RUN. Pour cette raison, il n’est pas possible de les utiliser.

Dans notre cas, cela n’a pas porté à conséquence. Le PIC se réveille toutes les secondes avec la RTCC. Il profite donc de faire une acquisition (*single-shot*) pendant l’un de ces réveils. L’évolution d’une température étant un process lent, nous avons décidé d’effectuer cette capture toutes les minutes. Pour faire une acquisition « single-shot » de la température et remettre le capteur de température en standby, il suffit d’avoir mis capteur en standby au préalable et d’écrire dans le registre « One-Shot » afin que le capteur mette à jour ses registres de température. On peut ensuite aller lire la valeur de la température mesurée.

La comparaison de cette température est effectuée avec les seuils haut et bas définis par l’application Android dans la mémoire EEPROM. En cas de sortie de cet intervalle, un message NDEF est enregistré dans l’EEPROM avec l’heure de l’évènement.

Afin de ne pas saturer la mémoire EEPROM, nous avons décidé d’enregistrer le premier franchissement de seuil. Ce paramètre peut être modifié aisément au sein du code du PIC. Cette information est facilement visible au sein de l’application.

## Mémoire EEPROM

### Améliorations

Plusieurs améliorations des librairies étaient à faire. Premièrement, le code était assez brouillon, peu structuré et très peu commenté ; il était assez difficile de s’y retrouver.

Ensuite, il fallait pouvoir enregistrer un NDEF record à chaque évènement non autorisé. Au début nous avions pensé à écrire un message NDEF contenant un NDEF record à chaque évènement. Après l’implémentation, nous nous sommes rendu compte qu’Android ne supporte qu’un seul message NDEF par tag. Il a donc fallu redévelopper les librairies afin de modifier le message NDEF en ajoutant un NDEF record à chaque évènement, qui était plus compliqué à implémenter. En effet, à chaque évènement il fallait modifier la valeur du « TLV length » et le header des autres records pour tenir compte de la position des NDEF records dans le message NDEF.

Nous avons aussi pensé à une meilleure organisation de la mémoire de l’EEPROM. Il y aura d’abord tout le début de la mémoire dédiée à l’enregistrement des évènements sous forme d’NDEF records (blocs 0 à 2041). Le reste de la mémoire sera dédiée aux bytes de configuration des différents périphériques ainsi que des bytes « drapeaux ». Les bytes de configuration sont les accélérations maximums pour chaque axe, l’accélération maximum pour tous les axes (*Thresold\_x\_y\_z*) pour l’accéléromètre, la date et l’heure pour la RTC et les températures maximum et minimum (température minimum non implémentée). Les bytes pour la date et l’heure servent aussi à écrire la date et l’heure toutes les 10 min afin d’avoir un contrôle sur le bon fonctionnement de la carte pendant toute la durée du transport. Les bytes « drapeaux » servent à pouvoir faire un lien de communication entre le micro-contrôleur et l’application Android. Le byte « UP/DOWN » indique qu’un ou plusieurs évènements non autorisés ont eu lieu, et sert aussi à garder cet information en mémoire lorsqu’on reset le µC. Le byte « RF\_Change » sert à l’application Android à indiquer au µC qu’elle a changé des bytes de configuration ou qu’elle veut reseter le programme. Enfin, le byte « Memory Full » indique que la mémoire de l’EEPROM est pleine.

Nous avons aussi changé le format des données dans le payload des NDEF records. Avant les données pour un évènement étaient stockées sous forme de texte, qui était simplement affiché lors de la lecture par l’application Android de ST. Le stockage de données texte prend de la place, nous avons donc pensé à passer en format binaire, avec une structure particulière pour chaque évènement. Pour tous les évènements (température ou accélération), le payload commence par la date et l’heure. Ensuite nous avons prévu un byte « Type\_messsage » qui indique le type d’évènement (0x01 pour l’accéléromètre et 0x02 pour la température). Ensuite pour chaque évènement, soit on va indiquer la température, ou l’accélération et le retournement. La structure de la mémoire de l’EEPROM est représentée à la Figure 11.(pour plus de clarté se référer au document Excel joint au rapport).

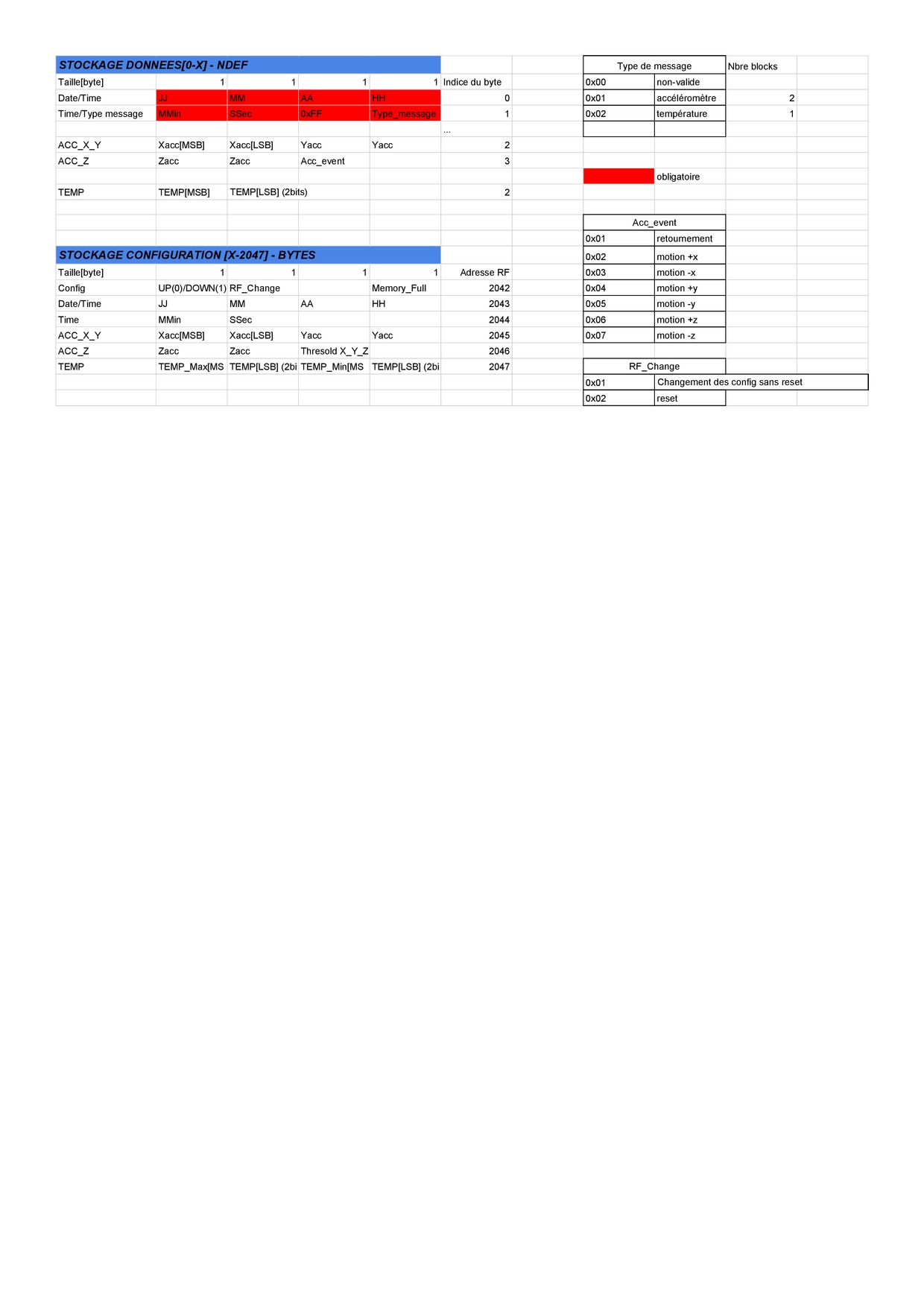


Figure : Organisation de la mémoire EEPROM

Comme nous travaillons au format binaire, il a aussi fallu changer le type de payload du NDEF record. On utilise maintenant le TNF 0x02 qui correspond au « Media-type », avec le type à « application/octet-stream ». Cela a demandé un traitement différent des données pour la construction du NDEF record.

**En résumé**, la mémoire de l’EEPROM est divisée en deux, avec une partie message NDEF et une partie « configuration ». Dans la partie message NDEF il y a **un seul message NDEF** balisé par un block TLV avant le message NDEF et par un bloc terminator TLV (0xFE) à la fin. Le message NDEF est **constitué de plusieurs NDEF records**, dont chaque record représente un évènement non autorisé. Le payload de chaque record contient des informations binaires sur l’évènement, dont la structure est représentée à la Figure 11.

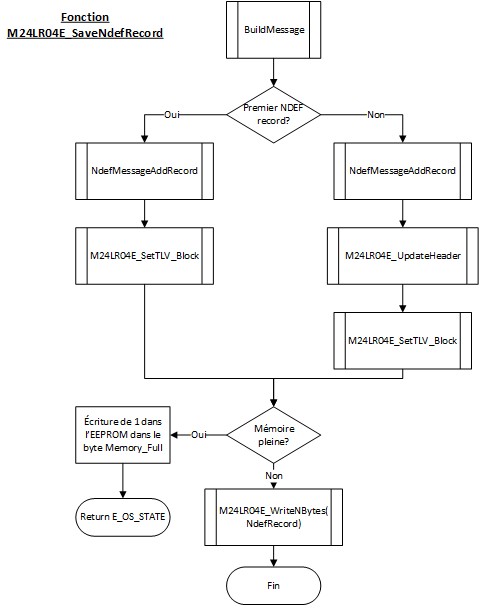
### Explication des librairies

Nous avons gardé la même structure qu’au départ :

* La librairie « M24LR04E\_R » s’occupe de tout ce qui est écriture/lecture dans l’EEPROM.
* La librairie « NDEFRecord » se comporte comme un objet instancié de manière statique qui représente le NDEF record à sauvegarder dans l’EEPROM.
* La librairie « NDEFMessage » s’occupe de la construction du record et du message NDEF, ainsi que la mise à jour des records au fur et à mesure de l’enregistrement des NDEF records et la mise à jour des blocks TLV.

Pour enregistrer un NDEF record lié à la température ou l’accéléromètre, on fait appel à la fonction *FXLS8471QSaveNdefMessage* ou *EMC1001SaveNdefMessage* qui va créer une structure de type *NDEFPayload\_t* et qui va remplir avec les données du payload. Il va ensuite faire appel à la fonction *M24LR04E\_SaveNdefRecord* qui va créer le NDEF record avec les données du payload et le stocker dans l’EEPROM.

Voici l’organigramme de la fonction *M24LR04E\_SaveNdefRecord*:



Tout d’abord on fait appel à la fonction *BuildMessage* qui va construire le payload (dans un tableau) à partir des données de la structure *NDEFPayload\_t* envoyée en paramètre à la fonction.

Ensuite on teste si c’est le premier NDEF record qu’on va stocker.   
Si oui, on va appeler les fonctions *NdefMessageAddRecord* et *M24LR04E\_SetTLV\_Block* en leur indiquant que c’est le premier NDEF record. La fonction *NdefMessageAddRecord* va construire le NDEF record. La fonction *M24LR04E\_SetTLV\_Block* va créer le premier block TLV et l’écrire dans l’EEPROM.  
Si ce n’est pas le premier NDEF record, on va appeler les fonctions *NdefMessageAddRecord* et *M24LR04E\_SetTLV\_Block* en leur indiquant que ce n’est pas le premier NDEF record, ainsi que la fonction *M24LR04E\_UpdateHeader*. La fonction *M24LR04E\_UpdateHeader* va mettre à jour le header du NDEF record précédent (pour indiquer que ce n’est plus le dernier).

Enfin, on teste si la mémoire de l’EEPROM va être pleine pour éviter un débordement sur la zone de configuration. Si l’EEPROM ne va pas être pleine, on va écrire le NDEF record dans l’EEPROM avec la fonction *M24LR04E\_WriteNBytes*.

## Android

L’objectif de l’application était de pouvoir lire les différents évènements survenus sur la carte, de pouvoir la configurer et de nettoyer les évènements de la mémoire pour une nouvelle livraison du colis.

L’application que nous avons créée a été développée sous Android Studio qui fonctionne sur le moteur d’IntelliJ et est basée sur l’application de chez STMicroelectronics.

Avant la conception de l’application, nous avons déterminé les différentes fonctions que l’on souhaitait implémenter dans celle-ci et des accès autorisés aux différents utilisateurs. Le diagramme de cas d’utilisation illustre les fonctions liées aux utilisateurs par la Figure 14.

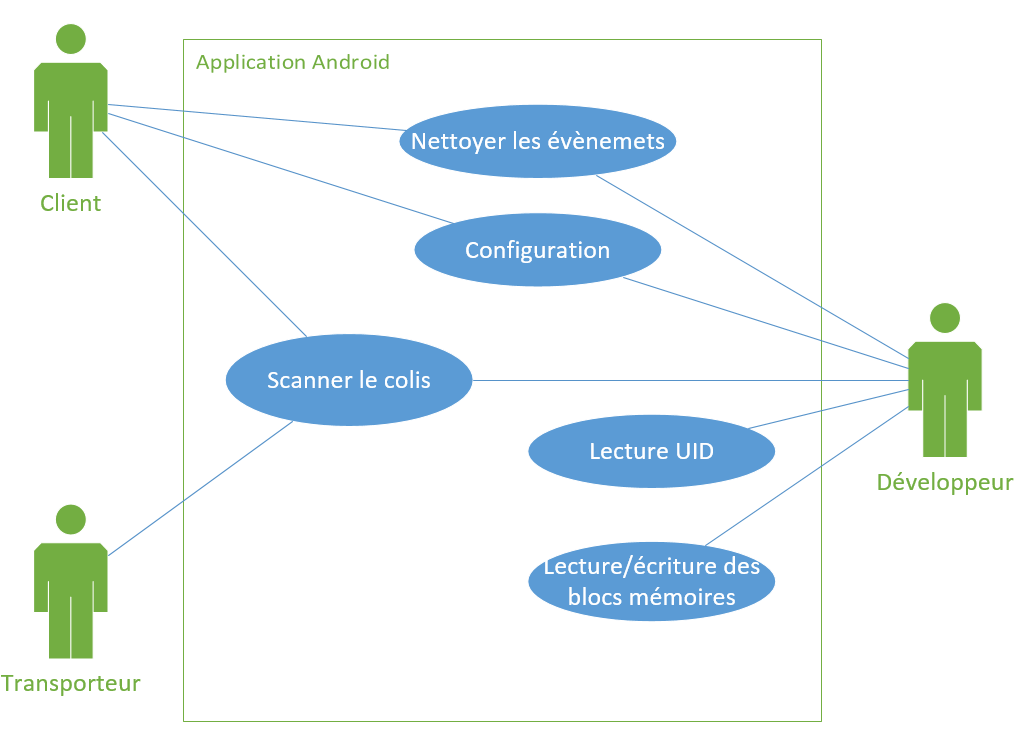


Figure : Diagramme de cas d'utilisation de l'application Android

Ce diagramme permet de spécifier les différentes fonctions à implémenter pour notre application et de déterminer le niveau d’accès à une fonction pour chaque utilisateur.

Le transporteur a la possibilité de scanner le colis pour déterminer si, avant le transport, le colis n’a subi aucun évènement non autorisé. Le client peut scanner le colis, lorsqu’il le réceptionne, ainsi que de le configurer pour un nouvel envoi. La configuration donne au client la possibilité d’imposer les conditions de transport pour le colis comme la sensibilité de l’accéléromètre et les valeurs de seuils minimum et maximum de température. Le développeur a accès à toutes les fonctions citées mais possède une série de fonctions supplémentaires pour le débogage et le teste de l’application. Ces fonctions comprennent la lecture ou l’écriture des blocs de la mémoire EEPROM sur carte et la lecture des informations relatives à la puce NFC appelée UID (Fabricant, taille de la mémoire, nombre de blocs mémoires, …).

Après avoir développé l’application, nous pouvons visualiser les interactions des différentes classes que compose l’application par la Figure 15.

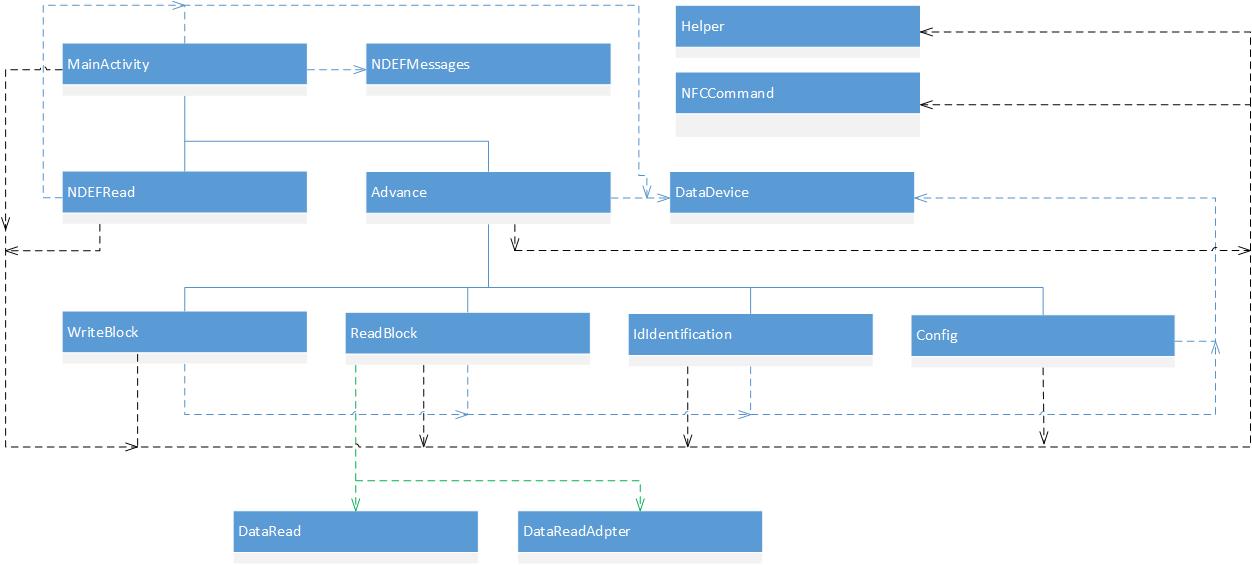


Figure : Diagramme de classe de l’application Android

Les classes ont besoin de certaines informations ou utilise des fonctions semblables. Pour éviter de la redondance du code, on peut centraliser ces fonctions pour qu’ils puissent être utilisables dans l’ensemble des autres classes. Le tableau suivant énonce l’utilité de chacune de ces classes.

|  |  |
| --- | --- |
| Nom de la classe | Description |
| MainActivity | C’est la classe principale. Elle permet le scan de la carte. |
| NDEFRead | Lorsqu’un colis est scanné et s’il y a eu un évènement non autorisé, c’est cette classe qui est appelée par la classe « MainActivity ». Il permet l’affichage du message de type NDEF sur l’écran du smartphone. |
| Advance | Cette classe donne accès au menu avancé. Il permet, par le bouton « Clean Event », le nettoyage des évènements survenus sur le colis. |
| WriteBlock | Accessible par la classe « Advance », il permet l’écriture d’un bloc mémoire. |
| ReadBlock | Accessible par la classe « Advance », il permet la lecture d’un ou plusieurs blocs mémoires. |
| IDIdentification | Accessible depuis la classe « Advance », il permet la lecture du tag NFC. Il donne des informations grâce à son identifiant UID. Les informations sont stockées dans la classe « DataDevice ». |
| Config | Accessible depuis la classe « Advance », il permet la configuration des paramètres de l’accéléromètre et des seuils de températures minimum et maximum. |
| DataDevice | Cette classe permet de récupérer des informations enregistrées au préalable par un scan du tag NFC. |
| DataRead | Il est utilisé par la classe « ReadBlock ». En scannant la mémoire, le numéro du block et sa valeur sont enregistrés dans une liste. |
| DataReadAdapter | Il est utilisé par le classe « ReadBlock ». Après avoir enregistré les valeurs des différents blocs mémoires dans une liste, cette classe permet l’affichage sous forme de liste déroulante à l’écran du smartphone. |
| NDEFMessages | Il est utilisé par la classe « MainActivity ». Cette classe contient plusieurs fonctions qui permettent des conversions des messages NDEF. |
| NFCCommand | Cette classe contient des fonctions qui permettent l’interaction entre le smartphone et le tag NFC. Il permet par exemple de lire ou écrire des blocs mémoires. |
| Helper | Cette classe contient des fonctions de conversion. |

# Résultats

## Liste des modifications apportées – nouvelles fonctionnalités

Voici la liste des modifications apportées – nouvelles fonctionnalités :

1. Android
   1. Visualisation des évènements avec l’heure à laquelle il s’est produit.
   2. Nettoyage des évènements qui se sont produits.
   3. Modification de la configuration :
      1. Seuil d’accélération.
      2. Seuil haut et bas de température.
   4. Fonctionnalités développeur : information sur l’appareil NFC, écriture dans des blocks ou lecture de blocks.
   5. Design graphique de l’application
2. PIC
   1. Fonctionnement en mode sleep : usage de la RTCC et timer pour réveiller le microcontrôleur -> optimisation de la consommation
   2. La librairie pour l’accéléromètre a été réécrite et en mode DEBUG elle affiche des informations sur l’UART. Il est possible de récupérer ces accélérations et d’activer la détection de chute libre/mouvement, changement d’orientation, simple/double tap, ainsi que de le mettre en SLEEP.
   3. La librairie pour le capteur de température le met en standby et elle permet de lire la valeur de la température.
   4. Réécriture de la tâche principale et nettoyage de code.
   5. Protection contre l’enlèvement de la batterie (grâce à l’écriture de la date et heure toutes les 10 minutes)

## PIC

### I²C

En analysant les trames à l’analyseur logique, il est apparu (voir Figure 16) qu’il y avait un délai important (en moyenne 5ms) entre deux trames I²C. Cela vient du driver I²C du PICOS. Il est donc important de le savoir car c’est 5 ms pendant que le PIC est potentiellement en mode RUN et dans l’optique d’optimisation de consommation, cela peut être un frein.

Toutefois, dans notre cas, lorsque l’on écrit dans la mémoire EEPROM, nous n’avons pas énormément à écrire dans le cas idéal où le transport se passe correctement. Dans le cas d’une accélération, il y aurait 15 bytes à écrire, et donc 75ms où le microcontrôleur serait en mode RUN. Il faudrait dans ce cas modifier le driver I²C fourni par PICOS pour qu’à l’intérieur de leur tâche, on le mette en mode SLEEP. Mais alors, l’intérêt d’utiliser le PICOS diminue si l’on commence à tout customisé pour l’application.

Cependant, d’après la datasheet de l’EEPROM, le temps maximal d’écriture est de 5 ms donc au final, c’est un mal nécessaire.

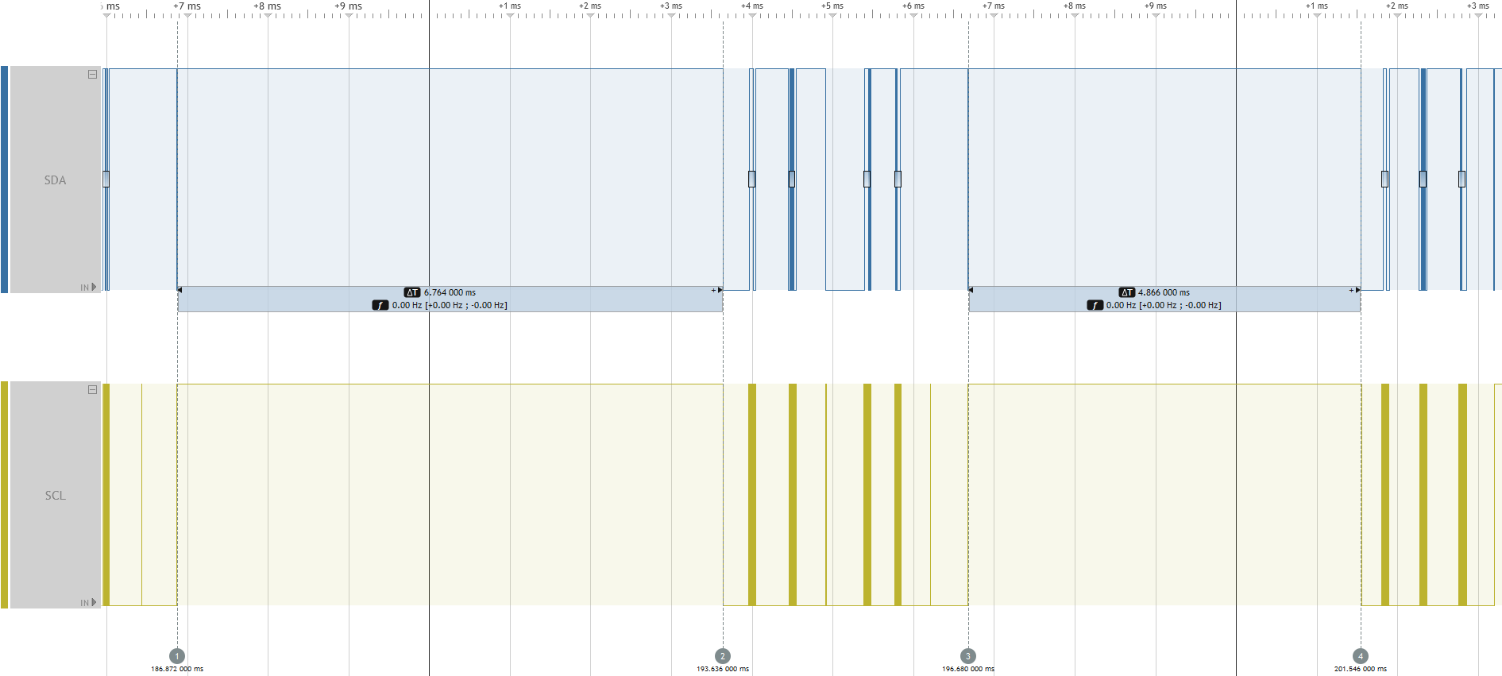


Figure Délai entre deux trames I²C

### Timer

L’utilisation d’un timer à la place d’un délai pour allumer-éteindre la LED a permis de diminuer le temps d’éveil du microcontrôleur de 100ms à moins d’une ms (cf. Figure 17) ! La capture a été faite sur un intervalle de temps de 5 secondes.

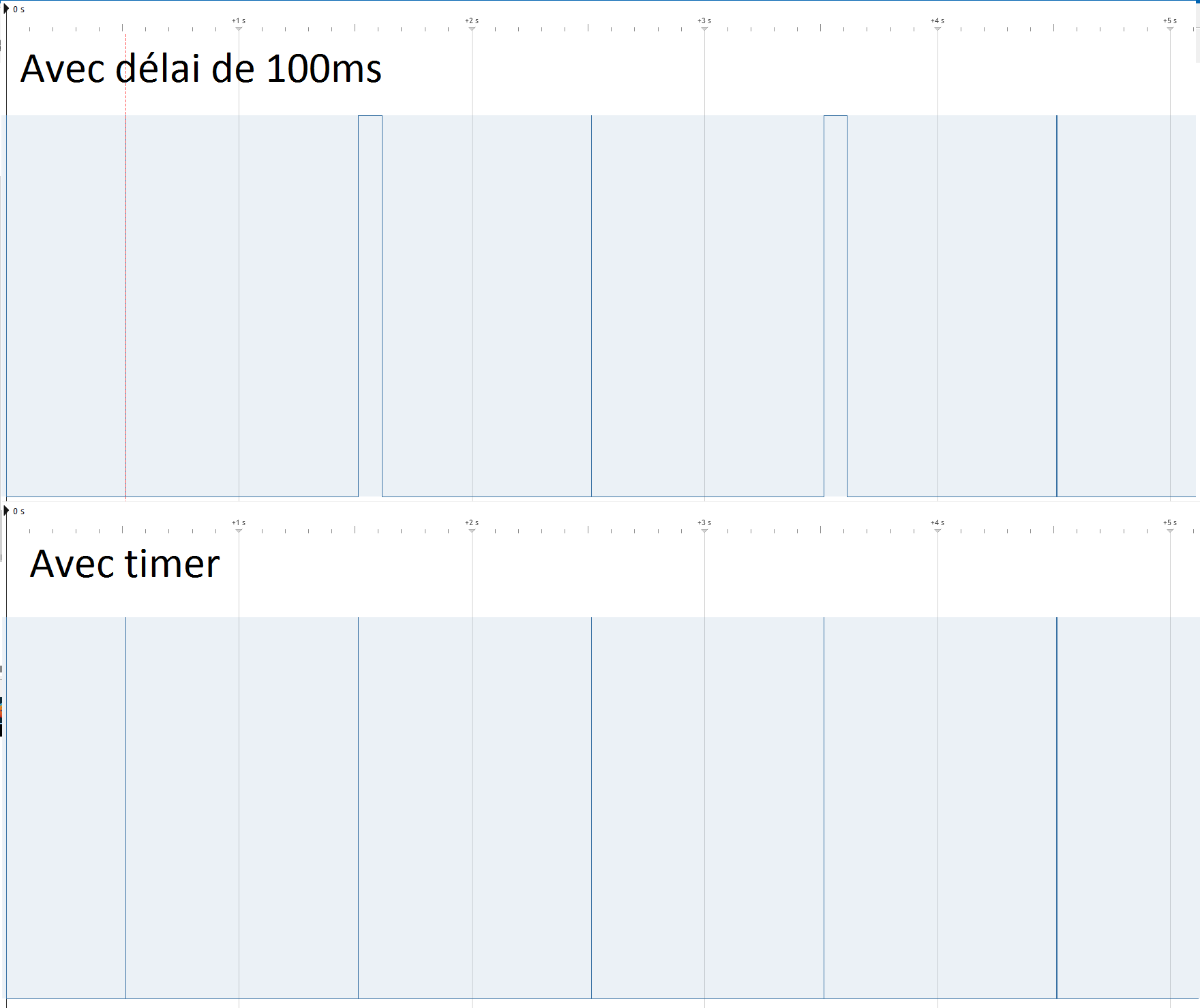


Figure Délai vs Timer comparaison

### Mode Sleep

L’utilisation du mode sleep a été préféré à celui du deep sleep. La première raison est technique : ce dernier est beaucoup plus contraignant. Le réveil du mode deep sleep n’est possible que par ces voies :

* INT0
* RTCC
* Watchog (utilisation non possible avec le PICOS)
* BOR[[2]](#footnote-2)

Or, dans notre cas, nous avons besoin du TIMER1 et d’une autre source d’interruption que le INT0 pour savoir si un utilisateur a été écrire dans l’EEPROM. Si les interruptions étaient groupées, il faudrait interroger l’accéléromètre et l’EEPROM pour savoir qui est la source d’interruptions.

La deuxième raison est exprimée au travers de la Figure 18. Celle-ci montre que dans le cas du PIC18F46J50, le mode deep sleep ne devient intéressant qu’à partir d’un temps au repos de 5,2 secondes. Or, dans notre cas, la LED doit clignoter régulièrement afin d’informer l’utilisateur que la carte fonctionne toujours correctement.

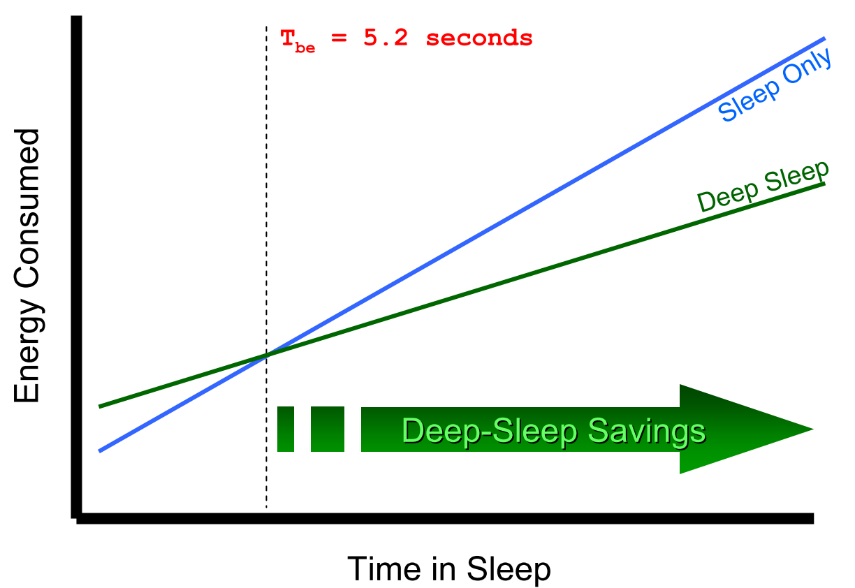


Figure 18 Comparaison sleep-deep sleep[[3]](#footnote-3)

La troisième raison est liée à la mémoire du microcontrôleur. Dans le cas du deep sleep, seul les 2 à 4 premiers bytes de la mémoire RAM seraient utilisables. Il faudrait donc obligatoirement aller lire à chaque réveil du microcontrôleur l’EEPROM externe. Dans notre cas, le temps de réveil ne serait pas critique. Il est de 1 à 10 µs en sleep et de 300 µs à 3 ms en deep sleep.

En contrepartie, le gain de consommation du PIC vis-à-vis des autres composants serait minime (cf. Tableau 1 du point 4.4). Le passage en mode deep sleep ferait gagner un peu plus de 3mA sur 40mA ce qui ferait une économie théorique de 7% de batterie. En pratique[[4]](#footnote-4), ce serait plutôt 5%.

### Accéléromètre

Lors des essais de détection d’accélération, nous nous sommes rendu compte que le capteur détectait des retournements alors que ce n’était que des accélérations. Et plus particulièrement, lors d’un agitement du capteur. Dans la note AN4068 de Freescale sur son accéléromètre : « If the device is set to go to sleep, reset the debounce counter before the device goes to sleep. This setting helps avoid long delays since the debounce will always scale with the current sample rate. » Mais en faisant ce qu’ils préconisaient, le capteur était trop sensible. Après plusieurs essais, nous sommes parvenus à avoir une valeur de *debounce* pour la détection de retournement permettant d’éviter ce genre de situation.

## Application Android

La conception de l’application n’a pas été facile. Nous avons rencontré plusieurs problèmes liés à la lecture des messages NDEF et de son affichage sur l’écran du smartphone. Nous avons aussi eu des problèmes dans la conversion de certaines données.

Après la résolution de ces problèmes, nous avons réussi à développer une application fonctionnelle et avec les différentes fonctions souhaitées tels que :

* La lecture de message NDEF ;
* L’effacement des évènements survenu sur le colis (Reset software) ;
* La configuration des valeurs de seuils de température et de la sensibilité de l’accéléromètre.

Du côté du développeur, nous avons implémenté des fonctions qui permettent de réaliser des tests sur la mémoire EEPROM :

* Lire/Ecrire sur la mémoire sous forme de block ;
* Lire les informations relatives au tag NFC (UID).

L’application se présente comme sur les captures d’écran ci-dessous. L’écran d’accueil de l’application se présente comme sur la Figure 19.

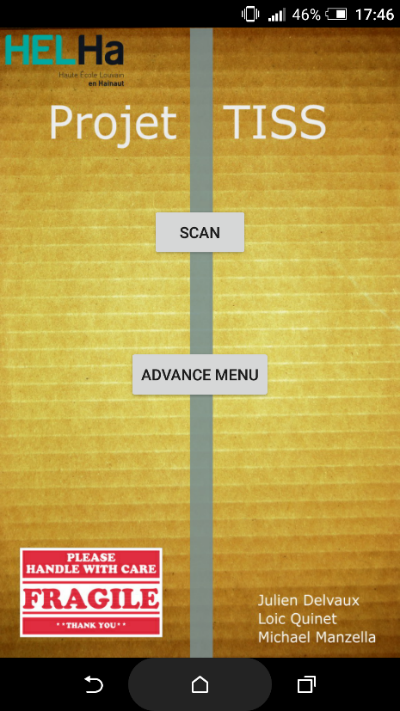


Figure : Ecran d'accueil de l'application Android

Cet écran permet le choix entre deux fonctionnalités : « Scan » et « Advance Menu ». Le premier bouton permet la lecture des évènements sur la carte. Si aucun évènement critique n’a eu lieu, l’application renverra comme message « Pas d’évènement critique ». S’il y a eu un évènement non autorisé sur le colis, l’application va scanner la mémoire EEPROM avec les messages correspondants aux différents évènements et l’affichera sur un autre écran. Le second bouton, « Advance menu », permet l’accès à des fonctions avancées (voir Figure 20).

|  |  |
| --- | --- |
| C:\Users\michael\AppData\Local\Temp\Temp2_documents-export-2016-04-24.zip\Screenshot_20160424-174655.png  Figure : Menu avancée (Advance menu) | C:\Users\michael\AppData\Local\Temp\Temp2_documents-export-2016-04-24.zip\Screenshot_20160424-174703.png  Figure : Ecran de configuration |

Ce menu permet plusieurs fonctions découpées en deux parties : configuration et développeur. La partie configuration est orientée « client » et permet par le premier bouton, « Clean Event », de réinitialiser la carte pour l’envoi d’un nouveau colis. Le second bouton, « Config », permet de régler les paramètres de sensibilité de l’accéléromètre (avec comme unité des mg) et les seuils de température du capteur en °C (voir Figure 21).

L’autre partie de l’écran du menu avancé, est dirigé plus coté « développer », pour les tests de lecture et écriture dans la mémoire EEPROM de la carte (Voir les figures suivantes).

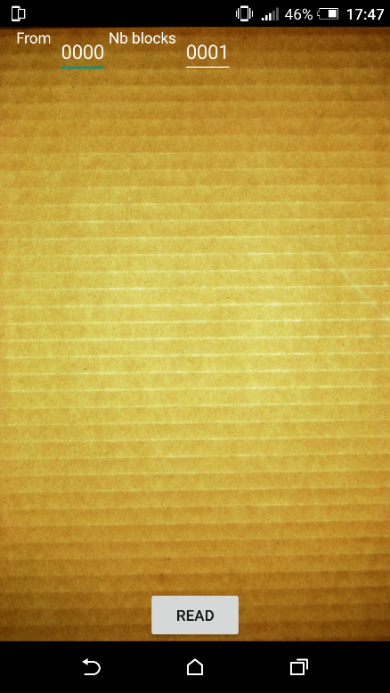


Figure : Ecran de lecture des blocs mémoires

La Figure 22 montre la lecture d’un à plusieurs blocs de la mémoire. La zone « From » donne à l’application, à partir de quels blocs mémoires commencés à lire. La zone « Nb blocks » donne le nombre de blocs à lire à partir de la valeur donné dans la zone précédente. Après avoir complété les deux paramètres, l’appui sur le bouton « Read » permet de visualiser le contenu de la mémoire.

|  |  |
| --- | --- |
| C:\Users\michael\AppData\Local\Temp\Temp2_documents-export-2016-04-24.zip\Screenshot_20160424-174723.png  Figure : Ecran d'écriture des blocs dans la mémoire EEPROM | C:\Users\michael\AppData\Local\Temp\Temp2_documents-export-2016-04-24.zip\Screenshot_20160424-174730.png  Figure : Ecran d'affichage des informations UID |

Le second bouton « Write Block », donne la Figure 23. Nous avons accès à deux zones de configuration. La première est le numéro du bloc de ma mémoire que l’on souhaite modifier. Le second est découpé en 4 parties. Le bloc de mémoire doit être configuré sur 4 bytes avec des valeurs hexadécimales. La valeur du bloc mémoire est effectuée après l’appui sur le bouton « Write ».

Le troisième bouton, « UID », permet d’avoir des informations sur le tag NFC comme son fabricant, le modèle, la taille de la mémoire EEPROM, le nombre de blocs mémoires,… (Voir la Figure 24)

## Consommation

Le Tableau 1 ci-dessous reprend la consommation théorique des différents composants utilisés dans le circuit.

|  |  |  |  |
| --- | --- | --- | --- |
| Référence | Type | Standby current [µA] | Supply current [µA] |
| PIC18F46J50 | Microcontrôleur | 3,2 (SLEEP) | 2800 (RUN) |
| EMC1001 | Température | 4,8 | 47 |
| FXLS8471Q | Accéléromètre | 2 | 100 |
| M24LR64E | EEPROM | >30[[5]](#footnote-5) | >200 |

Tableau Consommation théorique des différents composants

On remarque que le mode standby ou SLEEP est très intéressant pour l’optimisation de la consommation. Un facteur 10 est appliqué pour les capteurs tandis que c’est un facteur de l’ordre de 1000 pour le microcontrôleur !

Le Tableau 2 reprend la consommation théorique et celle mesurée dans différents cas. Le premier était lorsque l’on utilisait une fonction de délai pour faire clignoter la LED. La LED clignotait 100 ms toutes les 2 secondes. Suivant les évènements qui se sont passés, la LED clignotera rouge (dans le cas où il y a eu un retournement par exemple) et verte lorsque rien ne s’est passé. La consommation est relativement importante due au fait que le microcontrôleur reste en mode RUN pendant que la LED est allumé.

|  |  |  |
| --- | --- | --- |
|  | Courant théorique [µA] | Courant mesuré [µA] |
| Standby | >40 | 62 |
| RUN(LED rouge) | 2840+conso led | 3140 |
| RUN(LED verte) | 2840+conso led | 4450 |

Tableau Consommation du prototype avec l'utilisation d'une fonction delay

Une des premières voies d’amélioration est de le mettre en SLEEP pendant ce temps-là. Sa consommation est réduite de ± 2.8 mA dû au fait qu’il ne soit plus en mode RUN. Le Tableau 3 reprend ces résultats.

|  |  |  |
| --- | --- | --- |
|  | Courant théorique [µA] | Courant mesuré [µA] |
| Standby | >40 | 62 |
| SLEEP + LED rouge | 40+conso led | 450 |
| SLEEP + LED verte | 40+conso led | 1900 |

Tableau Consommation du prototype avec l'utilisation d'un timer

L’estimation de la durée de vie maximale de la batterie est reprise dans le Tableau 4. Celle-ci a été calculée en tenant compte que la LED est allumée pendant 50ms et que le microcontrôleur est en sleep le reste du temps. Dans le 3ème cas, la LED n’est plus allumée que pendant 25ms ce qui est largement encore visible. Le trajet a été supposé parfait : aucun évènement d’accéléromètre et de température. L’écriture dans l’EEPROM toutes les dix minutes de la date et l’heure n’a pas été prise en compte.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Run+LED verte | Sleep+LED verte(50ms) | Sleep+LED verte(25ms) |
| Consommation [mA] | 0.2814 | 0.1539 | 0.1064 |
| Durée [heures] | 426.4 | 779.7 | 1127.8 |
| Durée [jours] | 17jours 3/4 | 32jours 1/2 | 47 jours |

Tableau Durée de vie d'une batterie CR1632

Le Tableau 5 reprend l’estimation de durée la batterie dans le cas où la LED clignoterait en rouge (25ms) et le cas où la pile utilisée serait une CR2032 (220mAh) contre l’actuelle CR1632(120mAh) pour une durée de LED rouge allumée 25 ms (2) et une LED verte allumée pendant 50ms (3) et 25ms (4).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Sleep+LED rouge | Sleep+LED rouge(2) | Sleep+LED verte(3) | Sleep+LED verte(4) |
| Consommation [mA] | 0.06045 | 0.06045 | 0.1539 | 0.1064 |
| Durée [heures] | 1985 | 3639.3 | 1429.5 | 2067.6 |
| Durée [jours] | 82 jours 3/4 | 151 jours 1/2 | 59 jours 1/2 | 86 jours |

Tableau Durée de vie d'une batterie CR1632 avec LED rouge et d'une batterie CR2032 dans le cas d'un clignotement vert de 50ms et 25ms

# Conclusion

Ce projet nous a confronté à de nouveaux concepts, tels que la programmation de PIC, l'utilisation d'un PICOS, la communication NFC et le fonctionnement de trames NDEF, l'utilisation d'un accéléromètre et la programmation sur Android. Nous avons fait de nombreuses recherches avant de pouvoir nous lancer dans le projet afin de comprendre leur fonctionnement et la manière dont ils ont été implémentés. Il a fallu une réelle coordination et gestion de l'équipe pour arriver à terminer ce projet.

Dès le début du projet, les rôles de chacun ont été définis en fonction des affinités. Nous avons aussi créé une équipe sur Slack (plate-forme de communication collaborative qui permet centraliser le suivi et la gestion d'un projet) afin de pouvoir communiquer en dehors des séances de projet, et nous avons échangé pas loin de 1000 messages pour le projet.

Le gestionnaire de sources (Git) a permis d’améliorer la rapidité dans le *merge* (fusion) pour la partie du PIC. Après avoir compris son fonctionnement, il nous fallait moins d’une minute pour réunir nos codes sans se soucier des conflits ou de possibles oublis.

Cela a aussi demandé beaucoup de temps de se plonger dans le code déjà existant, au niveau du PIC et d'Android, d'autant plus que le programme en PIC n'était quasi pas commenté et très peu structuré. Mais au final nous sommes parvenus à obtenir un projet fonctionnel et respectant le cahier des charges demandé.

Évidemment il reste encore plusieurs améliorations à apporter. Voici quelques propositions :

* Détection de batterie faible grâce à l’usage du HLVD (*High/Low Voltage Detect*)
* Utilisation du mode « deep sleep » pour essayer d’encore diminuer un peu plus les consommations
* Switch pour facilement mettre le circuit hors tension et éviter une consommation de batterie sans avoir à enlever la pile ou mettre un isolant entre la pile et le contact.
* Circuit de protection contre une mise à l’envers de la pile.
* Disparition du bouton reset qui ne semble plus nécessaire puisque on reset par software
* Investiguer sur les LED afin de trouver une LED qui ne consomme pas trop quand elle clignote en vert
* Changer la pile pour une CR2032 : plus facile à trouver (plus standard) et autonomie plus grande (220mAh contre 120mAh pour la CR1632).
* Sécuriser le prototype en mettant un mot de passe afin d’éviter que n’importe qui puisse écrire dans l’EEPROM
* En cas de développement d’un autre PCB prototype, il serait intéressant d’avoir des broches mâles soudées, des points de masse ainsi que des labels pour chaque broche

Du point de vue de l’application Android :

* Amélioration de l’interface utilisateur de l’application Android
* Protéger certaines fonctionnalités de l’application par un mot de passe (Accès développeur) ;
* Lecture du fichier contenant les messages NDEF lu précédemment dans l’application ;
* Amélioration du code pour optimiser la taille de l’application.

# Bibliographie

1. ADAFRUIT (consulté le 03.03.16), *About the NDEF Format*  
   <https://learn.adafruit.com/adafruit-pn532-rfid-nfc/ndef>
2. GERBAUD Guillaume (consulté le 01.03.16), *NFC IV – Les types de messages du NFC Forum*, <http://blog.zenika.com/2012/04/24/nfc-iv-les-types-de-messages-du-nfc-forum-wkt/>
3. Microchip, “*Emerging Trends in Embedded Power Management: Microcontrollers, Memory & Analog*”
4. Microchip, Datasheet sur le composant «EMC1001»
5. Microchip, Datasheet sur le composant «PIC18F46J50 »
6. NFC FORUM (consulté le 01.03.16), *NFC Data Exchange Format (NDEF)*<http://www.eet-china.com/ARTICLES/2006AUG/PDF/NFCForum-TS-NDEF.pdf>
7. NFC FORUM (consulté le 03.03.16), *Type 4 Tag Operation Specification*, <http://apps4android.org/nfc-specifications/NFCForum-TS-Type-4-Tag_2.0.pdf>
8. NXP Semiconductors, Datasheet sur le composant «FXLS8471Q»
9. STMicroelectronics, Datasheet sur le composant «M24LR64E-R»

1. La RTCC peut réveiller le microcontrôleur toutes les 0.5 secondes, secondes, 10 secondes, minutes, 10 minutes … Dans notre cas, nous l’avons réglé sur 1 seconde. [↑](#footnote-ref-1)
2. Le BOR (*Brown-out Reset*) est un circuit qui monitore la tension en le comparant à une valeur de seuil. Lorsque cette tension est plus petite que le seuil, le BOR est activé. [↑](#footnote-ref-2)
3. Cette figure est issue d’un document de Microchip : « *Emerging Trends in Embedded Power Management: Microcontrollers, Memory & Analog »* [↑](#footnote-ref-3)
4. Cf. note en bas de page n°5, section 4.4. [↑](#footnote-ref-4)
5. Cette valeur est donnée pour une tension d’alimentation de 2.5 V. A 5.5 V, ce courant passe à 100µA. Il est donc impossible de donner une valeur précise dû au fait que la relation entre consommation de courant et tension d’alimentation n’est pas linéaire. [↑](#footnote-ref-5)