|  |  |
| --- | --- |
| 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](#_Toc450206369)

[2. Analyse de l’existant et cahier des charges 2](#_Toc450206370)

[2.1. Analyse de l’existant 2](#_Toc450206371)

[2.2. Cahier des charges 2](#_Toc450206372)

[2.3. PICOS18 2](#_Toc450206373)

[2.4. Capteurs 3](#_Toc450206374)

[2.4.1. Température 3](#_Toc450206375)

[2.4.2. Accéléromètre 3](#_Toc450206376)

[2.5. EEPROM 4](#_Toc450206377)

[2.6. Android 4](#_Toc450206378)

[3. Développement 5](#_Toc450206379)

[3.1. PICOS18 5](#_Toc450206380)

[3.2. Accéléromètre 7](#_Toc450206381)

[3.3. Capteur température 9](#_Toc450206382)

[3.4. Mémoire EEPROM 9](#_Toc450206383)

[3.5. Android 9](#_Toc450206384)

[4. Résultats 10](#_Toc450206385)

[4.1. Liste des modifications apportées – nouvelles fonctionnalités 10](#_Toc450206386)

[4.2. PIC 10](#_Toc450206387)

[4.2.1. I²C 10](#_Toc450206388)

[4.2.2. Timer 11](#_Toc450206389)

[4.2.3. Mode Sleep 12](#_Toc450206390)

[4.2.4. Accéléromètre 13](#_Toc450206391)

[4.3. Application Android 13](#_Toc450206392)

[4.4. Consommation 13](#_Toc450206393)

[5. Conclusion 15](#_Toc450206394)

[6. Bibliographie 16](#_Toc450206395)

# Introduction

# 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.

## Cahier des charges

Nos objectifs pour ce projet étaient les suivants :

1. Garder le PICOS18.
2. Optimiser la consommation afin de faire durer la batterie le plus longtemps possible.
3. Ajouter des fonctionnalités au prototype ; entre autre l’écriture de tous les évènements non-autorisés et l’amélioration de la détection de chocs et de retournements.
4. Développer l’application Android customisée pour ce projet.
5. Faire clignoter la LED à intervalles réguliers.

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.

## 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, celle-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.

## 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 1. 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 défini (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.

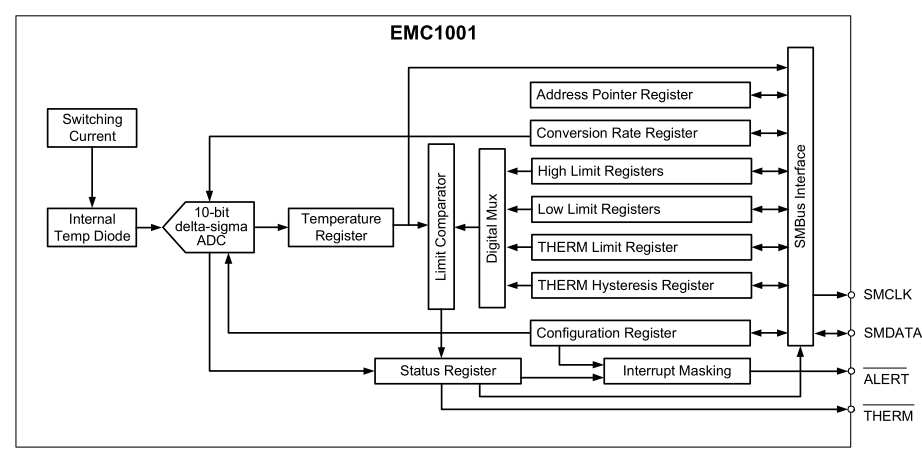


Figure 1 Schéma block interne du capteur de température

## 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 2. 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.

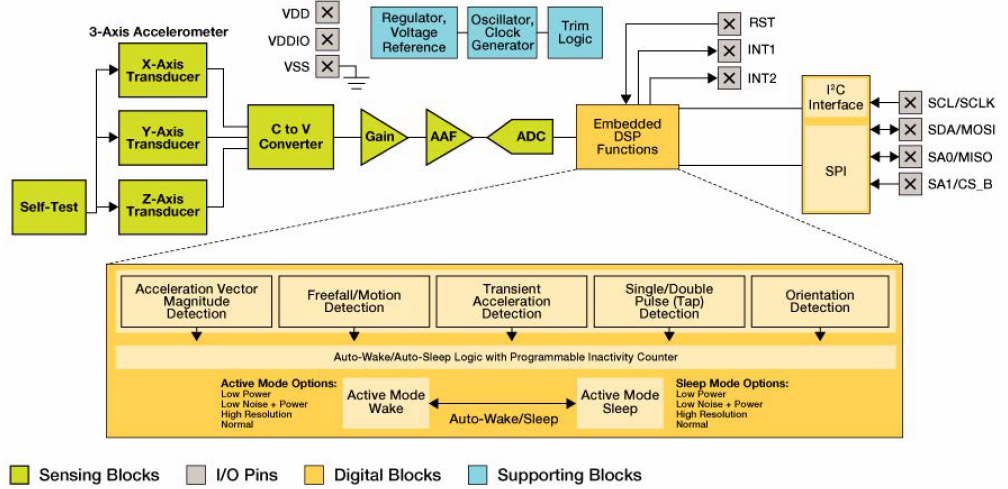


Figure 2 Schéma fonctionnement FXLS8471Q

## EEPROM

Les librairies déjà existantes en rapport à l’EEPROM et l’écriture de message NDEF n’a pas été facile à prendre en main. D’une part, il a fallu comprendre l’utilisation du driver I²C du PICOS18 afin de communiquer avec l’EEPROM. D’une 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 fonctionne 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 une sorte d’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.

## Android

# Développement

## PICOS18

Le microcontrôleur utilise le PICOS18 pour fonctionner. Dans notre cas, nous avons une tâche principale (*TASK\_Main*) et deux tâches secondaires nécessaire 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 3. 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 possible :

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.
3. Accéléromètre : Lorsqu’un retournement ou une détection d’accélération a été détectée, 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 3 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 grand 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 illustré par la Figure 4 :

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 5.



Figure 4 Initialisation de l'accéléromètre



Figure 5 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.

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 maximum 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 stocké sous forme de texte, qui était simplement affiché lors de la lecture par l’application Android de ST. Le stockage de données texte prends 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 représenté à la figure 6.

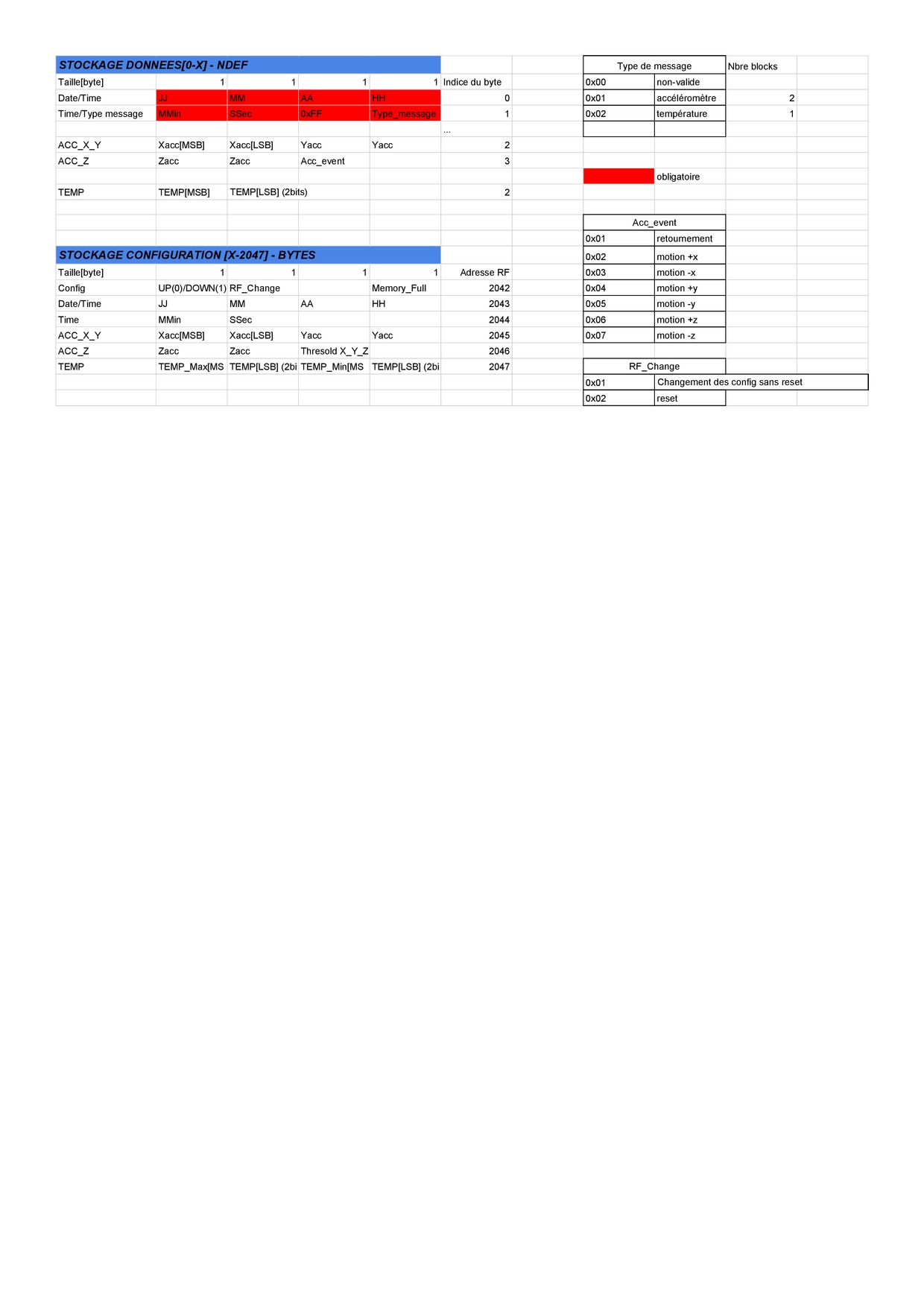


Figure 6: Organisation de la mémoire EEPROM

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 corresponds au « Media-type », avec le type à « application/octet-stream ». Cela a demandé un traitement différent des données pour la contruction du NDEF record.

### Explication des librairies

## Android

# 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 apparût (voir Figure 6) 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.

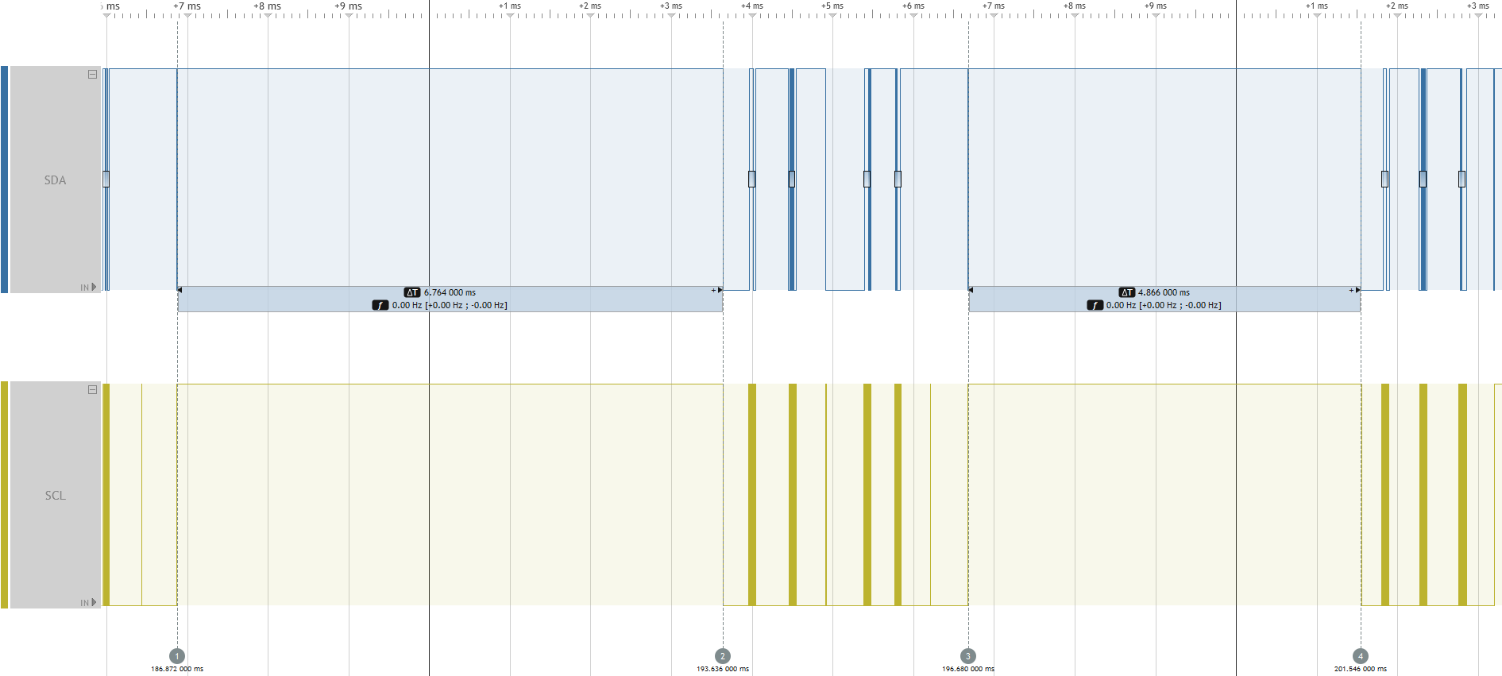


Figure 7 Délai entre deux trames I²C

### 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 7) ! La capture a été faite sur un intervalle de temps de 5 secondes.

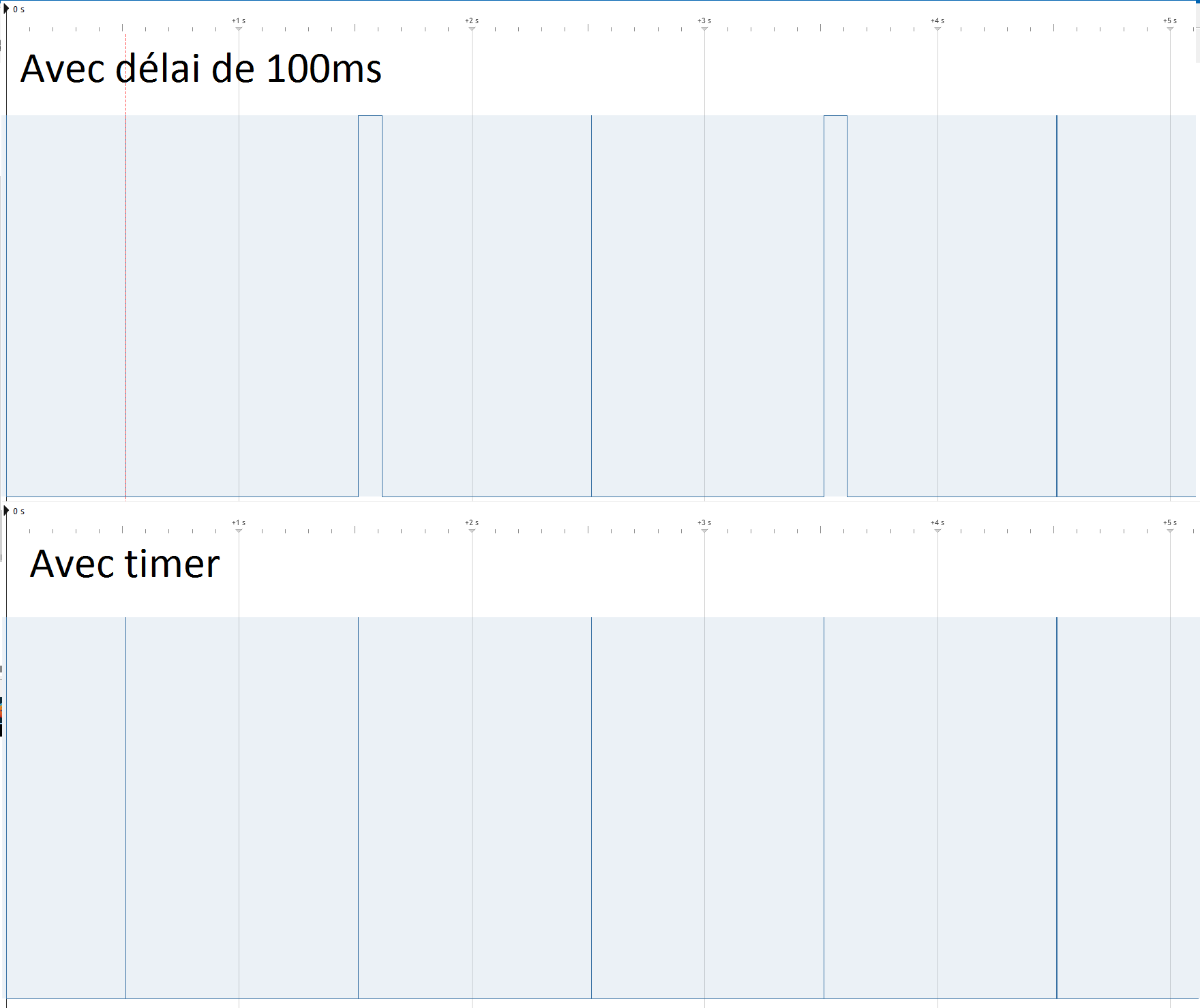


Figure 8 Délai vs Timer comparaison

### 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 8. 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.

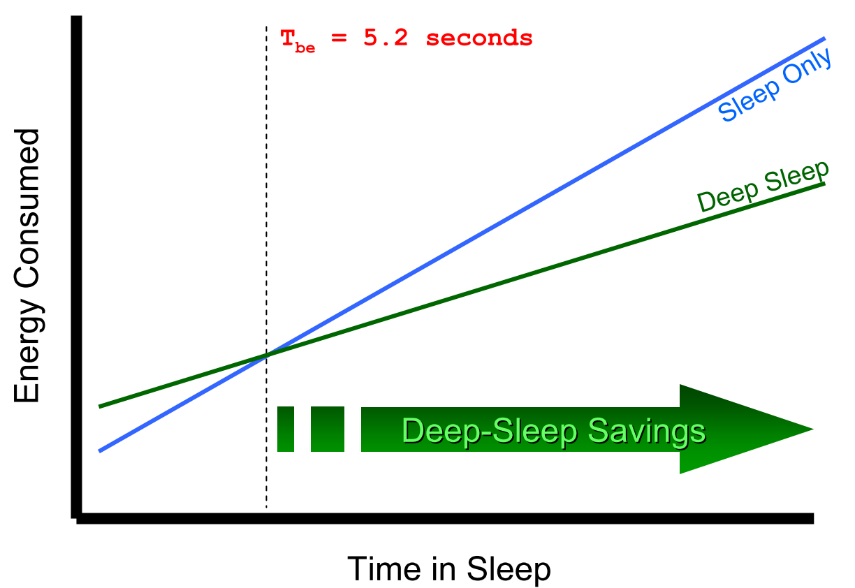


Figure 9 Comparaison sleep-deep sleep[[3]](#footnote-3)

La troisième raison est liée à la mémoire du microcontrôleur. Dans le cas du deep sleep, seul les 2 à 4 permiers 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 rendus 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

## 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 1 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 différent 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 passé, 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 allumée

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

Tableau 2 Consommation du prototype avec l'utilisation d'une fonction delay

Une des première voie 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 3 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 4 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 5 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

Liste d’améliorations possible :

* Détection de batterie faible grâce à l’usage du HLVD (*High/Low Voltage Detect*).
* Switch pour facilement « éteindre » le circuit 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.
* Mode Deep Sleep ?
* Disparition du bouton reset ?
* 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).
* Améliorer la sécurité du prototype ? 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 pins mâles soudés, des points de masse ainsi que des labels pour chaque pin.

# Bibliographie

|  |  |
| --- | --- |
| [ 1 ] | Microchip, “*Emerging Trends in Embedded Power Management: Microcontrollers, Memory & Analog*”. |

Datasheets des différents composants.

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)