|  |  |
| --- | --- |
| 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](#_Toc449644254)

[2. Analyse de l’existant et cahier des charges 2](#_Toc449644255)

[2.1. Analyse de l’existant 2](#_Toc449644256)

[2.2. Cahier des charges 2](#_Toc449644257)

[2.3. PICOS18 2](#_Toc449644258)

[2.4. Capteurs 3](#_Toc449644259)

[2.4.1. Température 3](#_Toc449644260)

[2.4.2. Accéléromètre 3](#_Toc449644261)

[2.5. EEPROM 4](#_Toc449644262)

[2.6. Android 4](#_Toc449644263)

[3. Développement 5](#_Toc449644264)

[3.1. PICOS18 5](#_Toc449644265)

[3.2. Accéléromètre 7](#_Toc449644266)

[3.3. Capteur température 9](#_Toc449644267)

[3.4. Mémoire EEPROM 9](#_Toc449644268)

[3.5. Android 9](#_Toc449644269)

[4. Résultats 10](#_Toc449644270)

[4.1. Liste des fonctionnalités implémentées 10](#_Toc449644271)

[4.2. PIC 10](#_Toc449644272)

[4.3. Application Android 12](#_Toc449644273)

[4.4. Consommation 12](#_Toc449644274)

[4.5. Conclusion 13](#_Toc449644275)

[5. Conclusion 14](#_Toc449644276)

[6. Bibliographie 15](#_Toc449644277)

# 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 du PICOS, était d’apprendre à manipuler le PIC avec lequel nous n’avions aucune connaissance.

Au niveau hardware, la carte électronique était déjà réalisée et avait été testée par Denis Alaime. Lors de notre 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. 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 :

* Mike 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. 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 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.

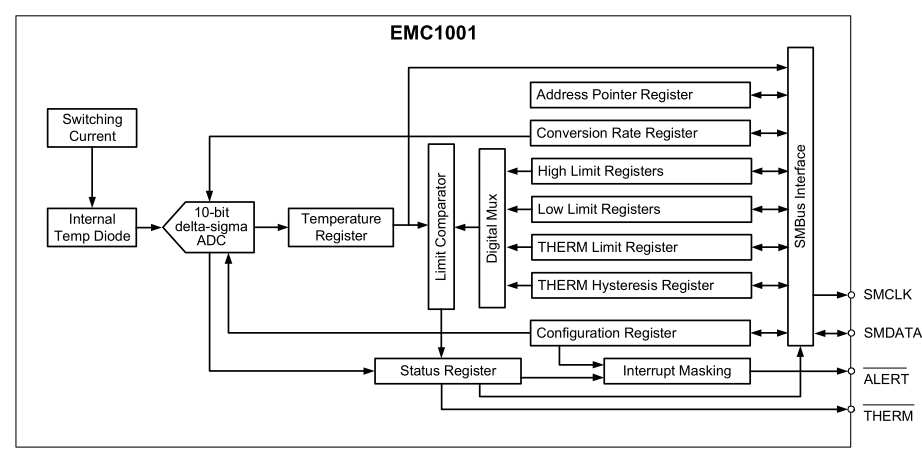


Figure Schéma block interne du capteur de température

## 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.

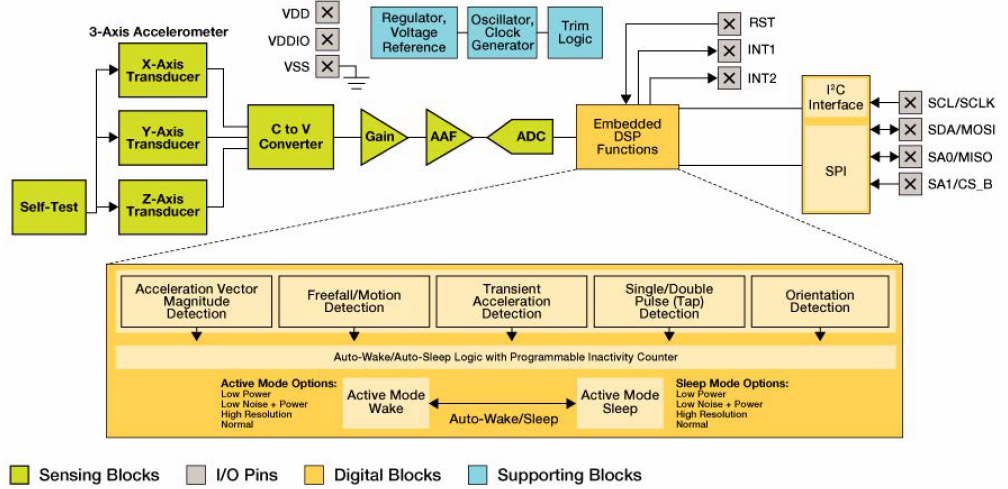


Figure Schéma fonctionnement FXLS8471Q

## EEPROM

## 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).

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, on va passer le microcontrôleur en mode *RUN* afin de traiter celle-ci. Il y a trois sources d’interruptions possible :

1. RTCC : la *Real-Time Clock and Calendar* 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) 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.
3. Accéléromètre : Lorsqu’un retournement ou une détection d’accélération a été détectée.

Le clignotement de la LED est effectué grâce au lancement d’un TIMER afin de réduire de manière significative la consommation du microcontrôleur.



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 ce n’était pas lisible et facile à 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 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.

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

## Android

# Résultats

## Liste des fonctionnalités implémentées

Voici la liste des fonctionnalités implémentées :

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.
2. PIC
   1. Fonctionnement en mode sleep : usage de la RTCC et timer pour réveiller le microcontrôleur.
   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.

## PIC

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 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. De plus, d’après la datasheet de celle-ci, le temps maximal d’écriture est de 5 ms donc au final, cela ne nous posait pas de problème.

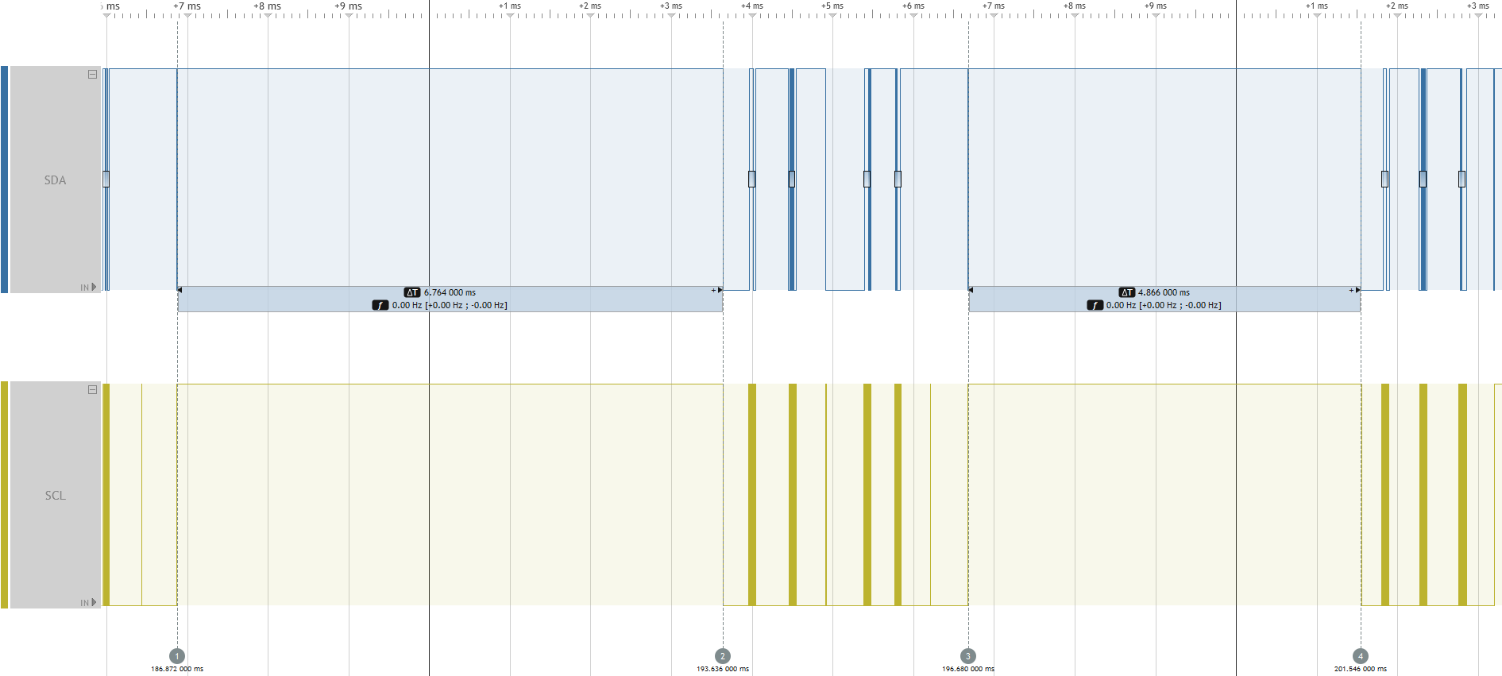


Figure 6 Délai entre deux trames I²C

Ensuite, 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 5 secondes.

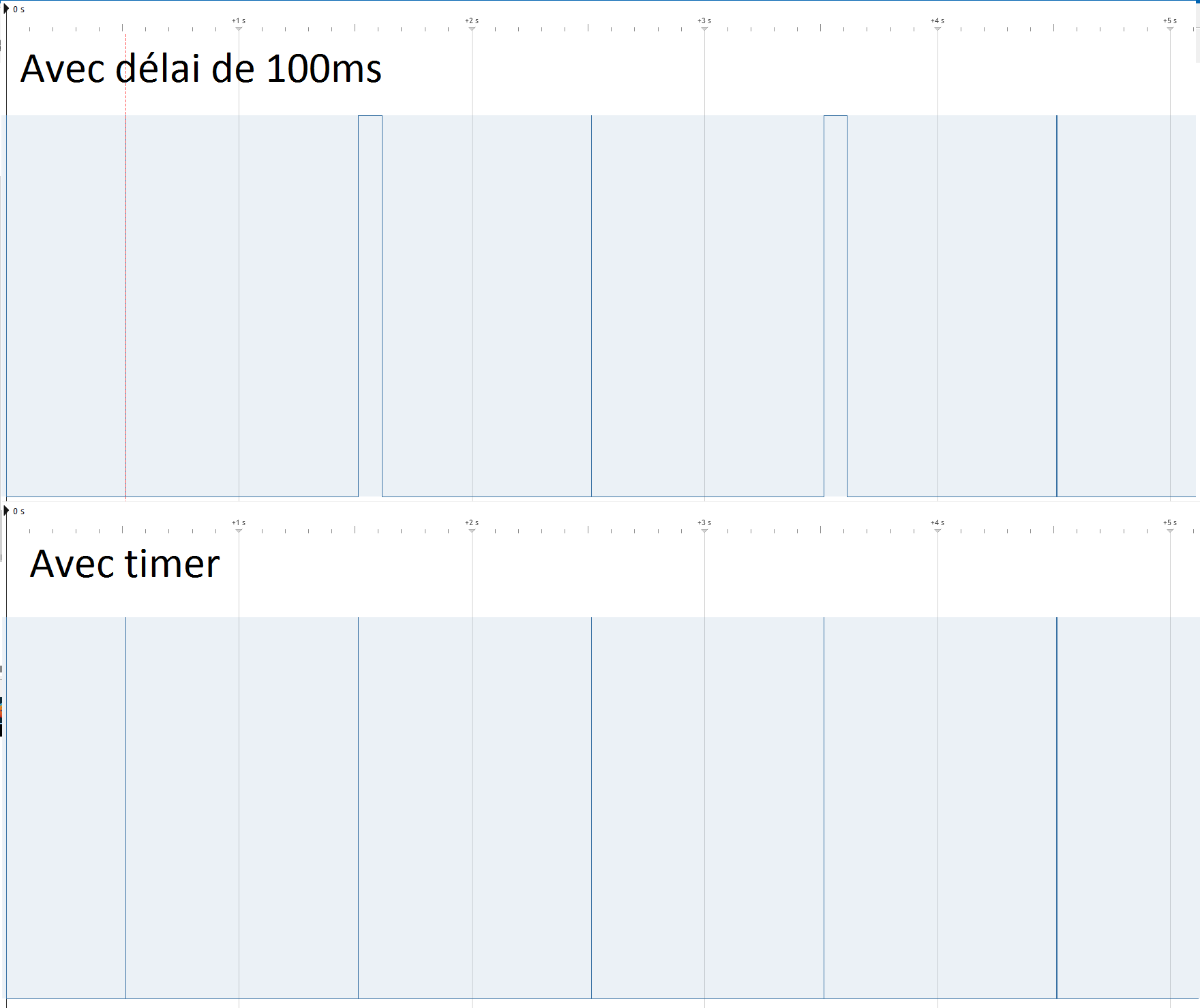


Figure 7 Délai vs Timer comparaison

## Application Android

## Consommation

Le tableau 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[[1]](#footnote-1) | >200 |

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 suivant 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.

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

La consommation est relativement importante due au fait que le microcontrôleur reste en mode RUN pendant que la LED est allumée. 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.

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

L’estimation de la durée de vie maximale de la batterie est reprise dans le tableau ci-dessous. 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 |

Le tableau suivant 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 de 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 |

## Conclusion

# Conclusion

# Bibliographie

Références principales :

|  |  |
| --- | --- |
| [ ] | L. Jones, “Coordination and Control for Multi-Quadrotor UAV Mission”, Thesis, Naval Postgraduate School, 2012. |

1. 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-1)