POLYTECH GRENOBLE

UNIVERSITÉ Grenoble Alpes

Rapport final du projet IESE : Optimisation de la Consommation Énergétique de Microcontrôleurs

Marcelo Viana da Silva, Qian Likun, Li Yihao

04.2109

Sommaire

1. Introduction

Actuellement, réaliser des objets autonomes en énergie devient très important. Outre les moyens de récupération d'énergie de l'environnement, il faut aussi faire en sorte que les objets connectés soient les moins énergivores possibles. Dans ce projet nous allons étudier la consommation sur une carte *Nucleo Low Power*. Selon le projet déjà amorcé l'année dernière, nous allons explorer et caractériser les différents modes de basse consommation d'une carte STM32L073RZ et réduire la consommation de la carte autant que possible.
Dans le projet précédent, la consommation de la carte en mode Standby était de 1,4 mA. Un des nos objectifs est donc d'arriver à une consommation au moins inférieure à celle de l'année dernière.

1.1 Organisation

Un plan réalisable peut améliorer l'efficacité de notre travail. Nous avons créé un schéma visuel pour le planning du projet sur Gantt (Figure 1).

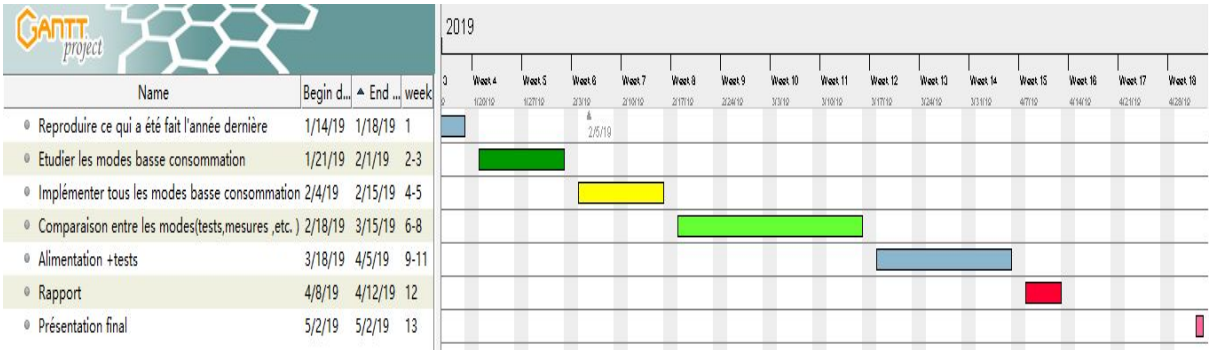

Figure 1 : Planning du projet sur Gantt

2. Choix du mode de basse consommation

Tout d'abord, nous avons étudié les modes de basse consommation pour ensuite les coder et les comparer pour trouver des solutions adaptés à réduire la consommation de la carte autant que possible.

D'après le *Reference Manual* de la STM32L0x3, nous voyons qu'il y a 5 modes de basse consommation selon la Figure 2.

Table 32. Summary of low-power modes

Mode name	Entry	Wakeup	Effect on V_{CORE} domain clocks	Effect on V_{DD} domain clocks	Voltage regulator
Low-power run	LPSDSR and LPRUN bits + Clock setting	The regulator is forced in Main regulator (1.8 V)	None	None	In low-power mode
Sleep (Sleep now or Sleep-on-exit)	WFI or Return from ISR	Any interrupt	CPU CLK OFF no effect on other clocks or analog clock sources	None	ON
	WFE	Wakeup event			
Low-power sleep (Sleep now or Sleep-on-exit)	LPSDSR bits + WFI or Return from ISR	Any interrupt	CPU CLK OFF no effect on other clocks or analog clock sources, Flash CLK OFF	None	In low-power mode
	LPSDSR bits + WFE	Wakeup event			
Stop	PDDS, LPSDSR bits + SLEEPDEEP bit + WFI, Return from ISR or WFE	Any EXTI line (configured in the EXTI registers, internal and external lines)	All V_{CORE} domain clocks OFF	HSI16(1), HSE and MSI oscillators OFF	In low-power mode
Standby	PDDS bit + SLEEPDEEP bit + WFI, Return from ISR or WFE	WKUP pin rising edge, RTC alarm (Alarm A or Alarm B), RTC Wakeup event, RTC tamper event, RTC timestamp event, external reset in NRST pin, IWDG reset			OFF

Figure 2 : 5 modes de basse consommation

Chaque mode permet la désactivation des divers périphériques disponibles sur la carte pour améliorer le gain énergétique, le mode *low-power Run* étant ce qui consomme plus et le mode *Standby* étant ce qui consomme moins.

Pour réveiller la carte, celle-ci disponibilise plusieures méthodes en utilisant, entre autres, des wakeup pins, RTC alarme ou RTC wakeup, par exemple.

Les modes les plus intéressants à utiliser sont le mode *Stop* et le mode *Standby* puisqu'ils permettent d'arriver à une consommation très faible. Nous étudierons donc principalement ces deux modes.

3. Etude du mode *Standby*

Le mode *Standby* est ce qui consomme moins par rapport aux autres modes puisqu'il permet de désactiver tous les périphériques ainsi que le processeur de la carte, les horloges et les registres. Les registres du *RTC*, *RTC backup* et le circuiterie de *Standby* sont les seules à être préservés et tous les pins sauf les wakeup pins et les pins relationnés au RTC sont mis en haute-impédance.

Pour entrer dans le mode *Standby*, nous adoptons les étapes suivantes :

- Le bit CWUF du registre PWR_CR (*Power Control Register*) est activé pour réinitialiser le drapeau de *wakeup*.
- Le bit ULP du registre PWR_CR est activé pour mettre le régulateur interne en mode *low-power*.
- Le bit PDDS du registre PWR_CR est activé pour sélectionner le mode *Standby* lors du “sommeil” de la carte.
- Le RTC est configuré pour produire une interruption toutes les *x* secondes.
- Les bits SLEEPDEEP et SLEEPONEXIT du registre SCB_SCR sont activés pour sélectionner le mode *low-power* de la carte et pour que le processeur puisse rentrer dans le mode *Standby* dès que l’interruption est traitée, respectivement.
- Enfin, les fonctions WFI (*Wait For Interrupt*) ou WFE (*Wait For Event*) sont utilisées pour entrer dans le mode *low-power*. Par rapport à WFI, lorsqu'on l'exécute, le microcontrôleur entre dans le mode *low-power* et sort dès qu'il y a une interruption sur NVIC. Pour WFE, le microcontrôleur sort du mode *low-power* dès qu’un événement s’est produit. Dans notre cas, nous avons choisi WFI puisqu’on utilise une interruption du *RTC wakeup* pour réveiller la carte.

Dès que la carte est réveillée du mode *Standby*, le bit SBF (*Standby Flag*) du registre PWR_CSR (*Power Control/Status register*) est mis en haut pour signaler que la carte est réveillée du mode. C’est la seule information qui différencie un réveil du mode *Standby* d’un reset puisque que tous les registres sont remis à zéro à chaque fois que la carte se réveille. Ainsi nous devons redéfinir ce bit en utilisant le bit CSBF (*Clear Standby Flag*) du registre PWR_CR.

Une étape très importante est celle de la configuration du RTC puisqu’il est la méthode utilisée pour réveiller la carte du mode *Standby*.
On remarque que les registres du RTC sont protégés contre des écritures indésirables donc il faut désactiver cette protection avant toute écriture dans les registres concernés.

Pour configurer le RTC, nous avons besoin de 2 registres : le RCC et le RTC. Le RCC permet de réinitialiser les registres du RTC, configurer l’horloge, enlever la protection d’écriture et activer le RTC. Le registre RTC permet de configurer tous les registres concernés. Le *Reference Manual* fourni des exemples de code pour plusieurs situations.

4. Etude du mode *Stop*

Ce mode consomme un peu plus que le mode *Standby* puisqu'il y a plus de fonctionnalités activées, notamment les registres et les E/S.

Dans le mode *Stop* tous les clocks dans le V_{CORE} sont arrêtés mais le contenu de la SRAM et des registres est préservé. Toutes les entrées/sorties sont aussi préservées.

Une fonctionnalité du mode *Stop* est de mettre le régulateur interne en mode *low-power* ce qui fait que la carte consomme moins. Le régulateur est mis en mode *low-power* en activant le bit LPSDSR du registre PWR_CR. Il peut fonctionner en mode *main* mais ça consomme beaucoup plus. La seule avantage de mettre le régulateur en mode *main* est de diminuer le temps de réveil de la carte.

Pour diminuer la consommation, V_{REFINT}, BOR, PVD et le capteur de température peuvent être éteints avant d'entrer dans le mode. Cette fonctionnalité est définie par le bit ULP du registre PWR_CR.

Dans le mode Stop, on peut sélectionner les fonctionnalités suivantes :
- Independent Watchdog (IWDG)
- RTC
- Oscillateur interne (LSI RC)
- Oscillateur externe (LSE OSC)

L'ADC, le DAC et le LCD peuvent consommer de l'énergie, nous pouvons donc les désactiver. Ça se fait à l'aide du bit ADON du registre ADC_CR2 et du bit ENx du registre DAC_CR qui doivent être mis à 0.

Quand nous sortons du mode, l'horloge du système est sélectionné comme MSI ou HSI16 RC oscillator selon le bit STOPWUCK du registre RCC_CFGR.

Pour entrer dans le mode :
- Il n'y a ni d'interruption ni d'évènement à traiter
- Le bit SLEEPDEEP est mis à 1 dans le registre Cortex -M0+ System Control
- Le bit PDDS = 0 dans le registre PWR_CR
- Le bit WUF = 0 dans le registre PWR_CSR
- MSI ou HSI16 RC est sélectionné comme l'horloge du système en mode Stop à l'aide du bit STOPWUCK du registre RCC_CFGR

Les différences principales entre le mode *Stop* et le mode *Standby* sont :
- Dans le mode standby le bit PDDS est mis en haut alors que dans le mode stop il est mis en bas. Le bit PDDS correspond à *Low Power Deep Sleep* et il sert à mettre la CPU en profond sommeil lors de l'entré au mode.
- Dans le mode *Stop* le bit LPSDSR est mis en haut afin de mettre le régulateur interne en mode *low-power* pendant le mode *Stop* ce qui n'est pas fait dans le mode *Standby* puisque le régulateur est éteint.

Selon le Reference Manual, pour réveiller la carte à travers d'une interruption de RTC Wakeup il faut configurer la ligne d'interruption EXTI line 20 pour être sensible à des fronts montants, activer l'interruption du RTC Wakeup dans le registre RTC_CR et configurer le RTC pour générer une interruption RTC Wakeup.

5. Etude du mode *Sleep*

Le mode *Sleep* peut être implémenté de deux façons distinctes : soit en mode *Sleep*, soit en mode *low power Sleep*.

Dans le mode *Sleep*, toutes les E/S ont le même stat comme dans le mode *Run* mais la CPU est éteinte. Pour entrer dans le mode, il faut mettre le bit SLEEPDEEP à 0 et le bit SLEEPONEXIT à 1 puis utiliser la fonction __WFI() pour mettre la carte en mode “veille”.

Dans le mode *low power Sleep*, la CPU est éteinte, le clock est limité en fréquence, le nombre de périphériques en exécution est aussi limité, le régulateur est mis en mode *low-power* et la mémoire Flash est arrêtée. Pour réaliser tout cela, nous utilisons les registres FLASH_ACR, RCC_APBxENR et PWR_CR.

6. Réalisation des codes sur Mbed

Mbed est une plateforme en ligne permettant de créer des applications embarqués à l’aide d’un IDE Web gratuit. Nous nous sommes servis de cette plateforme pour développer notre code.
Pour implémenter les modes de basse consommation, nous avons créé des libraries. Les librairies se trouvent dans les Annexes. Les fonctions écrites pour chaque librairie sont décrites ci-dessous.

6.1 Standby.h

Dans le fichier *standby.h* nous avons défini les fonctions suivantes :

i) config_RTC(time) : cette fonction permet de configurer le *wakeup timer* du RTC pour produire une interruption périodique définie par *time*.
ii) config_Standby() : cette fonction met les bits CWUF, ULP et PDDS à 1 pour configurer le mode Standby.
iii) enter_Standby() : cette fonction met les bits SLEEPDEEP et SLEEPONEXIT pour configurer le mode *Standby* comme le mode *low-power* utilisé lors d’une __WFI().
iv) standby_mode(time) : cette fonction inclut les autres fonctions pour permettre l’utilisation du mode *Standby* avec une seule instruction.
v) standby_verification() : cette fonction permet de vérifier si la carte s’est réveillée du mode *Standby* ou d’un *reset* à l’aide des flags SBF et WUF mentionnés dans la section 3.

6.2 Stop.h

Dans le fichier *stop.h*, les fonctions définies sont les mêmes sauf les fonction suivantes :

i) config_Stop() : cette fonction met l'ADC et le DAC à 0 ainsi que les bits PDDS et STOPWUCK à 0 et LSPDSR et ULP à 1 pour désactiver la fonction *profond sommeil* (éteint le processeur), sélectionner MSI pour l'horloge de *wakeup*, mettre le régulateur interne en mode *low-power* et pour mettre tous les périphériques compatibles en mode *low-power*. Enfin, elle réinitialise le WUF.

ii) config_EXTI() : cette fonction configure l'interruption EXTI line 20 pour être sensible à des fronts montants.

iii) stop_mode(time) : cette fonction inclut les autres fonctions pour permettre l'utilisation du mode *Stop* avec une seule instruction.

6.3 Sleep.h

Dans le fichier sleep.h, le cas est le même que le précédent, avec quelques modifications :

i) config_Sleep() : cette fonction met les bits PDDS et LSPDSR à 0 pour configurer le mode Sleep et le bit CWFU à 1 pour réinitialiser le drapeau WUF.

ii) sleep_mode(time) : cette fonction inclut les autres fonctions pour permettre l'utilisation du mode *Stop* avec une seule instruction.

7. Mesures avec les modes de basse consommation

Une fois les codes écrits et compilés, nous avons fait des mesures et simulations à l'aide de diverses outils fournies par STMicroelectronics.

7.1 La consommation - mesures sur MonitorPower

Pour tester nos solutions, nous avons réalisé des mesures à l'aide de l'outil STM32CubeMonitorPower et de la carte *Power consumption measurement* (Figure 3).

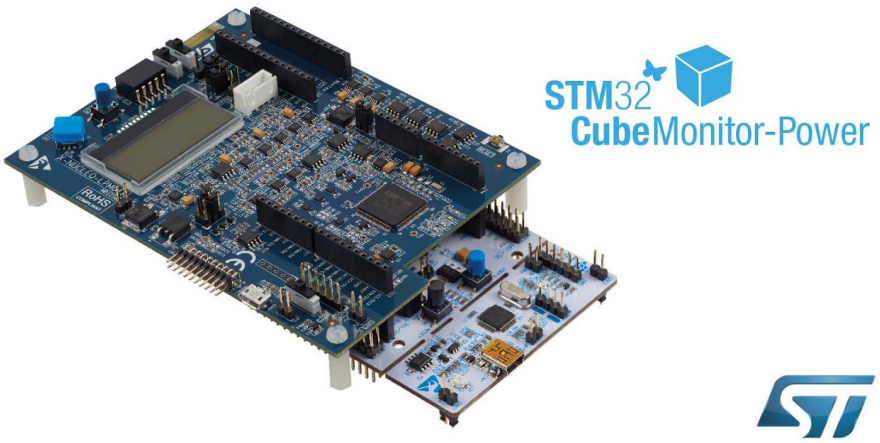

Figure 3 : carte STM32 Power Shield

La carte STM32L073RZ est équipée d'un module de communication USB (ST-LINK) permettant la programmation et la communication avec un ordinateur. Ce module est responsable pour une grande partie de la consommation de la carte donc nous avons décidé de l'enlever pour faire en sorte que la consommation puisse diminuer encore plus.
La documentation de la carte *Power Shield* fourni les étapes nécessaires pour enlever le module ST-LINK afin de réaliser des mesures au niveau du microcontrôleur sans débrancher physiquement le module. C'est une adaptation de l'alimentation de la carte pour qu'elle soit alimentée par le pin AREF à travers de la carte *Power Shield*.

Pour faire l'adaptation, il faut adopter les étapes suivantes :

- Retirer le cavalier JP6 : débrancher VDD du 3V3.
- Retirer le cavalier JP5 : débrancher +5V de E5V et de U5V.
- Enlever les *solder bridges* SB13/14 : débrancher STLINK_RX/STLINK_TX de Virtual Com Port.
- Retirer les cavaliers de CN2 : débrancher SWCLK/SWDIO de STLINK du MCU.
- Enlever le *solder bridge* SB15 : débrancher SWO de STLINK du MCU.
- Enlever le *solder bridge* SB12 : débrancher NRST de STLINK du MCU.

Les connecteurs, cavaliers et solder bridges concernés sont illustrés dans les Figures 4 et 5.

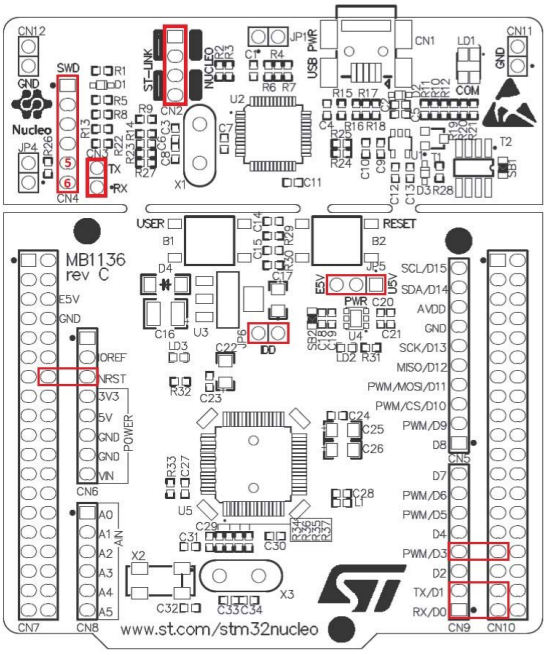

Figure 4 : Face supérieure de la carte STM32 avec les connecteurs et cavaliers nécessaires à l'adaptation d'alimentation en évidence

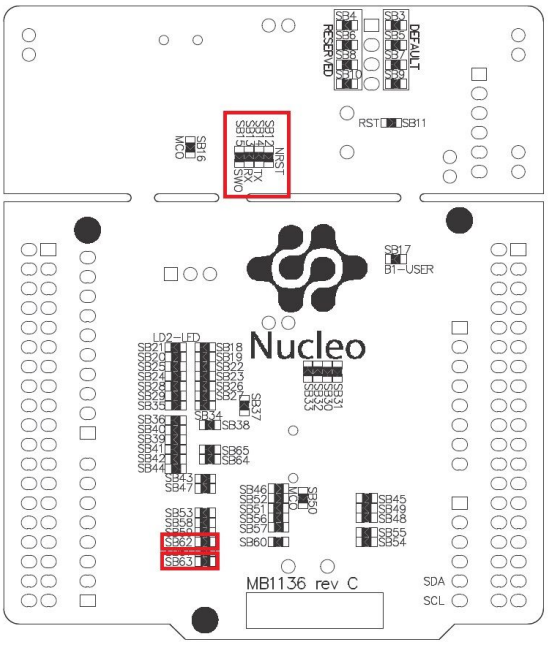

Figure 5 : Face inférieure de la carte STM32 avec les solder bridges nécessaires à l'adaptation d'alimentation en évidence

Ensuite, par rapport à la carte *Power Shield*, il faut configurer les cavaliers (Figure 6) comme suit :

- Mettre le cavalier JP9 : utiliser le mode d'alimentation par le pin AREF.
- Retirer le cavalier JP10 : désactiver le mode d'alimentation 3,3 V.
- Mettre le cavalier JP1 en position "normal".
- Mettre le cavalier JP4 afin d'avoir une capacitance de découplage sur la tension de sortie (V_{OUT}) pour éviter les oscillations dans les mesures.

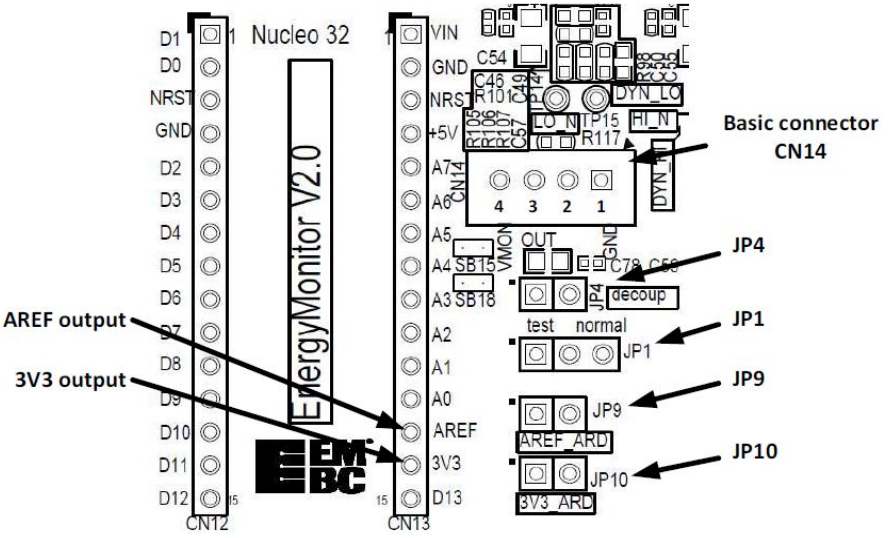

Figure 6 : Cavaliers JP1, JP4, JP9 et JP10 de la carte *Power Shield*

Pour programmer la carte STM32L073RZ, une fois les modifications faites, il faut rebrancher les cavaliers et s'en servir du connecteur CN4 (Figure 4) pour les autres connections.
Nous remarquons que pour rebrancher les STLINK_RX/STLINK_TX, il faut souder les solder bridges SB62 et SB63 (Figure 5).
Les connections nécessaires sont décrites ci-dessous :

- Remettre le cavalier JP6.
- Remettre le cavalier JP5.
- Brancher le pin TX du connecteur CN3 au pin RX du connecteur CN9 ou CN10.
- Brancher le pin RX du connecteur CN3 au pin TX du connecteur CN9 ou CN10.
- Remettre les cavaliers du connecteur CN2.
- Brancher le 5ème pin du connecteur CN4 (SWD) au pin NRST du connecteur CN6 ou CN7.
- Brancher le 6ème pin du connecteur CN4 (SWD) au pin PWM/D3 du connecteur CN9 ou CN10.

En premier test, nous avons fait des mesures avec des réveils périodiques toutes les 10 secondes (Figure 7).

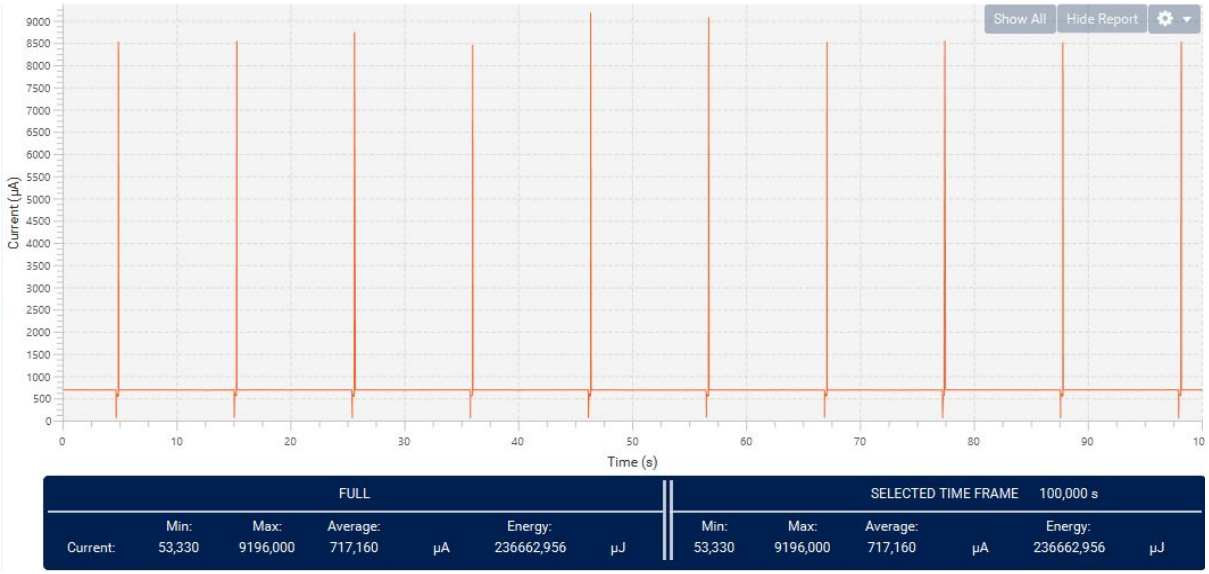

Figure 7 : Mesure de consommation de la carte en premier test

D'après la Figure 7 on remarque que la consommation moyenne est d'environ 717 uA et que la carte se réveil toutes les 10 secondes.

Pour voir la performance du mode de basse consommation dans une application plus réelle, nous avons décidé d'implémenter notre librairie *standby.h* dans un code déjà amorcé (Serres), une application qui récupère des données de capteurs et les envoie via communication LoRa vers un serveur web.

La Figure 8 présente la mesure de consommation de la carte avec cette nouvelle application.

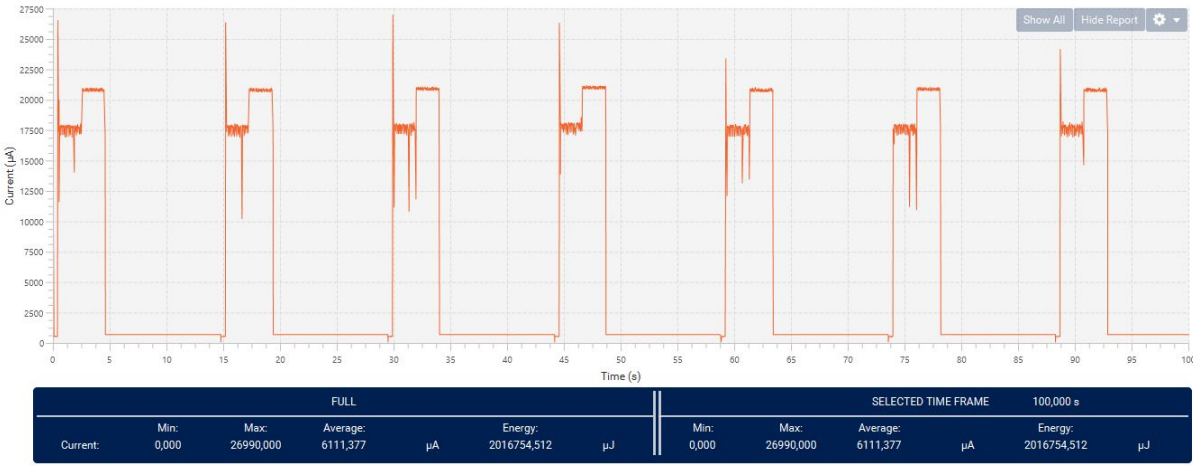

Figure 8 : Mesure de consommation de la carte avec le code Serres

Nous observons que la consommation en mode “veille” est d’environ 700 uA pendant 10 secondes. Dès que la carte se réveille, la consommation monte jusqu’à 17500 uA, au bout d’à peu près 2,5 secondes elle envoie les données et la consommation monte jusqu’à 21000 uA, elle reste avec cette consommation pendant 2 secondes avant de rentrer dans le mode *Standby*.

La figure ci-dessous illustre le résultat de la mesure de consommation en utilisant comme mode *low-power* le *Stop* et le *Sleep*.

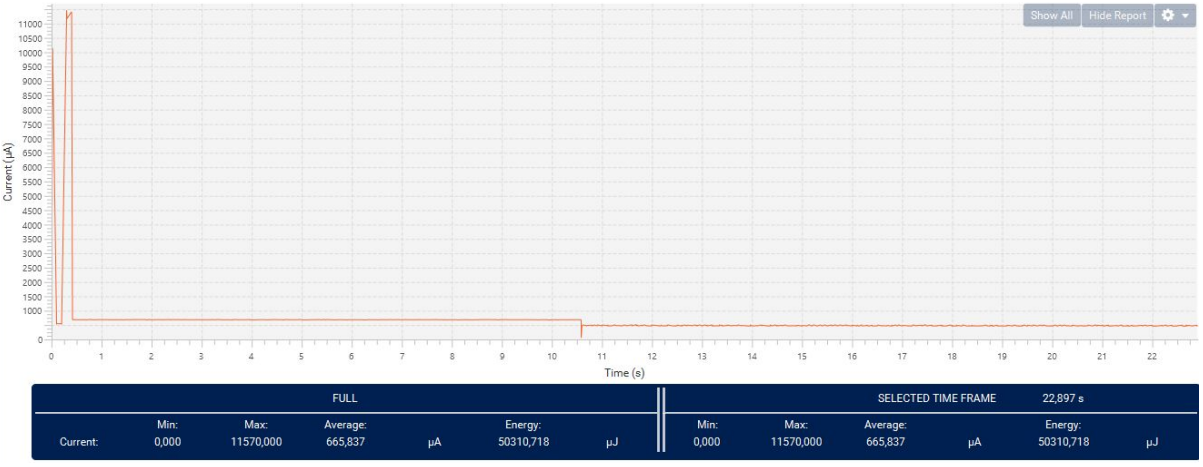

Figure 9 : Mesure de consommation de la carte en mode *Stop/Sleep*

Nous observons que la carte se réveille une fois et dès qu’elle rentre au mode *Stop/Sleep*, ne se réveille plus. Jusqu’à présent, nous n’avons pas trouvé ni de solution ni d’explication pour ce phénomène. Nous remarquons que le code est le même pour le mode Standby mais avec quelques modifications (Section 6).

En ce qui concerne la consommation en mode *Standby*, celle-ci étant importante par rapport à ce que nous attendions, nous avons observé qu'une ligne de code produit la différence (Figure 10).

```
69 int config_RTC(uint16_t time)
70 {
71     // Disable backup write protection. this bit must be set in order to enable
72     // RTC registers write access - Reference Manual 27.4.7 (pg 649)
73     _SET_BIT(PWR->CR, PWR_CR_DBP);
74     // RTC Reset - Reference Manual 7.3.21 (pg 219)
75     _SET_BIT(RCC->CSR, RCC_CSR_RTCRST);
76     _CLEAR_BIT(RCC->CSR, RCC_CSR_RTCRST);
```

Figure 10 : Extrait de la fonction config_RTC() avec ligne 73 en évidence

L'instruction de la ligne 73 correspond à la désactivation de la protection d'écriture des registres du RTC. Si nous enlevons l'instruction exécutée dans cette ligne de code, la carte entre en mode *Standby* avec une consommation qui varie de 0,8 uA à 1,7 uA (Figure 11), c'est bien le comportement que nous attendions sauf que la carte ne se réveille plus. Si nous remettons la ligne 73 du code, la carte se réveille mais avec une consommation de 700 uA en mode *Standby* (Figure 8). C'est une consommation très faible mais ce n'est pas exactement ce que nous attendions.
Nous avons observé aussi qu'il faut ajouter les lignes correspondantes à la réinitialisation des registres du RTC (lignes 75 et 76 dans la Figure 10), sans en avoir, le comportement est le même que sans la ligne 73.
Nous remarquons que cette réponse est différente de celle avec les modes *Stop*/*Sleep* puisque dans celle-ci la consommation est de 1 uA et dans celle-là la consommation est un peu en dessus de 700 uA.

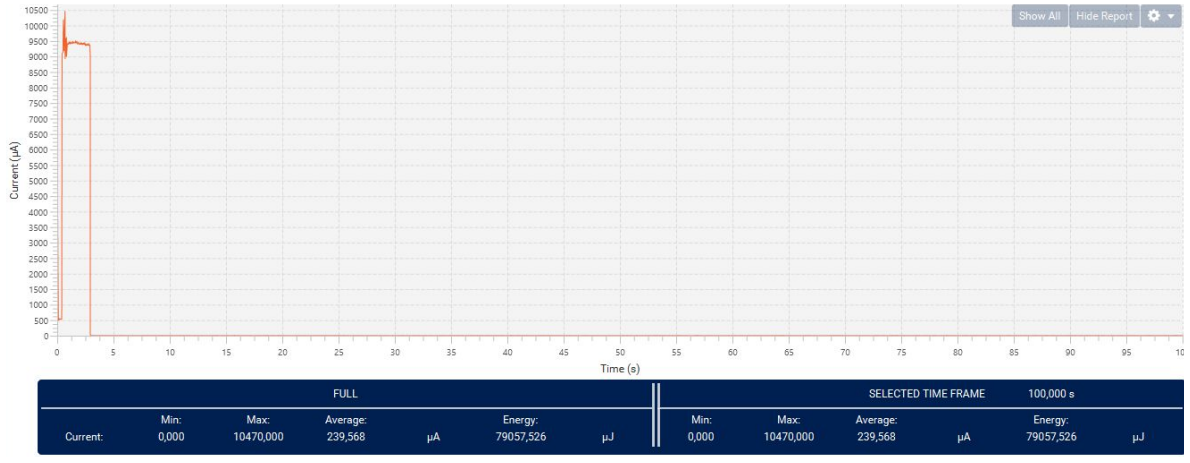

Figure 11 : Mesure de consommation de la carte avec la ligne 73 du fichier *standby.h* enlevée

En fin nous avons décidé d'utiliser le mode *Standby* puisqu'il permet une consommation très faible, même si cette valeur n'est pas exactement d'accord avec ce que nous attendions.

Nous avons ensuite passé à l'étape de calcul de l'autonomie de la carte avec une batterie.

7.2 La consommation - simulations sur CubeMX

Nous avons utilisé d'abord, à titre de comparaison, le logiciel CubeMX afin de simuler la durée de vie d'une batterie type LR20.
Nous avons choisi ce type de batterie pour plusieurs raisons, entre elles, puisque ce type de batterie est facile à trouver, elles ont une capacité élevée par rapport aux autres types de batterie de même voltage, elles ne sont pas très chères et elles nous étaient disponibles en moment de nos tests.

CubeMX est un outil de configuration graphique destiné à simplifier la génération d'un programme d'initialisation d'une STM32. Outre ces multiples fonctionnalités d'assistance à la programmation, ce logiciel intègre un calculateur de consommation de puissance qui permet à l'utilisateur d'estimer la durée de vie de sa batterie en fonction de sa technologie, le courant tiré par le microcontrôleur et ses périphériques ainsi que la température ambiante à ne pas dépasser.

Tout d'abord, après consulter judicieusement les datasheets des différents périphériques externes tels que les capteurs et le module LoRa, nous avons déterminé le temps d'une période en mode *Run* (allumage, mesure, transmission bit-à-bit des données, envoi des données) ainsi que leur consommation en courant.

Le tableau suivant présente les valeurs obtenues de cette recherche :

Tableau 1 : Paramètres à régler dans le logiciel CubeMX

Mode	Vdd	Range/Scale	Memory	CPU Freq	Clock Config	Peripherals	Step Current	Duration
RUN	3.0	Range-3-LOW	Flash	2 MHZ	HSE	RTC	42.3 mA	78.979 ms
STANDBY	3.0	No Range	n/a	0	LSI	RTC	691.16 uA	3600 s

Ces informations étant prédominantes pour les résultats, deux simulations ont été calculées, une avec les valeurs typiques et une autre avec les valeurs maximales (selon les *Datasheets* et mesures) :

Tableau 2 : Résultat de la simulation sur CubeMX

	Valeurs typiques	Valeurs maximales

Etape	Courant	Durée	Courant	Durée
Mode RUN	42.3 mA	78.979 ms	143.01 mA	101.338 ms
Mode STANDBY	691.16 uA	3600 s	702.96 uA	3600 s
Tº Max	99.16 ºC		85.27 ºC	
Courant moyen	692.07 uA		706.97 uA	
Durée de vie des batteries	3 ans, 3 jours, 14 heures		2 ans,11 mois,17 jours, 23 heures	

Nous remarquons que les valeurs ci-dessus sont par rapport à un cycle d'exécution de la carte, c'est-à-dire, carte en mode *Standby* puis réveillée une fois. La durée de vie de la batterie est calculée à partir de ces valeurs.
Afin de prolonger la durée de vie de la batterie, nous devons prolonger autant que possible la durée d'utilisation du mode *Standby* et réduire en même temps la durée d'utilisation du mode *Run*, comme ça, la consommation moyenne va se rapprocher de plus en plus de la consommation en mode *Standby* et ainsi augmenter la durée de vie de la batterie.

7.3 La consommation - calcul de durée de vie d'une batterie

Pour cette étape, nous avons utilisé deux batteries type LR20 (1,5 V) en série avec une capacité unitaire qui varie de 1200 mAh à 1800 mAh.

Pour connecter la carte à une batterie avec les modifications dans le module ST-LINK, nous avons branché la batterie entre les pins GND et le VDD du connecteur CN7.

Pour le calcul d'autonomie de batterie, nous avons considéré le programme Serres avec un temps de sommeil pris arbitrairement qui varie de 5 secondes à une heure. Nous avons calculé la moyenne des consommations en mode *Run* et en mode *Standby* pour les utiliser dans les calculs d'autonomie.

La figure suivante résume les résultats obtenus :

	C. réveil (uA)	T. réveil (s)	C. envoie (uA)	T. envoie (s)	C. run (uA)	T. run (s)
5s	15086,48	2,5	20884,952	2	17663,57867	4,5
10s	15086,48	2,5	20884,952	2	17663,57867	4,5
60s	15086,48	2,5	20884,952	2	17663,57867	4,5
120s	15086,48	2,5	20884,952	2	17663,57867	4,5
300s	15086,48	2,5	20884,952	2	17663,57867	4,5
3600s	15086,48	2,5	20884,952	2	17663,57867	4,5
	C. sommeil (uA)	T. sommeil (s)	Conso totale (uA)	Temps totale (s)	Durée max (ans)	Durée min (ans)
5s	709,84	5	8740,558316	9,5	0,23508733	0,156724887
10s	709,84	10	5971,345103	14,5	0,344109155	0,229406104
60s	709,84	60	1892,658977	64,5	1,085665482	0,723776988
120s	709,84	120	1322,625735	124,5	1,55357216	1,035714774
300s	709,84	300	960,3878621	304,5	2,139546533	1,426364355
3600s	709,84	3600	731,0057162	3604,5	2,810914436	1,873942957

Figure 12 : Résumé des calculs d'autonomie d'une batterie type LR20

A partir de cette figure nous voyons que la durée de vie d'une batterie varie de 2 mois à 3 ans en fonction du temps de sommeil choisi par l'application et de la capacité de la batterie utilisé pour alimenter la carte.

Conclusion

La carte STM32L0073RZ est polyvalente, elle offre plusieurs modes de basse consommation de puissance et avec ces modes il est possible d'arriver à des consommations très faibles et donc à une autonomie importante.
La caractérisation des modes basse consommation est importante puisqu'elle permet de bien choisir le mode selon les besoins du client et/ou de l'application.
Même avec un résultat différent de celui attendu, avec une consommation un peu plus importante, nous avons arrivé à des consommations très faibles, ce qui fait de la carte STM32 un bon choix pour n'importe quelle application de l'Internet des Objets qui exigent des dispositifs de plus en plus autonomes.
Pour une suite au projet développé cette année, nous suggérons les étapes suivantes :

- Etudier la consommation en mode Standby qui est toujours élevée par rapport à ce que l'on attendait.
- Etudier plus profondément les modes Stop/Sleep puis qu'ils n'ont pas marché.
- Etudier le mode d'alimentation le plus économique pour prolonger encore plus la durée de vie du système.

Annexes

standby.h

```
#ifndef _STANDBY_H
#define _STANDBY_H

#include "mbed.h"
#include "stm32l073xx.h"

#define _CLEAR_BIT(a, b) (a &=~ b)
#define _SET_BIT(a, b) (a |= b)

// function prototypes
void standby_mode(uint16_t time); // time in seconds
void standby_verification(void);

// other functions
void disable_RTC_protection_access(void);
void enable_RTC_protection_access(void);
int config_RTC(uint16_t time);
void enable_reg_access(void);
void disable_reg_access(void);
void config_Standby(void);
void enter_Standby(void);

void disable_RTC_protection_access(void)
{
/* Enable write access for RTC registers - Reference Manual 27.7.9
(pg 672, 649) */
    RTC->WPR = 0xCA;
    RTC->WPR = 0x53;
}

void enable_RTC_protection_access(void)
{
/* Disable write access for RTC registers */
    RTC->WPR = 0xFE;
    RTC->WPR = 0x64;
}

int config_RTC(uint16_t time)
{
     // Disable backup write protection. this bit must be set in
order to enable
     // RTC registers write access - Reference Manual 27.4.7 (pg
649)
    _SET_BIT(PWR->CR, PWR_CR_DBP);
     // RTC Reset - Reference Manual 7.3.21 (pg 219)
```

```
    _SET_BIT(RCC->CSR, RCC_CSR_RTCRST);
    _CLEAR_BIT(RCC->CSR, RCC_CSR_RTCRST);

      // Reset and Clock Control
      // Power interface clock enabled - Reference Manual 7.3.15
(pg 208)
    _SET_BIT(RCC->APB1ENR, RCC_APB1ENR_PWREN);
    enable_reg_access();
      // Select the RTC clock source: LSE oscillator (32768Hz) -
Reference Manual 7.3.21 (pg 220)
    _SET_BIT(RCC->CSR, RCC_CSR_RTCSEL_LSE);
      // Enable the external low-speed LSE oscillator - Reference
Manual 7.3.21 (pg 220)
      _SET_BIT(RCC->CSR, RCC_CSR_LSEON);

      disable_RTC_protection_access();

      // RTC configuration
      // Disable wake up timer to modify it - Reference Manual
27.4.7 (pg 650)
    _CLEAR_BIT(RTC->CR, RTC_CR_WUTE);
    while((RTC->ISR & RTC_ISR_WUTWF) != RTC_ISR_WUTWF) {
      // Wait until it is allowed to modify wake up reload value -
Reference Manual 27.7.4 (pg 667)
    }

      RTC->WUTR = time-1; // Wake up value reload counter [s] -
Reference Manual 27.7.6 (pg 669)
      RTC->PRER = 0x7F00FF; // ck_spre = 1Hz PREDIV_A = 0x7F(127)
PREDIV_S = 0xFF(255) using LSE (32768Hz). ck = (32768)/(PREDIV_A *
PREDIV_S)
      // OSEL (output selection) = 0x3 -> RTC_ALARM output = Wake
up timer - Reference Manual 27.7.3 (pg 662)
    _SET_BIT(RTC->CR, RTC_CR_OSEL);
      // WUCKSEL = 0x4 -> RTC Timer [1s - 18h] - Reference Manual
27.7.3 (pg 648, 664)
    _SET_BIT(RTC->CR, RTC_CR_WUCKSEL_2);
      // Enable wake up counter/interrupt - Reference Manual 27.7.3
(pg 650, 663, WUTIE 663)
    _SET_BIT(RTC->CR, RTC_CR_WUTE | RTC_CR_WUTIE);

      enable_RTC_protection_access();

      // Re-enable the RTC clock
      _SET_BIT(RCC->CSR, RCC_CSR_RTCEN);
      disable_reg_access();
      return 1; // Return TRUE
}
```

```
void enable_reg_access(void)
{
	// Disable backup write protection. this bit must be set in order to enable RTC registers write access - Reference Manual 27.4.7 (pg 649)
	// - Reference Manual 27.4.7 (pg 649)
    _SET_BIT(PWR->CR, PWR_CR_DBP);
}

void disable_reg_access(void)
{
	// Disable backup write protection. this bit must be set in order to enable RTC registers write access - Reference Manual 27.4.7 (pg 649)
	// - Reference Manual 27.4.7 (pg 649)
    _CLEAR_BIT(PWR->CR, PWR_CR_DBP);
}

void config_Standby(void)
{

	// Power configuration

	// Clear the WUF flag - Reference Manual 6.4.1 (PG 168)
    _SET_BIT(PWR->CR, PWR_CR_CWUF);
	//  V_{REFINT} (internal voltage reference for analog peripherals) is off in low-power mode - Reference Manual 6.2.4 (pg 151)
    _SET_BIT(PWR->CR, PWR_CR_ULP);
	// Enter Standby mode when the CPU enters deepsleep - Reference Manual 6.4.1 (PG 168)
    _SET_BIT(PWR->CR, PWR_CR_PDDS);
}

void enter_Standby(void)
{

	// System Control Block
	// Low power mode -> Deep Sleep
	_SET_BIT(SCB->SCR, SCB_SCR_SLEEPDEEP_Msk);
	// Reenter low-power mode after ISR
	_SET_BIT(SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk);
}

void standby_mode(uint16_t time)
{
	config_Standby();
	config_RTC(time);
	enter_Standby();
```

```
      __WFI(); // Waiting for Interruption -> Enter low-power mode
}

void standby_verification(void)
{
      // check standby flag
    // Reference Manual 6.3.10 <<Exiting Standby Mode>> (PG 162) &
    // Reference Manual 6.4.2(PG 170)
    if((PWR->CSR)&(PWR_CSR_SBF)) {
        // clear PWR Wake Up flag - Reference Manua 6.4.1 (PG 168)
        _SET_BIT(PWR->CR, PWR_CR_CWUF);
        // clear PWR Standby flag - Reference Manua 6.4.1 (PG 167)
        _SET_BIT(PWR->CR, PWR_CR_CSBF);

        printf("\nLa carte se reveille du mode standby\n");
    }
    else {
        printf("\nLa carte se reveille du power cycle\n");
    }
}

#endif // standby.h
```

stop.h

```
#ifndef _STOP_H
#define _STOP_H

#include "mbed.h"
#include "stm32l073xx.h"

#define _CLEAR_BIT(a, b) (a &=~ b)
#define _SET_BIT(a, b) (a |= b)

// Functions prototypes
void enter_stopmode(void);

// Other functions
void enable_RTC_reg_access(void);
void disable_RTC_reg_access(void);
void config_EXTI(void);
int config_RTC(uint16_t time);
void config_Stop(void);
void enter_Stop(void);
void stop_mode(int time);


void enable_RTC_reg_access(void)
{
     /* Enable write access for RTC registers - Reference Manual
27.7.9 (pg 672) */
    RTC->WPR = 0xCA; // - Reference Manual 27.4.7 (pg 649)
    RTC->WPR = 0x53;
}

void disable_RTC_reg_access(void)
{
    /* Disable write access for RTC registers */
    RTC->WPR = 0xFE;
    RTC->WPR = 0x64;
}

void config_EXTI(void)
{
        NVIC_EnableIRQ(RTC_IRQn); // Enable Interrupt on RTC
        NVIC_SetPriority(RTC_IRQn,0); // Set priority for RTC
     // Enable RTC alarm going through EXTI 20 line to NVIC
           _SET_BIT(EXTI->IMR, EXTI_IMR_IM20); // IMR = Interrupt
Mask Register
     // Select Rising Edge Trigger
         _SET_BIT(EXTI->RTSR, EXTI_IMR_IM20); // Select Rising Edge
Trigger (ligne 20 sensible à des fronts montants)
}
```

```
int config_RTC(uint16_t time)
{
     // Disable backup write protection. this bit must be set in order to enable RTC registers write access - Reference Manual 27.4.7 (pg 649)
    _SET_BIT(PWR->CR, PWR_CR_DBP);

    // RTC Reset - Reference Manual 7.3.21 (pg 219)
    _SET_BIT(RCC->CSR, RCC_CSR_RTCRST);
    _CLEAR_BIT(RCC->CSR, RCC_CSR_RTCRST);

    // Reset and Clock Control
   // Power interface clock enabled - Reference Manual 7.3.15 (pg 208)
    _SET_BIT(RCC->APB1ENR, RCC_APB1ENR_PWREN);
//  Select the RTC clock source: LSE oscillator (32768Hz) - Reference Manual 7.3.21 (pg 220)
    _SET_BIT(RCC->CSR, RCC_CSR_RTCSEL_LSE);
// Enable the external low-speed LSE oscillator - Reference Manual 7.3.21 (pg 220)
    _SET_BIT(RCC->CSR, RCC_CSR_LSEON);

    enable_RTC_reg_access();

    // RTC configuration
// Disable wake up timer to modify it - Reference Manual 27.4.7 (pg 650)
    _CLEAR_BIT(RTC->CR, RTC_CR_WUTE);
    while((RTC->ISR & RTC_ISR_WUTWF) != RTC_ISR_WUTWF) {
         // Wait until it is allowed to modify wake up reload value - Reference Manual 27.7.4 (pg 667)
    }

     RTC->WUTR = time-1; // Wake up value reload counter [s] - Reference Manual 27.7.6 (pg 669)
     RTC->PRER = 0x7F00FF; // ck_spre = 1Hz PREDIV_A = 0x7F(127) PREDIV_S = 0xFF(255) using LSE (32768Hz). ck = (32768)/(PREDIV_A * PREDIV_S)
// OSEL (output selection) = 0x3 -> RTC_ALARM output = Wake up timer - Reference Manual 27.7.3 (pg 662)
    _SET_BIT(RTC->CR, RTC_CR_OSEL);
// WUCKSEL = 0x4 -> RTC Timer [1s - 18h] - Reference Manual 27.7.3 (pg 648, 664)
    _SET_BIT(RTC->CR, RTC_CR_WUCKSEL_2);
// Enable wake up counter/interrupt - Reference Manual 27.7.3 (pg 650, 663, WUTIE 663)
    _SET_BIT(RTC->CR, RTC_CR_WUTE | RTC_CR_WUTIE);
```

```
    disable_RTC_reg_access();

     // Re-enable the RTC clock
    _SET_BIT(RCC->CSR, RCC_CSR_RTCEN);
    return 1; // Return TRUE
}

void config_Stop(void)
{
    uint32_t temp_reg = 0;

    // disable DAC
    _CLEAR_BIT(DAC->CR, (DAC_CR_EN1 | DAC_CR_EN2));
    // disable ADC
    _CLEAR_BIT(ADC1->CR, ADC_CR_ADEN); // ADC->CR2, ADC_CR_ADON
    // select the regulator state in stop mode
    temp_reg = PWR->CR;
    // PDDS = 0
        _CLEAR_BIT(temp_reg,  (PWR_CR_PDDS  |  PWR_CR_LPSDSR));  //
ENLEVER pour le mode standby

    // select MSI as wakeup from stop mode
    _CLEAR_BIT(RCC->CFGR, RCC_CFGR_STOPWUCK);

       //  Set  clear  wake-up  flag,  low-power  deepsleep  run  bit
(regulator in low power mode)
    // and V_{REFINT}
           _SET_BIT(temp_reg,   (PWR_CR_CWUF   |   PWR_CR_LPSDSR   |
PWR_CR_ULP));
    // store the new value
    PWR->CR = temp_reg;
}

void enter_Stop(void)
{
    // System control block
    _SET_BIT(SCB->SCR, SCB_SCR_SLEEPDEEP_Msk);
// Reenter low-power mode after ISR
    SET_BIT(SCB->SCR, SCB_SCR_SLEEPONEXIT_Msk);
}

void stop_mode(int time)
{
    config_EXTI(); // configure the interruptions
    config_Stop();
    config_RTC(time); // configure the RTC
    enter_Stop();
    __WFI();
}
```

```
#endif // stop.h
```

sleep.h

```
#ifndef _SLEEP_H
#define _SLEEP_H

#include "mbed.h"
#include "stm32l073xx.h"

#define _CLEAR_BIT(a, b) (a &=~ b)
#define _SET_BIT(a, b) (a |= b)

// Function prototypes
void sleep_mode(uint16_t time); // time in second

// other functions
void enable_RTC_reg_access(void);
void disable_RTC_reg_access(void);
int config_RTC(uint16_t time);
void enable_bkp_access(void);
void config_Sleep(uint16_t time);
void config_EXTI(void);
void enter_Sleep(void);


void enable_RTC_reg_access(void)
{
     /* Enable write access for RTC registers - Reference Manual
27.7.9 (pg 672) */
    RTC->WPR = 0xCA; // - Reference Manual 27.4.7 (pg 649)
    RTC->WPR = 0x53;
}

void disable_RTC_reg_access(void)
{
    /* Disable write access for RTC registers */
    RTC->WPR = 0xFE;
    RTC->WPR = 0x64;
}

int config_RTC(uint16_t time)
{
     // Disable backup write protection. this bit must be set in
order to enable RTC registers write access - Reference Manual
27.4.7 (pg 649)
    _SET_BIT(PWR->CR, PWR_CR_DBP);

    // RTC Reset - Reference Manual 7.3.21 (pg 219)
    _SET_BIT(RCC->CSR, RCC_CSR_RTCRST);
    _CLEAR_BIT(RCC->CSR, RCC_CSR_RTCRST);
```

```
    // Reset and Clock Control
     // Power interface clock enabled - Reference Manual 7.3.15
(pg 208)
    _SET_BIT(RCC->APB1ENR, RCC_APB1ENR_PWREN);
    enable_bkp_access();
//  Select the RTC clock source: LSE oscillator (32768Hz) -
Reference Manual 7.3.21 (pg 220)
    _SET_BIT(RCC->CSR, RCC_CSR_RTCSEL_LSE);
// Enable the external low-speed LSE oscillator - Reference Manual
7.3.21 (pg 220)
    _SET_BIT(RCC->CSR, RCC_CSR_LSEON);

    enable_RTC_reg_access();

    // RTC configuration
// Disable wake up timer to modify it - Reference Manual 27.4.7
(pg 650)
    _CLEAR_BIT(RTC->CR, RTC_CR_WUTE);
    while((RTC->ISR & RTC_ISR_WUTWF) != RTC_ISR_WUTWF) {
        // Wait until it is allowed to modify wake up reload value
- Reference Manual 27.7.4 (pg 667)
    }

     RTC->WUTR = time-1; // Wake up value reload counter [s] -
Reference Manual 27.7.6 (pg 669)
     RTC->PRER = 0x7F00FF; // ck_spre = 1Hz PREDIV_A = 0x7F(127)
PREDIV_S = 0xFF(255) using LSE (32768Hz). ck = (32768)/(PREDIV_A *
PREDIV_S)
// OSEL (output selection) = 0x3 -> RTC_ALARM output = Wake up
timer - Reference Manual 27.7.3 (pg 662)
    _SET_BIT(RTC->CR, RTC_CR_OSEL);
// WUCKSEL = 0x4 -> RTC Timer [1s - 18h] - Reference Manual 27.7.3
(pg 648, 664)
    _SET_BIT(RTC->CR, RTC_CR_WUCKSEL_2);
// Enable wake up counter/interrupt - Reference Manual 27.7.3 (pg
650, 663, WUTIE 663)
    _SET_BIT(RTC->CR, RTC_CR_WUTE | RTC_CR_WUTIE);

    disable_RTC_reg_access();

// Re-enable the RTC clock
    _SET_BIT(RCC->CSR, RCC_CSR_RTCEN);
    return 1; // Return TRUE
}

void enable_bkp_access(void)
{
```

```
    // Disable backup write protection. this bit must be set in
order to enable RTC registers write access - Reference Manual
27.4.7 (pg 649)
    _SET_BIT(PWR->CR, PWR_CR_DBP);
}

void config_Sleep(void)
{

    // Power configuration

    enable_bkp_access();
// Clear the WUF flag - Reference Manual 6.4.1 (PG 168)
    _SET_BIT(PWR->CR, PWR_CR_CWUF);
// V_{REFINT} (internal voltage reference for analog peripherals)
is off in low-power mode - Reference Manual 6.2.4 (pg 151)
    _SET_BIT(PWR->CR, PWR_CR_ULP);
    _CLEAR_BIT(PWR->CR, (PWR_CR_PDDS | PWR_CR_LPSDSR));
}

void config_EXTI(void)
{

        NVIC_EnableIRQ(RTC_IRQn); // Enable Interrupt on RTC
        NVIC_SetPriority(RTC_IRQn,0); // Set priority for RTC
// Enable RTC alarm going through EXTI 20 line to NVIC
        _SET_BIT(EXTI->IMR, EXTI_IMR_IM20);
// Select Rising Edge Trigger
        _SET_BIT(EXTI->RTSR, EXTI_IMR_IM20);
}

void enter_Sleep(void)
{

        // System Control Block
// Low power mode -> Deep Sleep
        _CLEAR_BIT(SCB->SCR, SCB_SCR_SLEEPDEEP_Msk);
}

void sleep_mode(uint16_t time)
{
        config_EXTI();
        config_Sleep();
        config_RTC(time);
        enter_Sleep();
          __WFI(); // Waiting for Interruption -> Enter low-power
mode
}
```

```
#endif // sleep.h
```