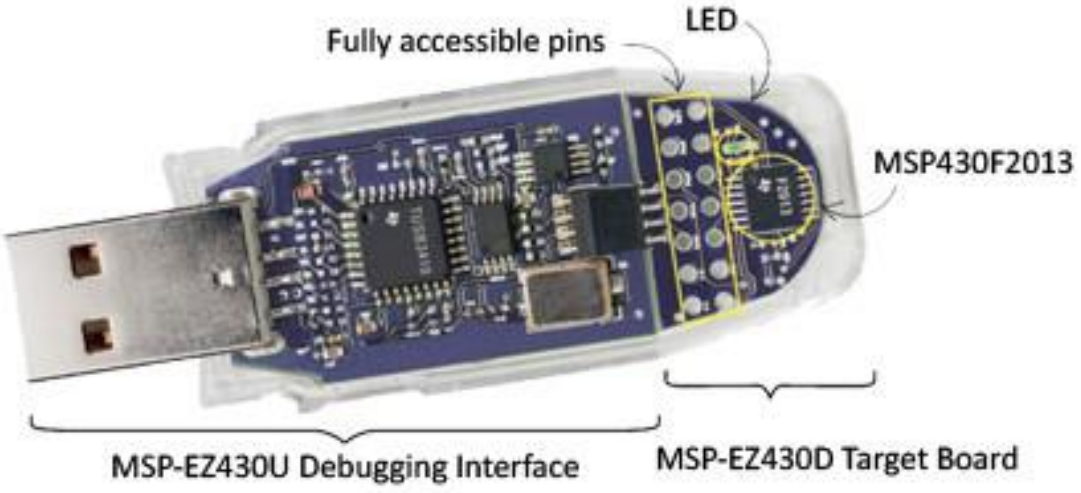
MSP430-F2012 de TEXAS INSTRUMENT

Travaux Pratiques du Semestre 7

Filtrage numérique par microcontrôleur



Sommaire

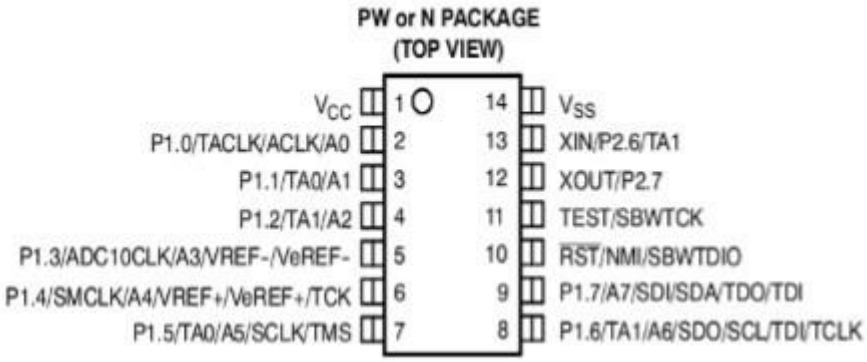


Schéma de brochage du MSP430

Introduction aux microcontrôleurs

Un microcontrôleur (noté µC) est un circuit intégré rassemblant plusieurs éléments électriques dans une puce de dimension réduite. L'intérêt premier est de disposer d'une grande puissance de calcul avec un encombrement réduit. Ces ressources sont accessibles depuis les broches du circuit intégré, on y trouve aussi des broches pour l'alimentation électrique ainsi qu'un accès à la programmation de ce circuit.

Le mot clé du paragraphe précédent est « programmation » car s'il était anciennement possible de créer des circuits électroniques montés sur des cartes, on ne pouvait jusque-là pas modifier leur comportement. A partir du moment où les composants sont soudés sur la carte, il est invraisemblable de vouloir changer leur fonctionnement. Les microcontrôleurs répondent à cette problématique en proposant un unique circuit que l'on peut éventuellement intégrer au sein d'une carte comportant des composants additionnels. Le comportement complet du circuit est modifiable en changeant quelques lignes du programme exécuté par le microcontrôleur.

Afin que le programme puisse agir sur le circuit, il faut des sorties électriques mais aussi des entrées pour que les actions sur les sorties puissent être situationnelles. On distingue donc plusieurs ressources disponibles sur ce type de circuit :

- Une unité de calcul appelée CPU (Central Processing Unit)
- Les ports d'entrées et de sorties
- Une horloge pour cadencer les instructions
- Une mémoire de stockage pour le programme (mémoire morte)
- Une mémoire de stockage des variables (mémoire vive)

Tous ces composants sont reliés entre eux par des voies de communications appelées « BUS ». Il en existe trois types :

- Le bus d'adresse qui fait office de sélecteur pour choisir les cases mémoires et les périphériques souhaités.
- Le bus de données faisant transiter les informations d'un élément à l'autre du microcontrôleur. Ces données peuvent être des instructions ou des données à traiter par le programme.
- Le bus de contrôle gérant les interruptions et les états de lecture et écriture.

Certaines gammes de microcontrôleurs possèdent deux bus de données pour séparer les données d'exécutions et les instructions de programmes. Les transferts de données se font en parallèle. On parle alors d'une « architecture Harvard », ce qui permet de réduire le temps de chaque instruction (contrairement à une « architecture de von Neumann » qui est plus économique).

Les microcontrôleurs sont présents partout, en particulier là où on ne s'y attend pas. On peut citer les télécommandes de télévision ou les souris d'ordinateur pour les signaux envoyés, les bouilloires et les radiateurs pour les réglages de température voire même les stylos pour créer un effet lumineux au bout du crayon :

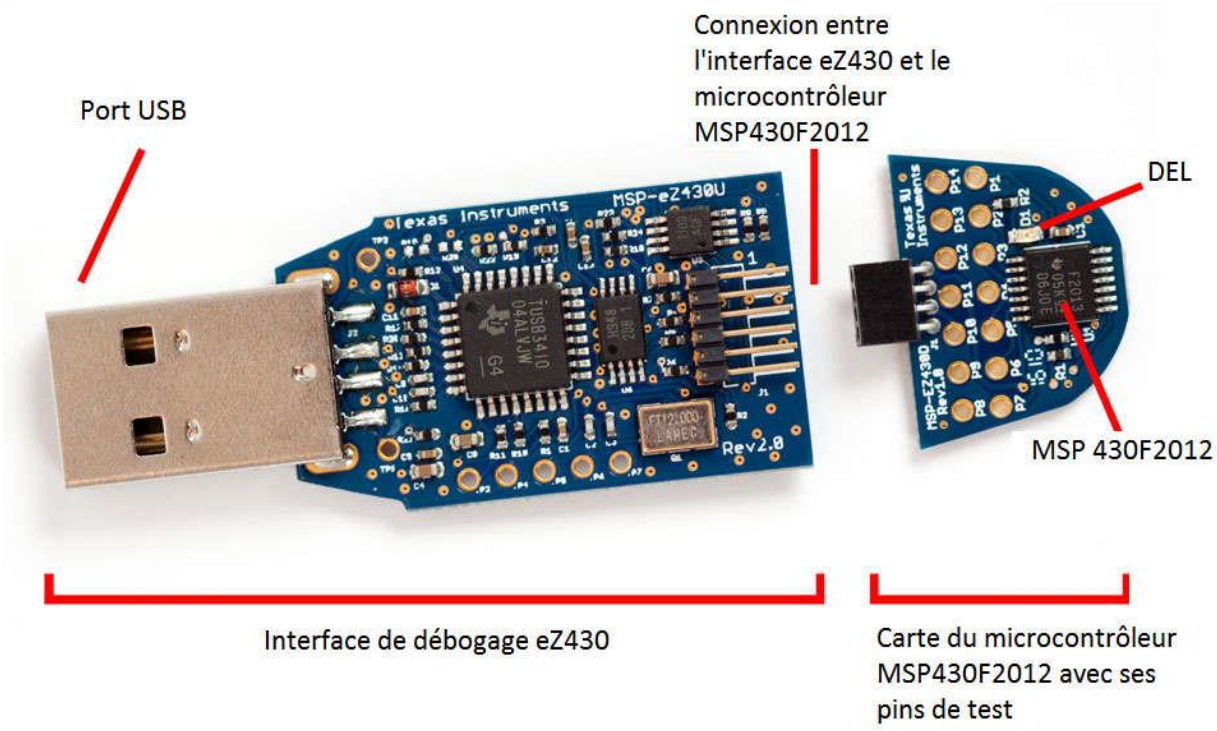
D’autres applications plus complexes utilisent un grand nombre de ces microcontrôleurs. Chacun d’eux est spécialisé dans une tâche et leur assemblage organise la structure d’un système très puissant. Dans le cas d’une voiture, il existe de nombreux µC (plusieurs dizaines/centaines) liés ensemble et chacun d’eux exerce une fonction bien précise.

Nombre d’électroniciens débutants ou confirmés se sont rapidement intéressés aux microcontrôleurs. La carte Arduino est devenu le porte-étendard de cet intérêt croissant porté aux circuits qu’il est devenu si simple de réaliser et de modifier. En plus d’être abordable est très simple de compréhension, il est désormais possible de réaliser des applications très diverses avec très peu de composants externes. Voici une carte Arduino dont on distingue le microcontrôleur au centre et les broches d’entrées et de sorties sur les côtés :

D’autres microcontrôleurs sont connus comme le PIC (Peripheral Interface Controller) de Microchip mais celui que nous allons utiliser par la suite est le MSP430 de Texas Instruments.

Le µC MSP430

Le MSP430F2012 est un microcontrôleur développé par Texas Instruments faisant partie de la famille des MSP430. L'atout majeur de cette gamme de µC est sa consommation très faible d'énergie, ce qui en fait un outil adapté pour les applications embarquées nécessitant une forte autonomie.



Le kit utilisé pour cette série de TP est constitué de deux partie :

- L'interface de débogage servant à faire la liaison entre le microcontrôleur et l'ordinateur d'où s'effectue la programmation.
- Le microcontrôleur lui-même ainsi que ses connexions aux broches physiques et une diode électroluminescente.

<u>Caractéristiques techniques :</u>

Ce µC travaille avec des formats de variable de 16 bits, c'est-à-dire que les nombres utilisés par ce µC seront de ce format. Par exemple, un entier non-signé peut contenir n'importe quelle valeur allant de 0 à 2^{16}-1, ce qui fait 65 536 possibilités.

Le MSP430 travaille sous une faible tension avoisinant les 3,3 volts. Pour économiser son énergie, il existe plusieurs modes de « veille » que l'on appelle « Low Power Mode » dans le programme. Ces modes de veilles permettent d'économiser du courant. Lorsque le µC travaille à 1MHz (sous 2,2V), la consommation de courant est de l'ordre de 220 µA et cette consommation chute à 0,5 µA lors du passage en mode de veille. Ce qui fait respectivement une consommation de 484 µW et 1.1 µW. A titre de comparaison une calculatrice « type collège » consomme 600 µW et une calculatrice « graphique » 100 fois plus.

Bien entendu, il faut que le MSP430 puisse être réactif lorsque l'on le réveille de son mode économie d'énergie. Le temps de réveil garanti par la documentation est de moins de 1 µs, ce qui correspond à une période d'horloge lorsqu'elle tourne à 1MHz.

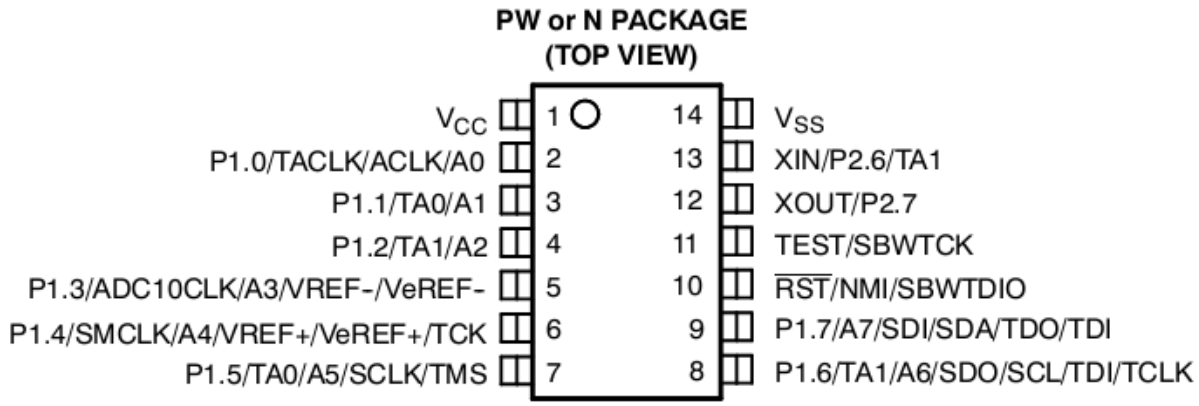
Et la fréquence nominale d'horloge dans tout cela ? Il existe plusieurs configurations d'horloges dont la plus rapide arrive jusqu'à 16MHz. Cela permet de sélectionner plusieurs signaux d'horloges en fonction des besoins.

Le MSP430 contient aussi un capteur de température interne, ce qui est pratique et permet de ne pas brancher de circuit externe. De plus, il y a aussi un Convertisseur Analogique Numérique (CAN) sur 12 bits à approximation successives, cela permet au microcontrôleur de lire des entrées analogiques. Il n'y a pas besoin de l'outil inverse : le Convertisseur Numérique Analogique (CNA) car cela peut se faire grâce à une Modulation de Largeur d'Impulsion (MLI).

Le programme est stocké dans une mémoire de 2 ko avec en plus 256 octets de mémoire flash. Enfin, le µC dispose de 128 octets de mémoire vive.

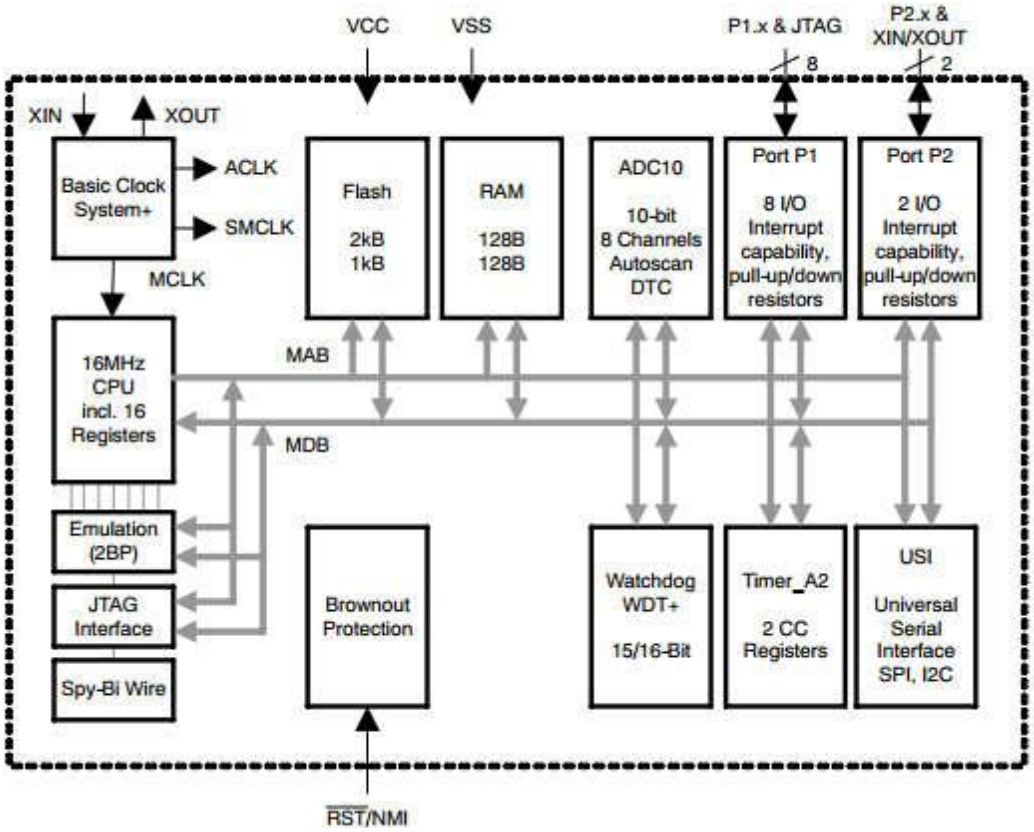
Fonctionnement des entrées/sorties :

Sur ce microcontrôleur, on distingue les connexions physiques (broches) et les connexions logiques. Les connexions physiques représentent ce que l'on peut observer sur une broche, on les numérote de P1 à P14.



On peut voir que chaque broche physique (ici notées de 1 à 14) est connectée à plusieurs ressources internes dont les entrées/sorties sont numérotées de P1.0 à P1.7. Pour choisir la ressource associée à la broche physique, on effectue une étape de configuration en début de programme.

Résumé des ressources internes au MSP430 :



Chacune des ressources internes présentées précédemment se trouve sur ce schéma. On peut y distinguer notamment :

- Le bus d'adresse (MAB) et le bus de données (MDB)
- Le système de gestion des horloges (Basic Clock System)
- Le calculateur (CPU)
- Les mémoires mortes (Flash) et vives (RAM)
- L'étage de conversion analogique-numérique (ADC10)
- Les ports d'entrées-sorties (Port P1 et Port P2)
- Le contrôle de plantage du programme (Watchdog)
- Le timer pour générer des signaux carrés de rapport cycliques variables (Timer_A1)
- Les bus de communications externes (Universal Serial Interface)

Introduction au traitement de signal : Notions de filtrage

Signal analogique :

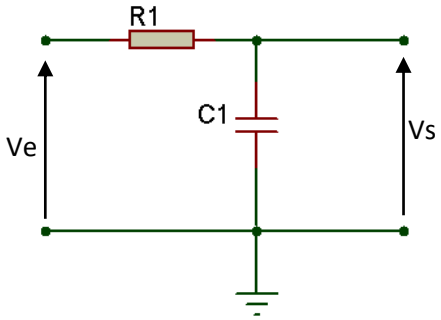
Un signal est constitué d'une somme d'informations, mais peut aussi contenir des perturbations nommées bruits, se glissant entre la source et le récepteur. Un signal électrique sera donc de même, une somme d'informations, véhiculées par de la tension (ex : principe du télégraphe) ou du courant (ex : boucle 4-20mA de capteur). Dans le but de ne conserver que les informations voulues, il est nécessaire de traiter le signal pour éliminer les parasites. Dans le cas présent, on s'intéressera au filtrage du signal, qui est l'une des opérations possibles de la discipline du traitement des signaux.

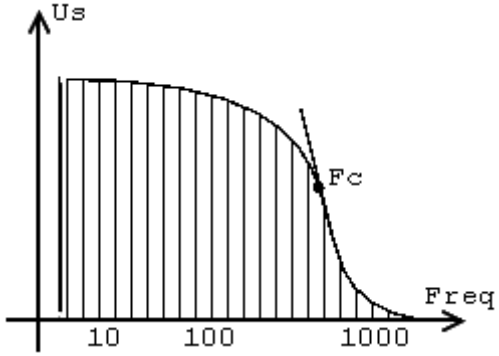
Les filtres en électronique réalisent une opération de mise en forme du signal, en traitant des valeurs du signal d'entrée successives relevée dans le temps. On obtient par ce traitement un signal de sortie comprenant uniquement les informations souhaitées.

Il existe donc divers types de filtres dont les principaux sont :

- Les filtres passe-bas (ou coupe-haut) ne laissant passer que les fréquences inférieures à la fréquence de coupure. (Ex : atténue les aigues en signal audio).
- Les filtres passe-haut (ou coupe-bas) ne laissant passer que les fréquences supérieures à la fréquence de coupure. (Ex : atténue les graves en signal audio).
- Les filtres coupe-bande (ou trappe, cloche) laissant passer que les fréquences en dehors d'une plage définie. On pourra notamment supprimer des bruits à une fréquence connue, par exemple le 50 Hz présent dans tout environnement possédant une alimentation au réseau de distribution électrique.
- Les filtres passe-bande ne laissant passer que les fréquences sur une plage définie (complémentaire au coupe-bande). On utilisa principalement ce filtre pour la réalisation des radios, TV analogique, pour contribuer à la réception en ne conservant qu'une plage précise (Ex : changement de chaine, de station, etc…)

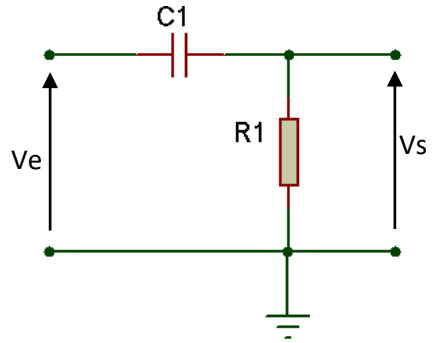
Le filtre sera réalisé au besoin avec des composants actifs et passifs tels que des résistances, bobines, condensateurs (passifs, pour l'atténuation) et des amplificateurs opérationnels ou AOP (actifs, pour l'amplification) en général. Chacun de ces composants influe par sa présence sur une fréquence dite de coupure. Cette fréquence, combinée au type de filtre réalisé permettra d'obtenir le traitement escompté. Elle correspond au point d'inflexion de la courbe de la bande passante (diagramme de Bode) du filtre, et se calcule (pour les circuits RC) par la relation : $Fc = \frac{1}{2\pi RC}$, avec R la valeur de la résistance en Ohm (Ω) et C la capacité du condensateur en Farad (F). A cette fréquence particulière, l'atténuation du signal sera de -3 dB.

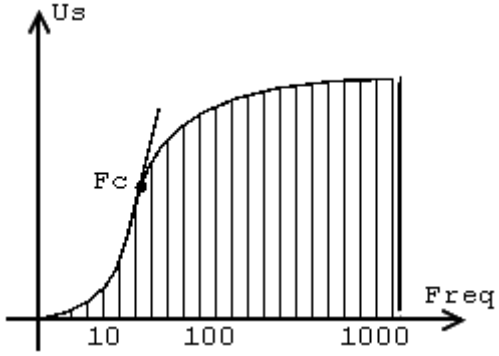




Passe-bas :

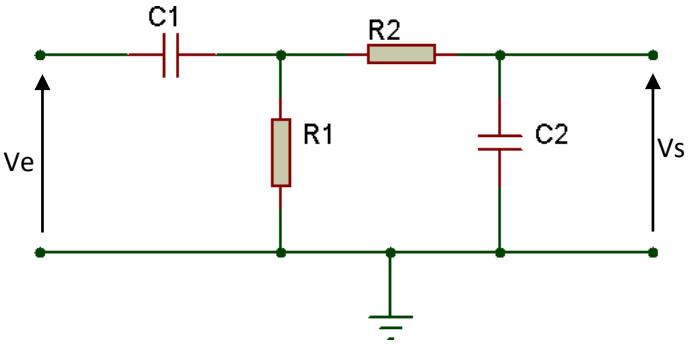
Le condensateur favorise l'écoulement à la masse des fréquences les plus élevées, et laisse passer les plus basses. Les fréquences sont atténuées peu avant la fréquence de coupure définie par les caractéristiques de R1 et C1.

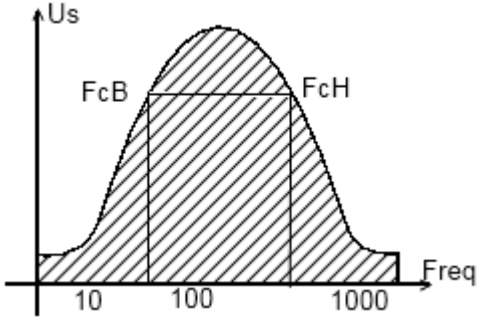




Passe-haut :

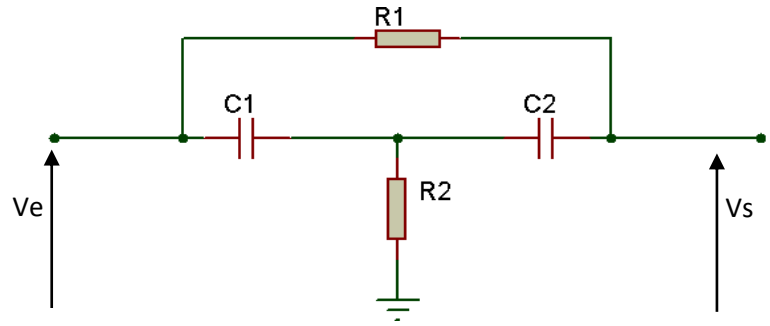
Le condensateur laisse passer les fréquences élevées, tout en atténuant les fréquences basses. On obtient une caractéristique d'atténuation symétrique à celle du passe-bas.

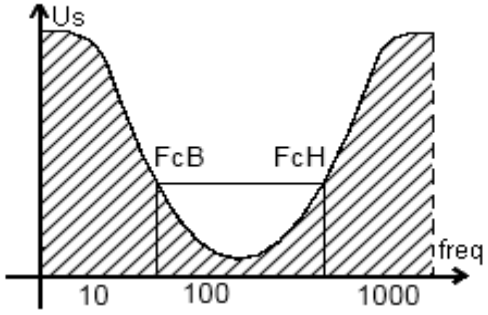




Passe-bande :

Il s'agit en réalité de la combinaison du filtre passe-bas et passe-haut, on atténue donc les fréquences situées avant FcB, fréquence de coupure basse du filtre [C1,R1], et celles situées après FcH, fréquence de coupure haute du filtre [R2,C2].



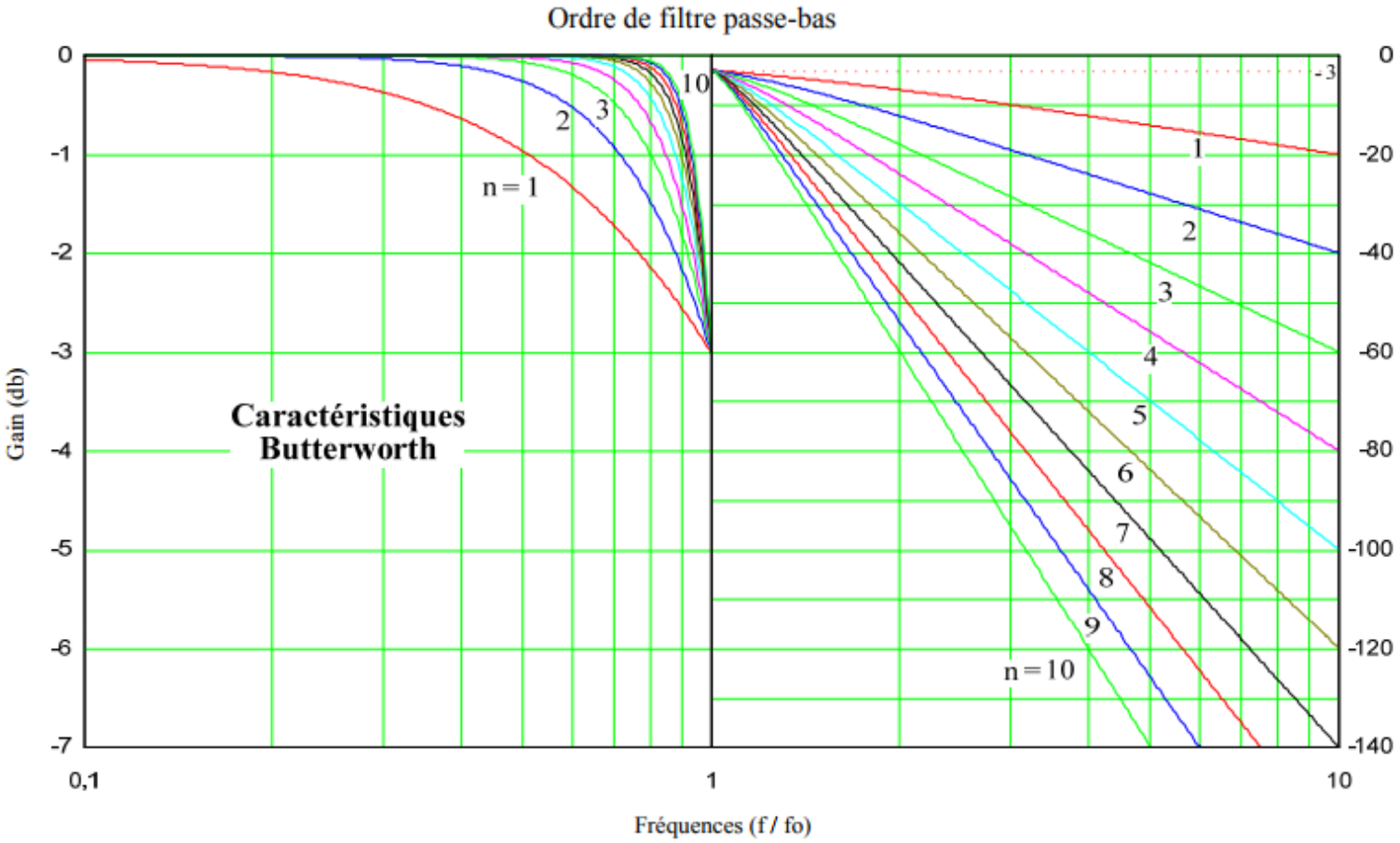


Coupe-bande :

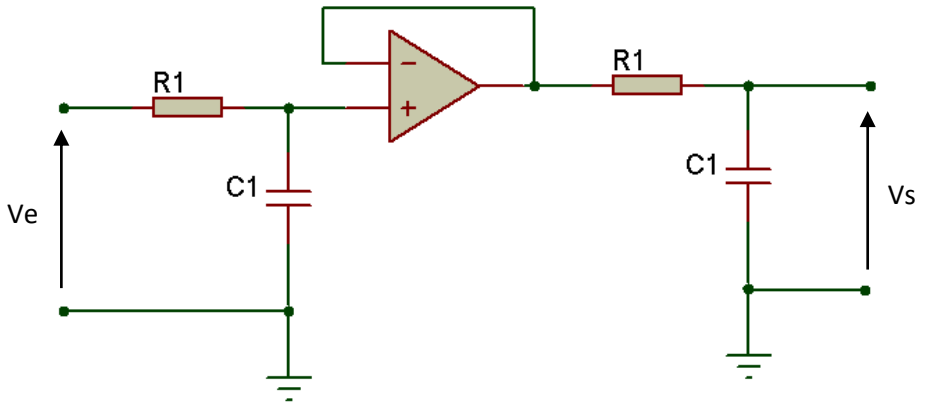
Ce filtre est constitué de deux passe-haut, ne possédant pas la même fréquence de coupure. Ici, le condensateur C1 attenue les fréquences inférieures à la fréquence de coupure FcB du filtre [C1,R1]. Les fréquences inférieures à FcB sont envoyées vers la sortie via la résistance R1. Les fréquences ayant franchies C1 sont alors de nouveaux filtrées par C2, qui atténue les fréquences inférieures à FcH, et n'envoie vers la sortie que les fréquences supérieures à FcH.

La gamme de fréquence visible en sortie est donc : [0 ;FcB[U]FcH ;+∞[.

Chacun des filtres cités précédemment est également caractérisé par un ordre. Ceux vus ci-dessus sont d'ordre 1. Par l'augmentation de l'ordre du filtre, on augmentera alors sa précision : atténuation plus proche de la fréquence de coupure dans la plage conservée, ce qui implique moins de perte de données, et atténuation plus forte après la fréquence de coupure, ce qui inhibera d'autant plus les parasites persistant éventuels. Ci-dessous sont visibles les caractéristiques des bandes passantes de filtres passe-bas de différents ordres selon le modèle de Butterworth, qui est le plus communément utilisé. Le terme « n » correspond à l'ordre du filtre. L'atténuation du filtre peut être estimée de la sorte : on aura $Gain(dB) = -20n\ dB/Décade$, avec « n » l'ordre du filtre.

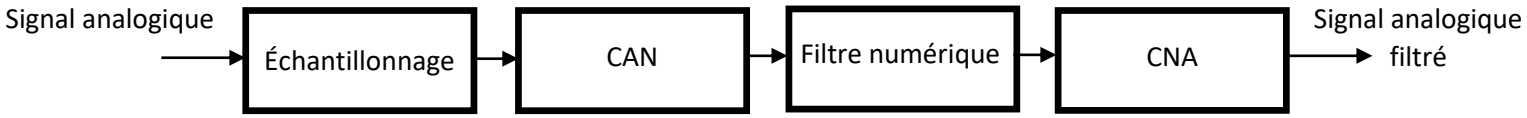


Pour réaliser un filtre d'ordre 2, on placera deux filtres du premier ordre en série, reliés par un amplificateur opérationnel en mode suiveur, comme sur le modèle suivant. Les valeurs des caractéristiques des résistances et des condensateurs sont les mêmes (pour les composants similaires). Dans un cas général d'ailleurs, les filtres d'ordre supérieur à deux sont constitués de filtres d'ordre 1 et/ou 2 mis en série. (Equivalent à la factorisation des polynômes ex : ordre 5 =ordre 2*ordre 2*ordre1)



Présentation du filtrage numérique

Contrairement aux filtres analogiques, qui sont réalisés à l'aide d'un agencement de composants physiques (résistance, condensateur, bobines, AOP, etc..) les filtres numériques sont réalisés soit par des circuits intégrés dédiés, processeurs programmables, soit par logiciel informatique. Pour la réalisation de ce type de filtrage, on retrouve en général une conception de ce type :



Le signal analogique est échantillonné à intervalles réguliers (fréquence d'échantillonnage fe). Le signal ainsi obtenu est converti en valeurs numériques, par le CAN (Convertisseur Analogique Numérique), qui pourront être traitées au travers de la fonction de transfert du filtre numérique. Après traitement, le signal numérique peut de nouveau être converti en signal analogique filtré par le CNA (Convertisseur Numérique Analogique) et être utilisé. Pour certaines applications, on utilisera directement le signal numérique, comme pour un capteur de température à afficheur digital par exemple.

Le filtrage numérique consiste donc à modifier le signal numérique, de manière à obtenir différents effets, comme le filtrage passe-bas, passe-bande, etc… vu précédemment de façon analogique, par une succession d'opérations mathématiques sur un signal discret. C'est-à-dire qu'il modifie le contenu spectral du signal d'entrée en atténuant ou éliminant certaines composantes spectrales indésirées. Ils peuvent théoriquement réaliser la totalité des effets de filtrage pouvant être définis par des fonctions mathématiques ou des algorithmes.

Les deux principales limitations des filtres numériques sont la vitesse et le coût. La vitesse du filtre est limitée par la vitesse du processeur (horloge, clock). Pour ce qui est du coût, celui-ci dépend du type de processeur utilisé. Il existe des microprocesseurs spécialisés dans le filtrage numériques (DSP Digital Signal Processor), capables de traiter des signaux numériques en temps réel avec une cadence très élevée (plusieurs millions d'échantillons par seconde).

Ils permettent d'obtenir des filtrages très efficaces, beaucoup plus facilement qu'avec les filtres analogiques, et à un coût bien plus faible. Par ailleurs, un filtre numérique est défini par une suite de coefficients, ce qui implique une facilité de modifications de filtrage importante et une modularité sans égale en analogique. Il est de plus, au moyen de carte telles que des microcontrôleurs, possible de programmer un filtre, et de modifier celui-ci sans même avoir à modifier la carte électronique.

Prise en main de l'interface de développement, du kit et des E/S

Pour commencer, nous allons faire clignoter la diode électroluminescente intégrée au MSP430. On dispose pour cela du programme suivant, les commandes sont en vert et les commentaires en gris :

```
//*******************************************************************************
// MSP430x2xx Demo - Software Toggle P1.0
//
//*******************************************************************************
#include "msp430x20x2.h"
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
        P1DIR = 0x01; // Set P1.0 to output direction
        unsigned int i;
        for (;;)
        {
                P1OUT ^= 0x01; // toggle P1.0 with exclusive-OR
                for (i=0 ; i<10000 ; i++);
        }
}
```

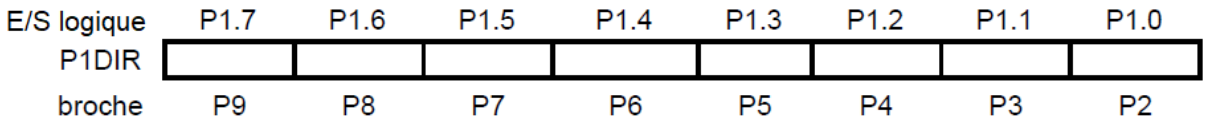
La commande « #include "msp430x20x2.h" » sert à inclure les bibliothèques pour les utiliser lors de la suite de la programmation.

Le programme en lui-même se situe entre les accolades de la fonction « main() ». Il n'y a pas besoin d'envoyer des paramètres lors de l'appel de cette fonction car on utilise le mot clé « void ».

La première instruction de la fonction fait mention du registre « WDTCTL », nommé WatchDog Timer ConTroL. A ce registre on affecte les commandes de « WDTPW » et de « WDTHOLD ». Le Mot clé « WatchDog Timer PassWord (WDTPW) » n'est ici que pour autoriser une action sur le registre du watchdog, on y affecte la mise en pause de celui-ci par l'ajout du « WatchDog Timer HOLD (WDTHOLD) ». Le tout permet de désactiver le watchdog pour la suite du programme.

Explication du watchdog : Un chien de garde (en anglais watchdog) est une part de circuit électronique ou de logiciel dont l'objectif est de s'assurer que le programme ne reste pas bloqué dans une étape précise. Si cela arrive, le système est censé redémarrer.

La ligne suivante « P1DIR = 0x01; » initie la configuration des registres internes du microcontrôleur. Les registres fréquemment utilisés pour contrôler les entrées et les sorties du µC sont configurables sur 8 bits. Chacun d'eux correspond à une entrée/sortie logique.



Le registre P1DIR sert à indiquer le mode de fonctionnement d'une entrée/sortie logique. Si l'on place le bit à la valeur 1, la fonction logique sera une sortie, si c'est à l'état 0, ce sera une entrée. On peut ainsi définir les huit fonctions logiques en une seule fois grâce à un seul octet. Par convention usuelle, on écrira cet octet sous la forme hexadécimale (chiffres de 0 à F). Par exemple le « 0x01 » est composé de « 0x » pour indiquer l'hexadécimal et de deux chiffres pouvant être décomposés en nombres de 4 bits comme il suit : 0 -> 0000 et 1 -> 0001.

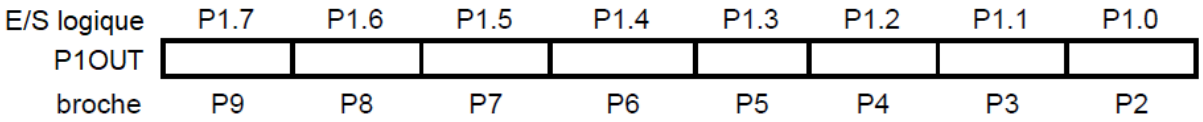
La broche P2 associée à la fonction logique P1.0 est donc définie comme une sortie numérique.

On définit ensuite un entier non-signé avec la commande « unsigned int i; ». Puisque le MSP430 travaille avec des nombres de 16 bits, on obtient une variable avec 2^16 possibilités c'est-à-dire 65 536 nombres possibles. Puisqu'elle est non-signée et de type entière, la plage de valeurs s'étend entre 0 et 65 535 avec un pas de 1. « i » représente uniquement le nom de la variable.

Nous avons abordé la partie initialisation du programme, les instructions mises en jeu ne sont exécutés qu'une seule fois. On passe maintenant à la seconde partie qui fonctionne en boucle tant que le µC est sous tension. Pour cela, on a besoin de ce que l'on appelle une boucle infinie.

Cette boucle infinie est constituée d'une instruction « for (;;) » et d'une paire d'accolades. Toutes les instructions se trouvant entre cette paire d'accolades seront exécutées à l'infini jusqu'à l'arrêt du µC. On aurait aussi pu écrire « while (1); » ce qui est parfaitement équivalent.

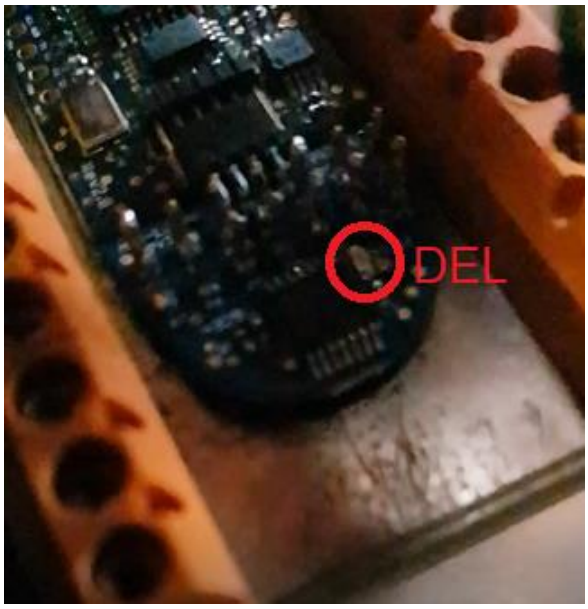
L'avant dernière instruction est la suivante « P1OUT ^= 0x01; ». On modifie la valeur binaire du registre P1OUT. Ce registre a pour finalité d'associer une résistance de tirage à la sortie. Cela place la sortie à l'état haut ou à l'état bas.



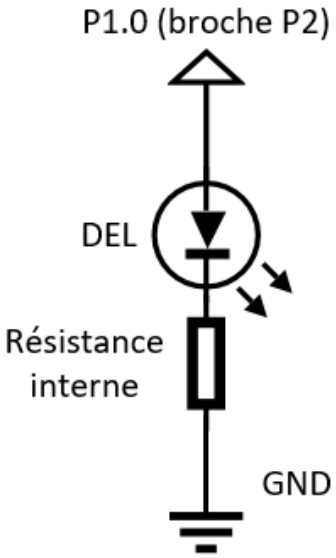
La partie « ^= » est l'abréviation de « P1OUT = P1OUT ^ 0x01; », c'est un ou exclusif (XOR). Dans le cas présent on applique un filtre à la valeur P1OUT sur le bit de poids faible associé à P1.0. Le résultat est que l'on inverse la valeur logique de ce bit. Par défaut, les valeurs du registre sont à l'état 0. Au premier tour de boucle, P1.0 passe à l'état 1. Au second, P1.0 retourne à l'état 0 et ainsi de suite.

L'ultime commande est la boucle « for (i=0 ; i<10000 ; i++); » sans accolades, ce qui signifie que la boucle n'engendre pas d'actions. Par contre, le temps d'exécution de chaque étape de cette boucle n'est pas nul, ce qui fait que lorsqu'on demande au microcontrôleur de compter jusque 10 000, cela lui prend un peu de temps.

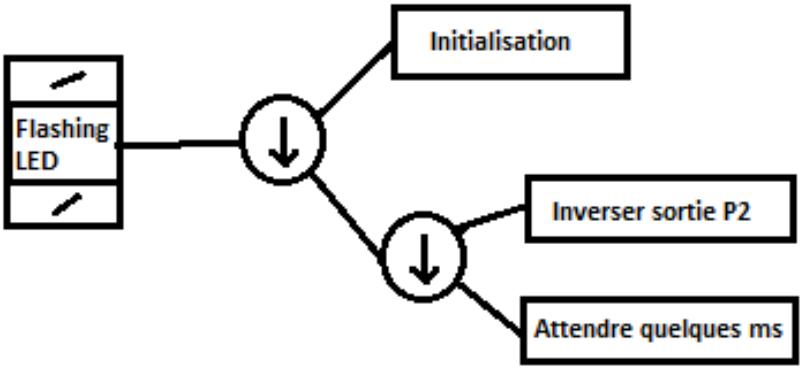
En résumé le programme exécuté permet de faire clignoter la DEL connectée à la broche P2 avec une période proportionnelle au nombre inscrit dans la boucle « for » et avec un rapport cyclique de 0.5. Voici en images le résultat du clignotement de la DEL :



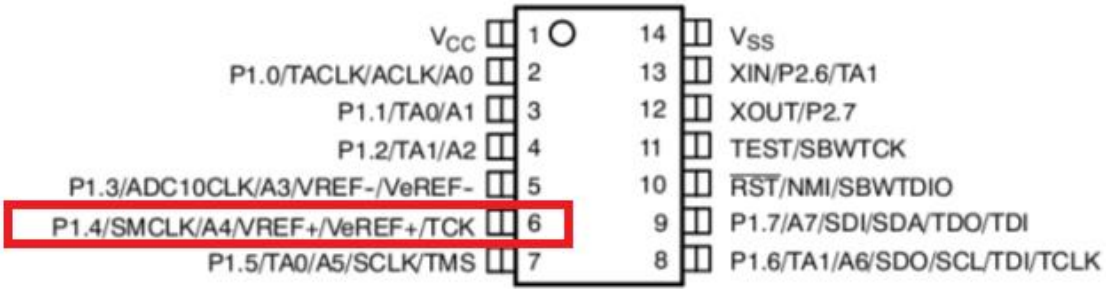
La DEL concernée est connectée sur la broche P2 avec le schéma de câblage suivant :



Par rétro-ingénierie, nous établissons ensuite l'algorithme du programme :



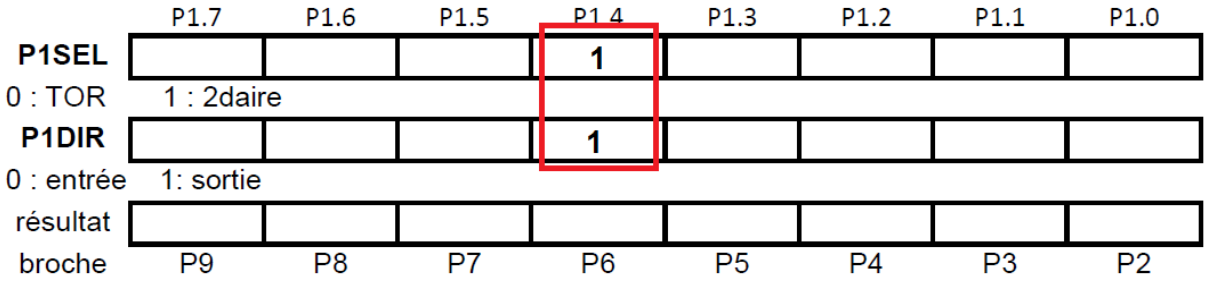
Pour la suite, on adjoint à ce programme le signal d'horloge SMCLK visualisable par oscilloscope sur la broche P6. Pour que le signal soit connecté à la broche de sortie, il faut préciser au multiplexeur ce que l'on souhaite voir. Par défaut, ce sont les entrées/sorties numériques qui sont actives, c'est pour cela que l'on n'avait pas besoin de cette étape pour le programme précédent.



Information : SMCLK est l'horloge utilisée pour le timer du watchdog lorsque celui-ci est actif.

Puisque la broche P6 sera utilisée par le signal de SMCLK, on ne pourra plus utiliser la broche P1.4 pendant ce temps. Pour configurer la fonction associée à P6, on doit modifier le registre P1SEL. Celui-ci fonctionne de manière identique aux autres, lorsque l'on place un bit à l'état bas logique (0), la broche est connectée à la fonction d'entrée/sortie tout ou rien (TOR). Si on le place à l'état haut (1), on utilisera la fonction secondaire. Il est nécessaire de placer un 1 pour P6 et on laisse à 0 pour le reste.

On précisera aussi dans le registre P1DIR que l'on visualise le signal sur une sortie.



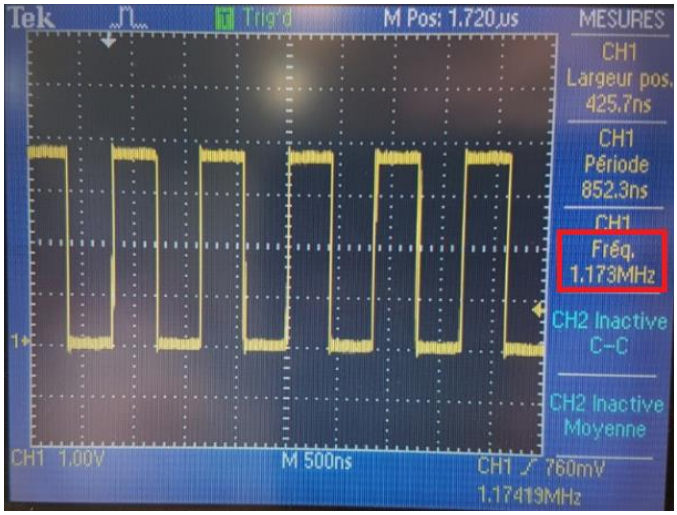
Traduit en langage informatique on utilise la commande « P1SEL = 0x10; », que l'on placera dans la phase d'initialisation.

Pour délivrer un signal carré de rapport cyclique 50% sur la broche P4, on procède de la même manière que pour le clignotement de la DEL en changeant la configuration de la broche de sortie concernée (et donc de la fonction entrée/sortie TOR concernée).

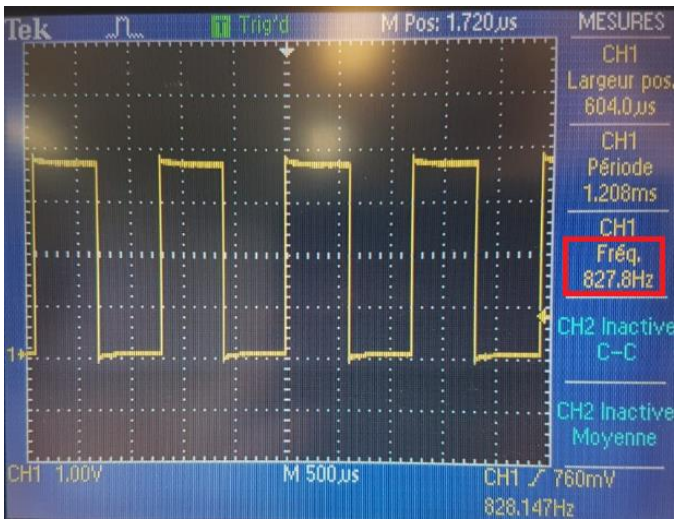
On obtient donc le programme suivant :

```
#include "msp430x20x2.h"
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
        P1DIR = 0x14; // Set P1.2 and P1.4 to output direction
        P1SEL = 0x10; // On place le signal SMCLK sur la broche P6
        unsigned int i;
        for (;;)
        {
                P1OUT ^= 0x04; // toggle P1.2 with exclusive-OR
                for (i=0 ; i<100 ; i++); // 10k = 8Hz -> 100 = 827Hz
        }
}
```

On observe à l'oscilloscope le signal d'horloge SMCLK en positionnant la sonde de tension sur la broche physique P6 du µC. Le signal obtenu est de fréquence 1,173 MHz :



On observe de la même manière l'oscillogramme du signal délivré par la sortie TOR sur la broche physique P4. Celui-ci a été réglé à 827 Hz pour correspondre au programme disponible au-dessus :



On cherche ensuite à regrouper les applications suivantes. Pour cela, on réunit dans un même programme la génération du signal SMCLK, le clignotement de la DEL en P1.0 et le signal carré sur P1.2. Afin de réaliser le clignotement et le signal carré pour qu'ils puissent avoir des périodes différentes, on va devoir utiliser un séparateur de compteur. Il n'est plus envisageable de disposer un « for (i=0 ; i<100 ; i++); » sans le décomposer.

Au lieu de ne déclarer que la variable i, nous allons associer un compteur pour chaque signal TOR. On ajoute donc une variable nommée « j » que l'on initialisera à la valeur 0. On incrémentera chaque variable lors de chaque tour de boucle et on inversera la valeur de la sortie souhaitée lorsque le contenu de la variable associée atteindra la valeur définie comme maximum. On obtient le programme suivant :

```
#include "msp430x20x2.h"
int main(void)
{
        WDTCTL = WDTPW + WDTHOLD;  // Stop watchdog timer
        P1DIR = 0x15;     // Set P1.0 to output direction
        P1SEL = 0x10;
        unsigned int i = 0;  // Initialisation de i à la valeur 0
        unsigned int j = 0;  // Initialisation de i à la valeur 0
        for (;;)
        {
                if(i==500) // La valeur associée de P1.0 est 500
                {
                        P1OUT ^= 0x01;       // toggle P1.0 with exclusive-OR
                }
                else
                {
                        i++;
                }
                if(j==100) // La valeur associée de P1.2 est 100
                {
                        P1OUT ^= 0x04;       // toggle P1.2 with exclusive-OR
                }
                else
                {
                        j++;
                }
        }
}
```

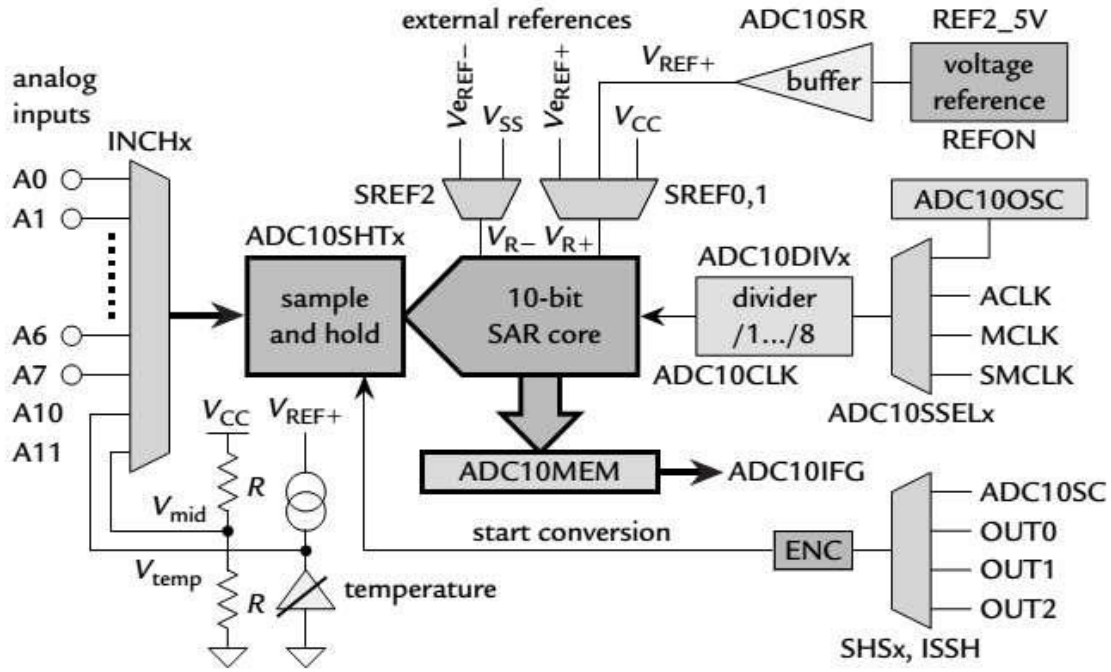
Dans ce dernier exemple, la fréquence du signal carré généré est 5 fois plus élevée que la fréquence de clignotement de la DEL. Ceci clôture l'étape de prise en main du MSP430 et des outils de développement.

Programmation de l'ADC10, de l'Echant/Bloqueur et du MUX

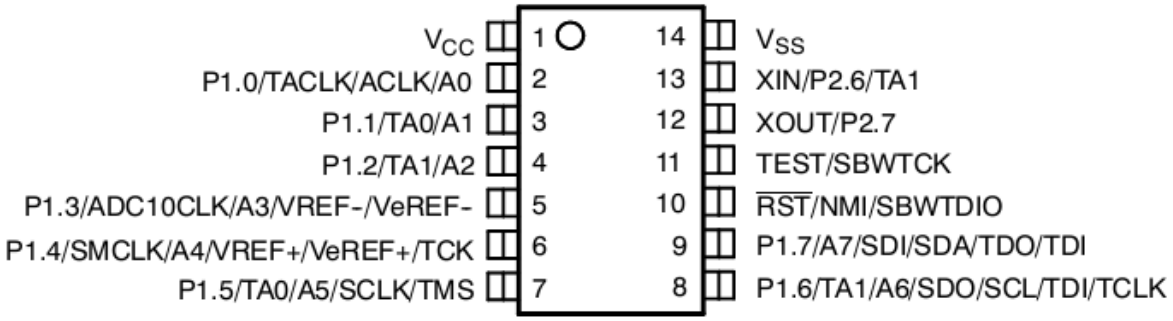
Pour commencer à filtrer un signal numérique, il faut pouvoir l'acquérir mais cette acquisition est faite à partir d'un signal analogique. Nous avons donc besoin de traduire ce signal analogique en signal numérique, c'est le rôle du CAN. La ressource du MSP430 qui joue ce rôle est nommée l'ADC10 (Analog-to-Digital Converter, 10 bits).

Puisque l'ADC10 travaille sur 10 bits, il renvoie en sortie une valeur parmi une liste de 2^{10} possibles, c'est-à-dire 1 024 valeurs possibles. Or le µC est alimenté par une tension de 3,3 V, la résolution de ce convertisseur est donc d'environ 3 mV par marche.

On a le schéma de fonctionnement de l'ADC10 du MSP430 :

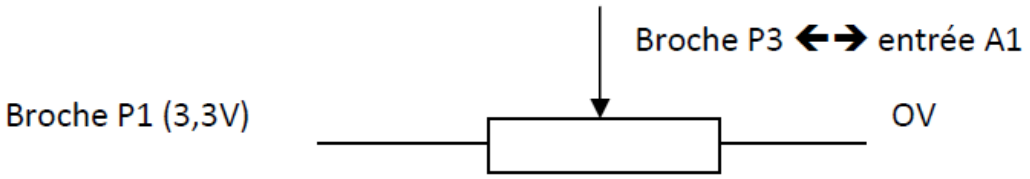


Les entrées analogiques de ce convertisseur sont nommées de A0 à A11. Les deux dernières sont connectées à des ressources internes au microcontrôleur, les autres peuvent être associées à des broches d'entrées/sorties, notées sur le schéma suivant :



L'échantillon analogique réalisé est gardé en mémoire pendant un temps compris entre 4 et 64 fois la période de l'horloge ADC10CLK. Plus ce temps sera long, plus l'échantillon sera précis. C'est l'échantillonneur bloqueur. L'échantillon analogique est ensuite converti en un nombre numérique de 10 bits et placé dans la mémoire nommée ADC10MEM.

Pour tester le fonctionnement du CAN, nous allons acquérir un signal analogique compris entre 0 V et 3,3 V. Ceci sera appliqué à l'aide d'un potentiomètre branché sur les deux tensions extrémums et dont le curseur sera connecté à la broche physique P3, correspondant à l'entrée analogique A1.



On dispose du programme suivant :

```
#include <msp430x20x2.h>
unsigned int analog_chan_1=0;
/************************************/
/* ADC10 Interrupt Service Routine */
/************************************/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10ISR(void)
{
        analog_chan_1=ADC10MEM;
}
/*****************/
/* Main routine */
/*****************/
void main(void)
{
        unsigned int i;
        WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
        P1DIR = BIT0 + BIT3; // On configure les broches P2 et P5 (P1.0 et P1.3) en sortie
        P1SEL = BIT3; // P5 (P1.3) est configuré en sortie secondaire (ADC10CLK pour visualisation)
        P1OUT = !BIT0; // On s'assure que P2 (P1.0) est en sortie avec pull-down

        /* Clocks init. */
        DCOCTL = CALDCO_16MHZ; // Set DCO to 16 MHz (MCLK = SMCLK )

        /* Analog to Digital SAR Converter Init. */
        ADC10AE0 = BIT1; // On configure l'ADC10AE0 en entrée analogique P3 (reliée à A1)

        // Paramétrage registre de contrôle 0 :
        ADC10CTL0 = SREF_0 + ADC10SHT_1 + ADC10ON + ADC10IE;
        // SREF_0 : SREFx = 000 On prend VCC en tension de référence
        // ADC10SHT_1 : On configure le sample and hold time à 8* la période de ADC10CLK
        // ADC10ON : Permet d'activer l'ADC10
        // ADC10IE : Permet de traiter automatiquement la gestion de l'interruption
```

```
        // Paramétrage du registre de contrôle 1 :
        ADC10CTL1 = INCH_1 + SHS_0 + ADC10SSEL_0 + ADC10DIV_0 + CONSEQ_0 ;
        // INCH_1 : On sélectionne la voie A1 (broche physique P3)
        // SHS_0 : On utilise le front montant de ADC10SC
        // ADC10SSEL_0 : On utilise l'horloge ADC10OSC
        // ADC10DIV_0 :  On divise l'horloge par 1
        // CONSEQ_0 : Le mode choisi est séquence de canaux converti une fois

        __bis_SR_register(GIE); // Enable Global Interrupt
        while(1)
        {
                // On place la valeur de i en fonction de l'entrée analogique
                i=(1023-analog_chan_1)*30+9000;
                do (i--); // On décrémente pour avoir un timer « fait maison »
                while (i !=0);
                P1OUT ^= 0x01; // toggle P1.0 clignotement de la DEL
                ADC10CTL0 |= ENC + ADC10SC;
                // On replace ENC à 1 pour préparer la conversion suivante et on déclenche en
                // plaçant ADC10SC à 1
        }
}
```

Explication des étapes de l'initialisation :

Pour l'initialisation du programme, on commence par déclarer un entier non-signé nommé i, on désactive aussi de watchdog. On configure la broche P2 en sortie TOR P1.0 avec une résistance de tirage vers le bas (pull-down) pour en faire une sortie à 0 V. On configure aussi la broche P5 en sortie secondaire pour visualiser le signal d'horloge de ADC10CLK.

Les notations sont ici différentes avec les mots clés « BIT0 » et « BIT3 » mais l'action reste identique que respectivement « 0x01 » et « 0x08 ». L'action « !BIT0 » signifie que l'on applique un NON logique sur l'ensemble du registre initialisé à la valeur « 0x01 ». Cela permet de faire passer le code binaire de « 0000 0001 » à « 1111 1110 », ce qui est identique à l'affectation de « 0xFE ».

Ensuite, on modifie le registre « DCOCTL » par la commande « DCOCTL = CALDCO_16MHZ; », ce qui affecte une fréquence d'horloge de 16 MHz à l'oscillateur RC interne réglable nommé DCO, permettant d'agir sur les horloges MCLK et SMCLK.

On affecte l'entrée analogique du convertisseur en paramétrant l'ADC10 pour qu'il utilise l'entrée A1, cela se fait avec la commande « ADC10AE0 = BIT1; ».

On va ensuite influer sur les deux registres ADC10CTL0 et ADC10CTL1 du convertisseur pour effectuer tous les autres réglages. Pour le premier registre, on affecte quatre commandes par la ligne « ADC10CTL0 = SREF_0 + ADC10SHT_1 + ADC10ON + ADC10IE; ». La première de ces commandes est la sélection des tension extremums. On choisit uniquement SREF_0 ce qui veut dire que la valeur

binaire de SREF est « 000 ». On aurait aussi pu l'écrire !SREF2 + !SREF1 + !SREF0. Le résultat est donc constitué de la première ligne du tableau suivant :

- **Choisir un VREF externe** par **SREFx**

15-13	SREFx	Select voltage reference:	V_{R+}	V_{R-}
		SREF2 SREF1 SREF0 = 000 ⇒	V_{CC}	V_{SS}
		SREF2 SREF1 SREF0 = 001 ⇒	V_{REF+}	V_{SS}
		SREF2 SREF1 SREF0 = 010 ⇒	Ve_{REF+}	V_{SS}
		SREF2 SREF1 SREF0 = 011 ⇒	Buffered Ve_{REF+}	V_{SS}
		SREF2 SREF1 SREF0 = 100 ⇒	V_{CC}	V_{REF-}/Ve_{REF-}
		SREF2 SREF1 SREF0 = 101 ⇒	V_{REF+}	V_{REF-}/Ve_{REF-}
		SREF2 SREF1 SREF0 = 110 ⇒	Ve_{REF+}	V_{REF-}/Ve_{REF-}
		SREF2 SREF1 SREF0 = 111 ⇒	Buffered Ve_{REF+}	V_{REF-}/Ve_{REF-}

Modes unipolaires : codes 0 à 3 Modes bipolaires : codes 4 à 7.

Nous appliquons donc la tension V_{CC} (3,3 V) en tension de référence et V_{SS} (0 V) en tension minimale. On s'intéresse ensuite à la durée d'échantillonnage. Quatre durées sont possibles grâce au réglage de deux bits distincts. Dans le cas présent on appelle la valeur binaire de 1 (traduit « 01 ») ce qui correspond à la deuxième ligne du tableau suivant :

- **La durée d'échantillonnage** est programmable par **ADC10SHTx** :

12-11	ADC10SHTx	ADC10 sample-and-hold time:
		ADC10SHT1 ADC10SHT0 = 00 ⇒ 4 x ADC10CLKs
		ADC10SHT1 ADC10SHT0 = 01 ⇒ 8 x ADC10CLKs
		ADC10SHT1 ADC10SHT0 = 10 ⇒16 x ADC10CLKs
		ADC10SHT1 ADC10SHT0 = 11 ⇒ 64 x ADC10CLKs

Plus la durée est longue, meilleure est la précision de l'échantillon acquis.

Nous configurons donc la période d'échantillonnage à 8 fois la période de l'horloge ADC10CLK. La mention d'ADC10ON est présente pour activer le convertisseur ADC10. De plus, la commande ADC10IE permet d'activer la gestion des interruptions (IE = Interrupt Enabled).

Le second registre de configuration du convertisseur est modifié quant à lui par cinq commandes : « ADC10CTL1 = INCH_1 + SHS_0 + ADC10SSEL_0 + ADC10DIV_0 + CONSEQ_0 ; ». La première, INCH_1, configure la voie échantillonnée. Dans cette configuration où quatre bits sont présents mais seules 12 possibilités coexistent. Pour le cas présent, nous appelons la première valeur ce qui traduit en binaire « 0001 » correspond à la deuxième ligne du tableau suivant :

- **Voie(s) échantillonnées :**

Les bits **INCHx** permettent de sélectionner 1 voie :

15-12	INCHx	Input channel select:
		INCH3 INCH2 INCH1 INCH0 = 0000 ⇒ A0
		INCH3 INCH2 INCH1 INCH0 = 0001 ⇒ A1
		INCH3 INCH2 INCH1 INCH0 = 0010 ⇒ A2
		INCH3 INCH2 INCH1 INCH0 = 0011 ⇒ A3
		INCH3 INCH2 INCH1 INCH0 = 0100 ⇒ A4
		INCH3 INCH2 INCH1 INCH0 = 0101 ⇒ A5
		INCH3 INCH2 INCH1 INCH0 = 0110 ⇒ A6
		INCH3 INCH2 INCH1 INCH0 = 0111 ⇒ A7
		INCH3 INCH2 INCH1 INCH0 = 1000 ⇒ Ve_{REF+}
		INCH3 INCH2 INCH1 INCH0 = 1001 ⇒ V_{REF-}/Ve_{REF-}
		INCH3 INCH2 INCH1 INCH0 = 1010 ⇒ Temperature sensor
		INCH3 INCH2 INCH1 INCH0 = 1011 ⇒ $(V_{CC} - V_{SS})/2$

Le registre de SHS (Sample and Hold Source) définit le déclencheur source de début de conversion. Dans le cas présent, SHS est définit à la valeur binaire 0, ce qui correspond à la première ligne du tableau suivant :

- **Start Conversion :** sur ↑ du signal SHI (Sample & Hold Input) lui-même choisi par les bits 10 et 11 SHSx de l'ADC10CTL0 :

11–10	SHSx	Sample-and-hold source:		
		SHS1 SHS0 = 00	⇒	bit ADC10SC
		SHS1 SHS0 = 01	⇒	TIMER_A Output Unit 1
		SHS1 SHS0 = 10	⇒	TIMER_A Output Unit 0
		SHS1 SHS0 = 11	⇒	TIMER_A Output Unit 2

On se réfère au front montant du bit ADC10SC dont la valeur sera pilotée par programmation.

On s'intéresse ensuite à ADC10SSEL qui est ajusté par la valeur binaire 0. Ce qui correspond à la première ligne du tableau suivant :

- **CLK de l'ADC10 :**

L'ADC10CLK est choisie parmi plusieurs signaux grâce aux bits **ADC10SSELx** de ADC10CTL1 :

4–3	ADC10SSELx	ADC10 clock source:		
		ADC10SSEL1 ADC10SSEL0 = 00	⇒	ADC10OSC
		ADC10SSEL1 ADC10SSEL0 = 01	⇒	ACLK
		ADC10SSEL1 ADC10SSEL0 = 10	⇒	MCLK
		ADC10SSEL1 ADC10SSEL0 = 11	⇒	SMCLK

L'horloge de référence qui deviendra ADC10CLK est reprise à partir de ADC10OSC. On divise cette horloge avec la commande ADC10DIV en la plaçant à la valeur binaire 0. Le fait de placer cette valeur divise l'horloge par 1, ce qui revient à ne pas la diviser. Le dernier réglage est nommé CONSEQ_0 et précise que l'acquisition se fait échantillon par échantillon.

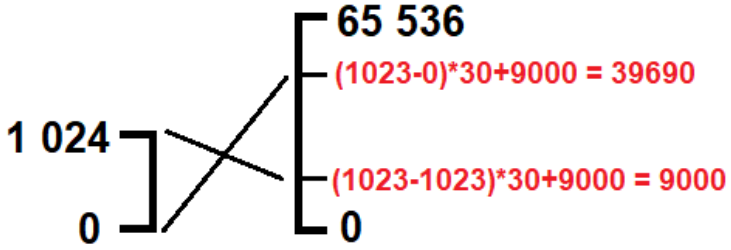
Pour terminer l'initialisation, on autorise le programme à être interrompu par le gestionnaire d'interruptions à l'aide de la ligne « __bis_SR_register(GIE); ».

On va ensuite détailler les étapes du programme se trouvant dans la boucle infinie.

Explication des étapes du programme :

Pour le début du tour de boucle on commence par l'affectation de la variable entière i :

« i=(1023-analog_chan_1)*30+9000; ». Cette variable contiendra un nombre inversement proportionnel à la valeur contenue dans l'ADC10MEM (via analog_chan_1) en adaptant les bornes des variables. Voici le schéma de la mise à l'échelle des variables (à gauche ADC10MEM et à droite i) :



Maintenant que nous avons obtenu une variable de format entier non-signé utilisable pour créer un timer approximatif. On décrémente ainsi la variable jusqu'à atteindre 0, étape où l'on arrête d'attendre

et où l'on passe à la suite du programme. Plus la variable contient une valeur élevée, plus on attend longtemps. On place les deux lignes suivantes :

« do (i--); // On décrémente pour faire un timer
  while (i !=0); // Tant qu'il en reste à décrémenter »

Pour finir, on inverse l'état de la DEL avec la commande « P1OUT ^= 0x01; » et on recommence une conversion avec « ADC10CTL0 |= ENC + ADC10SC; » pour replacer ENC à 1 et déclencher l'ADC10 sur le front montant d'ADC10SC. Les caractères « |= » sont un raccourci pour dire « ADC10CTL0 = ADC10CTL0 | (ENC + ADC10SC); ». C'est l'opérateur logique OU qui s'effectue bit par bit (en anglais OR) permettant d'activer le bit de sortie si au moins un des deux bits d'entrée est à l'état haut.

Indépendamment du programme en boucle faisant clignoter la DEL, il faut pouvoir remettre à jour le contenu de la variable analog_chan_1 à chaque fin de conversion. C'est le rôle de l'interruption.

Explication de l'interruption :

Une interruption est l'arrêt temporaire de l'exécution du programme pour pouvoir exécuter un autre programme que l'on appelle routine ou service d'interruption. Ici, à chaque fin de conversion on applique l'interruption suivante :

```
#pragma vector=ADC10_VECTOR
__interrupt void ADC10ISR(void)
{
        analog_chan_1=ADC10MEM;
}
```

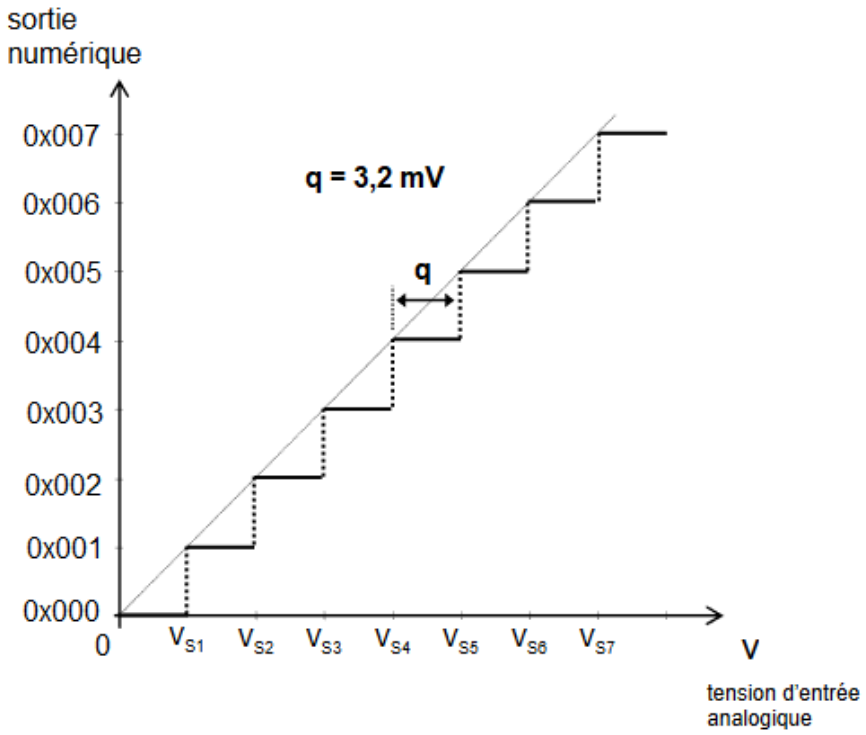
Le contenu de l'interruption (entre accolades) replace le contenu de l'ADC10MEM dans la variable analog_chan_1.

Questions :

Quelle est la plage de tension que ADC10 peut convertir ? Quel est l'intervalle des valeurs de N, en décimal et en hexadécimal ? Tracez N = f(UA1). Justifiez l'appellation « mode de conversion unipolaire ».

Comme expliqué précédemment, les tensions mises en jeu vont de 0 volt à 3,3 volts. Puisque les valeurs de N vont de 0 à 1023 (2^{10} possibilités car c'est sur 10 bits), ou de 0x000 à 0x3FF en hexadécimal, l'écart de tension entre deux valeurs consécutives est de 3,3 / 1024 soit 3,2 millivolts.

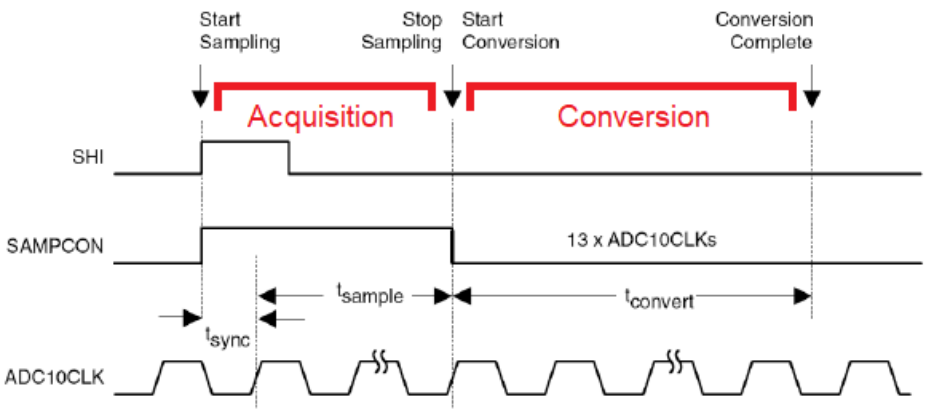
Voici la caractéristique de la valeur hexadécimale en fonction de la tension d'entrée N = f(V_{A1}) sur les huit premières valeurs :



Un mode de conversion bipolaire permet de convertir des valeurs de tension négatives. Ce qui n'est pas le cas ici. Lorsque les tensions mises en jeu sont strictement positives on parle de convertisseur unipolaire.

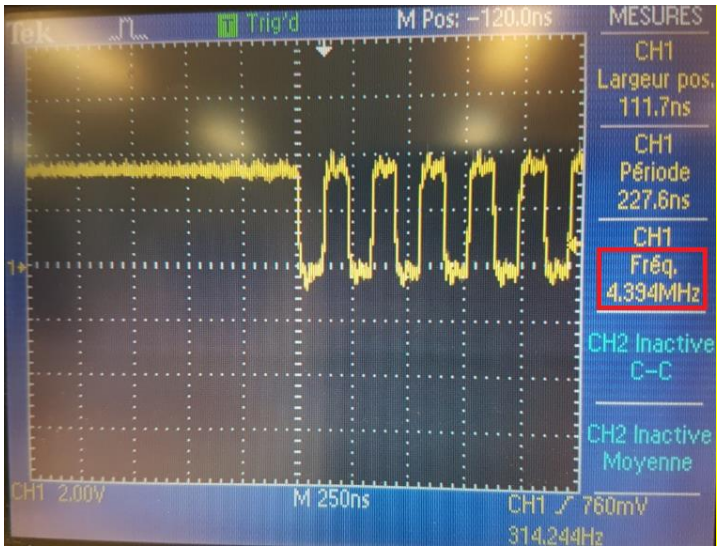
Quelle est la valeur de la fréquence de l'horloge ADC10CLK ? Recopiez alors les chronogrammes se trouvant dans les « Notes de lecture MSP » et chiffrez la durée d'acquisition d'un échantillon et la durée de conversion.

La valeur de la fréquence de l'horloge ADC10CLK est de 4MHz. Cela se justifie avec la division par 4 de la fréquence de base qui est de 16 MHz. La période de l'horloge ADC10CLK est donc de 250 nanosecondes. La durée d'échantillonnage est configurée sur 8 fois la durée de l'horloge ADC10CLK ce qui fait huit fois la période de 250 ns, on ajoute à cela une période pour le temps de synchronisation. La durée d'acquisition est finalement de 2,25 µs. La durée de conversion est fixe et est de 13 fois la période de l'horloge ADC10CLK. Le temps de conversion est de 3,25 µs.

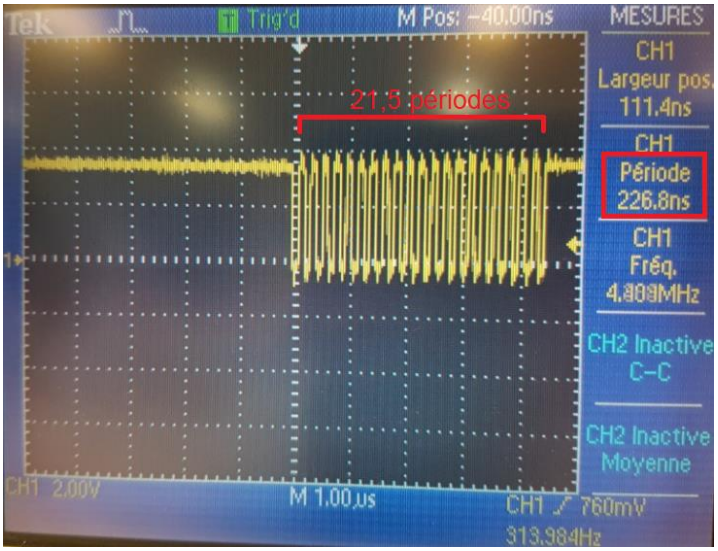


Mesurez la fréquence ADC10CLK présente en broche physique P5. Justifiez le nombre de périodes des salves. Enregistrez une copie-écran de ce signal.

La fréquence mesurée de l'horloge ADC10CLK sur la broche physique P5 est de 4.4MHz. Nous avons mesuré 230 ns par période comme sur l'oscillogramme suivant :



On compte 21 périodes plus un front montant sur l'oscillogramme suivant. Chacune des périodes dure 230 ns, pour obtenir la durée de la salve, on multiplie par 21,5. Cela nous donne le résultat de 4,945 µs par salve. Les 21,5 périodes sont constituées des 13 périodes de conversion, ajouté aux 8 périodes d'échantillonnage ainsi qu'à la demi-période de synchronisation. Voici l'oscillogramme répertoriant la salve :



Programme du bonus avec le capteur de température :

Dans la suite du sujet, on se propose d'utiliser le capteur de température interne au MSP430 pour récupérer une indication de température. Nous avons créé le programme suivant :

```
#include <msp430x20x2.h>
unsigned int analog_chan_1=0;
```

```
/***********************************/
/* ADC10 Interrupt Service Routine */
/***********************************/
#pragma vector=ADC10_VECTOR // Interruption déclenchée par le timer
__interrupt void ADC10ISR(void)
{
        analog_chan_1=ADC10MEM;
}
/*****************/
/* Main routine */
/*****************/
void main(void)
{
        float i;
        WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
        P1DIR = BIT0 + BIT3;    // On configure les broches P2 et P5 (P1.0 et P1.3) en sortie
        P1SEL = BIT3;    // P5 (P1.3) est configuré en sortie secondaire (ADC10CLK pour visualisation)
        P1OUT = !BIT0; // On s'assure que P2 (P1.0) est en sortie avec pull-down

        /* Clocks init. */
        DCOCTL = CALDCO_16MHZ;  // Set DCO to 16 MHz (MCLK = SMCLK )

        /* Analog to Digital SAR Converter Init. */
        ADC10AE0 = BIT1; // On configure l'ADC10AE0 en entrée P3

        // Pour la T°
        ADC10CTL0 = SREF_1 | ADC10SHT_3 | ADC10SR | ADC10IE;
        ADC10CTL1 = INCH_10 | SHS_0 | ADC10DIV_3 | ADC10SSEL_0 | CONSEQ_0;

        __bis_SR_register(GIE); // Enable Global Interrupt
        while(1)
        {
                i=analog_chan_1*3.22266; // i est la valeur de tension en mV
                i=(i-986)/3.55; // i devient la valeur de température en °C
        }
}
```

Pour la partie de la récupération de la valeur de température, il faut modifier les registres d'initialisations de l'ADC10 afin de le connecter avec le capteur de température (A10) et non plus sur la broche d'entrée P3 (A1). De plus, il faut aussi utiliser la même plage de tension que pour le capteur de température. On associera donc la tension de référence maximum à V_{REF+}.

Pour la partie calcul de la variable contenant la valeur de température en °C, on s'aide de la formule : Vcapteur (mV) = 3,55*T(°C) + 986.

On déclare un nombre à virgule flottante nommé i. Ce nombre a pour but de retranscrire la valeur de tension en millivolts délivrée par le capteur à partir de la valeur de sortie de l'ADC10. On le multiplie donc par 3 300 (tension de référence) et on le divise par 1 024 (nombre de possibilités du compteur), ce qui revient à appliquer la commande « i=analog_chan_1*3.22266; ».

Pour trouver la température, on retravaille la formule donnée pour avoir la température en fonction de la tension :

$V_{capteur}(mV) = 3,55*T(°C) + 986$
$V_{capteur}(mV) - 986 = 3,55*T(°C)$
$T(°C) = (V_{capteur}(mV) - 986) / 3,55$

On applique la dernière ligne qui est le calcul de la température : « i=(i-986)/3.55; ».

Maintenant que nous avons appliqués l'équivalent d'un convertisseur analogique-numérique, il reste deux étapes : le filtrage numérique et la conversion numérique-analogique. Puisque l'étape de filtrage dépend du filtre que l'on souhaite appliquer, nous allons réaliser l'étape de reconversion analogique directement à partir de la valeur numérique récupérée en sortie de l'ADC10. Cela créera un filtre avec une fonction de transfert de 1.

Programmation du Timer

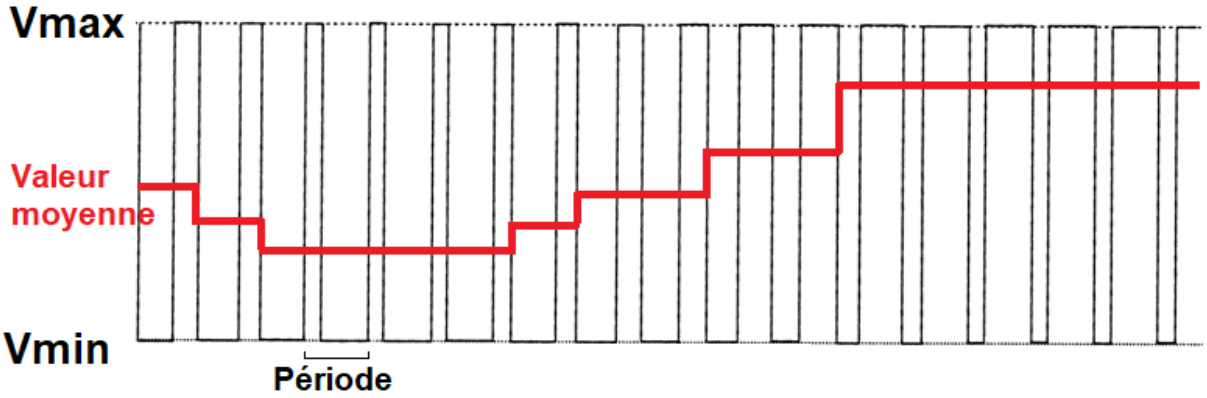
Pour Effectuer un convertisseur numérique-analogique, nous allons devoir utiliser la technique de la modulation de largeur d'impulsion.

Explication du principe de la MLI :

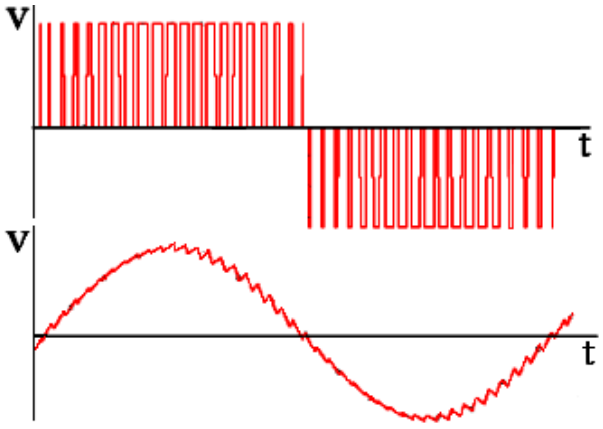
La Modulation de Largeur d'Impulsion (MLI) est appelée en anglais Pulse Width Modulation (PWM). C'est une technique utilisée pour créer des signaux analogiques lorsque l'on ne dispose que de circuits à fonctionnement tout ou rien. C'est une technique bien répandue dans le milieu des µC.

Dans d'autres cas, on est amené à utiliser des MLI pour certaines applications à command numériques comme la commande de servomoteurs par exemple.

Le signal est constitué de trois grandeurs principales : l'amplitude, la fréquence du signal numérique et son rapport cyclique. La fréquence est généralement fixe est doit être suffisamment élevée pour ne pas interférer avec les plus grandes fréquences souhaitées du signal numérique. Le rapport cyclique est ce qui est le plus susceptible d'évoluer, il peut varier de 0 à 100% et cette variation peut être moyennée pour obtenir une valeur de tension du signal en fonction de l'amplitude de celui-ci.



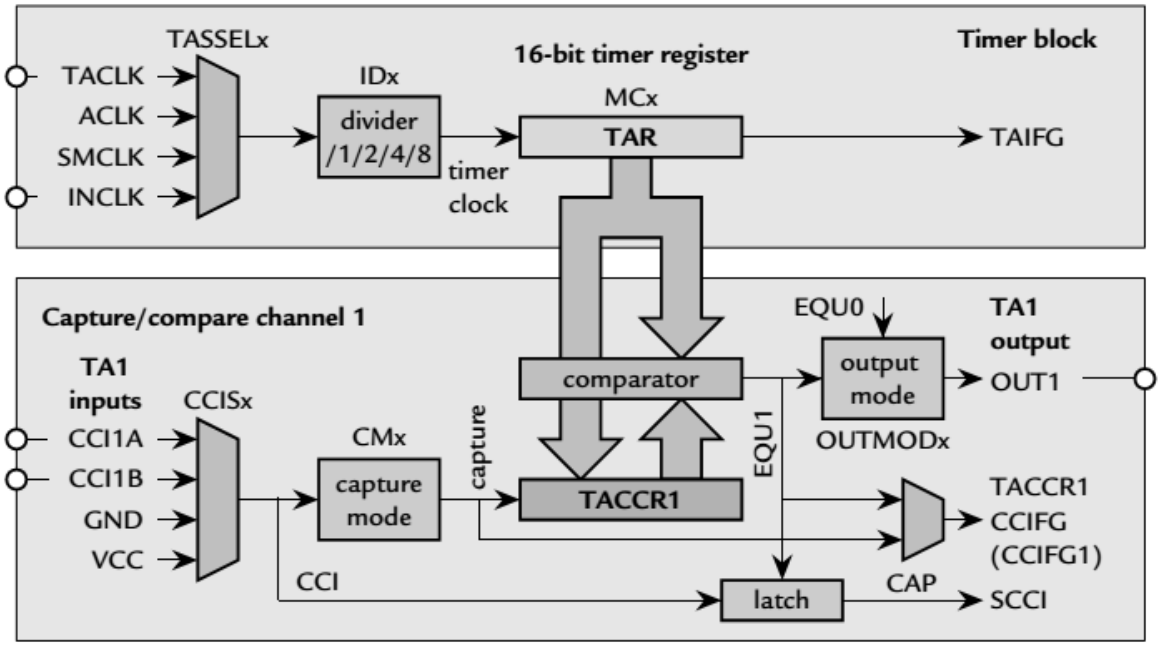
En lissant la valeur moyenne du signal obtenu par MLI, on peut reconstituer le signal analogique souhaité comme dans l'exemple suivant où l'on reconstruit une sinusoïde à partir du signal MLI :



Elaboration du timer :

A l'aide du Timer du MSP430, nous allons créer un signal de modulation de largeur d'impulsion. Ceci permettra de reconstruire par la suite un signal lissé pour obtenir un signal analogique. L'utilisation du Timer aura pour effet d'agir sur le signal carré de manière plus précise que le délai de boucle utilisé précédemment.

Les ressources du Timer sont multiples. On les retrouve dans le schéma suivant :



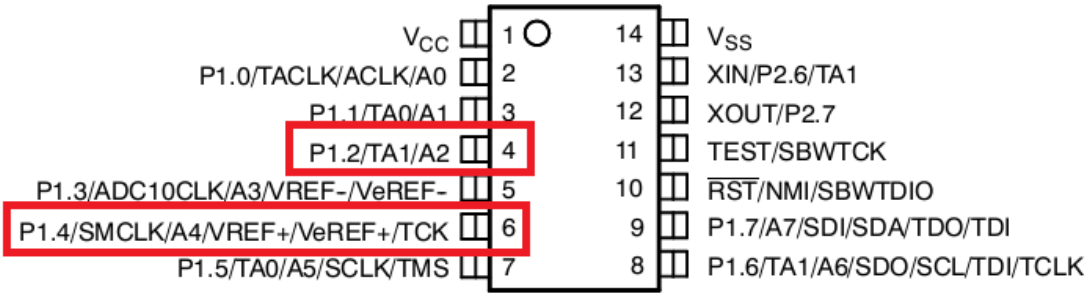
En premier lieu, il est nécessaire de choisir une horloge de base pour commencer le comptage. On peut diviser cette horloge pour obtenir une fréquence plus basse. Ensuite, on choisit le mode de fonctionnement parmi trois : Le mode continu qui compte jusqu'à la valeur maximum puis recommence en partant de 0. Le mode Up qui compte jusqu'à une valeur prédéfinie avant de revenir à 0. Et le mode Up/Down qui compte jusqu'à la valeur définie puis décompte pour revenir à la valeur 0.

Pour mettre en œuvre les paramétrages évoqués, on a le programme suivant :

```
#include <msp430x20x2.h>
unsigned char DCO[8]= {0x00,0x20,0x40,0x60,0x80,0xA0,0xC0,0xE0};
unsigned char RSEL[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
void main(void)
{
        WDTCTL = WDTPW + WDTHOLD; // Stop Watch Dog Interrupt
        /* Parametrages des sorties */
        P1DIR |= 0x14; // 0001 0100 -> On place P4 et P6 en sortie
        P1SEL |= 0x14; // 0001 0100 -> On place les sorties P4 et P6 en secondaires
        // P4 représente le timer et P6 le SMCLK
        // Basic Clock System
        BCSCTL1 = XT2OFF + RSEL[7]; // Pas de Quartz externe -> RSEL[7]
        DCOCTL = DCO[0];    // cf Notes de Lecture, Chap.5 CLOCK SYSTEM
        // DC0CTL = 0x00 première ligne du tableau des frequences -> RSEL[7] d'où 714kHz
```

```
        /* Reglages TimerA */
        TACTL = TASSEL_2 + ID_0 + MC_1;
        CCTL1 = OUTMOD_7;
        CCR0 = 100 ; // Période du PWM = 100 µs avec 100 pour 100µs et 9.5kHz
        // pour 1000 on a 963 Hz
        CCR1 = 4;    // duty cycle 4%
        __low_power_mode_1();
}
```

Pour commencer, comme à notre habitude, on désactive le watchdog. On configure ensuite les broches P4 et P6 en sorties secondaires en paramétrant les registre P1DIR et P1SEL. La sortie secondaire de P4 est TA1, ce qui correspond à la sortie du timer_A. La sortie secondaire de P6 est le signal d'horloge SMCLK.



On désactive ensuite le quartz externe par la commande « BCSCTL1 = XT2OFF + RSEL[7]; » ce qui a aussi pour effet de sélectionner la huitième valeur du tableau RSEL. Cette valeur est donc celle de ce tableau prédéfini : « RSEL[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}; ». On conserve en mémoire la valeur décimale 7.

La ligne suivante « DCOCTL = DCO[0]; » procède de même pour la sélection de la valeur dans le tableau de DCO. Ces valeurs sont hexadécimales et se trouvent dans le second tableau définit en initialisation de variables « unsigned char DCO[8]= {0x00,0x20,0x40,0x60,0x80,0xA0,0xC0,0xE0}; ». On sélectionne la première valeur de ce tableau, ce qui correspond à la valeur 0x00.

La combinaison des deux valeurs sélectionnées permet de définir la fréquence de base du DCO que l'on utilisera. Le tableau suivant résume toutes les combinaisons paramétrables de fréquences possibles :

FREQUENCE du DCO en MHz

		RSEL0	RSEL1	RSEL2	RSEL3	RSEL4	RSEL5	RSEL6	RSEL7
		0x00	0x01	...					0x07
DCO0	0x00	0,106	0,132	0,187	0,263	0,367	0,520	0,714	1,000
DCO1	0x20	0,114	0,143	0,202	0,284	0,396	0,562	0,771	1,080
DCO2	0x40	0,124	0,154	0,218	0,307	0,428	0,607	0,833	1,166
DCO3	0x60	0,134	0,160	0,230	0,320	0,450	0,630	0,870	1,220
DCO4	0x80	0,144	0,173	0,248	0,346	0,486	0,680	0,940	1,318
DCO5	0xA0	0,156	0,187	0,268	0,373	0,525	0,735	0,937	1,423
DCO6	0xC0	0,168	0,202	0,290	0,403	0,567	0,794	1,110	1,537
DCO7	0xE0	0,182	0,218	0,313	0,435	0,612	0,857	1,199	1,660

		RSEL8	RSEL9	RSEL10	RSEL11	RSEL12	RSEL13	RSEL14	RSEL15
		0x08	0x09	0x0A	...				0x0F
DCO0	0x00	1,430	2,050	2,940	3,761	5,060	6,803	9,900	13,550
DCO1	0x20	1,544	2,214	3,175	4,062	5,465	7,347	10,692	14,634
DCO2	0x40	1,668	2,391	3,300	4,387	5,902	7,935	11,547	15,805
DCO3	0x60	1,730	2,480	3,540	4,550	6,100	8,170	11,900	16,000
DCO4	0x80	1,868	2,678	3,823	4,914	6,588	8,824	12,852	17,280
DCO5	0xA0	2,018	2,893	4,129	5,307	7,115	9,529	13,880	18,662
DCO6	0xC0	2,179	3,124	4,459	5,732	7,684	10,292	14,991	20,155
DCO7	0xE0	2,354	3,374	4,816	6,190	8,299	11,115	16,190	21,768

L'intersection des deux valeurs donne une fréquence de 1 MHz, l'oscillateur DCO sera configuré pour tourner à cette fréquence.

On paramètre les autres éléments du timer par la commande suivante « TACTL = TASSEL_2 + ID_0 + MC_1; ». Le registre de contrôle du timer (Timer_A ConTroL) prends comme affectations trois paramètres.

Le premier de ces paramètres est « TASSEL_2 » qui sélectionne les horloges possibles pour la configuration du timer. Quatre horloges sont disponibles. En prenant la valeur binaire de 2, on sélectionne l'horloge SMCLK comme nous le montre le tableau suivant :

Bit		Description		
9-8	TASSELx	Timer_A clock source:		
		TASSEL1 TASSEL0 = 00	⇒	TACLK
		TASSEL1 TASSEL0 = 01	⇒	ACLK
		TASSEL1 TASSEL0 = 10	⇒	SMCLK
		TASSEL1 TASSEL0 = 11	⇒	INCLK

Le paramètre « ID_0 » permet de choisir la division appliquée au signal d'horloge. Il existe quatre choix possibles. Dans le cas présent, nous sélectionneront celui associé à la valeur binaire 0, c'est celui de la première ligne du tableau suivant :

Bit		Description		
7-6	IDx	Clock signal divider:		
		ID1 ID0 = 00	⇒	/ 1
		ID1 ID0 = 01	⇒	/ 2
		ID1 ID0 = 10	⇒	/ 4
		ID1 ID0 = 11	⇒	/ 8

Le taux de division appliqué est « 1 », cela signifie que l'on ne divise pas la fréquence d'horloge appliquée au timer.

Le dernier paramètre fait état du mode de conversion appliqué au timer. Le « MC_1; » sélectionne la valeur binaire 1 parmi les quatre disponibles : les trois modes présentés plus haut ainsi que le mode arrêt pour mettre en pause le timer. Le tableau suivant met en évidence la sélection du mode Up :

Bit		Description
5-4	MCx	Clock timer operating mode: MC1 MC0 = 00 ⇒ Stop mode MC1 MC0 = 01 ⇒ Up mode MC1 MC0 = 10 ⇒ Continuous mode MC1 MC0 = 11 ⇒ Up/down mode

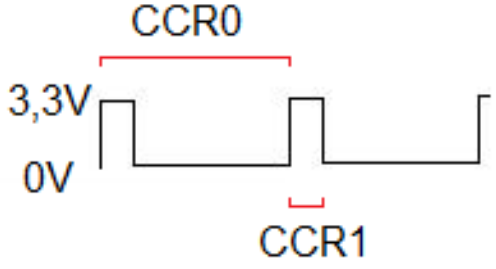
Il faut ensuite choisir le mode associé au mode de comptage, il en existe huit. La commande « CCTL1 = OUTMOD_7; » permet de définir la dernière :

OUTMODx	Mode	Description
000	Output	The output signal OUTx is defined by the bit OUTx
001	Set	OUTx = 1 ⇒ timer = TACCRx OUTx = 0 ⇒ timer = 0 or until another output mode is selected and affects the output
010	Toggle/Reset	OUTx = toggle ⇒ timer = TACCRx OUTx = 0 ⇒ timer = TACCR0
011	Set/Reset	OUTx = 1 ⇒ timer = TACCRx OUTx = 0 ⇒ timer = TACCR0
100	Toggle	OUTx = toggle ⇒ timer = TACCRx The output period is double the timer period
101	Reset	OUTx = 0 ⇒ timer = TACCRx OUTx = 1 ⇒ another output mode is selected and affects the output
110	Toggle/Set	OUTx = toggle ⇒ timer = TACCRx OUTx = 1 ⇒ timer = TACCR0
111	Reset/Set	OUTx = 0 ⇒ timer = TACCRx OUTx = 1 ⇒ timer = TACCR0

Ce mode permet d'envoyer un état haut en sortie TA1 lorsque l'on atteint la valeur haute du comptage et d'envoyer un état bas lorsque l'on atteint une valeur balise.

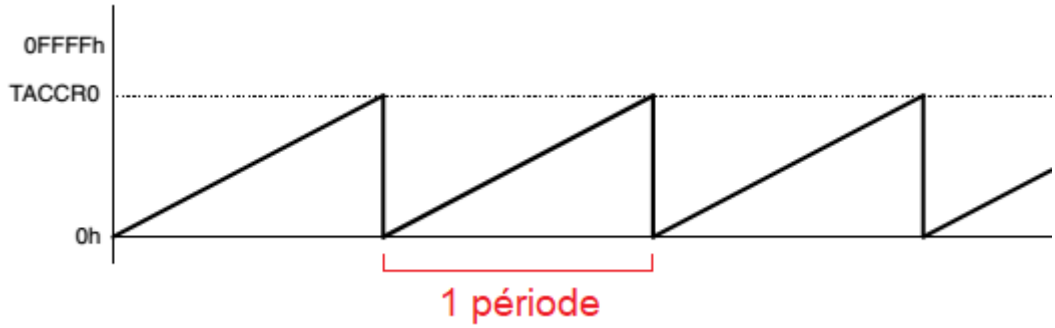
Maintenant que tous les réglages sont faits, il reste à définir le temps de cycle et le rapport cyclique du signal que l'on souhaite générer. Puisque l'on travaille en mode Up, il faut choisir une valeur de pic. Cette valeur maximale de comptage est matérialisée par le registre nommé CCR0, elle est comprise entre 0 et 65 535.

Pour le second registre nommé CCR1, cela définit le rapport cyclique. Sa valeur doit obligatoirement être comprise entre 0 et CCR0. Plus sa valeur est proche de 0, plus le rapport cyclique sera faible car le temps passé à l'état haut sera petit. A contrario, plus la valeur est proche de CCR0, plus le rapport cyclique sera élevé car on déclenche l'état bas plus tardivement.

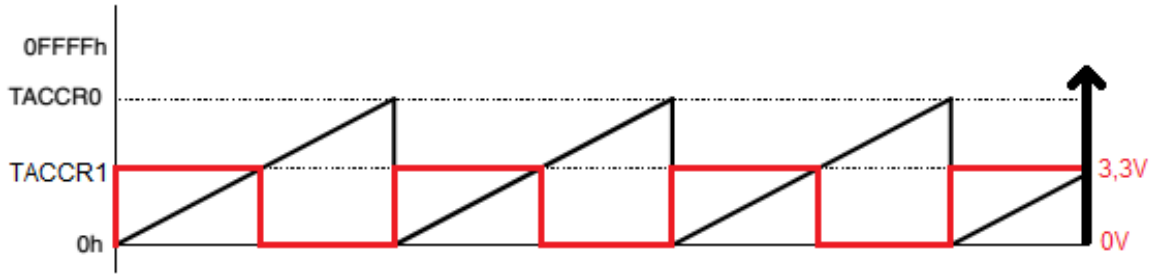


On place finalement le MSP430 en veille pour économiser de l'énergie. Le timer continuera à fonctionner normalement si le mode de veille n'est pas trop profond. Cette économie d'énergie ne peut pas se faire dans le cas d'une MLI utilisant une boucle « for » ou via une boucle « while ». On place le µC en veille en utilisant la commande « __low_power_mode_1(); ».

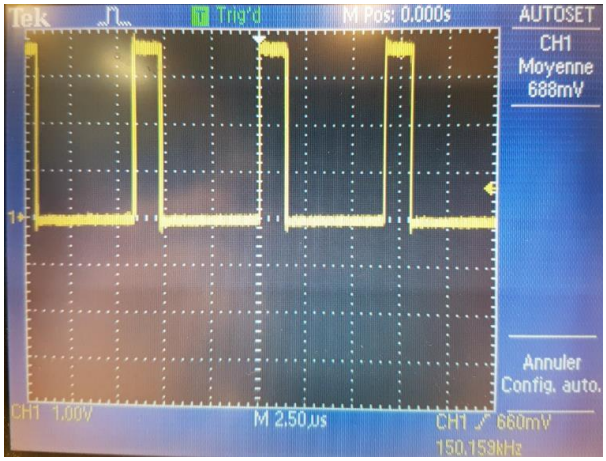
Pour obtenir le chronogramme de TAR, on prend en compte le type de comptage. Dans le comptage Up, on compte jusqu'à la valeur maximale programmée par le registre CCR0, une fois cette valeur atteinte, on reprend le comptage depuis le début. Le chronogramme est le suivant :

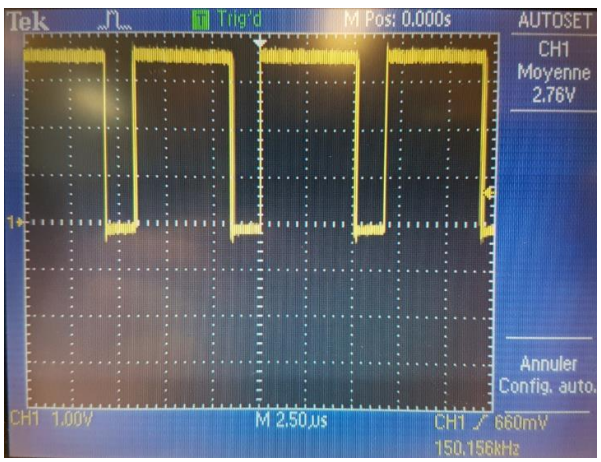


Maintenant, pour obtenir l'oscillogramme de la sortie TA1, on se base sur l'oscillogramme de TAR et on place le signal à l'état haut entre le début de période et la valeur fixée de CCR1. On passe le signal TA1 à l'état bas pour le reste de la période.



En choisissant des valeurs au hasard pour les registres CCR0 et CCR1, On obtient les oscillogrammes suivants pour les rapports cycliques de 25% (CCR1 = 1/4 de CCR0) et 75% (CCR1 = 3/4 de CCR0) :



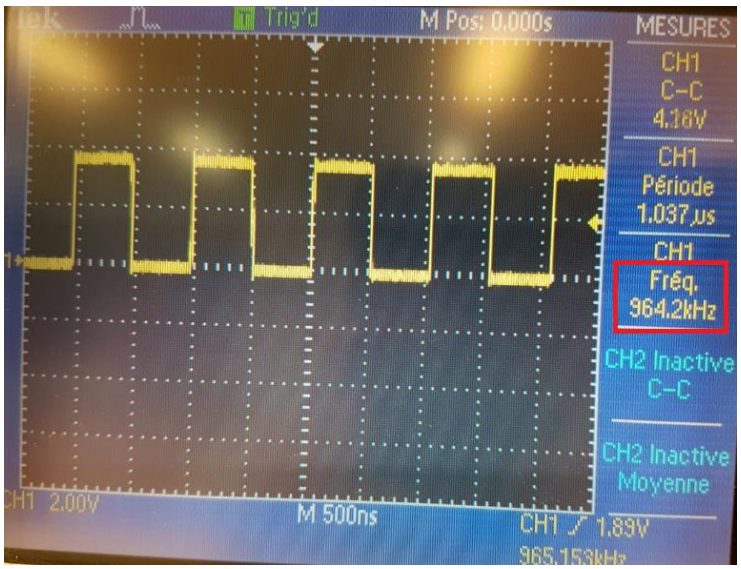


Nous avons ensuite essayé de paramétrer une période de 100 µs. Puisque l'horloge de base est réglée à 1 MHz sans division, sa période est de 1 µs. C'est la période d'une « marche » du timer. En appliquant le nombre 100 au registre CCR0 pour compter 100 fois cette période, nous retrouvons par la mesure une fréquence de 9,5 kHz ce qui fait approximativement une période de 105 µs.

Le calcul est validé expérimentalement en prenant une deuxième valeur CCR0 = 1 000. La fréquence obtenue est de 963 Hz, ce qui donne une période de 1 038 µs.

Mais revenons sur le cas de la période de 100 µs. On peut régler le rapport cyclique en ajustant la valeur de CCR1. Pour un rapport cyclique de 50%, on place CCR1 à la valeur 50, pour un rapport cyclique de 25%, on place CCR1 à la valeur 25 et ainsi de suite.

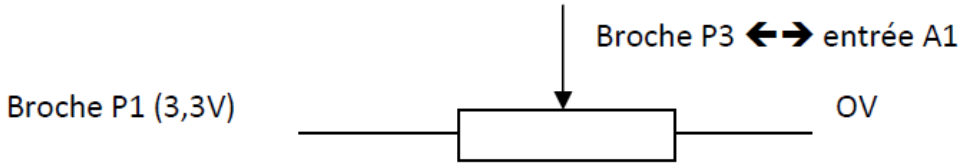
On peut visualiser l'oscillogramme de l'horloge SMCLK sur la broche P6. C'est un signal d'horloge tout à fait normal avec une fréquence mesurable proche du mégahertz. La valeur de 964 kHz est ainsi mesurée :



L'étape suivante est de créer le signal de sortie en modulation de largeur d'impulsion à partir d'un signal d'entrée analogique récupéré par le convertisseur ADC10. On choisira un potentiomètre afin de disposer d'une valeur moyenne stable.

<u>Création du filtre de fonction de transfert 1 :</u>

On réutilise le câblage du potentiomètre utilisé lors du paramétrage de l'ADC10.



Le programme suivant est constitué de la fusion entre le précédent et celui de l'acquisition du signal analogique avec l'ADC10 :

```
#include <msp430x20x2.h>
```

```
unsigned char DCO[8]= {0x00,0x20,0x40,0x60,0x80,0xA0,0xC0,0xE0};
unsigned char RSEL[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
double X; // ADC10MEM est codé sur 10 bits donc 2 octets ==> double donc entre 0 et 2047
/*********************************************/
/* ADC10 Interrupt Service Routine */
/*********************************************/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10ISR(void)
{
        X = ADC10MEM * 1.6;
        if (X > 1600) X = 1600 ;     // Butées pour éviter plantage prgr
        if (X < 1) X = 1 ;
        CCR1 = X; // on règle le rapport cyclique par le biais de X
        ADC10CTL0 |= ENC + ADC10SC;
}
/******************/
/* Main routine */
/******************/
void main(void)
{
        WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
        // Parametrages des entrées/sorties
        P1DIR = 0x1D; // 0001 1101 -> P6 P5 P4 et P2 en sortie donc P3 en entrée
        P1SEL = 0x1E; // 0001 1110 -> P6 P5 P4 et P3 en secondaire donc P2 en sortie TOR
        P1OUT = 0x00; // 0000 0000 -> rési de pull-down partout
        /* P6 = visualisation de SMCLK
           P5 = visualisation de l'ADC10CLK
           P4 = visualisation de la sortie timer TA1
           P3 = entrée analogique pour le CAN
           P2 = sortie TOR de la DEL
        */
        // Basic Clock System
        BCSCTL1 = XT2OFF + RSEL[15]; // Pas de Quartz externe
        DCOCTL = DCO[3]; // cf Notes de Lecture, Chap.5 CLOCK SYSTEM
        // RSEL = 15 et DCO = 0x60 -> 16MHz
        // Reglages TimerA
        TACTL = TASSEL_2 + MC_1;
        CCTL1 = OUTMOD_7 ;
        CCR0 = 1600 ; // Période du signal PWM = 100 µs
        CCR1 = 800;  // CCR1 PWM duty cycle
        // Analog to Digital SAR Converter Init.
        ADC10AE0 = BIT1; // enable chan1 (P1.1 = br. P3) as analog input
        // On récupère l'entrée analogique en P3
        ADC10CTL0 = SREF_0 + ADC10SHT_1 + ADC10ON + ADC10IE;
        ADC10CTL1 = INCH_1 + SHS_1 + ADC10DIV_1 + ADC10SSEL_0 + CONSEQ_0;
        // Commentaires sur la programmation de l'ADC10 :
        __bis_SR_register(GIE);
        ADC10CTL0 |= ENC + ADC10SC;
        for(;;)
        {
                __low_power_mode_1() ; // mise en veille du µC jusqu'à l'interruption
        }
```

}

On configure les entrées et sorties suivantes :

- P2 en sortie Tout ou Rien (visualisation de la DEL)
- P3 en entrée analogique reliée à A1 pour l'acquisition de l'ADC10
- P4 en sortie du timer TA1 où l'on visualise la MLI
- P5 en sortie secondaire pour la visualisation de l'horloge ADC10CLK
- P6 en sortie secondaire pour la visualisation de l'horloge SMCLK

On ajuste l'oscillateur DCO avec les composantes RSEL = 15 et DCO = 0x60. On retrouve la fréquence associée dans le tableau suivant :

FREQUENCE du DCO en MHz

		RSEL0	RSEL1	RSEL2	RSEL3	RSEL4	RSEL5	RSEL6	RSEL7
		0x00	0x01	...					0x07
DCO0	0x00	0,106	0,132	0,187	0,263	0,367	0,520	0,714	1,000
DCO1	0x20	0,114	0,143	0,202	0,284	0,396	0,562	0,771	1,080
DCO2	0x40	0,124	0,154	0,218	0,307	0,428	0,607	0,833	1,166
DCO3	0x60	0,134	0,160	0,230	0,320	0,450	0,630	0,870	1,220
DCO4	0x80	0,144	0,173	0,248	0,346	0,486	0,680	0,940	1,318
DCO5	0xA0	0,156	0,187	0,268	0,373	0,525	0,735	0,937	1,423
DCO6	0xC0	0,168	0,202	0,290	0,403	0,567	0,794	1,110	1,537
DCO7	0xE0	0,182	0,218	0,313	0,435	0,612	0,857	1,199	1,660

		RSEL8	RSEL9	RSEL10	RSEL11	RSEL12	RSEL13	RSEL14	RSEL15
		0x08	0x09	0x0A	...				0x0F
DCO0	0x00	1,430	2,050	2,940	3,761	5,060	6,803	9,900	13,550
DCO1	0x20	1,544	2,214	3,175	4,062	5,465	7,347	10,692	14,634
DCO2	0x40	1,668	2,391	3,300	4,387	5,902	7,935	11,547	15,805
DCO3	0x60	1,730	2,480	3,540	4,550	6,100	8,170	11,900	16,000
DCO4	0x80	1,868	2,678	3,823	4,914	6,588	8,824	12,852	17,280
DCO5	0xA0	2,018	2,893	4,129	5,307	7,115	9,529	13,880	18,662
DCO6	0xC0	2,179	3,124	4,459	5,732	7,684	10,292	14,991	20,155
DCO7	0xE0	2,354	3,374	4,816	6,190	8,299	11,115	16,190	21,768

Cette fréquence d'oscillateur est de 16 MHz.

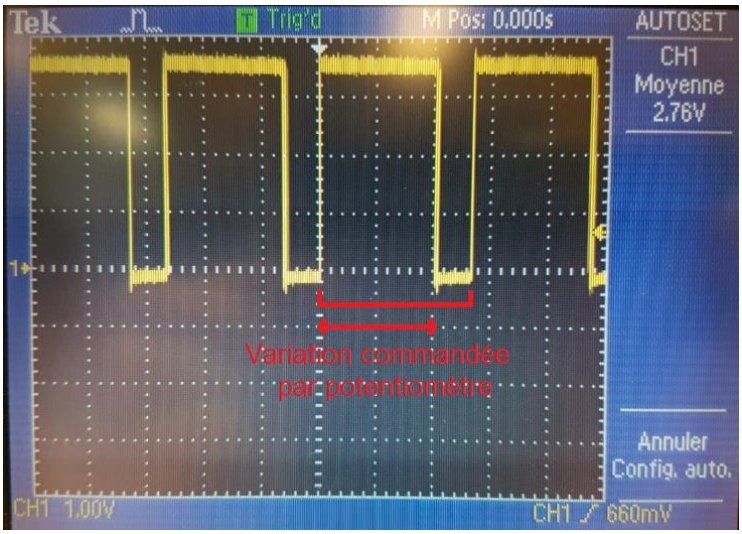
L'étape d'initialisation de ce programme met en place les configurations du convertisseur ADC10 ainsi que celles du timer A.

L'étape de boucle infinie est simplement constituée de l'instruction de mise en veille du µC.

La partie principale active de ce programme est placée dans l'interruption. L'ADC10 Interuption Service Routine (« ADC10ISR ») est exécutée lors de la fin de la conversion du signal analogique échantillonné. Lorsque la conversion de l'entrée analogique est terminée, on récupère ce nombre et l'on s'assure

qu'il est compatible avec la plage de valeurs de CCR1 (entre 0 et CCR0). Une fois que ceci est accompli, on affecte le nombre récupéré à l'entrée du timer pour qu'il puisse être envoyé en sortie par MLI. Le timer fera son travail grâce au paramétrage en initialisation.

Le timer délivre en sortie un signal carré dont la largeur d'impulsion est commandée par l'entrée analogique du potentiomètre en broche P3. On visualise sur la broche P4 le signal de sortie, ce qui nous donne l'oscillogramme suivant :



La variable X qui fait le lien entre le convertisseur et le timer est déclarée de la manière suivante : « double X; ». La principale différence avec une variable « int » est son format. Double est le format le plus grand disponible en langage C et contrairement au format entier, le format double prend en compte les nombres à virgules.

Au niveau du compilateur, un message d'erreur est indiqué en anglais « implicit conversion from floating point to integer » sur la ligne « CCR1 = X; ». Le registre CCR1 est de type integer (entier) et on cherche à lui affecter une variable de type double. Pour convertir des formats de variables il existe des fonctions spécialisées mais dans le cas présent on peut se contenter d'un arrondi de la valeur. La conversion implicite du type de valeur arrondit celle d'entrée pour sortir la valeur la plus proche disponible dans le format entier.

On a placé une paire de contrôle pour vérifier que la valeur de CCR1 (variable) ne dépasse jamais celle de CCR0 (fixe) et reste tout le temps positive. On a ces tests dans le programme qui sont matérialisés par les commandes « if (X > 1600) X = 1600 ; » et « if (X < 1) X = 1 ; ».

La durée totale d'une conversion implique la somme des trois étapes. Dans cette exemple, on doit prendre le temps d'acquisition et le temps de conversion de l'ADC10. On l'ajoute au temps de filtrage (négligeable dans le cas présent) et au temps d'une période du timer.

Ceci constitue le « décalage » dans le temps de la sortie par rapport à l'entrée. Dans le cas d'un filtre on parle de phase.

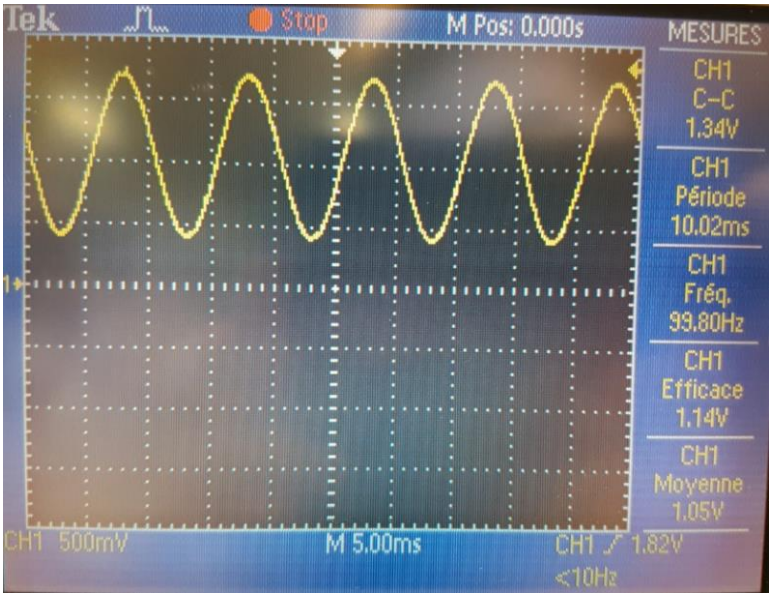
- Une période de modulation de largeur d'impulsion est réglée sur 100 µs.
- L'horloge ADC10CLK est configurée pour 16 MHz.

- La durée d'échantillonnage est de 8 fois la fréquence d'ADC10CLK.

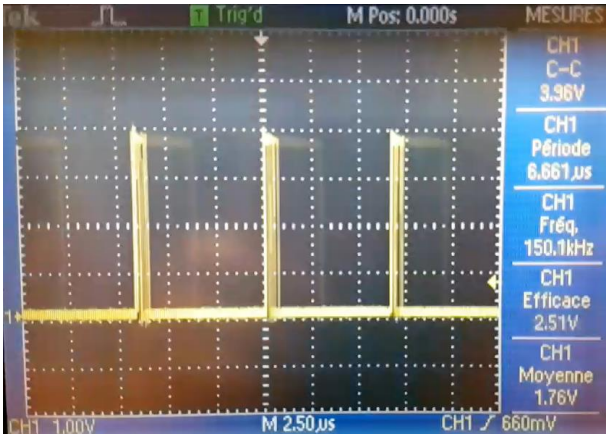
On calcule la période nécessaire pour l'échantillonnage et la conversion est de 21,5 fois la période de l'ADC10CLK ce qui fait 1,344 µs. Pour avoir la période totale, on l'ajoute au 100 µs. Le temps total du décalage T_{sortie}-$T_{entrée}$ est d'environ 102 µs.

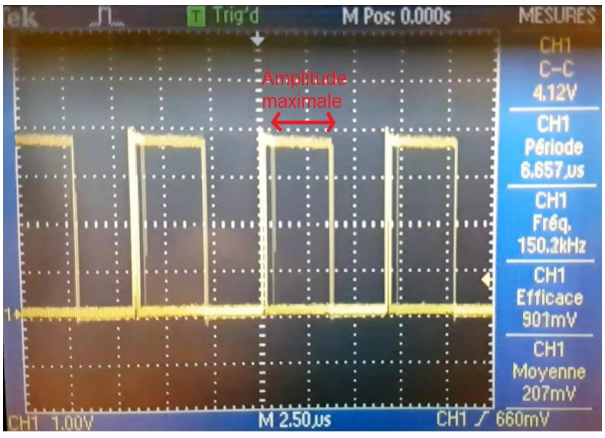
Etude du filtre avec GBF :

On déconnecte ensuite un potentiomètre de la broche P3 et on règle le Générateur Basse Fréquence (GBF) pour délivrer une onde sinusoïdale de 100 Hz d'une amplitude allant de 0 V jusque 3,3 V. Pour ne pas abimer le MSP430, la tension maximale doit être comprise entre ces deux valeurs. Le signal du GBF mesuré à l'oscilloscope est le suivant :

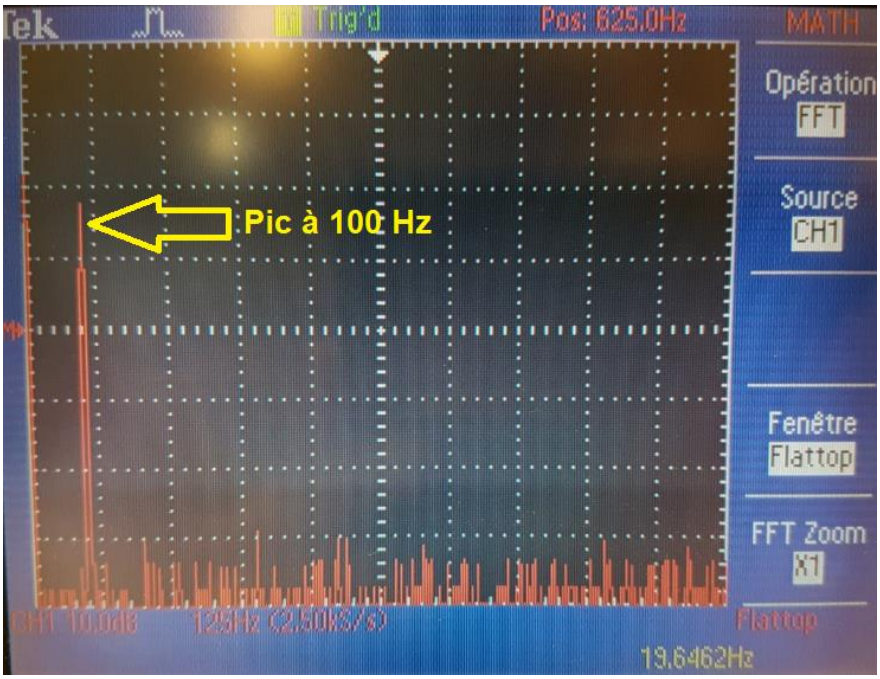


Le µC accomplit le travail pour lequel il a précédemment été programmé et délivre en sortie un signal carré dont la largeur d'impulsion est modulée sur le modèle de la tension d'entrée. Cette largeur évolue progressivement depuis un faible rapport cyclique vers un fort rapport cyclique avant de revenir au stade de départ. Les oscillogrammes suivants ont été réalisés à partir de l'observation de ces évolutions liées au GBF :



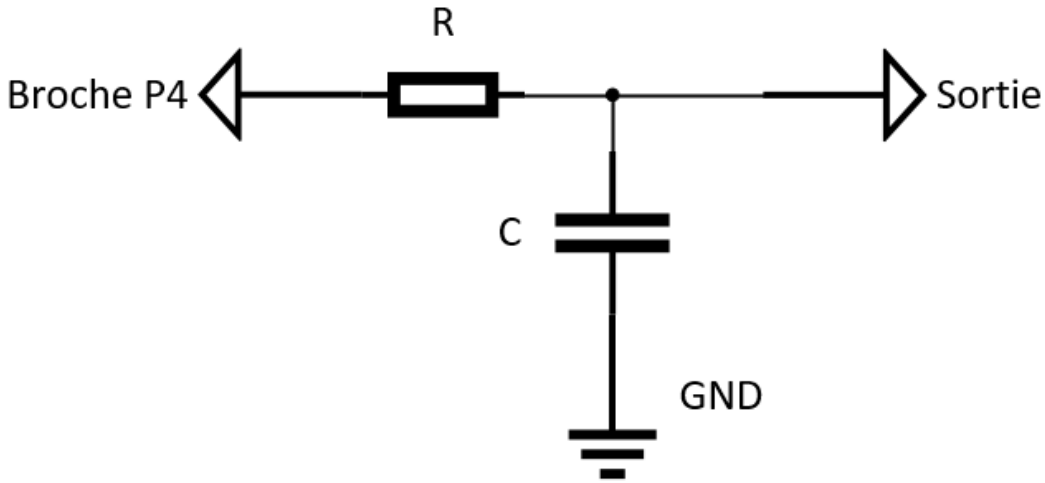


On arrange ensuite les réglages de l'oscilloscope pour obtenir la transformée de fourrier rapide (FFT : Fast Fourrier Transform). Voici la capture d'écran associée. On y observe un pic important à 100 Hz, ce qui retraduit le signal d'entrée reçu du GBF.

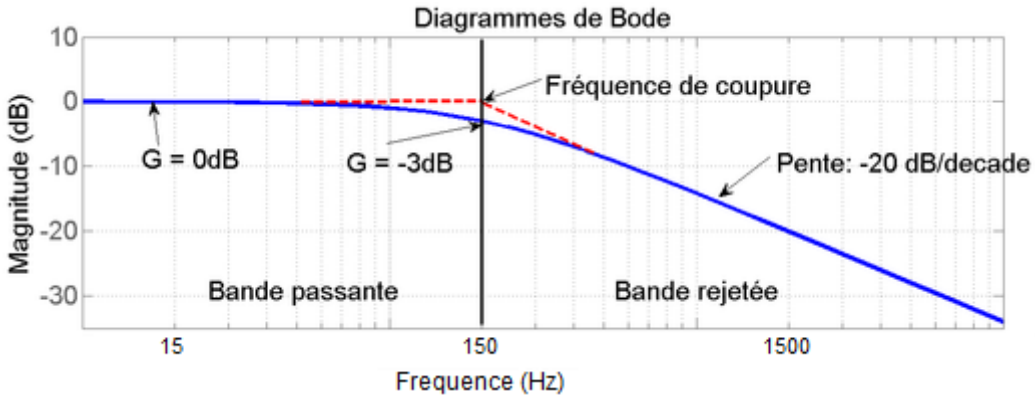


Récupération du signal de sortie depuis la MLI :

Le signal de sortie que l'on cherche à reconstruire possède une fréquence de 100 Hz. Le signal modulé par la largeur d'impulsion a une fréquence de 10 kHz. Pour reconstruire ce signal, on applique en sortie du microcontrôleur un filtre passe-bas passif du premier ordre pour enlever les hautes fréquences et lisser la tension de sortie pour récupérer un signal de basse fréquence. Le filtre sera constitué d'une résistance et d'un condensateur comme sur le schéma suivant :



On trace le diagramme de Bode théorique de ce filtre pour ajuster au mieux les caractéristiques des composants. En bleu on a la caractéristique réelle du filtre et en rouge la caractéristique idéale :



La fréquence de coupure doit être placée au plus près du 100 Hz pour atténuer au maximum les parasites sur la fréquence 10 kHz. A cause de la caractéristique non idéale du filtre RC, il y a une petite atténuation avant la fréquence de coupure. Pour limiter l'atténuation au 100 Hz on choisira une fréquence légèrement plus élevée.

Nous choisirons arbitrairement la fréquence de coupure de la cellule RC à 200 Hz.

La formule de la relation entre la fréquence de coupure et les valeurs de la résistance et du condensateur est donnée par :

$F_c = 1/(2\pi*R*C)$

On transforme cette formule pour obtenir la valeur de R et de C en fonction de la fréquence de coupure par :

$R*C = 1/(2\pi*F_c)$

Puisque la fréquence de coupure est de 200 Hz, on a : $R*C = 1/400\pi$

On choisit arbitrairement la valeur de R pour calculer la valeur du condensateur. Parmi les valeurs normalisées de la série E12 on choisit celle de 1 kΩ.

<u>Explication des séries de valeurs normalisées :</u>

Pour les montages électroniques, on utilise des composants dont les valeurs sont standardisées. Chaque application nécessite d'avoir des valeurs de composants adaptés ainsi qu'une gamme de précision. Les constructeurs ont défini plusieurs gammes allant de composant peu chers et peu précis jusqu'aux composants d'instrumentation dont la précision est la meilleure possible.

Lorsque l'on fait le choix d'utiliser des composants peu précis (tolérance élevée), il n'est pas nécessaire de posséder une grande gamme de choix. A contrario, en instrumentation une différence d'une dizaine d'Ω sur des composants de l'ordre de la dizaine de kΩ peut changer le comportement d'un montage. Il est dans ce cas exclu de choisir une valeur proche de celle souhaitée. Plus la gamme de composant est d'une fine précision, plus le nombre de valeurs intermédiaires est disponible.

Les noms des séries sont normalisés avec la lettre E puis le nombre de composants disponibles dans une décade. Par exemple, dans la série E24, on pourra choisir 24 valeurs différentes entre 1 Ω et 10 Ω, il en va de même pour 24 autres valeurs entre 10 Ω et 100 Ω et ainsi de suite.

Le tableau suivant nous décrit les séries avec le nombre de valeurs disponibles, leurs précisions et les applications courantes.

Série	n/décade	Tolérance	Observation
E3	3	±50 %	N'est plus utilisée
E6	6	±20 %	Vieux postes à lampes
E12	12	±10 %	Extraite de la série E24
E24	24	±5 %	Electronique grand public
E48	48	±2 %	Prototypage
E96	96	±1 %	Filtres BF et précision
E192	192	±0,5 %	Instrumentation

La série E24 est celle la plus couramment utilisée en électronique grand public. Les valeurs disponibles sur une décade sont les suivantes :

E24 (±5%) : 100 - 110 - 120 - 130 - 150 - 160 - 180
200 - 220 - 240 - 270 - 300 - 330 - 360 - 390
430 - 470 - 510 - 560 - 620 - 680 - 750 - 820 - 910

Revenons maintenant sur l'application concrète du filtre RC. La valeur du condensateur est calculée comme il suit :

$C = 1/(400\pi*1000)$

$= 1/(400000\pi)$

$= 800$ nF

Les condensateurs sont plus chers (et donc plus rares) lorsque leurs valeurs sont élevées, on optimise le circuit en divisant la valeur du condensateur par 10 et en contrebalançant par la multiplication par 10 de la valeur de la résistance.

On a donc les valeurs de R = 10 kΩ et C = 80 nF

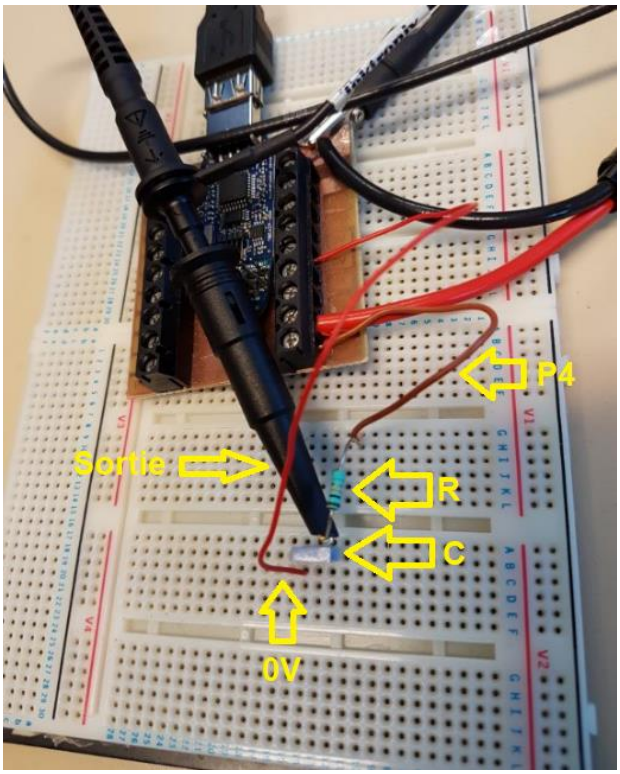
Cependant les contraintes techniques liées aux valeurs normalisées de la série E12 nous donne la valeur approchée du condensateur de C = 100 nF. On garde la valeur de résistance de R = 10 kΩ.

Puisque les valeurs théoriques ont changé, on doit recalculer la valeur de la fréquence de coupure afin de vérifier qu'elle ne s'éloigne pas trop de la valeur souhaitée :

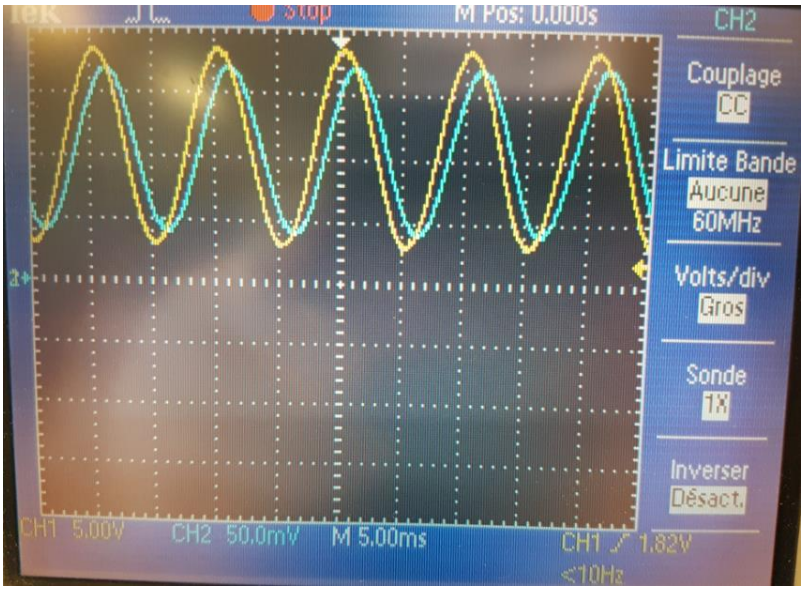
$F_c = 1/(2\pi*R*C) = 1/(2\pi*10^{4}*10^{-7}) = 159$ Hz

La valeur de 159 Hz est encore plus avantageuse que 200 Hz car on reste au-dessus de la fréquence de coupure théorique de 100 Hz. De plus, on se décale en s'éloignant des bruits à atténuer en haute fréquence.

Après réalisation du montage, on obtient le résultat suivant :



L'oscillogramme des tensions mesurées donne en jaune la tension de sortie du GBF et en cyan la tension de sortie du filtre RC en sortie du microcontrôleur :



Explication complémentaire sur l'atténuation :

Nous avons calculé une atténuation théorique du filtre de -20 dB par décade à partir de la fréquence de coupure du filtre. On peut s'amuser à anticiper la valeur de la tension de sortie sous différentes fréquences. La formule donnée est la suivante :

Gain(dB) = 20 log ($V_{sortie}/V_{entrée}$)

Nb : Le log utilisé est le logarithme de base 10, ce qui veut dire que l'opération inverse (fonction réciproque) de log(x) est 10^x.

Prenons la fréquence de coupure à 150 Hz de la cellule RC câblée avec une tension C-C (crête à crête) de 3 V. Pour calculer la valeur de tension, on se réfère au gain de -20 dB au bout d'une décade. Ce qui fait à 1 500 Hz :

-20 = 20 log (Vs/3)

log (Vs/3) = -20 / 20 = -1

$Vs/3 = 10^{-1}$

Vs = 1/3 * 0.1 = 33 mV

On peut faire le même calcul à 15 kHz en prenant un gain de -40 dB :

$Vs = 1/3 * 10^{(-40/20)}$

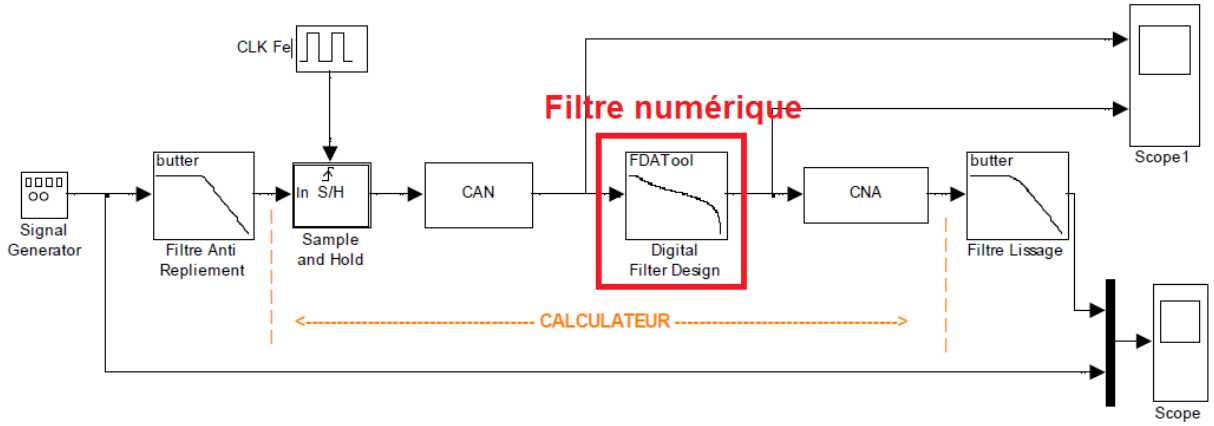
$Vs = 1/3 *10^{-2}$

Vs = 3,3 mV

Ceci conclut la réalisation du filtre numérique de fonction de transfert 1. L'étape finale consiste à adapter ce filtre pour que la fonction de transfert soit changée numériquement en fonction des besoins de l'application.

Synthèse et réalisation du filtre numérique

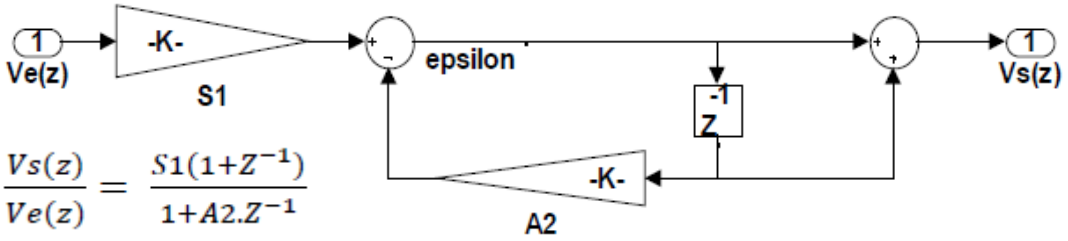
Le filtre numérique à réaliser a été simulé sous MATLAB avant sa réalisation. Le schéma complet de la simulation est le suivant. La partie du filtrage numérique se trouve au centre du montage :



L'algorithme de simulation du filtre utilise deux constantes nommées S1 et A2. Leurs valeurs sont définies à l'avance par des nombres à virgule flottante dont la précision est élevée. On donne :

- S1 = 0,030468747091253828
- A2 = -0,9390625058174924

L'algorithme et la fonction de transfert sont les suivants :



$$\frac{Vs(z)}{Ve(z)} = \frac{S1(1+Z^{-1})}{1+A2.Z^{-1}}$$

On note V(z) = v(n.Te) et z^{-1}.V(z) = v((n-1).Te), on a donc l'équation de récurrence :

$$V_{s(z)} * (1 + A2 * Z^{-1}) = V_{e(z)} * S1(1 + Z^{-1})$$

$$V_{s(z)} + V_{s(z)} * A2 * Z^{-1} = S1(V_{e(z)} + V_{e(z)}Z^{-1})$$

$$V_{s(z)} = S1\left(V_{e(z)} + V_{e(z)}Z^{-1}\right) - V_{s(z)} * A2 * Z^{-1}$$

On a alors : $V_s(nTe) = S1(V_e(nTe) + V_e((n-1)Te)) - V_s((n-1)Te) * A2$

Soit : **vs(n.Te) = S1.[ve(n.Te) + ve{(n-1).Te}] – A2.vs{(n-1).Te}**

L'équation de récurrence nous montre qu'en plus de devoir stocker les constantes S1 et A2, nous devons garder en mémoire les valeurs de Ve et de Vs du calcul précédent. Lors de l'étape calculatoire, on doit donc stocker au début la valeur de l'échantillon d'entrée récupéré puis en fin de calcul la valeur de l'échantillon de sortie envoyé.

Dans le programme, nous nommerons ces variables « Ve_mem » et « Vs_mem ». Les formats des quatre variables et constantes gardées en mémoire pour le calcul seront identiques au format de la variable récupérée en sortie de l'ADC10MEM. C'est-à-dire le format « double ».

On modifie le programme du chapitre précédent en effectuant le calcul dans l'interruption juste avant l'affectation de la valeur de CCR1. Voici le programme complet :

```
#include <msp430x20x2.h>
unsigned char DCO[8]= {0x00,0x20,0x40,0x60,0x80,0xA0,0xC0,0xE0};
unsigned char RSEL[16] = {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
double X ; // ADC10MEM est codé sur 10 bits donc 2 octets ==> double donc entre 0 et 2047
double S1 = 0.030468747091253828 ; // Coefficient du filtre en entrée
double A2 = -0.9390625058174924 ; // Coefficient du filtre de retour
double Ve_mem = 0;
double Vs_mem = 1;
/*******************************************/
/* ADC10 Interrupt Service Routine */
/*******************************************/
#pragma vector=ADC10_VECTOR
__interrupt void ADC10ISR(void)
{
        // X = ve(n.Te) -> entrée du filtre
        // CCR1 = vs(n.Te) -> sortie du filtre
        // vs(n.Te) = S1.[ ve(n.Te) + ve{(n-1).Te}] – A2.vs{(n-1).Te}
        // On définit des mémoire de :
        // ve([n-1].Te) = Ve_mem
        // vs([n-1].Te) = Vs_mem
        // CCR1 = S1*(X+Ve_mem)-A2*Vs_mem
        X = ADC10MEM / 10;
        if (X > 100) X = 100 ;     // Butées pour éviter plantage prgr
        if (X < 1) X = 1 ;
        //CCR1 = X; // on règle le rapport cyclique par le biais de X
        CCR1 = S1*(X+Ve_mem)-(A2*Vs_mem);
        Vs_mem = S1*(X+Ve_mem)-(A2*Vs_mem);
        Ve_mem = X;
        ADC10CTL0 |= ENC + ADC10SC;
}
/******************/
/* Main routine */
/******************/
void main(void)
{
        WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer
        // Parametrages des entrées/sorties
        P1DIR = 0x1D; // 0001 1101 -> P6 P5 P4 et P2 en sortie donc P3 en entrée
        P1SEL = 0x1E; // 0001 1110 -> P6 P5 P4 et P3 en secondaire donc P2 en sortie TOR
        P1OUT = 0x00; // 0000 0000 -> rési de pull-down partout
        /*  P6 = visualisation de SMCLK
           P5 = visualisation de l'ADC10CLK
           P4 = visualisation du timer
```

```
        P3 = entrée analogique pour le CAN
        P2 = sortie TOR
    */
    // Basic Clock System
    BCSCTL1 = XT2OFF + RSEL[15]; // Pas de Quartz externe
    DCOCTL = DCO[3]; // cf Notes de Lecture, Chap.5 CLOCK SYSTEM
    // RSEL = 15 et DCO = 0x60 -> 16MHz
    // Reglages TimerA
    TACTL = TASSEL_2 + MC_1;
    CCTL1 = OUTMOD_7 ;
    CCR0 = 100 ; // Période du signal PWM = 100 µs
    CCR1 = 50;  // CCR1 PWM duty cycle
    // Analog to Digital SAR Converter Init.
    ADC10AE0 = BIT1; // enable chan1 (P1.1 = br. P3) as analog input
    // On récupère l'entrée analogique en P3
    ADC10CTL0 = SREF_0 + ADC10SHT_1 + ADC10ON + ADC10IE;
    ADC10CTL1 = INCH_1 + SHS_1 + ADC10DIV_1 + ADC10SSEL_0 + CONSEQ_0;
    // Commentaires sur la programmation de l'ADC10 :
    __bis_SR_register(GIE);
    ADC10CTL0 |= ENC + ADC10SC;
    for(;;)
    {
        __low_power_mode_1() ; // mise en veille du µC jusqu'à l'interruption
    }
}
```

Premièrement, dans l'étape de définition des variables, nous avons affectés des valeurs à Ve_mem et à Vs_mem. Cela s'explique par le déroulement du programme dans l'interruption. En effet, on commence par utiliser les valeurs de ces variables dans le calcul avant de les affecter. Il est indispensable que l'on ait conscience de ce que contiennent ces variables au début.

On initialise la variable d'entrée à 0 car on peut se permettre au lancement du microcontrôleur d'affecter une valeur de tension de 0 V. Par contre, pour la valeur de sortie, il n'est pas possible de l'affecter à 0 car elle est multipliée par une des constantes. Pour ne pas perturber le calcul, on affecte à la valeur de 1.

Pour calculer la valeur de CCR1, on reprend la formule de récurrence en affectant les variables suivantes aux éléments de la formule :

$$vs(n.Te) = S1.[\ ve(n.Te) + ve\{(n-1).Te\}] - A2.vs\{(n-1).Te\}$$

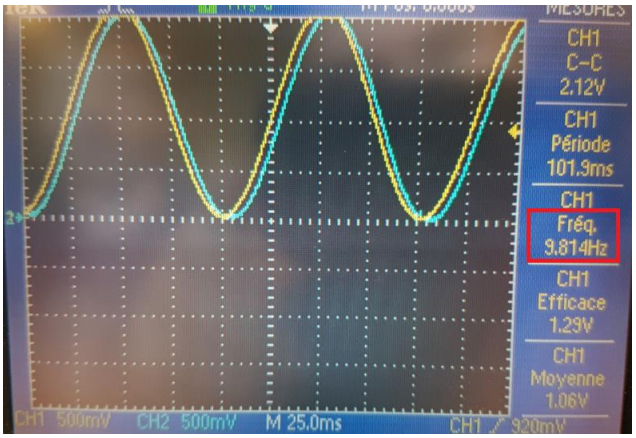
- CCR1 = vs(n.Te)
- S1 = S1
- ADC10MEM = Ve(n.Te)
- Ve_mem = ve((n-1).Te)
- A2 = A2
- Vs_mem = vs((n-1).Te

On retient la formule suivante avec les noms des variables de programmation :

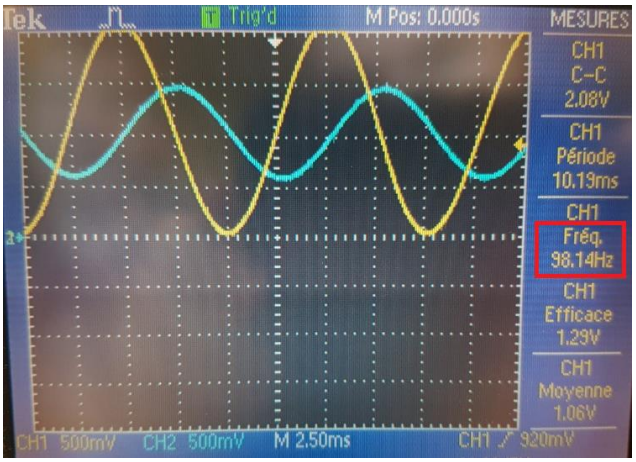
CCR1 = S1.(ADC10MEM + Ve_mem) - A2.Vs_mem

En ajustant les butées, on reprogramme cette formule dans le programme d'interruption et on réenregistre ensuite les nouvelles valeurs de Ve_mem et Vs_mem pour préparer le prochain appel de l'interruption par « Vs_mem = S1*(X+Ve_mem)-(A2*Vs_mem); » et « Ve_mem = X; ».

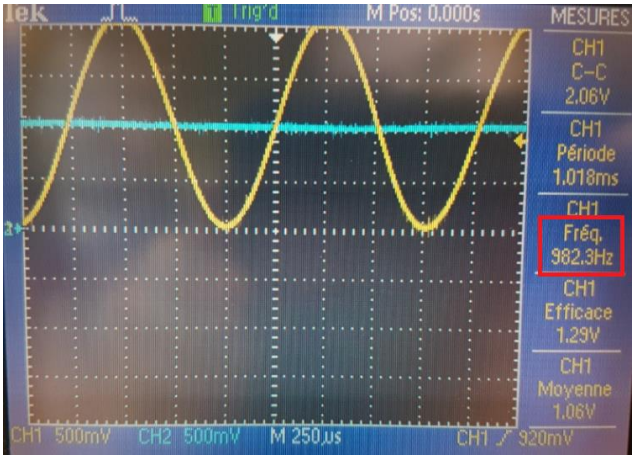
Le montage du filtre RC en sortie de MLI reste pour l'instant le même que celui du chapitre précédent. Nous avons pris les chronogrammes pour trois fréquences : à 10 Hz, à 100 Hz et à 1 kHz :



Au niveau de la basse fréquence, on observe qu'il n'y a strictement aucune atténuation du signal. En revanche, il existe un léger décalage dans le temps, ce qui est très certainement dû au temps de conversion et au temps de calcul du signal.



Pour la fréquence de 100 Hz, l'amplitude du signal est divisée par 2. Cela est dû aux deux filtres en série (filtre numérique + cellule RC). Les fréquences de coupure des deux filtres sont de 100 Hz. Pour arriver à ne garder que la fréquence de coupure du filtre numérique sans que le filtre RC perturbe ce signal, il faut augmenter la fréquence de coupure du filtre RC.



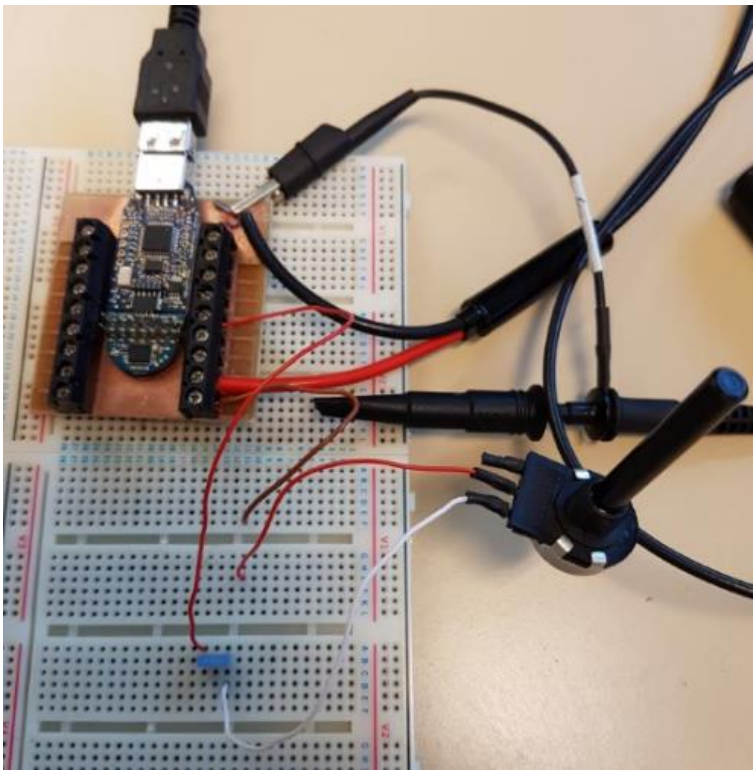
Sur une fréquence de 1 kHz, on observe une atténuation telle que le signal a été transformée en signal continu.

On va maintenant ajuster la fréquence de coupure du filtre RC à 500 Hz. On reprend la formule de calcul comme il suit :

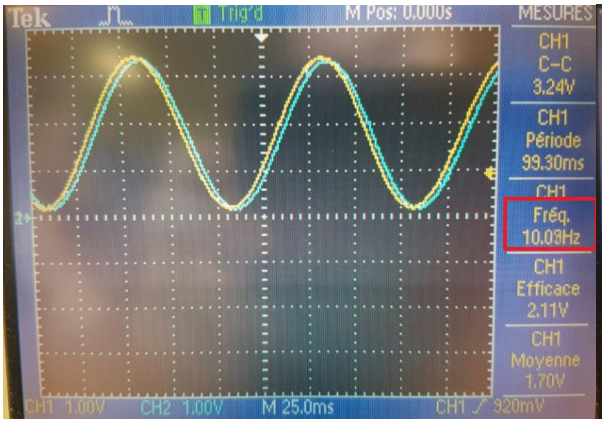
$R = 1/(2\pi*C*F_c)$

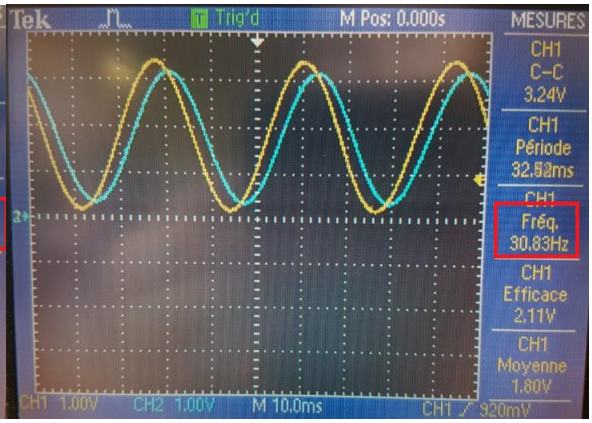
$R = 1/(2\pi*10^{-7}*500) \sim= 3\ 200\ \Omega$

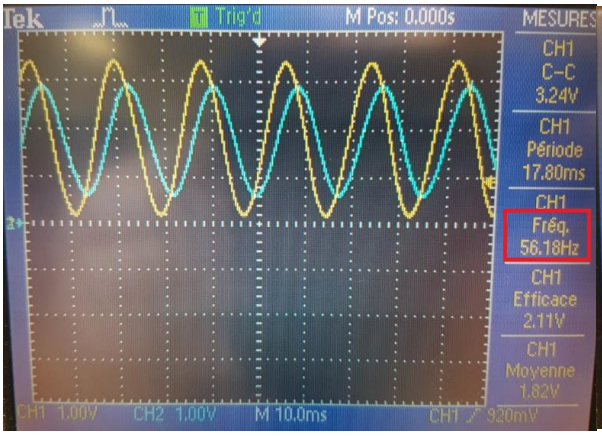
Pour des raisons de manque de temps, nous n'avons pas eu l'occasion de rechercher les bons composants résistifs. Nous avons pris un potentiomètre que nous avons ajustés à l'ohmmètre puis nous l'avons intégré au circuit à la place de la résistance comme sur le montage suivant :

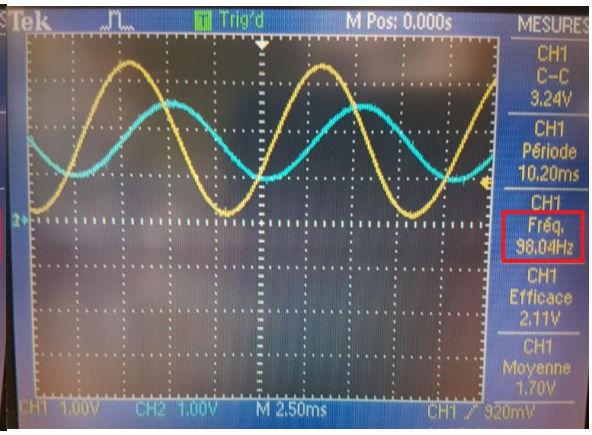


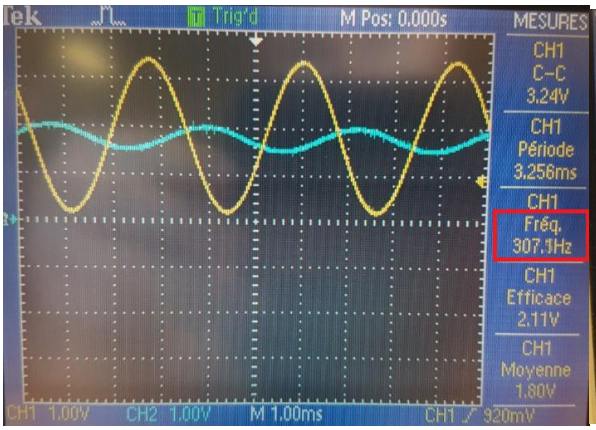
Cela nous a permis de relever les oscillogrammes sur les fréquences de 10 Hz, 30 Hz, 60 Hz, 100 Hz, 300, Hz, 600 Hz et 1 kHz. Les voici dans l'ordre :

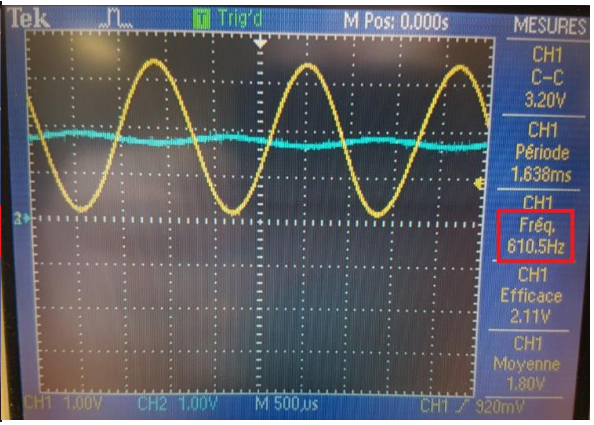


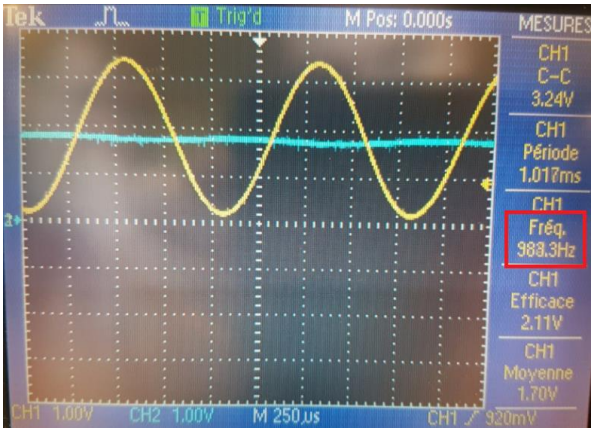












On observer dès le premier coup d'œil que l'amplitude du signal de sortie du filtre diminue fortement à mesure que la fréquence du signal d'entrée augmente. Ce phénomène est plus prononcé sur la plage

de fréquence à partir de 100 Hz. Par contre, malgré la fréquence de coupure du filtre numérique réglée pour 100Hz, l'atténuation commence à être présente longtemps avant. On l'observe déjà sur l'oscillogramme de la fréquence 30 Hz.

Cette présence de l'atténuation plusieurs dizaines de de Hertz avant la fréquence de coupure, prouve que le filtre est bien d'un ordre relativement faible (premier ordre souhaité) car comme décrit en introduction, plus l'ordre d'un filtre est élevé, plus la pente sera raide et plus les premières atténuations seront perceptibles proche de cette fréquence de coupure.

Conclusion sur le filtrage numérique et les microcontrôleurs

Pour conclure, nous avons vu au travers des divers apports théoriques et pratiques que le filtrage numérique apparaît comme une solution à la fois économique et ergonomique au traitement des signaux. Au-delà d'un coût réduit et d'un gain de place non négligeable, il offre une modularité sans égale de part sa réalisation basée sur une ou plusieurs fonctions de transfert. Enfin il peut, suivant les applications et l'envergure des installations/signaux, être mis au point au moyen d'une simple carte électronique équipée d'un microprocesseur.

Au cours des séances de travaux pratiques, notre étude a portée sur la compréhension et l'utilisation du microcontrôleur MSP430 F2012. Par l'utilisation de celui-ci, nous avons tenté de mettre en place une solution de filtrage de signal en provenance d'un GBF (Générateur Basse Fréquence). Pour réaliser le filtrage numérique, il est indispensable d'utiliser des convertisseurs analogique/numérique ou numérique/analogique, non présent ici. Nous avons donc simulé ceux-ci au moyen de l'ADC10 (échantillonneur bloqueur + CAN intégré au microcontrôleur) et de l'utilisation de la MLI, via le timer. Enfin, nous avons réalisé un filtre RC passe-bas analogique du 1er ordre pour lisser la sortie modulée en largeur d'impulsion.

Au moyen des différentes connaissances acquises et des montages mis en place, il nous a finalement été possible de filtrer le signal analogique du GBF, et de récupérer le signal filtré en sortie. En comparant les signaux d'entrée et de sortie à l'oscilloscope, les résultats sont sans appel : l'atténuation est bien réalisée telle qu'encodée dans le filtre numérique.

Pour terminer l'utilisation du kit comprenant le microcontrôleur MSP430 peut permettre la réalisation de filtre numérique, en mettant en jeu des consommations électriques moindre et un encombrement dérisoire. Il convient donc parfaitement à des fins de réalisations d'applications embarquées.

Ressources bibliographiques utilisées :

Le grand livre d'Arduino (E. Bartmann, Eyrolles)

MSP430 µC Basics (J. Davies, format numérique)

Notes de lecture MSP430 (F. Robert, format numérique)

Cours C, C++ (A. Canteaut, format numérique)

TP ENL µC (F. Lefebvre – F.Robert, format numérique)

Ressources numériques utilisées :

https://fr.wikipedia.org/wiki/Microcontrôleur

http://www.ti.com/product/MSP430F2012

http://www.gecif.net

https://openclassrooms.com

http://www.positron-libre.com/cours/electronique