
LIVRET DE TRAVAUX PRATIQUES
TRAITEMENT DU SIGNAL NUMÉRIQUE



Département d'Électronique
ECE École d'Ingénieurs - *Engineering School*

Année 2021-2022

Règlement

Voici quelques consignes importantes à suivre pendant les séances de TP tout au long de l'année scolaire:

1. Il est impératif de respecter les horaires des séances de TP: **aucun retard ne sera toléré.**
2. Interdiction formelle de manger ou de boire dans les salles de TP. Les bouteilles d'eau sont tolérées.
3. **Les préparations théoriques doivent être effectuées avant chaque séance de TP** et pourront être ramassées en début de cours. Tout TP non préparé sera sanctionné par la note 0.
4. La note de TP tiendra compte de l'**assiduité**, de la **préparation théorique** et la **qualité** des compte-rendus.
5. Certains compte-rendus seront à préparer et ramassés. Il est de votre devoir de prendre des notes sur les autres TPs afin de vous préparer au mieux à l'Examen de TP de fin de semestre.

— notation —

Vous recevrez à l'issue de ce semestre trois notes: une note de suivi, une note d'examen de TP et une note de projet comme suit:

note	description
NS	Note de suivi pour les rapports de TP, les éventuelles interros et votre implication
PRJ	Note du projet semestriel (moyenne pondérée de la soutenance (50%) et du rapport(50%))

Programme des séances de TP

— Arduino Avancé —



TP 1 - Conversion analogique numérique p. 7



TP 2 - Transformée de Fourier rapide p. 17

— Filtrage numérique —



TP 3 - Les filtres RIF p. 27



TP 4 - Les filtres RII p. 37

Conversion analogique numérique

Motivation

Les cartes Arduino nano disposent d'un Convertisseur Analogique Numérique de 10 bits intégré au microcontrôleur ATmega328. Le circuit présenté sur la Figure 1.1 (a) permet de numériser le signal provenant d'un microphone KY-037.

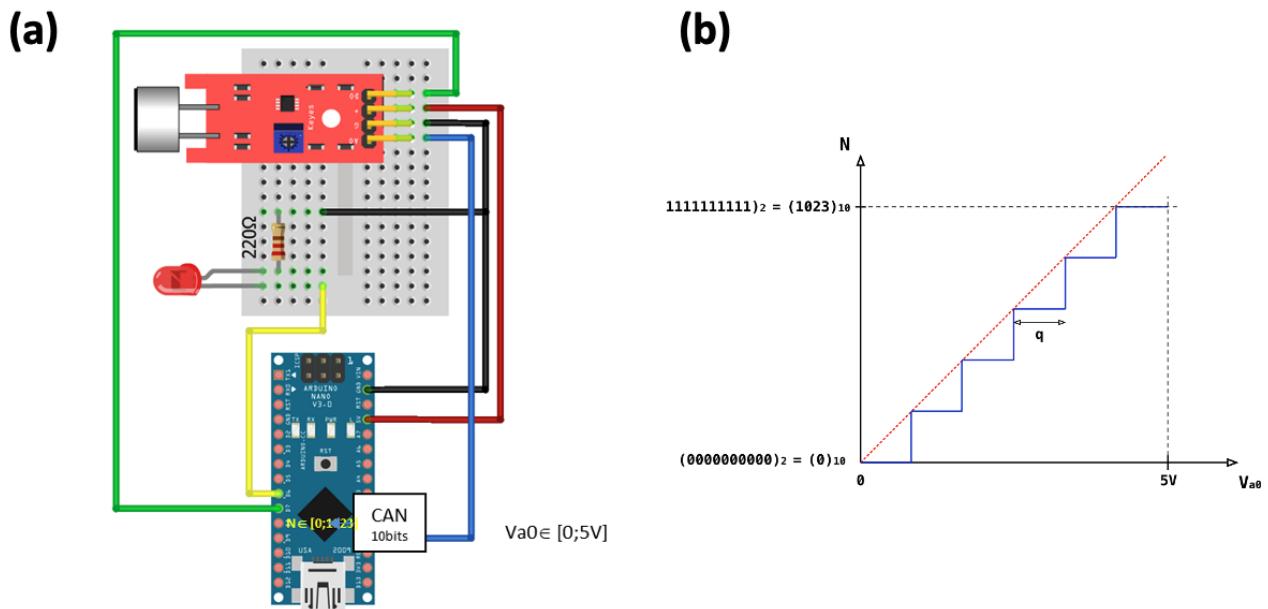


Figure 1.1: (a) Circuit permettant de numériser un signal analogique émanant d'un microphone KY-037 et (b) caractéristique du CAN de l'ATMega328P.

Le CAN de l'ATMega328P a une résolution de 10 bits (voir Figure 1.1 b), ce qui donne la possibilité d'avoir des $2^{10} = 1024$ combinaisons possibles en sortie avec un pas de $q = \frac{V_{REF}}{2^{10}} = 4,88 \text{ mV}$.

La fonction `analogRead()` permet de simplement obtenir un entier de 0 à 1023 image de la tension convertie. Elle est cadencée de sorte à obtenir un échantillon toutes les $115 \mu\text{s}$, soit $f_e \approx 9 \text{ kHz}$.

Le but de ce TP est d'étudier les différents moyens d'améliorer la précision et surtout la fréquence d'échantillonnage du CAN de l'ATMega328P.

I. Réglage de la rapidité de l'échantillonnage

L'échantillonnage est fait à l'aide du Timer1 qui comporte un compteur 16 bits (registre **TCNT1**). Nous allons l'utiliser dans le mode **CTC** (clear timer on compare match), avec une valeur maximale fixée par le registre **OCR1A**. Le compteur est incrémenté d'une unité à chaque coup d'horloge jusqu'à atteindre la valeur fixée par **OCR1A** (voir Figure 1.2).

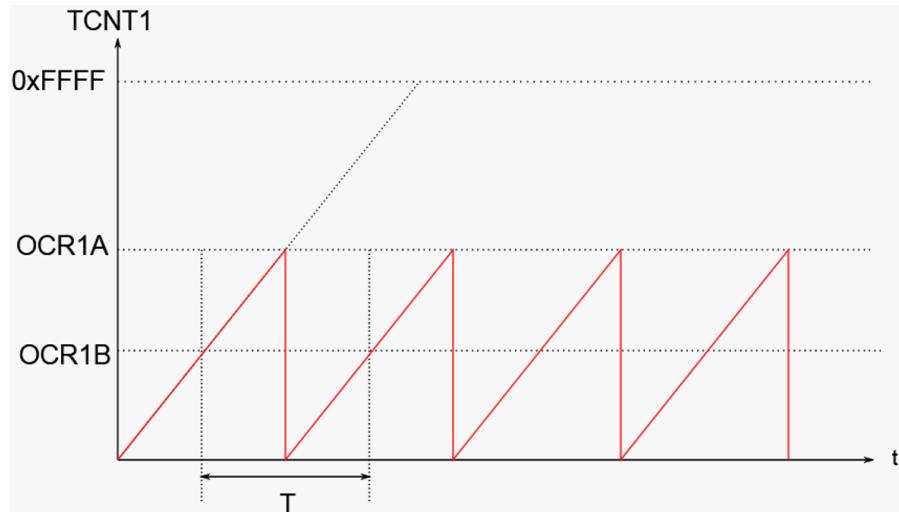


Figure 1.2: Évolution temporelle du compteur TIMER1 en fonction des registres OCR1A et OCR1B.

La période d'échantillonnage est égale à la période de l'horloge multipliée par la valeur maximale. On utilisera aussi le registre **OCR1B**, dont la valeur sera fixée à la moitié de **OCR1A**.

Lorsque **TCNT1=OCR1B**, une interruption sera déclenchée. Le conversion analogique-numérique sera aussi déclenchée par cette condition.

A. Les timers

L'intérêt d'un timer est qu'il compte sans cesse et que pendant ce temps, le programme peut réaliser autre chose, ce qui n'est pas possible si on utilise la fonction `delay()` qui est bloquante et qui ne permet pas de faire autre chose pendant ce temps d'attente. Le temps que le timer met pour compter 256 coups dépend bien sûr de la fréquence de l'horloge ; à 16 Mhz, c'est très rapide, mais il est possible de diviser cette fréquence d'horloge grâce à des circuits internes au microcontrôleur appelés prédiviseur (prescaler).

On peut alors diviser la fréquence de base (16 MHz) par 8, 32, 64, 128, 256 ou 1024 ; pour cela, il faut utiliser intelligemment d'autres registres de contrôle associés au **timer**. Par exemple, si on règle de prédiviseur pour diviser la fréquence par 1024, le **timer** comptera donc à une fréquence de 15625 Hz. Le module Arduino nano est construit autour du microcontrôleur AVR ATmega328P qui possède 3 timers :

- Le **timer0**, sur 8 bits, utilisé par les fonctions `delay()`, `millis()` et `micros()`. Il commande également des PWM (Pulse Width Modulation ou Modulation par Largeur d'Impulsion) sur les broches 5 et 6.
- Le **timer1**, sur 16 bits, qui compte de 0 à 65535 (0 à FFFF en hexadécimal) et qui est utilisé par la bibliothèque Servo ou bien pour de la PWM sur les broches 9 et 10.
- Le **timer2**, sur 8 bits, qui est utilisé par la fonction `Tone()` ou bien pour de la PWM sur les broches 3 et 11.

B. Les registres de contrôle

Le tableau suivant donne les différents registres de contrôle associés à chaque **Timer** ; nous verrons le rôle de chaque registre tout en nous limitant à ce qui est vraiment à connaître pour réaliser la fonction d'échantillonnage.

Timer0	Timer1	Timer2	Rôle
TCNT0	TCNT1L	TCNT2	Timer (bit 0 à 7)
-	TCNT1H	-	Timer (bit 8 à 15)
TCCROA	TCCR1A	TCCR2A	Registre de contrôle
TCCR0B	TCCR1B	TCCR2B	Registre de contrôle
-	TCCR1C	-	Registre de contrôle
OCROA	OCR1AL	OCR2A	Output Compare (bit 0 à 7)
-	OCR1AH	-	Output Compare (bit 8 à 15)
OCROB	OCR1BL	OCR2B	Output Compare (bit 0 à 7)
-	OCR1BH	-	Output Compare (bit 8 à 15)
-	ICR1L	-	Input Capture (bit 0 à 7)
-	ICR1H	-	Input Capture (bit 8 à 15)
TIMSK0	TIMSK1	TIMSK2	Interrupt Mask
TIFR0	TIFR1	TIFR2	Interrupt Flag

Le timer1 est un timer à 16 bits et composé de deux registres de 8 bits, l'un donnant les 8bits LSB (les bits 0 à 7), l'autre donnant les 8 bits MSB (les bits 8 à 15), comme présenté sur la Figure 1.3.

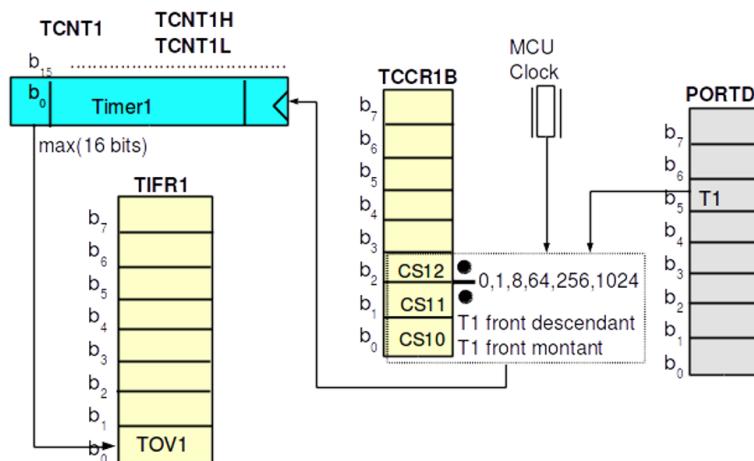


Figure 1.3: Architecture du Timer 1 de l'ATmega328P.

Le tableau suivant fait état des différents registres liés au Timer1 :

Registre	fonctionnalité
ASSR	Asynchronous Status Register
GTCCR	General Timer/Counter Control Register
TCNT	Timer/Counter (Register),
TCCR	Timer/Counter Control Register,
OCR	Output Compare Register,
ICR	Input Capture Register,
TIMSK	Timer/Counter Interrupt Mask Register
TIFR	Timer/Counter Interrupt Flag Register.

Les registres **OCR** et **ICR** ont des rôles particuliers dont nous n'aborderons pas dans ce Lab. **TIMSK** et **TIFR** seront utilisés pour que le timer puisse générer des interruptions. La configuration du Timer1 dans ce mode se fait avec les bits **WGM13**, **WGM12**, **WGM11** et **WGM10** (Waveform Generation Mode).

Les deux premiers sont les bits 4 et 3 du registre **TCCR1B**; les deux derniers sont les bits 1 et 0 du registre **TCCR1A**. Dans le cas présent, il faut mettre à 1 le bit **WGM12**.

```
TCCR1A = 0;
TCCR1B = 0;
TCCR1B |= (1 << WGM12);
```

L'horloge utilisée par le **Timer** est l'horloge principale de l'Arduino nano (16Mhz), à laquelle on peut faire subir une division de fréquence. Le choix de l'horloge se fait sur les bits **CS12**, **CS11**, **CS10** du registre **TCCR1B**. Le tableau suivant contient les facteurs de division, l'indice du tableau correspondant au 3 bits ci-dessus:

- **CSn2 = 0, CSn1 = 0, CSn0 = 0** : timer inactif
- **CSn2 = 0, CSn1 = 0, CSn0 = 1** : fréquence f principale
- **CSn2 = 0, CSn1 = 1, CSn0 = 0** : fréquence f/8
- **CSn2 = 0, CSn1 = 1, CSn0 = 1** : fréquence f/64
- **CSn2 = 1, CSn1 = 0, CSn0 = 0** : fréquence f/256
- **CSn2 = 1, CSn1 = 0, CSn0 = 1** : fréquence f/1024

Dans ce cas le diviseur (prescaler) pourra être défini par un tableau à 6 valeurs:

```
uint16_t diviseur[6] = {0,1,8,64,256,1024};
```

Nous considérons que la période d'échantillonnage en microsecondes soit dans une variable 32 bits/periode. La valeur de **OCR1A** et du sélecteur d'horloge pourront être codés comme suit:

```
uint32_t top = (F_CPU/1000000*period);
int clock = 1;
while ((top>0xFFFF)&&(clock<5)) {
    clock++;
    top = (F_CPU/1000000*period/diviseur[clock]);
}
OCR1A = top;
```

La macro F_CPU contient la fréquence de l'Arduino en Hz (16 MHz). On attribue au registre OCR1B la moitié de OCR1A:

```
OCR1B = top >> 1;
```

Pour qu'une interruption se déclenche lorsque le compteur est égal à **OCR1B**, il faut activer le bit correspondant dans le registre **TIMSK1** (Timer interrupt mask) :

```
TIMSK1 = (1 << OCIE1B);
```

Il faut aussi penser à activer les interruptions avec :

```
sei();
```

Pour déclencher le **Timer1**, il faut écrire les 3 bits définissant l'horloge :

```
TCCR1B |= clock;
```

Pour le stopper, il faut mettre ces 3 bits à zéro :

```
TCCR1B &= ~0x7;
```

II. Réglage de la précision du convertisseur

Les arduino nano comportent un convertisseur analogique-numérique 10 bits (ADC). La configuration de l'ADC se fait à l'aide de 3 registres 8 bits :

- **ADMUX** : ADC Multiplexer Selection Register
- **ADCSRA** : ADC Control and Status Register A
- **ADCSRB** : ADC Control and Status Register B

Le convertisseur est associé à un multiplexeur-amplificateur, qui permet de choisir l'entrée analogique à utiliser, ou les deux entrées dans le cas d'une mesure différentielle. Il est aussi possible d'appliquer un gain au signal analogique avant la conversion. Le multiplexeur se configure avec un code 6 bits. Pour connaître ce code, il faut consulter la documentation du microcontrôleur. Cette dernière est présentée sur la Figure 1.4.

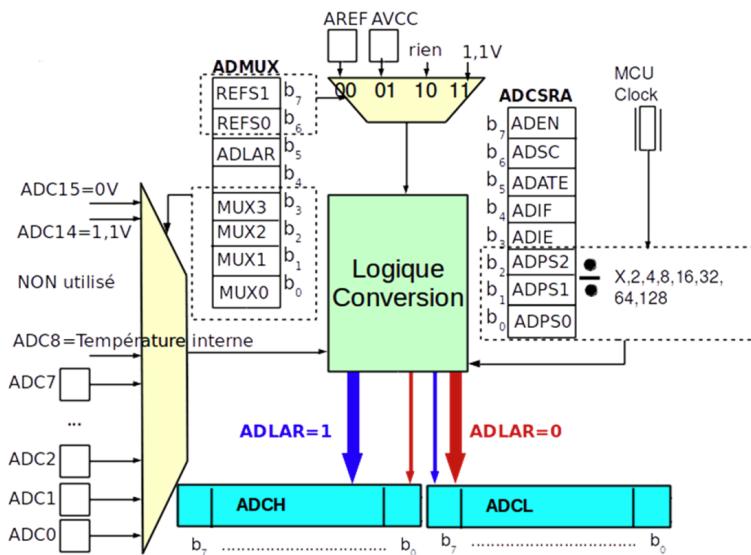


Figure 1.4: Diagramme structurel du convertisseur analogique-numérique de l'ATMega328P.

Pour le ATmega328, les codes 0 à 7 correspondent aux bornes A0 à A7 utilisées en entrée simple. Les 5 premiers bits du code sont les 5 premiers bits du registre **ADMUX**. Le dernier bit du code (poids fort) est le bit 2 du registre **ADCSRB**.

Voici comment se fait la configuration du multiplexage :

```
uint8_t multiplex = 0; //  
ADMUX = 0; //  
ADCSRB = 0; //  
ADMUX |= multiplex; //  
ADCSRB |= (multiplex & 0b00100000) >> 2; //  
uint8_t multiplex = 0; //  
ADMUX = 0; //  
ADCSRB = 0; //  
ADMUX |= multiplex; //  
ADCSRB |= (multiplex & 0b00100000) >> 2; //
```

La tension de référence de l'ADC est choisie par les deux bits **REFS0** et **REFS1** du registre **ADMUX**. On choisit ici AVCC :

```
ADMUX |= (1 << REFS0);
```

La conversion se fait en 10 bits. Le résultat d'une conversion est dans les deux registres 8 bits **ADCH** (4 bits de poids fort) et **ADCL** (8 bits de poids faible). Il faut lire d'abord le registre **ADCH**. Pour obtenir la vitesse de numérisation la plus grande possible, on fera une numérisation en 8 bits. Dans ce cas, on a intérêt à décaler les bits vers la gauche pour que la valeur 8 bits se trouve entièrement dans le registre **ADCH**. Ce réglage est obtenu par :

```
ADMUX |= (1 << ADLAR);
```

Dans le registre **ADSRA**, on doit activer l'ADC, l'auto-déclenchement et les interruptions :

```
ADCSRA |= (1 << ADEN); // enable ADC
ADCSRA |= (1 << ADATE); // Auto Trigger Enable
ADCSRA |= (1 << ADIE); // interrupt enable
```

Dans le cas de l'auto-déclenchement matériel, la source du déclenchement doit être sélectionnée avec les bits **ADTS2**, **ADTS1** et **ADTS0** du registre **ADCSR**. On choisit de déclencher la conversion lorsque le compteur du Timer 1 (**TCNT1**) atteint la valeur de **OCR1B**.

```
ADCSR |= (1 << ADTS2) | (1 << ADTS0); // trigger source : timer 1 comp B
```

Le dernier paramètre à régler est la fréquence de l'horloge utilisée par l'ADC, qui est la fréquence principale de votre arduino nano divisée par un facteur 2,4,8,16,32,64,128. Les trois premiers bits **ADPS2**, **ADPS1** et **ADPS0** du registre **ADCSRA** définissent le code correspondant (1,2,3,4,5,6,7). Voici la configuration de ce code :

```
uint8_t prescaler = 4;
ADCSRA |= prescaler;
```

La valeur utilisée par la fonction analogRead est 7, soit un préfacteur de 128. Cette valeur conduit à une résolution effective de 10 bits mais la conversion est trop lente pour un échantillonnage aux fréquences de l'ordre de 40 kHz. On sera donc amené à choisir un préfacteur plus petit pour accélérer la conversion, sachant qu'il y aura une perte de résolution, non gênante puisqu'on lira seulement les 8 bits de poids fort.

La conversion est effectuée automatiquement lorsque **TCNT1=OCRB1**. Lorsque la conversion est terminée, une interruption est générée par l'ADC (bit **ADIE** activé ci-dessus). La lecture et la mémorisation de la valeur doit se faire dans la fonction d'interruption suivante :

```
ISR(ADC_vect)
{
    // ADC interrupt handler
    uint8_t x = ADCH; // lecture 8 bits
}
```

III. Mémorisation et transmission des échantillons

Les échantillons obtenus dans la fonction **ADC_vect** doivent être mémorisés dans un tampon. Nous choisissons un tampon de 256 éléments. Lorsqu'on utilise le tampon, par exemple pour le transmettre à l'ordinateur par la liaison série, il ne faut pas que l'ADC soit en train d'écrire dessus. Pour résoudre ce problème, on utilise la technique des tampons multiples. Voici comment se fait la déclaration d'un tableau à 4 buffers des indices associés :

```
#define NBUF 0x4
#define NBUF_MASK 0x3 // (NBUF-1)
#define BUF_SIZE 0x100
#define BUF_MASK 0xFF // (BUF_SIZE-1)
volatile uint8_t buf[NBUF][BUF_SIZE];
volatile uint8_t compteur_buf,compteur_buf_transmis;
volatile uint16_t indice_buf;
```

L'écriture dans le buffer en cours s'effectue l'aide de la fonction suivante :

```
ISR(ADC_vect) {
    buf[compteur_buf][indice_buf] = ADCH;
    indice_buf++;
    if (indice_buf == BUF_SIZE) {
        indice_buf = 0;
        compteur_buf = (compteur_buf+1)&NBUF_MASK;
    }
}
```

Lorsque compteur_buf_transmis est différent de compteur_buf, le buffer correspondant est prêt pour la transmission à l'ordinateur, qui se fait par :

```
Serial.write((uint8_t *)buf[compteur_buf_transmis],BUF_SIZE);
compteur_buf_transmis=(compteur_buf_transmis+1)&NBUF_MASK;
```

Les données sont donc transmises par bloc de 256 échantillons.

IV. Code d'échantillonnage et de quantification

A. présentation de la démarche

Afin de réaliser la fonction d'échantillonnage quantification, nous vous proposons d'implémenter les fonctions ci-dessous et de suivre la démarche :

```
#include "Arduino.h"
#define SET_ACQUISITION 100
#define STOP_ACQUISITION 101
#define NBUF 0x4
#define NBUF_MASK 0x3 // (NBUF-1)
#define BUF_SIZE 0x100
#define BUF_MASK 0xFF // (BUF_SIZE-1)
volatile uint8_t buf[NBUF][BUF_SIZE];
volatile uint8_t compteur_buf,compteur_buf_transmis;
volatile uint16_t indice_buf;
uint16_t diviseur[6] = {0,1,8,64,256,1024};
uint32_t nblocs;
uint8_t sans_fin;
uint8_t flag;
```

La fonction suivante initialise et déclenche le timer avec la période donnée :

```
void timer1_init(uint32_t period) {
    TCCR1A = 0;
    TCCR1B = 0;
    TCCR1B |= (1 << WGM12);
    uint32_t top = (F_CPU/1000000*period);
    int clock = 1;
    while ((top>0xFFFF)&&(clock<5)) {
        clock++;
        top = (F_CPU/1000000*period/diviseur[clock]);
    }
    OCR1A = top;
    OCR1B = top >> 1;
    TIMSK1 = (1 << OCIE1B);
    TCCR1B |= clock;
}
```

La fonction adc_int initialise l'ADC de votre arduino nano:

```
void adc_init(uint8_t multiplex, uint8_t prescaler) {
    ADMUX = (1 << REFS0);
    ADMUX |= multiplex & 0b00011111;
    ADMUX |= (1 << ADLAR); //left adjust
    ADCSRA = 0;
    ADCSRA |= (1 << ADEN); // enable ADC
    ADCSRA |= (1 << ADATE); // Auto Trigger Enable
    ADCSRA |= (1 << ADIE); // interrupt enable
    ADCSRA |= prescaler ;
    ADCSRB = 0;
    ADCSRB |= (multiplex & 0b00100000) >> 2;
    ADCSRB |= (1 << ADTS2)|(1 << ADTS0); // trigger source : timer 1 comp B
}
```

La fonction d'interruption vers ADC_vect est appelée lorsque la conversion est terminée :

```
ISR(ADC_vect) {
    buf[compteur_buf][indice_buf] = ADCH;
    indice_buf++;
    if (indice_buf == BUF_SIZE) {
        indice_buf = 0;
        compteur_buf = (compteur_buf+1)&NBUF_MASK;
    }
}
```

Il y a aussi une interruption COMPB générée par le Timer1 qu'il faut exécuter pour que votre ADC fonctionne. Dans la fonction d'interruption, on fait alterner la sortie PD5, ce qui permettra de contrôler le fonctionnement de l'échantillonneur à l'oscilloscope. Cette sortie est reliée à la pin 3 sur l'Arduino nano :

```
ISR(TIMER1_COMPB_vect) {
    if (flag==0) {
        flag = 1;
        PORTD &= ~(1<<PORTD3); // sortie 3 sur Arduino UNO
    }
    else {
        flag = 0;
        PORTD |= (1<<PORTD3);
    }
}
```

Pour stopper le timer1 et l'acquisition, nous proposons de mettre une nouvelle valeur dans le registre TCCR1B :

```
void stop_acquisition() {
    TCCR1B &= ~0x7;
}
```

La fonction setup() établit la communication série avec l'ordinateur

```
void setup() {
    char c;
    Serial.begin(115200);
    Serial.setTimeout(0);
    c = 0;
    Serial.write(c);
    c = 255;
    Serial.write(c);
    c = 0;
    Serial.write(c);
    compteur_buf = compteur_buf_transmis = 0;
    indice_buf = 0;
    DDRD |= 1 << PORTD3; // PD3 en sortie
    flag = 0;
}
```

Pour configurer l'acquisition et la démarrer, nous proposons d'envoyer les données suivantes vers l'Arduino :

- Le code de multiplexage, sous forme d'un entier 8 bits.
- Le prescaler d'horloge, sous forme d'un entier 8 bits.
- La période d'échantillonnage en microsecondes, sous forme d'un entier 32 bits.
- Le nombre de blocs à acquérir, sous forme d'un entier 32 bits. Un bloc est constitué de 256 échantillons. Si ce nombre est nul, l'acquisition se fait sans fin (la transmission aussi).

```
void lecture_acquisition() {
    uint32_t c1,c2,c3,c4, period;;
    uint8_t multiplex,prescaler;
    while (Serial.available()<2) {};
    multiplex = Serial.read(); prescaler = Serial.read();
    while (Serial.available()<4) {};
    c1 = Serial.read(); c2 = Serial.read(); c3 = Serial.read(); c4 = Serial.read();
    period = ((c1 << 24) | (c2 << 16) | (c3 << 8) | c4);
    while (Serial.available()<4) {};
    c1 = Serial.read(); c2 = Serial.read(); c3 = Serial.read(); c4 = Serial.read();
    nblocs = ((c1 << 24) | (c2 << 16) | (c3 << 8) | c4);
    if (nblocs==0) {
        sans_fin = 1;
        nblocs = 1;
    }
    else sans_fin = 0;
    compteur_buf=compteur_buf_transmis=0;
    indice_buf = 0;
    cli();
    timer1_init(period);
    adc_init(multiplex,prescaler);
    sei();
}
```

Le port série est lu de manière non bloquante. En cas de présence d'une des deux commandes, elle exécute l'action correspondante.

```

void lecture_serie() {
    char com;
    if (Serial.available()>0) {
        com = Serial.read();
        if (com==SET_ACQUISITION) lecture_acquisition();
        if (com==STOP_ACQUISITION) stop_acquisition();
    }
}

```

Voici la fonction loop, qui transmet les buffers sur le port série.

```

void loop() {
    if ((compteur_buf_transmis!=compteur_buf)&&(nblocs>0)) {
        Serial.write((uint8_t *)buf[compteur_buf_transmis],BUF_SIZE);
        compteur_buf_transmis=(compteur_buf_transmis+1)&NBUF_MASK;
        if (sans_fin==0) {
            nblocs--;
            if (nblocs==0) stop_acquisition();
        }
    }
    else lecture_serie();
}

```

B. Travail demandé

Q1. Après avoir connecté l'Arduino avec le capteur de son KY-038 ainsi que la LED et la résistance, vous pouvez commencer à recopier le script ci-dessous qui lit la valeur analogique et numérique du capteur de son et si la valeur numérique du capteur est HAUT, la LED rouge externe s'allume.

```

1 int led = 6;
2 int sdigital = 7;
3 int sanalog = A0;
4
5 void setup(){
6     Serial.begin(115200);
7     pinMode(led , OUTPUT);
8     pinMode(sdigital , INPUT);
9 }
10
11 void loop(){
12     int valdigital = digitalRead(sdigital);
13     int valanalog = analogRead(sanalog);
14     Serial.print(valanalog);
15     Serial.print("\t");
16     Serial.println(valdigital);
17     if (valdigital == HIGH)
18     {
19         digitalWrite (led , HIGH);
20         delay(1000);
21     }
22     else
23     {
24         digitalWrite (led , LOW);
25     }
26 }

```

Q2. Dans la dernière partie du script du programme, vous créez une structure if else où vous allumez la LED avec la fonction digitalWrite(led, HIGH) si la valeur numérique du module KY-038 passe de LOW

à HIGH. Avec la fonction delay, vous assurez que la LED est allumée pendant au moins 1 secondes. Si la valeur numérique du capteur est LOW, la LED est éteinte. Utiliser votre moniteur série de votre IDE Arduino pour afficher les variables **s analog** et **s digital**, vous pouvez voir les valeurs des capteurs dans le moniteur série de votre IDE Arduino. Pour faciliter la lecture, vous séparez les deux valeurs par un tabulateur.

Q3. Pour mieux comprendre le comportement des valeurs du capteur analogique, utilisez le serial tracer pour tracer le signal phonatoire que vous obtenez chaque fois que vous prononcez un mot au microphone. Pensez à rapprocher le microphone de votre bouche à chaque fois que vous prononcez un mot.

Q4. Pouvez-vous calculer le temps nécessaire pour que la fonction **analogRead()** puisse lire le signal de votre microphone?.

Dans le reste du TP, nous vous proposons de construire votre propre fonction de conversion analogique numérique et remplacer la fonction de **analogRead()**. Pour réaliser la nouvelle fonction, vous devez écrire les fonctions pour choisir le canal à sélection notée **setAnalogMux()**, la fonction de prédivision de la fréquence et choisir le mode de fonction de votre convertisseur, fonction pour fixer la résolution de l'ADC et la fonction d'activation et désactivation de votre ADC.

Q5. Pour réaliser votre fonction d'activation et désactivation, nous vous demandons de modifier l'état du bit **ADEN** dans le registre **ADCSRA** de compléter le code des deux fonctions ci-dessous et de tester leurs fonctionnement.

```

1 void ADC_enable() { // activer l'ADC
2     cli() ;
3
4
5
6     sei() ;
7 }
8

```

```

1 void ADC_disable()
2     cli() ;
3
4
5
6     sei() ;
7 }
8

```

Q6. Pour réaliser la nouvelle fonction conversion analogique numérique, nous vous demandons de créer nouveau fichier c arduino et d'écrire un nouveau type enum :**ADC_modes** qui contiendra tous les composants de votre ADC.

```

1 enum ADC_modes {
2     /* Pour \ setAnalogMux() */
3     ADC_A0,
4     ADC_A1,
5     ADC_A2,
6     ADC_A3,
7     ADC_A4,
8     ADC_A5,
9     ADC_A6,
10    ADC_A7,
11    ADC_SENSOR,
12    ADC_1V1,
13    ADC_GND,
14    /* Pour ADC_setReference() */
15    ADC_AREF, } ;
16
17

```

Q7. Pour sélectionner un canal à lequel le microphone est attaché, nous vous proposons de créer une fonction de multiplexage notée **setAnalogMux()**. Pour cela, nous vous demandons de compléter le code ci-dessous en utilisant la table avec les différentes valeurs de MUX.

```

1 void setAnalogMux(ADC_modes mux) {
2     cli() ;
3     switch (mux) {
4         case ADC_A0 :
5             ADMUX &= ~ ((1 << MUX3) | (1 << MUX2) | (1 << MUX1) | (1 << MUX0)) ;
6             break ;
7         .
8         .
9         .
10        .
11    }
12    sei() ;
13 }
14
15

```

MUX3	MUX2	MUX1	MUX0	sortie de l'ADC
0	0	0	0	ADC_A0
0	0	0	1	ADC_A1
0	0	1	0	ADC_A2
0	0	1	1	ADC_A3
0	1	0	0	ADC_A4
0	1	0	1	ADC_A5
0	1	1	0	ADC_A6
0	1	1	1	ADC_A7
1	0	0	0	ADC_SENSOR
1	1	1	1	ADC_GND
1	1	1	0	ADC_1V1

Q8. Pour choisir correctement la fréquence d'échantillonnage de votre ADC, la combinaison des 3bits ADPS sont proposés dans la table dessous :

ADPS2	ADPS1	ADPS0	prescaler
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Pour configurer les bits ADPS [2:0], compléter la fonction ci-dessous:

```

1 void ADC_setPrescaler(byte prescl) {
2     cli() ;
3     switch (prescl) {
4         case 2 : // (defaut)
5             ADCSRA &= ~((1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0)) ;
6             break ;
7         case 4 :
8         .
9         .
10    }
11    sei() ;
12 }

```

Q9. Pour que vous puissiez utiliser une 3 tensions de référence, pensez à rajouter une fonction **ADC_setReference()** pour sélectionner l'une des trois tensions de références AREF, 1.1V ou AVCC. Nous vous proposons de compléter la fonction **ADC_setReference()** ci-dessous:

```

1 void ADC_setReference(ADC_modes ref) {
2     cli() ;
3     switch (ref) {
4         case ADC_1V1 :
5
6
7             break ;
8         case ADC_AREF :
9
10
11             break ;
12         case ADC_VCC :
13
14
15             break ;
16     }
17     sei() ;
18 }
```

Q10. Pour réaliser une conversion analogique numérique simple, nous avons pensé à vous proposer un premier exemple de code de votre nouvelle fonction de conversion analogique numérique

```

1 void setup() {
2     Serial.begin(115200) ;
3     ADC_enable() ;
4     ADC_setPrescaler(32) ;
5     ADC_setReference(ADC_AREF) ;
6     setAnalogMux(ADC_A0) ;
7 }
```



```

9 void loop() {
10     ADC_startConvert() ;
11     while (!ADC_available()) ;
12     Serial.println(ADC_read()) ;
13 }
```

La fonction **ADC_available()** vérifie à chaque fois que l'ADC n'est pas occupé à faire une conversion analogique numérique et la fonction **ADC_read()**, récupère juste la valeur de l'ADC codée sur 10bits.

Q11. Pour modifier la résolution de votre ADC le bit **ADLAR** peut être mis à 0 pour une résolution à 10bit ou à 1 pour une résolution à 8bits. Nous vous demandons de bien intégrer cette fonction avec le reste du code et de vérifier à chaque fois par l'affichage sur le moniteur série du contenu d registre ADC.

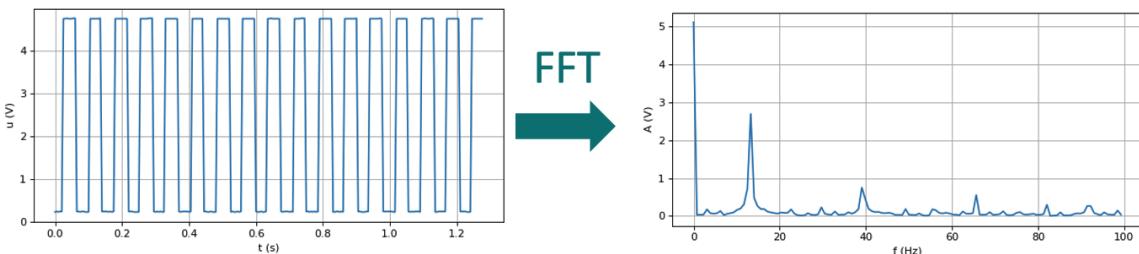
Q12. Utiliser votre programme pour lire et tracer avec le traceur série les signaux phonatoire. Pouvez-vous justifier à chaque le choix de vos paramétrés de résolution de votre ADC et la fréquence de votre prescalar.

TP n° 2

Transformée de Fourier rapide (FFT)

Motivation

Le but de ce TP est de montrer comment analyser le signal phonatoire au moyen de la transformée de Fourier rapide. Une analyse spectrale nécessite de pouvoir mémoriser un grand nombre d'échantillons afin d'effectuer le calcul de la FFT.



Si N est le nombre d'échantillons et T_e la période d'échantillonnage, la résolution fréquentielle est l'inverse de la durée totale en d'autre terme elle est de $1/(NT_e)$. On se fixe pour objectif d'analyser un signal phonatoire dont la fréquence fondamentale est inférieure à 8KHz.

Afin de maximiser la résolution fréquentielle, on choisit la fréquence d'échantillonnage la plus basse possible, soit $f_e = 16\text{KHz}$. Le signal analysé peut cependant contenir des composantes de fréquences supérieures à 8Khz, qui pourraient se replier dans la bande de fréquence analysée.

Pour éviter le repliement, nous allons effectuer la numérisation avec un sur-échantillonnage d'un facteur 3 à 4, c-à-d une fréquence $f_e = 32\text{Khz}$.

L'arduino nano possède un microcontrôleur 8 bits (Atmega328P) et 2Ko de RAM, ce qui permet de traiter $N_e = 256$ échantillons et permettra donc d'analyser un signal audio avec une bonne précision relative de fréquence.

Le microcontrôleur Atmel ATmega328P utilisé sur l'Arduino Uno possède un module de conversion analogique-numérique (ADC) capable de convertir une tension analogique en un nombre de 10 bits de 0 à 1023 ou un nombre de 8 bits de 0 à 255. L'entrée du module peut être sélectionnée pour provenir de l'une des six entrées de la puce.

Bien que cela donne au microcontrôleur la possibilité de convertir les signaux de six sources analogiques différentes en valeurs de 10 bits, seul un canal peut être converti à la fois. L'ADC peut convertir les signaux à un taux d'environ 15 kSPS (échantillons par seconde). Les entrées du module ADC apparaissent sur la carte Arduino comme des connexions A0 à A5. **Le but de ce TP est d'effectuer**

une Transformée de Fourier Rapide (FFT) sur un signal phonatoire issu du microphone KY-037 de votre kit. Le résultat de la FFT sera affiché sur un ecran OLED .

I. Numérisation du signal

La numérisation des signaux phonatoires par la conversion analogique numérique sur le CAN de votre Arduino (Figure 1), il doit également disposer de la gamme de tensions qu'il est censé traiter. Sinon, c'est comme si l'on demandait "Quelle est la hauteur de la hauteur ?". La plage est spécifiée au convertisseur en fournissant une tension de référence basse et haute. Le convertisseur attribue alors des valeurs numériques de manière linéaire entre la valeur numérique minimale et maximale lorsque le signal d'entrée passe de la référence basse à la référence haute. Si l'entrée est égale à la référence basse, la sortie est constituée de zéros. Si elle est égale à la référence haute, la sortie est constituée de tous les uns, et si l'entrée est à mi-chemin entre les références basse et haute, la sortie sera la valeur numérique au milieu de la plage numérique. Sur l'ATmega328P la référence basse est fixée à la masse mais il a la possibilité de sélectionner une des trois sources pour servir de référence haute.

- **AREF** : c'est une pin séparée sur l'atmega328P qui peut être utilisée pour fournir n'importe quelle tension de référence élevée que vous souhaitez utiliser tant qu'elle est dans la gamme de 1.0 V à VCC. Cette pin n'est pas directement accessible sur la carte Arduino nano.
- **AVCC** : cette broche sur le microcontrôleur fournit l'alimentation au circuit ADC sur la puce. Sur l'arduino nano, elle est connectée à VCC. C'est l'option la plus simple à utiliser si AVCC est connecté à VCC.
- **1.1V** : l'amtega328P possède une tension de référence interne de 1.1 V qui peut être utilisée.

La Figure 2.1 présente l'architecture du module CAN de l'ATMega328P.

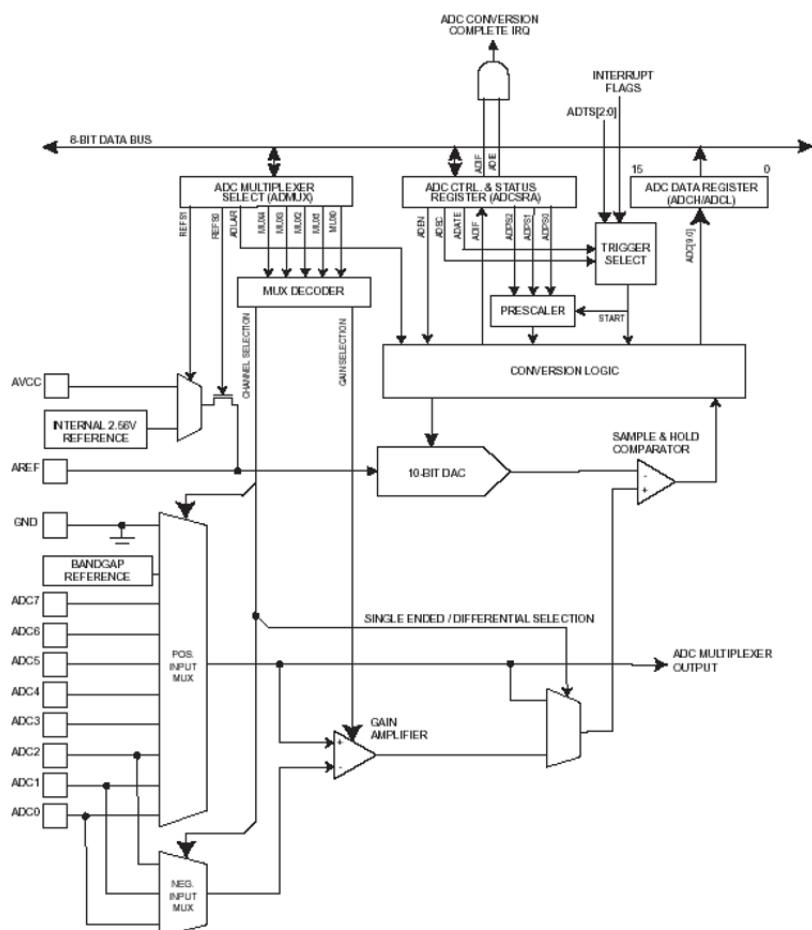


Figure 2.1: Schéma synoptique du CAN de l'ATmega328P.

A. Vitesse de conversion

L'ADC a besoin d'un signal d'horloge dans la gamme de 50kHz à 200kHz. Il n'y a pas d'entrée d'horloge ADC externe sur l'arduino nano, mais l'horloge est générée en interne à partir de la même horloge utilisée pour faire fonctionner le microcontrôleur atmega328P. Cette horloge CPU est trop rapide (16MHz sur le nano) donc la l'atmega328P inclut un "prescaler" ajustable pour diviser l'horloge CPU vers le bas à des valeurs acceptables. Le prescaler peut être réglé pour diviser par un choix de préscaler (2, 4, 8, 16, 32, 64, ou 128) et c'est au programmeur de sélectionner celui qui donne une horloge ADC dans la gamme acceptable.

B. Registres pour programmer l'ADC

L'interface du logiciel avec le module ADC se fait par le biais d'un groupe de registres. Les descriptions ci-dessous sont très brèves et ne couvrent que certaines des fonctions des registres. Pour une description complète, voir la fiche technique de l'ATmega328P. Dans la description ci-dessous, la notation avec des parenthèses et deux points comme "XYZ[2:0]" signifie la combinaison des bits XYZ2, XYZ1 et XYZ0. Par exemple, si nous disons "XYZ[2:0] = 5", cela signifie la valeur binaire de 5. (101) a été attribué aux trois bits XYZ (XYZ2 = 1, XYZ1 = 0, XYZ0 = 1).

(a) ADMUX : Registre de sélection du multiplexeur

Le registre **ADMUX** contient trois champs de bits pour contrôler divers aspects de la conversion de données.

7	6	5	4	3	2	1	0
REFS1	REFS0	ADLAR		MUX3	MUX2	MUX1	MUX0

- **Bits 7:6 :** Les bits **REFS1** et **REFS0** contrôlent la sélection de la source de référence haute.

REFS1	REFS0	Sélection de la référence
0	0	AREF
0	1	AVCC
1	1	Interne 1,1 V

- **Bit 5 :** Le bit **ADLAR** détermine si le résultat doit être ajusté à gauche ou à droite.

REFS1	REFS0	Sélection de la référence
0	0	AREF
0	1	AVCC
1	1	Interne 1,1 V

- **Bits 3:0 :** Les bits **MUX3** à **MUX0** contrôlent la sélection de l'une des six lignes d'entrée à connecter au circuit de numérisation.

ADLAR	Résultat de la conversion
0	Ajusté à droite pour des résultats sur 10 bits
1	Ajusté à gauche pour les résultats sur 8 bits

(b) ADCSRA - Registre A de contrôle et d'état

Le module ADC de l'ATmega328P possède deux registres de contrôle et d'état, **ADCSRA** et **ADC-SRB**. Pour les opérations de base de l'ADC, seuls les bits du registre **ADCSRA** doivent être modifiés. Pour plus d'informations sur le registre **ADCSRB**, voir la fiche technique de l'ATmega328P.

7	6	5	4	3	2	1	0
ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0

Ce registre a des bits pour initier des actions et rapporter diverses conditions dans le module ADC. Les bits nécessaires pour ce Lab sont les suivants.

- **Bit 7 : ADEN** : Le bit **ADEN** active le module ADC. Il doit être mis à 1 pour effectuer toute opération ADC.
- **Bit 6 : ADSC** : La mise à 1 du bit **ADSC** initie une conversion unique. Ce bit restera à 1 jusqu'à ce que la conversion soit terminée. Si votre programme utilise la méthode d'interrogation pour déterminer quand la conversion est terminée, il peut tester l'état de ce bit pour déterminer quand il peut lire le résultat de la conversion dans les registres de données. Tant que ce bit est à 1, les registres de données ne contiennent pas encore de résultat valide.
- **Bit 5 : ADATE** : Lorsque le bit **ADATE** est mis à l'état haut, le déclenchement automatique de l'ADC est activé. L'ADC commencera une conversion sur un front montant du signal de déclenchement sélectionné. La source de déclenchement est sélectionnée en réglant les bits de sélection de déclenchement de l'ADC, **ADTS** dans **ADCSRB**.
- **Bit 4 : ADIF** : Le bit **ADIF** est activé lorsqu'une conversion ADC est terminée et que les registres de données sont mis à jour. L'interruption **ADIF** est exécutée si le bit **ADIE** et le bit I dans **SREG** sont activés. **ADIF** est effacé par le matériel lors de l'exécution du vecteur de traitement d'interruption correspondant. Attention, si vous effectuez une lecture-modification-écriture sur **ADCSRA**, une interruption en attente peut être désactivée. Ceci s'applique également si les instructions SBI et CBI sont utilisées
- **Bit 3 : ADIE** : La mise à 1 du bit **ADIE** active les interruptions. Une interruption sera générée à la fin d'une conversion. Le nom du vecteur d'interruption est "ADC_vect".
- **Bits ADSP 2:0 :** Ces bits déterminent le facteur de division entre la fréquence de l'horloge système et l'horloge d'entrée de l'ADC. Les bits **ADPS2**, **ADPS1** et **ADPS0** sélectionnent la valeur du diviseur du prescaler.

ADPS2	ADPS1	ADPS0	Facteur de division
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

(c) Registre de données ADCH et ADCL (octets haut et bas)

Les résultats d'une conversion sont stockés dans les deux octets du registre de données. Les bits les plus significatifs du résultat sont stockés dans **ADCH** et les bits les moins significatifs sont dans **ADCL**. Si vous souhaitez utiliser les résultats complets de la conversion 10 bits, le bit **ADLAR** du registre **ADMUX** doit être mis à zéro. Ainsi, le résultat de la conversion de 10 bits est justifié à droite dans la combinaison de 16 bits de **ADCH** et **ADCL** (appelée ADC dans votre code C/arduino). Les deux bits les plus significatifs sont dans **ADCH** (bits 1 et 0) et les huit bits inférieurs sont dans **ADCL**.

	7	6	5	4	3	2	1	0
ADCH							ADC9	ADC8
ADCL	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0

Lorsque l'on utilise les résultats 10 bits (**ADLAR**=0), les résultats peuvent être enregistré.

Ainsi, les bits d'ordre supérieur de **ADCH** sont stockés dans l'octet supérieur de la variable 16 bits, et les huit bits inférieurs de **ADCL** sont stockés dans l'octet inférieur de la variable. Dans de nombreux cas, seule une valeur de conversion de huit bits est nécessaire. Pour cette situation, mettez le bit **ADLAR** dans le registre **ADMUX** à l'état haut. Les résultats de la conversion seront justifiés à gauche avec les huit bits les plus significatifs dans **ADCH**. Le résultat de 8 bits peut alors être obtenu en lisant simplement le registre **ADCH** et en ignorant le contenu de **ADCL**.

Lorsque vous utilisez des résultats sur 8 bits (**ADLAR**=1), les résultats peuvent être lus à l'aide d'une variable intermédiaire :

```
unsigned char x ;
x = ADCH ;
```

C. Utilisation de l'ADC

(a) Configuration générale de l'ADC

Les étapes de base de la configuration du module ADC pour effectuer des conversions sont les suivantes.

- Configurez les bits **REFS**[1:0] pour sélectionner la référence à utiliser. L'utilisation de AVCC est recommandée.
- Activez ou désactivez le bit **ADLAR** selon que vous souhaitez utiliser des résultats de conversion de 10 bits ou 8 bits.
- Configurez les bits **MUX**[3:0] pour sélectionner le canal d'entrée à utiliser.
- Configurez les bits **ADPS**[2:0] pour sélectionner la valeur du prescaler d'horloge.
- Mettez le bit **ADEN** dans **ADCSRA** à un. Cela active l'ADC et vous êtes maintenant prêt à lancer une conversion.

Si vous utilisez des interruptions, procédez également comme suit :

- Écrire une routine de service d'interruption (ISR) pour l'interruption ADC_vect.
- Mettez le bit **ADIE** dans **ADCSRA** à un pour permettre au module d'interrompre.
- Activez les interruptions globales avec l'appel de fonction sei().

(b) par écoute active (*polling*)

Pour effectuer des conversions à l'aide de sondages, procédez comme suit :

1. Mettez le bit **ADSC** dans **ADCSRA** à un. Cela lance une conversion.
2. Entrer dans une boucle vérifiant l'état du bit ADSC. Tant qu'il est toujours à un, la conversion est en cours. Une fois que ce bit devient un zéro, la conversion est terminée.
3. Lire le résultat de **ADCH** (8 bits) ou **ADC** (10 bits).

(c) en utilisant les interruptions

Pour effectuer des conversions en utilisant des interruptions, procédez comme suit.

1. Mettez le bit **ADSC** dans **ADCSRA** à un. Cela lance la première conversion.
2. Entrez dans une boucle sans rien faire ou effectuez une autre tâche pour attendre que l'interruption se produise.
3. Dans l'ISR, lisez le résultat de **ADCH** (8 bits) ou **ADC** (10 bits). Si vous souhaitez lancer immédiatement une autre conversion, vous pouvez le faire dans l'ISR en activant à nouveau le bit **ADSC**.

Ci-dessous un exemple de code pour configurer votre ADC avec des interruptions :

```
#define M_PIN 0
void setup() {
    Serial.begin(115200); // Initialize serial port for debugging.

    // Setup the ADC for polled 10 bit sampling on analog pin 5 at 19.2 kHz.
    cli(); // Disable interrupts.
    ADCSRA = 0; // Clear this register.
    ADCSRB = 0; //
    ADMUX = 0; //
    ADMUX |= (M_PIN & 0x07); // Set A0 analog input pin.
    ADMUX |= (0 << REFS0); // Set reference voltage
    ADMUX |= (0 << ADLAR); // Right justify to get full 10 A/D bits.

    // Set ADC clock with 64 prescaler where 16mHz/64=250kHz
    // and 250khz/13 instruction cycles = 19.2khz sampling.
    ADCSRA |= bit (ADPS1) | bit (ADPS2);

    ADCSRA |= (1 << ADATE); // Enable auto trigger.
    ADCSRA |= (1 << ADEN); // Enable ADC.
    ADCSRA |= (1 << ADSC); // Start ADC measurements.
    sei(); // Re-enable interrupts.
}
```

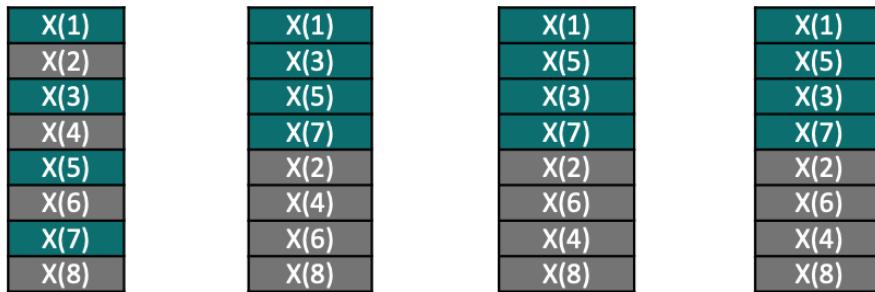
T1. Expliquer le rôle de chaque registre dans le code ci-dessus.

II. Calcul de la Transformée de Fourier

Dans cette partie du TP nous allons enregistrer des signaux sonores et analyser leurs composantes fréquentielles. La première partie importante pour effectuer une telle analyse de fréquence est l'algorithme de la transformée de Fourier rapide (FFT), qui convertit un signal temporel échantillonné en sa transformée de Fourier.

La forme la plus simple et la plus courante de l'algorithme FFT est appelée l'algorithme de Cooley-Tukey, qui fonctionne comme suit :

- Pour un signal temporel de N échantillons, séparez le signal en N signaux temporels différents, chacun ne comportant qu'un seul échantillon. Cette opération est effectuée de manière récursive, en divisant le signal en deux moitiés à chaque itération : un signal contenant les échantillons pour les indices pairs, un autre pour les indices impairs. Voici un exemple d'un signal à 8 échantillons:



- Prendre la composante de fréquence de chacun de ces signaux temporels à 1 échantillon, une étape très facile. Chaque valeur fréquentielle est juste égal au valeur du signal temporel lorsqu'il n'y a qu'un seul échantillon.
- Les signaux individuels du domaine fréquentiel doivent être reconstruits, dans l'ordre inverse de la séparation du signal temporel.

Cette explication est un peu simpliste, car elle ignore les nombres complexes et les détails de calcul de la FFT qui la rendent efficace. Essentiellement, la FFT exploite la propriété de la transformée de Fourier en temps discret selon laquelle, pour calculer la transformée de Fourier d'un signal, on peut combiner les transformées de signaux plus simples. L'utilisation de la récursion à cet égard réduit un problème très complexe sur le plan informatique à une complexité de seulement $O(n^* \log(n))$.

A.Bibliothèque Arduino FFT

Pour ce lab, nous utiliserons la bibliothèque FFT d'Arduino pour l'implémentation FFT. Pour mieux comprendre le fonctionnement de la FFT d'Arduino, nous avons téléchargé la bibliothèque et regarder l'exemple FFT05". Il faut prendre les précautions lorsque vous utilisez la bibliothèque Arduino FFT

1. La fréquence d'horloge du microcontrôleur Arduino est de 16 MHz.
2. L'ADC convertit une tension d'entrée analogique en une valeur numérique de 10 bits (cette ligne est copiée directement de la fiche technique).
3. Une conversion ADC normale prend 13 cycles d'horloge ADC et la première conversion prend 25 cycles.
4. Le facteur prescaler peut être modifié par les trois derniers bits de **ADCSRA**.
5. L'horloge d'entrée de l'ADC = fréquence de l'horloge système / facteur de division.
6. La conversion ADC fonctionne en continu en mode libre.
7. En augmentant la valeur du prescalaire, nous pouvons obtenir une résolution de fréquence plus élevée. Ceci va diminuer la fréquence d'échantillonnage du fft, ce qui peut poser problème lorsque nous échantillonons une composante haute fréquence conformément au théorème de Nyquist.

La sortie de l'algorithme FFT est la transformée de Fourier discrète, qui indique la puissance présente dans 256 "cases" de fréquence. La plage de fréquences totale analysée est la fréquence d'échantillonnage, et chaque bac décrit la puissance présente dans une plage de 1/256 de la fréquence d'échantillonnage.

Dans l'exemple du sketch, ADCSRA est réglé sur 0xe5, ce qui signifie que l'ADC est en mode libre et que le facteur prescaler est fixé à la valeur 32. Nous remarquons alors que la fréquence d'échantillonnage de la FFT est de 16MHz/32 prescalar/13 cycles ADC \approx 38461 Hz. Par conséquent, la largeur du bin de la FFT est de $38461 \text{ Hz} / 256 \approx 150 \text{ Hz}$. Si nous entrons un signal de 8000Hz, il est supposé être proche du bin 5 ($8000\text{Hz}/150 \approx 53$).

Pour tester notre hypothèse, nous vous demandons d'écrire un code C/Arduino pour générer un signal sinusoïdale sur la sortie A0, (pensez à utiliser un Oscilloscope pour bien vérifier l'allure de votre signal).

Sur un deuxième Arduino, nous vous demandons d'écrire le code pour acquérir, nous vous demandons d'écrire un code C/Arduino pour générer un signal sinusoïdale sur la sortie A0, (pensez à utiliser un Oscilloscope pour bien vérifier l'allure de votre signal).

B. Analyse FFT

L'objectif de cette sous-section est de détecter et de distinguer des sons à des fréquences différentes. Ceci doit également être fait en présence de bruit. Pour ce faire, nous allons utiliser le microphone KY 037 pour recueillir un enregistrement sonore dans le domaine temporel, et analyser sa transformée de Fourier et représenter son spectre d'amplitude sur un écran OLED. Dans la section suivante, nous vous demandons de suivre les différentes étapes pour arriver à la réalisation d'un spectromètre du signal phonatoire.

C. Travail demandé

Q1. Après avoir connecté l'Arduino avec le capteur de son KY-038 ainsi que l'écran OLED, vous pouvez commencer à recopier le script ci-dessous pour tester le fonctionnement de votre montage sur la figure ci-dessous. Assurez-vous du bon fonctionnement de votre montage par l'exécution de code arduino

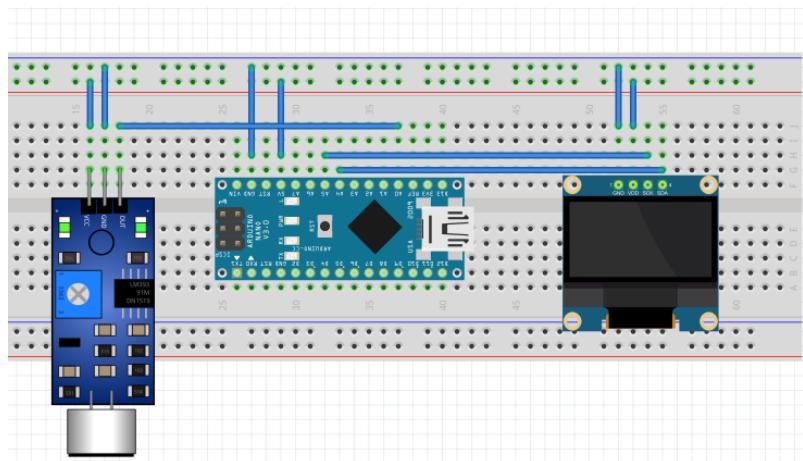


Figure 2.2: Circuit permettant de numériser un signal analogique émanant d'un microphone KY-037 et d'afficher son spectre sur un écran OLED.

ci-dessous. Le code arduino permettra d'afficher le signal phonatoire avec une échelle appropriée à votre écran OLED.

```

1 #include <Wire.h>
2 #include <Adafruit_GFX.h>
3 #include <Adafruit_SSD1306.h>
4
5 // defines for setting and clearing register bits
6 // defining pins for control

```

```

7 const int pinAnalogIn = A0;
8
9 const int bufferSize = 128;
10
11 const int minMode = 1;
12 const int maxMode = 7 + 500;
13
14 const int defaultMode = 7;
15
16 // initing display
17 Adafruit_SSD1306 display(128, 32, &Wire, -1);
18 byte buffer[bufferSize];
19
20 int mode = defaultMode;
21
22 void drawScreen(unsigned long duration) {
23     display.clearDisplay();
24     display.setTextColor(BLACK, WHITE);
25     for (int v = 5; v >= 0; v--) {
26         int tickY;
27         if (v == 5)
28             tickY = 0;
29         else if (!v)
30             tickY = display.height() - 1;
31         else
32             tickY = display.height() - display.height() * v / 5;
33         for (int x = 0; x < display.width(); x += 8)
34             display.drawPixel(x, tickY, WHITE);
35     }
36
37     for (int x = 0; x < bufferSize; x++) {
38         if (!x)
39             display.drawPixel(x, buffer[x], WHITE);
40         else
41             display.drawLine(x - 1, buffer[x - 1], x, buffer[x], WHITE);
42     }
43
44     display.setCursor(0, 0);
45     display.print("5V");
46     display.setCursor(0, display.height() - 7);
47     display.print(duration);
48     display.print(" us");
49     display.display();
50 }
51
52 void setup() {
53     // defining pins for control
54     pinMode(pinAnalogIn, INPUT);
55     display.begin(SSD1306_SWITCHCAPVCC, 0x3C, false);
56     display.setTextSize(1);
57     display.clearDisplay();
58 }
59
60 void loop() {
61
62     unsigned long delayUs = (mode > 7) ? (mode - 7) * 20 : 0;
63     unsigned long start = micros();
64     for (int x = 0; x < bufferSize; x++) {
65         buffer[x] = map(analogRead(pinAnalogIn), 0, 1023, display.height() - 1, 0);
66         if (delayUs)
67             delayMicroseconds(delayUs);
68     }
69     drawScreen(micros() - start);
70 }
71 }
```

Q1. Afin d'améliorer l'affichage du signal phonatoire sur l'écran OLED, pouvez modifier le code proposé pour avoir une meilleure échelle temporelle possible du signal phonatoire.

Q2. Dans ce même code remplacer la fonction analogRead les fonctions d'initialisation de l'ADC, de fonction de predvision et de déclenchement de l'échantillonnage. Vous pouvez utiliser les mêmes fonctions de vous avez développé dans le TP1

Q3. Ajouter les functions fft dans votre code pour calculer le spectre de votre signal numérique. Tracer à chaque fois le spectre sur traceur série et le signal sur l'écran OLED

Q3. Ajouter les functions fft dans votre code pour calculer le spectre de votre signal numérique. Tracer à chaque fois le spectre sur traceur série et le signal sur l'écran OLED

Q4. Utiliser un buffer circulaire avec votre ADC pour enregistrer le contenu du registre ADC lorsque le CAN fonctionne en mode libre. Vous pouvez ajouter ce buffer circulaire dans la fonction ISR ou simplement dans votre programme principal. La Taille de votre buffer circulaire est fixée N=1280. Vous pouvez choisir une taille inférieure à 1280.

```

1 #include <Wire.h>
2 #include <arduinoFFT.h>
3 #include <Adafruit_SSD1306.h>
4 Adafruit_SSD1306 display(-1);
5 #define SAMPLES 64
6 double vReal[SAMPLES];
7 double vImag[SAMPLES];
8 byte peak;
9 long maxpeak;
10 char buf[5];
11
12 arduinoFFT FFT = arduinoFFT();
13
14 void setup() {
15     byte x = 0;
16     display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
17     display.clearDisplay();
18     display.fillRect(0, 0, display.width() - 2, 11, WHITE);
19     display.setTextColor(BLACK);
20
21     x = 16; display.setCursor(x, 2);
22     display.print(F("AUDIO"));
23     x = 52; display.setCursor(x, 2);
24     display.print(F("SPECTROMETER"));
25
26     for (byte i = 0; i < SAMPLES / 2 - 1; i++) {
27         display.drawFastHLine(i * 4 + 1, display.height() - 1, 3, WHITE);
28     }
29     display.setTextColor(WHITE);
30     display.display();
31 }
32
33 void loop() {
34     for (byte i = 0; i < SAMPLES; i++) {
35
36         ///////////////////////////////////////////////////
37         vReal[i] = analogRead(A0);
38         ///////////////////////////////////////////////////
39         vImag[i] = 0;
40     }
41
42     FFT.DCRemoval();
43     FFT.Windowing(vReal, SAMPLES, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
44     FFT.Compute(vReal, vImag, SAMPLES, FFT_FORWARD);
45     FFT.ComplexToMagnitude(vReal, vImag, SAMPLES);
46
47

```

```

48     display.fillRect(0, 12, display.width() - 2, display.height() - 13, BLACK);
49     for (byte i = 0; i < SAMPLES / 2 - 1; i++) {
50         peak = map(vReal[i+2], 0, 1024, 0, 52);
51         display.fillRect(i * 4 + 1, abs(52 - peak) + 12, 3, peak, WHITE);
52     }
53     maxpeak= FFT.MajorPeak(vReal, SAMPLES, 5000);
54
55     sprintf(buf, "%04li ",maxpeak);
56     display.setCursor(72,16);
57     display.print(F("Peak:"));
58     display.print(buf);
59     display.display();
60 }
```

Pensez à remplacer la fonction analogRead par les fonctions que vous avez développé pour la question **Q2**. Ajoutez un buffer pour améliore le fonction d'acquisition.

Q6. Pouvez augmenter le nombre de points de votre fft à 128 au dans votre nouveau code arduino.

Q7. Parmi les fenetres proposés dans la bibliotheque arduino FFT pour calculer l'erreur entre le spectre avec la fenetre rectangulaire et les trois fenêtres de Hammjing, Hanning et Blackman.

Après avoir bien tester votre code, nous vous demandons d'évaluer l'erreur dans le cas d'une conversion analogique numérique à 8bit.

Q8. Quel rapport entre la résolution de votre CAN et la resolution spectrale. Expliquez

Q9. Modifier votre algorithme pour que l'analyseur de spectre soit le plus rapide possible. Quels sont les choix possibles, justifiez votre réponse.

TP n° 3

Filtrage RIF

Motivation

En traitement du signal, un filtre à réponse impulsionnelle finie ou filtre RIF est un filtre dont la réponse impulsionnelle est de durée finie. De façon générale le filtre RIF est décrit par la combinaison linéaire où les valeurs du signal de sortie sont représentées en fonction des valeurs du signal d'entrée. En utilisant le symbole de sommation, l'équation peut être réécrite de la façon suivante :

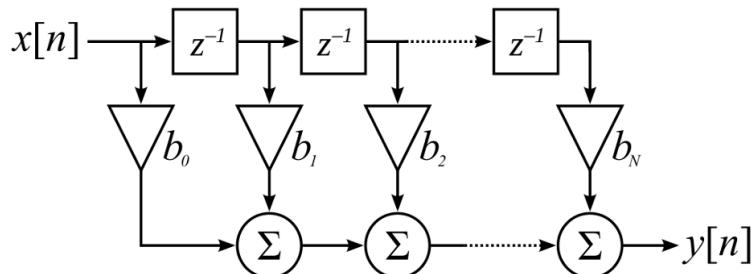
$$\text{output}[i] = \sum_{n=0}^{\text{FILTER_TAP_NUM}} \text{filter_taps}[i] * \text{data_input}[n - i] \quad (3.1)$$

avec :

- FILTER_TAP_NUM : Nombre de coefficients (ordre du filtre)
- filter_[i] : coefficients de la fonction de transfert du filtre.

C'est une convolution (discrète) entre le signal d'entrée et une fonction représentée par les valeurs du signal, celles-ci décrivant ainsi la réponse impulsionnelle du filtre. Puisque la réponse est une somme d'un nombre fini de valeurs, le filtre RIF est naturellement stable. Les filtres numériques peuvent être réalisés à l'aide de trois éléments ou opérations de base. Soit l'élément gain, l'élément de sommation et le retard unitaire. Ces éléments sont suffisants pour réaliser tous les filtres numériques linéaires possibles. La réalisation présentée dans la figure ci-dessous est une réalisation directe de type 1 du filtre RIF d'ordre N, avec un signal d'entrée noté $x[n]$, le signal de sortie $y[n]$ et les éléments de gain b_1, b_2, \dots, b_N .

Les opérations (les additions sont habituellement représentées par des cercles contenant un + et les multiplications par des triangles associés aux coefficients multiplicande) et les arcs les dépendances, c-à-d le flot des données issues du signal. Certains arcs sont valués d'un coefficient z^{-1} représentant un délai d'une période d'échantillonnage. Ce coefficient se représente également sous la forme d'un registre.



I. Rappels sur le filtre RIF

A. Propriétés

Les remarques générales suivantes peuvent être portées sur les filtres RIF.

- Les filtres RIF sont forcément stables, peu importe les coefficients utilisés.
- La complexité d'un filtre RIF est faible, ce qui peut être utile sur les plateformes limitées en puissance de calcul comme l'arduino nano.
- Généralement, les filtres RIF sont moins sensibles aux erreurs de quantification. L'absence de récursivité empêche les erreurs cumulatives.
- Un filtre RIF est moins sélectif. La transition entre la bande passante et la bande rejetée est moins rapide.I.
- Un filtre RIF peut avoir une réponse impulsionnelle symétrique et introduire un retard sur le signal mais aucun déphasage

B. Complexité d'implatation d'un filtre RIF

Un filtre RIF nécessite $N + 1$ opérations de multiplication, N opérations d'addition pour chaque nouvel échantillon à filtrer. On peut également exprimer la complexité en nombre de multiplication-accumulation (MAC), qui, dans le cas du filtre RIF, vaut $N + 1$. Le cout mémoire d'un filtrage RIF est de $2(N + 1)$ ($N + 1$ coefficients bi et $N + 1$ points mémoire pour le vecteur des entrées $x[i]$). Si la fréquence d'échantillonnage du signal vaut F_e , cela signifie que le calcul d'un filtre devra être réalisé en un temps :

$$t_{\text{calcul}} < T_e = \frac{1}{F_e} \quad (3.2)$$

Sur un microcontrôleur de type atmega328P capable d'exécuter une multiplication-accumulation (MAC) à chaque cycle, de puissance de calcul P_{calcul} exprimée en MIPS (Million d'Instruction Par Seconde), le temps de calcul sera :

$$T_{\text{calcul}} = (N + 1) T_{\text{cycle}} = \frac{(N + 1)}{P_{\text{calcul}}} \quad (3.3)$$

Aussi, la puissance de calcul pour l'implantation d'un filtre RIF vaut :

$$P_{\text{calcul}} (\text{MIPS}) > \frac{(N + 1) f_e}{106} \quad (3.4)$$

II. Conception d'un filtre RIF

Nous vous proposons de réaliser un filtre numérique RIF de type passe-bas avec les spécifications suivantes:

- $f_c = 500$ Hz
- nombre de coefficients : entre 5 et 15

Pour ce faire, nous allons directement récupérer les coefficients de l'outil en ligne suivant : <http://t-filter.engineerjs.com/>

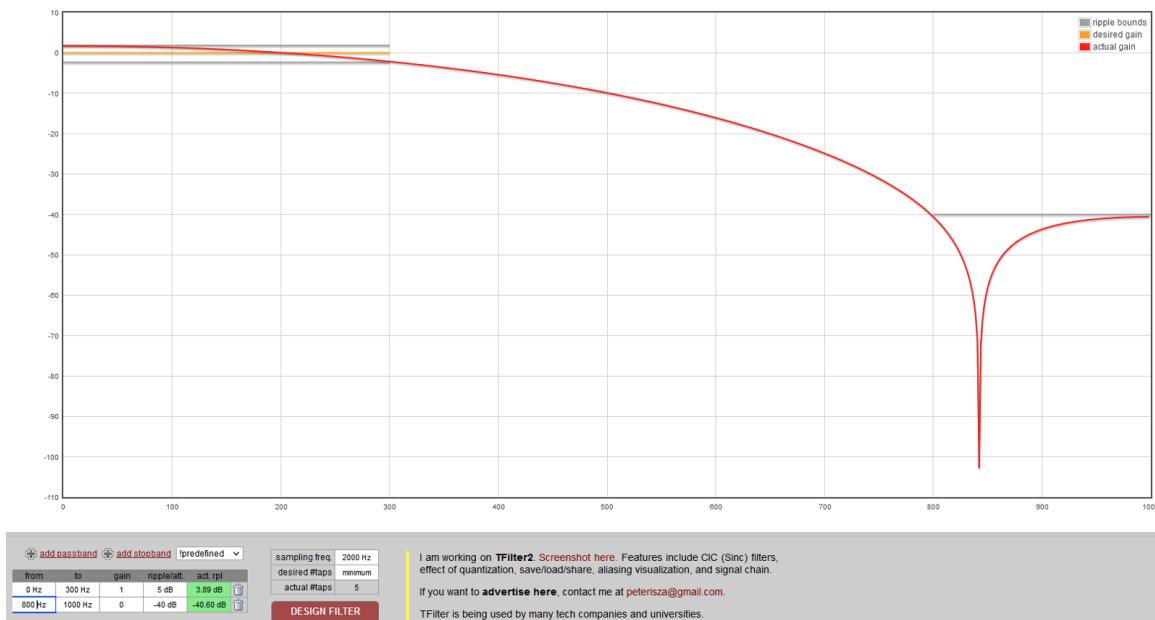


Figure 3.1: Calcul des coefficients d'un filtre numérique à partir d'un gabarit d'un filtre passe-bas

T1. À l'aide du calculateur de filtre en ligne, calculer le nombre les coefficients de votre filtre et préciser le tableau des valeurs à utiliser par la suite créer un fichier C/arduino et ajouter les deux lignes de codes suivantes avec les valeurs obtenues, comme suit :

```
#define FILTER_TAP_NUM ...  
  
static uint16_t filter_taps[FILTER_TAP_NUM] = { ... };
```

A chaque fois que nous vous demandrons de concevoir un filtre avec des spécifications fréquentielles, vous devez passer par un logiciel ou un site pour calculer les coefficients de votre filtre. Vous pouvez toujours utiliser d'autres outils sous python, scilab ou matlab et récupérer les valeurs des gains et l'ordre de votre filtre.

III. Fonctions de filtrage RIF

Le filtrage que nous proposons de réaliser doit respecter les formes canoniques directe de type I & type II, qui opère avec un accumulateur à 16/8 bits (soumis ou non au risque de débordement). Les coefficients du filtre sont des entiers codés sur des entiers signés à 16 bits. Si l'on doit implémenter le même filtrage pour des échantillons codés sur 10 bits, il suffit de décaler les variables déclarées à 16bits de 4bits à droite. Dans la suite, nous vous proposons de réaliser les fonctions de filtrage RIF de manière à utiliser les tampons (buffers) mémoires. A chaque nous vous proposons de matérialiser un algorithme sur vos arduino nano et d'étudier bien quelques propriétés.

NB : N'oubliez pas de cabler votre arduino et microphone Ky037 avant de commencer votre TP.

A. Filtre RIF à Tampon FIFO

Les filtres RIF utilisent l'algorithme FIFO (First Input- First Output) pour la réalisation de filtrage. Lors de l'utilisation du FIFO le FIR n'a besoin de sauvegarder que les échantillons N-1 précédents en mémoire, l'algorithme FIFO mettra à jour les valeurs en mémoire lorsqu'un nouvel échantillon est introduit. Ceci consiste à décaler tous les échantillons du tableau en les copiant à l'emplacement décalé et en stockant la nouvelle valeur au début du tableau. Il s'agit d'un tampon FIFO. La figure ci-dessous

montre un schéma de la mise en œuvre du filtre FIR en utilisant la méthode de décalage des tableaux. Dans la figure ci-dessous, le '*' indique une multiplication. La nouvelle valeur est décalée dans le point `data_in[0]` et toutes les autres sont décalées vers la droite.

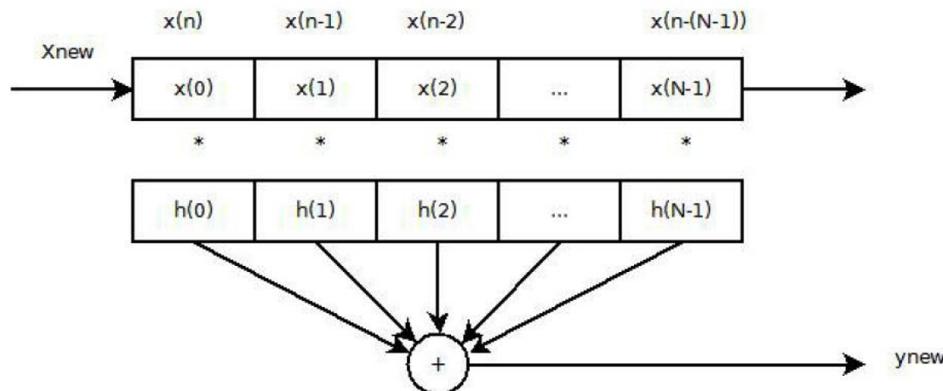


Figure 3.2: Tampon de données après le stockage des valeurs M

Le code ci-dessous montre comment mettre en œuvre un filtre FIR en utilisant la mémoire sous forme d'un tampon FIFO :

```
#define FILTER_TAP_NUM 13
static int16_t filter_taps[FILTER_TAP_NUM] = { 1,2,3,4,5,6,7,6,5,4,3,2,1};
int16_t data_input[FILTER_TAP_NUM];
int16_t output;
char i;

void setup() {
    //Serial setup at 115200 bauds
    Serial.begin(115200); // use the serial port

    // Setup the ADC for polled 10 bit sampling on analog pin 5 at 19.2kHz.
    cli(); // Disable interrupts.
    TIMSK0 = 0; // turn off timer0 for lower jitter
    ADCSRA = 0xe5; // set the adc to free running mode
    ADCSRA |= bit (ADPS1) | bit (ADPS2); // Set ADC clock with 64 prescaler
                                            //where 16mHz/64=250kHz and 250khz/13 instruction cycles = 19.2khz sampling
    ADMUX = 0x40; // use adc0
    DIDR0 = 0x01; // turn off the digital input for adc0
    sei(); // Re-enable interrupts.
}

void loop() {
    cli(); // UDRE interrupt slows this way down on arduino1.0
    while(!(ADCSRA & 0x10)); // wait for adc to be ready
    ADCSRA = 0xf5; // restart adc
    byte m = ADCL; // fetch adc data
    byte j = ADCH;
    int16_t k = (j << 8) | m; // form into an int
    k -= 0x0200; // form into a signed int
    k <= 6; // form into a 16b signed int
    data_input[0] = k; // put real data into even bins

    for(i=FILTER_TAP_NUM-2 ; i>=0 ; i--)
        data_input[i+1] = data_input[i] ;
    // Output Filtered data
    for(i=0;i<FILTER_TAP_NUM;i++)
    {
        output = output + filter_taps[i]*data_input[i] ;
    }
    Serial.println(data_input[0]);
    Serial.println(output);
    sei();
}
```

Q1. Inspirez-vous du code ci-dessus pour écrire un programme c/arduino pour le filtrage d'un signal numérique codé sur 16 bits à l'aide de votre nouveau filtre RIF, la fréquence de coupure du filtre numérique est fixée à 4 kHz et la fréquence d'échantillonnage de votre système de numérisation (Arduino nano) est autour de 19,2 kHz.

Q2. Complétez votre code de manière à tracer la réponse impulsionale sur votre moniteur série.

Q3. Pouvez-vous corriger le résultat de la multiplication $\text{filter_taps}[i]*\text{data_input}[i]$ qui déborde des 16bit proposés, remplacer la ligne de code $\text{output} = \text{output} + \text{filter_taps}[i]*\text{data_input}[i]$; par votre nouvelle ligne de code.

Q4. À l'aide de la fonction `millis()`, mesurer le temps nécessaire pour le filtrage de chaque échantillon, tracer la courbe temps de filtrage/échantillon dans le traceur série. Pouvez-vous améliorer l'algorithme votre fonction de filtrage RIF afin de réduire le temps de filtrage/échantillon.

Q5. Pouvez-vous libérer le processeur de votre arduino nano pour qu'il puisse faire autre chose ?

Q6. À quelle des deux structures de filtre RIF correspond votre algorithme RIF simple FIFO.

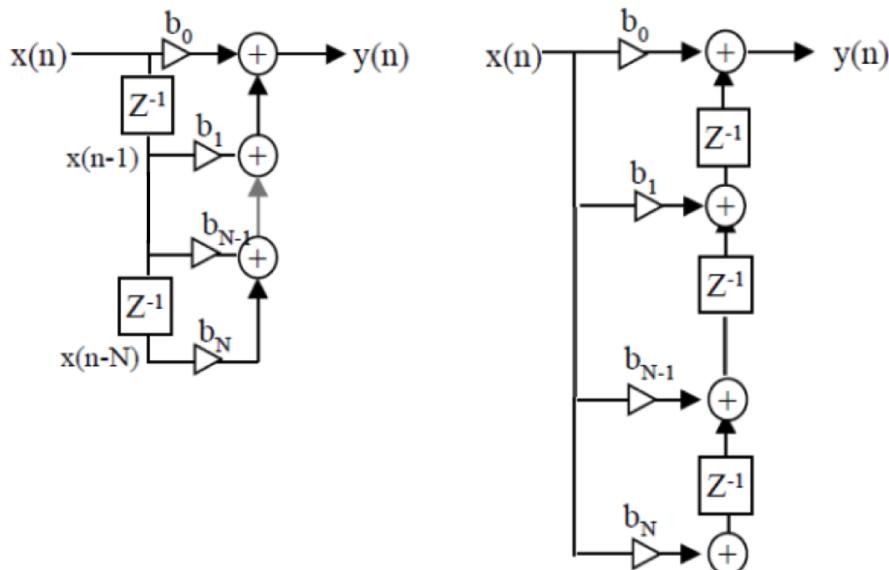


Figure 3.3: Tampon de données après le stockage des valeurs M

B. Filtre RIF à tampon FIFO double

Pour améliorer le rendement de la fonction de filtrage RIF par tampon FIFO est de créer un tampon FIFO deux fois plus grand que les données à stocker. Dans ce cas, le décalage des données peut être effectué après la réception d'un bloc de données. Les figures suivantes montrent le tampon de données au fur et à mesure que les données sont reçues et stockées. La figure ci-dessous montre la mémoire tampon lorsque la première nouvelle valeur est reçue. Bien entendu, la mémoire tampon est mise à zéro avant le début du de l'opération de filtrage. La zone ombrée montre la partie du tampon qui est utilisée pour l'algorithme.

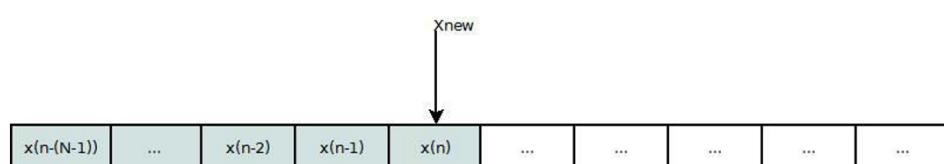


Figure 3.4: Tampon de données au début

La figure ci-dessous montre le tampon après réception de M échantillons. Notez qu'il n'est pas nécessaire de décaler les données dans la mémoire tampon. La zone ombrée est celle qui est utilisée par le filtre.

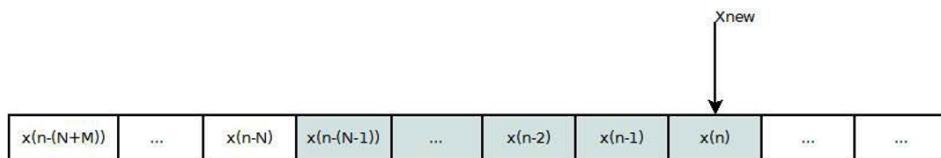


Figure 3.5: Tampon de données après le stockage des valeurs M

La figure ci-dessous montre le tampon lorsque le nouvel échantillon est stocké dans le dernier élément du tampon. À ce stade, le déplacement des données est nécessaire pour supprimer les anciennes données. Les anciennes données se trouvent au début de la mémoire tampon. Ces données sont simplement écrasées par celles qui se trouvent à la fin de la mémoire tampon.

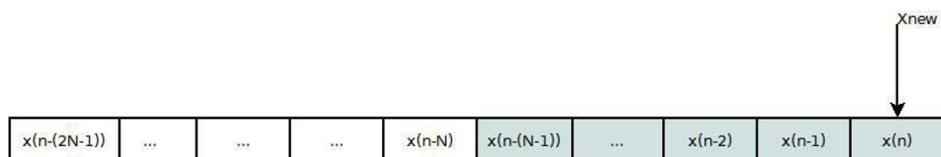


Figure 3.6: Tampon de données lorsque l'indice atteint la fin du tampon

La figure ?? montre la mémoire tampon après la copie des données. L'algorithme va maintenant utiliser les données au début de la mémoire tampon comme il l'a fait dans la figure ???. Remarquez qu'un grand bloc de données est copié en une seule fois.



Figure 3.7: Tampon de données montrant les données qui sont copiées de la fin au début

Le code c/arduino suivant montre comment mettre en œuvre le filtre FIR en utilisant le tampon de type FIFO à double taille.

```
#define FILTER_TAP_NUM 13
static int16_t filter_taps[FILTER_TAP_NUM] = { 1,2,3,4,5,6,7,6,5,4,3,2,1};
int16_t data_input[2*FILTER_TAP_NUM];
int16_t output;
uint16_t n, i=0;

void setup() {
    //Serial setup at 115200 bauds
    Serial.begin(115200); // use the serial port
    // Setup the ADC for polled 10 bit sampling on analog pin 5 at 19.2kHz.
    cli(); // Disable interrupts.
    TIMSK0 = 0; // turn off timer0 for lower jitter
    ADCSRA = 0xe5; // set the adc to free running mode
    ADCSRA |= bit (ADPS1) | bit (ADPS2); // Set ADC clock with 64 prescaler
    //where 16mHz/64=250kHz and 250khz/13 instruction cycles = 19.2khz sampling
    ADMUX = 0x40; // use adc0
    DIDR0 = 0x01; // turn off the digital input for adc0
    sei(); // Re-enable interrupts.
}
```

```

void loop() {
    cli(); // UDRE interrupt slows this way down on arduino1.0
    while(1)
    {
        while(!(ADCSRA & 0x10)); // wait for adc to be ready
        ADCSRA = 0xf5; // restart adc
        byte m = ADCL; // fetch adc data
        byte j = ADCH;
        int16_t k = (j << 8) | m; // form into an int
        k -= 0x0200; // form into a signed int
        k <= 6; // form into a 16b signed int
        data_input[i] = k; // put real data into even bins

        // Filtrer les données output = 0 ;
        for(n=0;n<FILTER_TAP_NUM;n++)
            output = output + filter_taps[i]*data_input [i-n] ;
        i++;
        if(i>=2*FILTER_TAP_NUM){
            for(n=FILTER_TAP_NUM-2 ; n>=0 ; n--)
                data_input[n+FILTER_TAP_NUM+1] = data_input[n] ;
            i = FILTER_TAP_NUM - 1 ;
            break;
        }
        Serial.println(data_input[0]);
        Serial.println(output);
    }
    sei();
}

```

Q7. A partir de votre code c/arduino déduire l'algorigramme de votre filtre RIF double tampon ?

Q8. Modifiez le tableau filter_taps[] FILTER_TAP_NUM pour votre filtre correspond à un filtre RIF avec une fréquence de coupure 5 kHz et une largeur de bande transition de 2 kHz. La fréquence d'échantillonnage de votre système de numérisation (Arduino nano) est toujours maintenue à 19,2 kHz.

Q9. Complétez votre code (avec les mêmes données fréquentielles de Q8) de manière à tracer la réponse impulsionale sur votre moniteur série. Comparer la réponse impulsionale du RIF double FIFO avec celle du RIF simple FIFO, quelle différence entre les deux ?

Q10. Si l'index i-n n'est plus positif, comment devrez-vous modifier l'accumulation output dans votre programme ?

Q11. Pouvez-vous améliorer l'algorithme votre fonction de filtrage RIF afin d'économiser la mémoire utilisée par le double tampon ?

C. Filtre RIF à tampon circulaire

Pour créer un tampon circulaire pour filtre RIF, vous devez conserver un index qui indique où la dernière entrée a été stockée. Cet index est incrémenté et utilisé pour stocker la nouvelle valeur. Lorsque l'index atteint la fin du tableau, il est enroulé autour du début du tableau. Cet algorithme est utilisé pour les filtres FIR :

$$y[k] = \sum_{k=0}^{N-1} h[k]x[(\text{index} - k)]_N \quad (3.5)$$

où $[...]$ _N correspond à un modulo-N. Cela suppose que l'index est incrémenté au fur et à mesure que de nouvelles données sont stockées dans la mémoire tampon. Lors de l'implémentation du filtre FIR en utilisant un tampon circulaire, notez que les indices du tampon de coefficient allent en avant et les index de la mémoire tampon des données en arrière.

```

#define FILTER_TAP_NUM 13
static int16_t filter_taps[FILTER_TAP_NUM] = { 1,2,3,4,5,6,7,6,5,4,3,2,1};
int16_t data_input[2*FILTER_TAP_NUM];
int16_t output;
uint16_t n,l, i=0;

void setup() {
    //Serial setup at 115200 bauds
    Serial.begin(115200); // use the serial port
    // Setup the ADC for polled 10 bit sampling on analog pin 5 at 19.2kHz.
    cli(); // Disable interrupts.
    TIMSK0 = 0; // turn off timer0 for lower jitter
    ADCSRA = 0xe5; // set the adc to free running mode
    ADCSRA |= bit (ADPS1) | bit (ADPS2); // Set ADC clock with 64 prescaler
    //where 16MHz/64=250kHz and 250khz/13 instruction cycles = 19.2khz sampling.
    ADMUX = 0x40; // use adc0
    DIDR0 = 0x01; // turn off the digital input for adc0
    sei(); // Re-enable interrupts.
}

void loop() {

    cli(); // UDRE interrupt slows this way down on arduino1.0
    while(1)
    {
        while(!(ADCSRA & 0x10)); // wait for adc to be ready
        ADCSRA = 0xf5; // restart adc
        byte m = ADCL; // fetch adc data
        byte j = ADCH;
        int16_t k = (j << 8) | m; // form into an int
        k -= 0x0200; // form into a signed int
        k <= 6; // form into a 16b signed int
        data_input[i]= k;// put real data into even bins
        // Filter the data
        output = 0;
        for(l=0;l<FILTER_TAP_NUM;l++){
            if((i-l)<0){
                output = output + filter_taps[l]*data_input[i-l+FILTER_TAP_NUM];
            }
            else {
                output = output + filter_taps[l]*data_input[i-l];
            }
        }
        i = (i+1)%FILTER_TAP_NUM;
        Serial.println(data_input[0]);
        Serial.println(output);
    }
    sei();
}

```

Q12. A partir de votre code c/arduino déduire l'algorigramme de votre filtre RIF à tampon circulaire ?

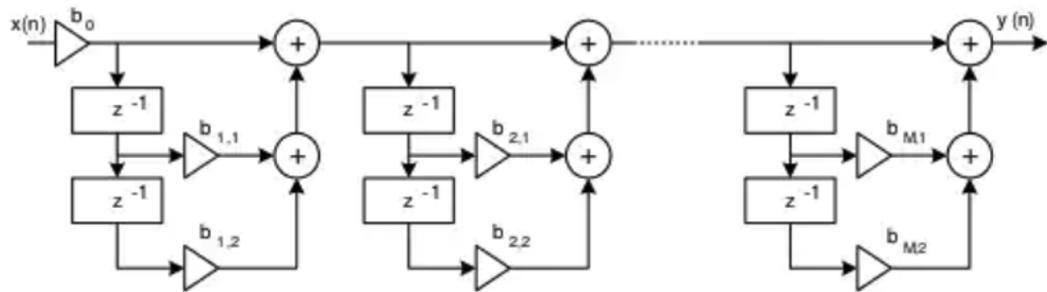
Q13. Modifiez le tableau filter_taps[] FILTER_TAP_NUM pour votre filtre correspond à un filtre RIF avec une fréquence de coupure 9Khz et une largeur de bande transition de 1Khz. La fréquence d'échantillonnage de votre système de numérisation (Arduino nano) est toujours maintenue à 19,2Khz.

Q14. Complétez votre code (avec les mêmes données fréquentielles de Q8) de manière à tracer la réponse impulsionale sur votre moniteur série. Comparer la réponse impulsionale du RIF tampon circulaire avec celle du RIF double tampon FIFO, quelle différence entre les deux ?

Q15. Pouvez-vous améliorer l'algorithme votre fonction de filtrage RIF afin d'économiser la mémoire utilisée par le double tampon.

D. Filtres RIF en cascade

Dans cette partie, nous vous proposons de réaliser une fonction de filtrage par la mise en cascade de 3 filtres RIF. Le premier filtre RIF à gauche à une fréquence de coupure 4 kHz, le filtre RIF du milieu à une fréquence de coupure de 2 kHz et le dernier filtre RIF à une fréquence de coupure de 1 kHz.



Q16. Donner une description de tout le processus de conception de filtre.

Q17. Un écrire un code c/arduino pour réaliser cette fonction de filtrage.

TP n° 4

Filtrage RII

Motivation

Dans ce TP, nous allons faire le filtrage d'un signal numérisé en $Q=8$ bits par échantillons. Les calculs seront faits uniquement en flottants 32 bits. Le filtrage en flottants donnera les meilleurs résultats en terme de qualité de filtrage, mais il sera parfois nécessaire de calculer avec des entiers pour obtenir une vitesse de traitement suffisante, surtout pour les grandes fréquences d'échantillonnage et les filtres RII à grand nombre de coefficients. Les réalisations de filtrage implémentées sont les réalisations directes de type I et II et avec un nombre réduit de coefficients. Dans une première étape, nous rappelons qu'il est toujours nécessaire de configurer le convertisseur analogique numérique et d'utiliser les timers pour définir les périodes d'échantillonnage.

I. Acquisition de signaux analogiques

A. Configuration du convertisseur et des interruptions

La première fonction configure et déclenche le Timer1 pour une période d'échantillonnage **period** :

(a)

```
void timer1_init(uint32_t period)
{
    TCCR1A = 0; //
    TCCR1B = 0; //
    TCCR1B |= (1 << WGM12);
    uint32_t top =(F_CPU/1000000*period);
    int clock = 1;//
    while ((top>0xFFFF)&&(clock<5))
    {
        clock++;//
        top = (F_CPU/1000000*period/diviseur[clock]);
    }
    OCR1A = top;// 
    OCR1B = top >> 1;// 
    TIMSK1 = (1 << OCIE1B);//
    TCCR1B |= clock;//
}
```

(b)

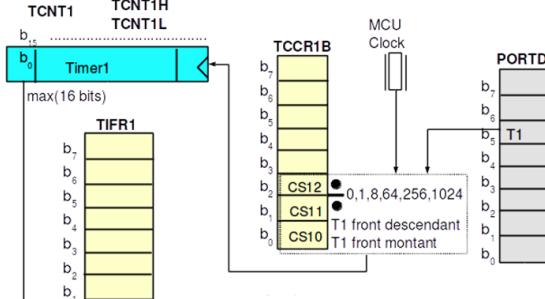


Figure 4.1: (a) Code permettant de configurer le Timer1 et (b) Vue synoptique de Timer1.

Q1. Rappelez l'utilité d'utiliser un timer d'arduino pour la fonction d'échantillonnage ?

Q2. Combien de fois peut-on utiliser la fonction timer1_init dans un programme de filtrage ?

Q3. Proposez une valeur de prescaler pour aider la fonction timer1_init ?

Le timer1 et l'acquisition s'arrêteront en même temps que les trois bits de registre TCCR1B sont mis à jour avec le code 0x7 :

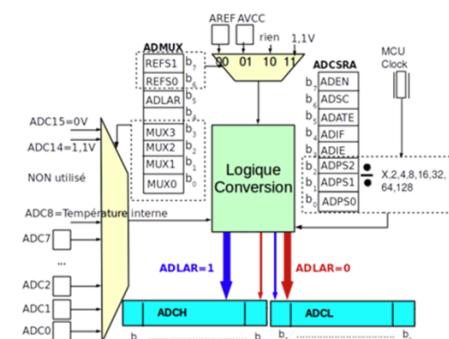
```
void stop_acquisition()
{
    TCCR1B &= ~0x7;
}
```

L'ADC est initialisé avec le code de configuration du multiplexeur et la fonction de traitement du signal qui sera appelée pour chaque échantillon obtenu. Pour une entrée simple, le code de multiplexage est simplement le numéro de l'entrée.

(a)

```
void adc_init(uint8_t multiplex, uint8_t prescaler,void (*isr)())
{
    ADMUX = (1 << REFS0);
    ADMUX |= multiplex & 0b00001111;
    ADMUX |= (1 << ADLAR); //left adjust
    ADCSRA = 0;
    ADCSRA |= (1 << ADEN); // enable ADC
    ADCSRA |= (1 << ADATE); // Auto Trigger Enable
    ADCSRA |= (1 << ADIE); // interrupt enable
    ADCSRA |= prescaler ;
    ADCSRB = 0;
    ADCSRB |= (multiplex & 0b00010000) >> 2; // trigger source : timer 1 comp B
    ADCSRB |= (1 << ADTS2)|(1 << ADTS0);
    isrCallback = isr;
}
```

(b)



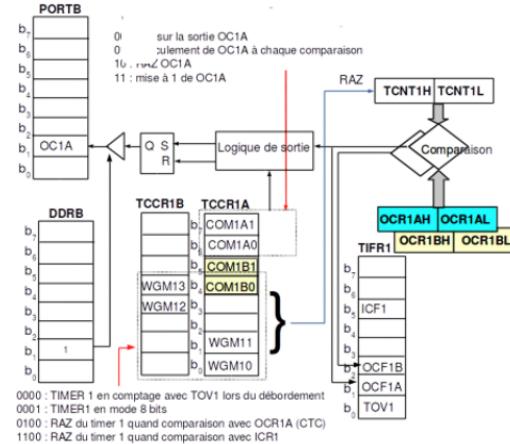
Q4. Expliquer pourquoi nous avons du modifier l'instruction `ADMUX |= multiplex & 0b00001111;`

Le timer1 déclenche aussi une interruption synchrone avec l'échantillonnage. La fonction ISR pourra être ajoutée pour alterner le niveau de la sortie 3.

(a)

```
ISR(TIMER1_COMPB_vect)
{
    if (flag==0)
    {
        flag = 1;
        PORTB &= ~(1<<PORTB3); // sortie numérique 3
    }
    else {
        flag = 0;
        PORTB |= (1<<PORTB3);
    }
}
```

(b)



Q5. Modifier la sortie PORTD3, donner le nouveau code arduino.

B. Acquisition de filtrage à RII

La fonction de filtrage à RII est appelée par interruption, juste après la fin de l'opération réalisée par l'ADC. Le filtrage que nous proposons de réaliser doit respecter les formes canoniques directe de type I & type II, qui opère avec un accumulateur à 16 bits. Le filtrage en 32 bits est moins rapide que le filtrage 16 bits, mais il permet d'encoder les coefficients du filtre avec plus de bits, ce qui réduit les erreurs d'arrondis.

Pour la réalisation directe de type I, le filtrage en 32 bits s'impose pour éviter les débordements de l'accumulateur. Nous vous proposons ci-dessous le code pour le filtrage directe de type I. La différence avec le cas 16 bits est l'utilisation de coefficients et d'un accumulateur 32 bits au lieu de 16 bits. Les tampons sont toujours en 8 bits, car la numérisation de fait en 8 bits.

Pour une numérisation en plus que 8 bits, il faudra définir des tampons 16 bits. Dans cette réalisation, il ne se produit pas de débordement et on peut donc choisir 32-8=24 bits pour les coefficients, ce qui permet de réaliser des filtrages très fins. La réalisation directe de type II est souvent soumise au débordement, donc il faut réduire le nombre de bits pour les coefficients, mais la marge est beaucoup plus grande que dans le cas 16 bits. On peut par exemple choisir 20 bits pour les coefficients, ce qui laisse 4 bits de sécurité pour éviter les débordements.

(a)

```
void adc_filtage_32bits_FD1() {
    int32_t accum;
    int16_t sortie;
    uint8_t i,k;
    x[J_tampon] = ADCH-offset;
    accum = 0;
    i = J_tampon;
    for (k=0; k<nb; k++) {
        accum += x[i]*bb[k];
        i--;
    }
    i = J_tampon-1;
    for (k=1; k<na; k++) {
        accum -= y[i]*aa[k];
        i--;
    }
    sortie = accum >> ab_shift;
    y[J_tampon] = sortie;
    J_tampon++;
    if (--i_reduc == 0) {
        i_reduc = reduc;
        buf[compteur_buf][indice_buf] = sortie+offset;
        indice_buf++;
        if (indice_buf == BUF_SIZE) {
            indice_buf = 0;
            compteur_buf = (compteur_buf+1)&NBUF_MASK;
        }
    }
}
```

(b)

```
void adc_filtage_32bits_FD2() {
    int32_t accum;
    int16_t sortie;
    uint8_t i,k;
    accum = (ADCH-offset)*aa[0];
    i = J_tampon-1;
    for (k=1; k<na; k++) {
        accum -= ww[i]*aa[k];
        i--;
    }
    ww[J_tampon] = accum >> ab_shift;
    i = J_tampon;
    accum = 0;
    for (k=0; k<nb; k++) {
        accum += ww[i]*bb[k];
        i--;
    }
    J_tampon++;
    sortie = accum >> ab_shift;
    if (--i_reduc == 0) {
        i_reduc = reduc;
        buf[compteur_buf][indice_buf] = sortie+offset;
        indice_buf++;
        if (indice_buf == BUF_SIZE) {
            indice_buf = 0;
            compteur_buf = (compteur_buf+1)&NBUF_MASK;
        }
    }
}
```

Q6. Expliquer le fonctionnement des fonctions de filtrage adc_filtage_32bits_FD1/ et adc_filtage_32bits_FD2.

C. Fonction de filtrage à RII

Pour initialiser la fonction de filtrage, nous vous proposons d'initialiser les variables dans la fonction setup. Le port D3 est configuré en sortie.

```
void setup() {
    char c;
    Serial.begin(115200);
    Serial.setTimeout(0);
    c = 0;
    Serial.write(c);
    c = 255;
    Serial.write(c);
    c = 0;
    Serial.write(c);
    compteur_buf = compteur_buf_transmis = 0;
    indice_buf = 0;
    DDRB |= 1 << PORTD3; // PB3 en sortie
    flag = 0;
}
```

La fonction suivante permet de configurer et de déclencher une acquisition sans filtrage. Les informations envoyées par l'ordinateur sont :

- Le code du multiplexeur (8 bits).
- Le code du facteur diviseur de l'horloge de l'ADC (8 bits).
- La période d'échantillonnage en microsecondes (16 bits).
- Le nombre de blocs de 64 échantillons à transmettre (32 bits). Si cette valeur est nulle, la transmission se fait sans fin.

- Le facteur de réduction d'échantillonnage pour les données à transmettre (8 bits).

L'appel de cli() désactive les interruptions, l'appel sei() active les interruptions.

```

void lecture_acquisition() {
    uint32_t c1,c2,c3,c4;
    uint8_t multiplex,prescaler;
    uint32_t period;
    while (Serial.available()<2) {};
    multiplex = Serial.read();
    prescaler = Serial.read();
    while (Serial.available()<4) {};
    c1 = Serial.read();
    c2 = Serial.read();
    c3 = Serial.read();
    c4 = Serial.read();
    period = ((c1 << 24) | (c2 << 16) | (c3 << 8) | c4);
    while (Serial.available()<4) {};
    c1 = Serial.read();
    c2 = Serial.read();
    c3 = Serial.read();
    c4 = Serial.read();
    nblocs = ((c1 << 24) | (c2 << 16) | (c3 << 8) | c4);
    while (Serial.available()<1);
    reduc = Serial.read();
    i_reduc = reduc;
    if (nblocs==0)
    {
        sans_fin = 1;
        nblocs = 1;
    }
    else sans_fin = 0;
    compteur_buf=compteur_buf_transmis=0;
    indice_buf = 0;
    cli();
    timer1_init(period);
    adc_init(multiplex,prescaler,adc);
    sei();
}

```

Pour amorcer configurer et déclenche l'acquisition avec un filtrage en 16 bits. La fonction lecture_filtrage_32bits() (voir le code arduino) doit fournir :

- Le nombre de coefficients a_k du filtre.
- Les coefficients a_k .
- Le nombre de coefficients b_k du filtre.
- Les coefficients b_k .
- Le décalage s appliqué aux coefficients.
- Le décalage à appliquer aux valeurs du signal avant le filtrage, correspondant au niveau 0 du signal.
- Le type de filtrage 1 = forme directe I, 2 = forme directe II

Q7. Quel différence entre le filtrage en 32bit type I et type II ? Le processus de filtrage est organisé par la fonction lecture_serie(), cette fonction détermine la tâche à réaliser par la saisie d'un caractère de commande (les codes sont définis dans le fichier example). La fonction de configuration correspondante est appelée.

La fonction loop transmet un bloc d'échantillons à l'ordinateur, lorsqu'il y a en un disponible, ou lit le port série. La taille des blocs est définie en entête (BUF_SIZE).

```

void lecture_serie() {
    char com;
    if (Serial.available()>0) {
        com = Serial.read();
        if (com==SET_ACQUISITION) lecture_acquisition();
        if (com==STOP_ACQUISITION) stop_acquisition();
        if (com==SET_FILTRAGE_32BITS) lecture_filtage_16bits();
    }
}

void loop() {
    if ((compteur_buf_transmis!=compteur_buf)&&(nblocs>0)) {
        Serial.write((uint8_t *)buf[compteur_buf_transmis],BUF_SIZE);
        compteur_buf_transmis=(compteur_buf_transmis+1)&NBUF_MASK;
        if (sans_fin==0) {
            nblocs--;
            if (nblocs==0) stop_acquisition();
        }
    }
    else lecture_serie();
}

```

Q8. Charger le code arduino à l'aide de votre éditeur, vérifier bien que votre Arduino est bien connecté au port COM de numéro le plus élevé, téléverser votre premier code sur votre Arduino.

II. Filtre de Butterworth d'ordre 3

Nous proposons à présent de réaliser un filtre numérique RII de type passe-bas à partir d'un modèle de filtre d butterworth d'ordre 3 avec les spécifications suivantes:

- $f_s = 8$ [kHz]
- Fonction de transfert du filtre analogique équivalente :

$$H(s) = \frac{Y(s)}{U(s)} = \frac{1}{1 + 1 \cdot \frac{s}{\omega_c} + 1 \cdot \left(\frac{s}{\omega_c}\right)^2} \quad (4.1)$$

- Fonction de transfert $H(z)$ après transformation bilinéaire est de la forme :

$$H(z) = \frac{0,239 \cdot 1 + z^{-1}}{1 - 0,5219 \cdot z^{-1}} \cdot \frac{0,06986 \cdot 1 + 2z^{-1} + z^{-2}}{1 - 1,2759 \cdot z^{-1} + 0,5553 \cdot z^{-2}} = \frac{0,239 \cdot (z + 1)}{z - 0,5219} \cdot \frac{0,06986 \cdot z^2 + 2z + 1}{z^2 - 1,2759 \cdot z + 0,5553} \quad (4.2)$$

Q9. Déterminer les coefficients du filtre RII.

Q10. Donner l'ordre du filtre RII

Q11. Télécharger le code arduino de la plateforme moodle et modifier les sections nécessaire et supprimer les codes unutiles pour la réalisation de votre filtrage

Q12. Introduisez les paramètres de votre filtre dans le code arduino

Q13. Modifier votre code arduino pour afficher les valeurs d'entrée sur terminal