TP microcontrôleur 16 bits : filtrage de signaux numériques

J.-M Friedt, 17 novembre 2009

Objectif de ce TP :
- exploiter les fonctions de traitement du signal de GNU/Octave pour concevoir un filtre
- implémenter ce filtre dans un microcontrôleur 16 bits
- valider expérimentalement le fonctionnement du filtre

1 Conception d'un filtre de réponse finie (FIR)

Le sujet de ce TP n'est pas de développer la théorie des filtres numériques mais uniquement de rappeler quelques uns des concepts utiles en pratique :
- un filtre de réponse impulsionnelle finie [1] (Finite Impulse Response, FIR) b_m est caractérisé par une sortie y_n à un instant donné n qui ne dépend que des valeurs actuelle et passées de la mesure x_m, $m \leq n$:

$$y_n = \sum_{k=0..m} b_k x_{n-k}$$

  L'application du filtre s'obtient pas convolution des coefficients du filtre et des mesures.
- un filtre de réponse impulsionnelle infinie (IIR) a une sortie qui dépend non seulement des mesures mais aussi des valeurs passées du filtre.
- GNU/Octave (et Matlab) proposent des outils de synthèse des filtres FIR et IIR. Par soucis de stabilité numérique et simplicité d'implémentation, nous ne nous intéresserons qu'aux FIR. Dans ce cas, les coefficient de récursion sur les valeurs passées du filtre (nommées en général a_m dans la littérature) seront égaux à $a = 1$.

Le calcul d'un filtre dont la réponse spectrale ressemble à une forme imposée par l'utilisateur (suivant un critère des moindres carrés) est obtenu par la fonction `firls()`. Cette fonction prend en argument l'ordre du filtre (nombre de coefficients), la liste des fréquences définissant le gabarit du filtre et les magnitudes associées. Elle renvoie la liste des coefficients `b`.

L'application d'un FIR de coefficients `b` (avec `a=1`) ou de façon plus générale d'un IIR de coefficients `a` et `b` sur un vecteur de données expérimentales `x` s'obtient par `filter(b,a,x) ;`

Tout filtre se définit en terme de fréquence normalisée : la fréquence 1 correspond à la demi-fréquence d'échantillonnage (fréquence de Nyquist) lors de la mesure des données.

Exercices :

1. en supposant que nous échantillonnons les données sur un convertisseur analogique-numérique à 16000 Hz, identifier les coefficients d'un filtre passe-bande entre 700 et 900 Hz. Combien faut-il de coefficients pour obtenir un résultat satisfaisant ?
2. Quelle est la conséquence de choisir trop de coefficients ? Quelle est la conséquence de choisir trop peu de coefficients ?
3. ayant obtenu la réponse impulsionnelle du filtre, quelle est sa réponse spectrale ?
4. valider sur des données (boucle sur les fréquences) simulées la réponse spectrale du filtre

Au lieu de boucler sur des fréquences et observer la réponse du filtre appliqué à un signal monochromatique, le `chirp` est un signal dont la fréquence instantanée évolue avec le temps. Ainsi, la fonction

```
chirp([0 :1/16000 :5],40,5,8000) ;
```

génère un signal échantillonné à 16 kHz, pendant 5 secondes, balayant la gamme de fréquences allant de 40 Hz à 8 kHz.

Exercices :

1. appliquer le filtre identifié auparavant au chirp, et visualiser le signal issu de cette transformation
2. que se passe-t-il si la fréquence de fin du chirp dépasse 8 kHz dans cet exemple ?

2 Application pratique du filtre

Exercices : Implémenter sur MSP430

1. sachant que les acquisitions se font sur 12 bits, quelle est la plage de valeur sur laquelle se font les mesures ?
2. comment exploiter les coefficients numériques du filtre pour un calcul sur des entiers sur 16 bits (short) ? sur 32 bits (long) ?

1. l'échantillonnage périodique de mesures analogiques acquises sur ADC0
2. l'affichage du résultat de ces mesures
3. le filtrage de ces mesures par un FIR synthétisé selon la méthode vue plus haut
4. l'affichage du résultat de ce filtrage
5. l'affichage de l'amplitude du signal résultant du filtrage
6. appliquer ce programme à une mesure sur un chirp généré par la carte son du PC au moyen de `audacity`

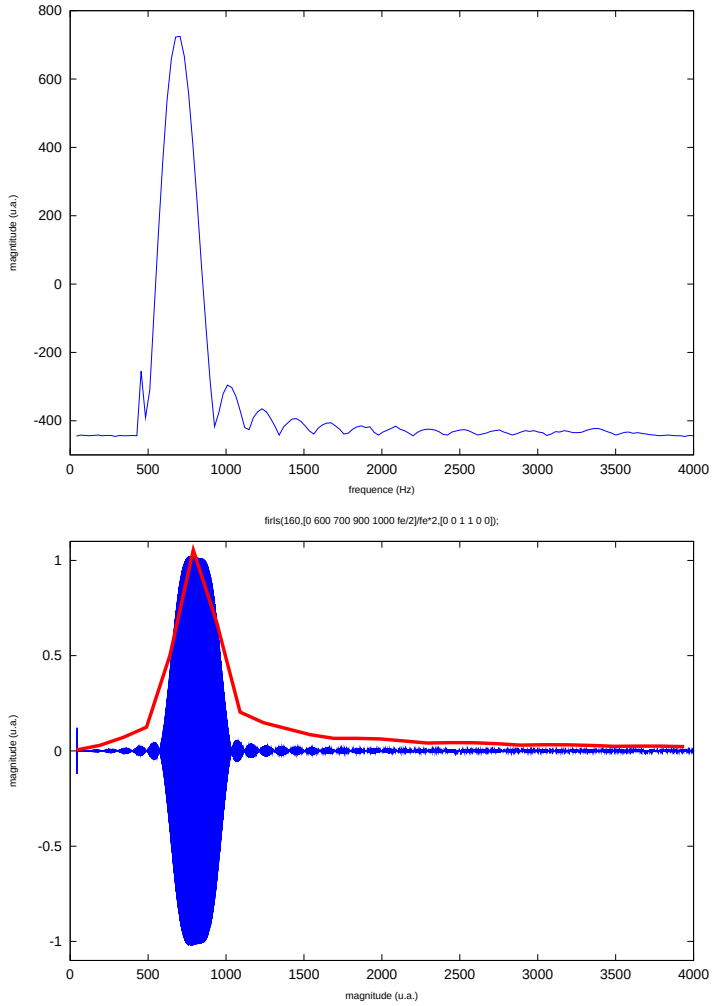
7. comparer l'expérience et la simulation.

Exemple de programme et résultats :

```
#include  <msp430x14x.h>
#include  <string.h>
#include  <signal.h>
#include <io.h>
#include <stdio.h>

#undef debug

void Sommeil(void);
void Pause_TIMERB(unsigned short Compteur);
void Pause_ms(unsigned short ms);
interrupt(TIMERB0_VECTOR ) wakeup Vector_TIMERB0(void);
void init_horloge(void);
void init_PORT(void);
void init_TIMER(void);

void init_rs1_9600(){
 UCTL1=SWRST;    // SetupUART1: mov.b #SWRST,&UCTL1   ; cf p.13-4
 UCTL1|=CHAR; //   bis.b #CHAR,&UCTL1
 UTCTL1=SSEL0;   //   mov.b #SSEL0,&UTCTL1   ; 32768 Hz ACLK
 UBR01=0x03;   //   mov.b #0x03,&UBR01   ; 4800 bauds
 UBR11=0x00;   //   mov.b #0x00,&UBR11
 UMCTL1=0x4A; //   mov.b #0x4A,&UMCTL1
 UCTL1&=~SWRST; //   bic.b #SWRST,&UCTL1
 ME2|=UTXE1+URXE1;   //   bis.b #UTXE1+URXE1,&ME2
}

void snd_rs1(unsigned char cara)
{while ((IFG2 & UTXIFG1)==0); // jz rs_tx1
 TXBUF1=cara;
}

void str_rs1(unsigned char* cara)
{int k=0;
 do {snd_rs1(cara[k]);k++;} while (cara[k]!=0);
}

void init_PORT(void){
  P1SEL = 0x00;
  P1DIR = 0xFF;
  P1OUT = 0x00;

  P3SEL=0xFE; //   mov.b #0xF0,&P3SEL   ; P3.6,7 = USART select
  P3DIR=0x5B; //   mov.b #0x50,&P3DIR   ; P3.6 = output direction
  P3OUT=0x01; //   mov.b #0x00,&P3OUT   ; P3.6 = output val
}

void init_ADC(void){
  ADC12CTL0=SHT0_6+REFON+ADC12ON; // V_REF=1.5 V
  ADC12CTL1=SHP;                  // mov.w    #SHP,&ADC12CTL1
  ADC12MCTL0=INCH_0+SREF_1;       // mov.b    #INCH_10+SREF_1 &ADC12MCTL0
  ADC12CTL0|=ENC;                 // bis.w    #ENC,&ADC12CTL0
}

unsigned short temperature()
{ADC12CTL0|=ADC12SC; //   start conversion
 do {} while ((ADC12CTL1 & 1)!=0);
 return(ADC12MEM0);
}

interrupt(TIMERA0_VECTOR ) wakeup Vector_TIMERA0(void)
{ }

void init_horloge(void){
  BCSCTL1 = 0x87;
  BCSCTL2 = SELM1 + SELS;
  WDTCTL = WDTPW + WDTHOLD;
}

long filtre(unsigned short *in,long *fil,long nin,long nfil,long *sortie)
{unsigned short k,j;long s=0,max=-2500;
 if (nin>nfil)
  {for (k=0;k<nin-nfil;k++)
    {s=0;
     for (j=0;j<nfil;j++)
         s+=((((long)fil[j]*(long)in[nfil-1+k-j])))/256;
     s=s/256;
```

```
      sortie[k]=s;
      if (max<s) max=s;
     }
   }
   return(max);
}

void init_TIMER(void){
 TACTL=TASSEL0 + TACLR;
 TACCTL0|=CCIE; // CCR0 interrupt enabled
 TACCR0=1;
 TACTL|= MC_0;
}

#define NB 110
#define NBFIL 61

int main (void)
{
 char chaine[30];
 unsigned short tableau[NB],k=0;
 long sortie[NB],max;

// a=(round(firls(60,[0 500 700 900 1000 32768/16]/32768*16,[0 0 1 1 0 0])*65536)')
/*long fil200[NBFIL]={-284,-343,193,700,362,-457,-665,-167,108,8,390,1092,\
     557,-1361,-2181,-294,2040,1797,-110,-628,-77,-1315,-3220,-821,\
    5539,7177,-1131,-10626,-8081,5196,12938,5196,-8081,-10626,-1131,7177,\
    5539,-821,-3220,-1315,-77,-628,-110,1797,2040,-294,-2181,-1361,\
    557,1092,390,8,108,-167,-665,-457,362,700,193,-343,-284};
*/

// a=(round(firls(60,[0 500 700 900 1000 32768/2]/32768*2,[0 0 1 1 0 0])*65536)')
long fil200[NBFIL]={-384,-543,-695,-836,-964,-1074,-1164,-1229,-1270,-1283, \
 -1268,-1224,-1153,-1054,-930,-783,-616,-433,-237,-34,173,379,578,767,941,\
 1096,1228,1334,1411,1459,1475,1459,1411,1334,1228,1096,941,767,578,379,173,\
 -34,-237,-433,-616,-783,-930,-1054,-1153,-1224,-1268,-1283,-1270,-1229,-1164,\
 -1074,-964,-836,-695,-543,-384};

/* freq=[50:50:3000];
   k=1;
   for f=freq
     x=sin(2*pi*f*[0:8/32768:0.1]);
     y=filter(a,1,x);
     mm(k)=max(y(100:200));k=k+1;
   end
   plot(freq,mm)
*/

 init_horloge();
 init_PORT();
 init_ADC();
 init_rs1_9600();
 init_TIMER();
 eint();

 TACTL|= MC_1 + TACLR;
 sprintf(chaine,"%%long:%d int:%d short:%d\r\n", \
    sizeof(long),sizeof(int),sizeof(short));
 str_rs1(chaine);

 while (1) {
   if(P4IN==0x04)
     {for (k=0;k<NB;k++)
       {LPM3;
        tableau[k]=temperature();
        P1OUT^=0xff;
       }
#ifdef debug
       for (k=0;k<NB;k++)
        {sprintf(chaine,"%d %d\r\n",k,tableau[k]);
         str_rs1(chaine);
        }
#endif
      max=filtre(tableau,fil200,NB,NBFIL,sortie);
#ifdef debug
      for (k=0;k<NB-NBFIL;k++)
       {sprintf(chaine,"%d %ld\r\n",k,sortie[k]);
        str_rs1(chaine);
       }
```

```
#endif
      sprintf(chaine,"%% %ld\r\n",max);
      str_rs1(chaine);
     }
   else {P1OUT=0x00;}
 }
 return(0);
}
```




```
fe=16000;
ffin=4000;
fdeb=40;

b=firls(160,[0 600 700 900 1000 fe/2]/fe*2,[0 0 1 1 0 0]);
x=chirp([0:1/fe:5],fdeb,5,ffin);
f=linspace(fdeb,ffin,length(x));
plot(f,filter(b,1,x));

freq=[fdeb:150:ffin];
k=1;
for f=freq
  x=sin(2*pi*f*[0:1/fe:1]);
  y=filter(b,1,x);
  sortie(k)=max(y);
  k=k+1;
end
hold on;plot(freq,sortie,'r')
xlabel('frequence (Hz)')
ylabel('magnitude (u.a.)')
```

En haut à gauche : mesure expérimentale de la réponse d'un FIR passe bande conçu pour être passant entre 700 et 900 Hz. La mesure est effectuée au moyen d'un chirp de 40 à 4000 Hz en 30 secondes généré par une carte son d'ordinateur. En bas à gauche : modélisation de la réponse d'un filtre passe bande entre 700 et 900 Hz, en bleu par filtrage d'un chirp, en rouge par calcul de l'effet du filtre sur quelques signaux monochromatiques. Ci-dessus : programme GNU/Octave de la simulation.

3 Conception d'un filtre de réponse infinie (IIR)

GNU/Octave propose les fonctions `cheb` et `butter` pour concevoir des filtres IIR. Une fois les coefficients b et a (cette fois a est un vecteur) l'IIR se tient compte non seulement des dernières observations mais aussi des valeurs passées du filtre :

$$y_n = \sum_{k=0..m} b_k x_{n-k} - \sum_{k=1..m} a_k y_{n-k}$$

Exercice :
Exploiter ces fonctions pour concevoir un filtre aux performances comparables aux IIR proposés ci-dessus, et implémenter ces filtres dans le MSP430 pour en valider le fonctionnement.

4 Expression explicite dans le domaine fréquentiel

L'efficacité des filtres vus auparavant est leur application dans le domaine temporel : une fois les coefficients a et b identifiés, l'application du filtre consiste en N multiplication pour un filtre d'ordre N.

Le passage dans le domaine fréquentiel par transformée de Fourier nécessite un calcul considérablement plus long. L'approche naïve de la transformée de Fourier nécessite N multiplications, puisque chacun des N termes en sortie nécessite lui-même N multiplications

$$X_n = \sum_{k=0..N-1} x_k \exp\left(-2i\pi k \times n/N\right)$$

L'exploitation de la transformée de Fourier rapide (FFT) exploite les symétries de la formule vue ci-dessus pour utiliser autant de fois que nécessaire les mêmes termes $x_k \exp\left(-2\pi k \times n/N\right)$ qui apparaissent plusieurs fois : on notera [2, p.432] la

relation suivante entre les termes pairs F_n^p et les termes impairs F_n^i de la transformée de Fourier F_n

$$X_n = \sum_{k=0..N-1} x_k \exp\left(2i\pi k \times n/N\right)$$

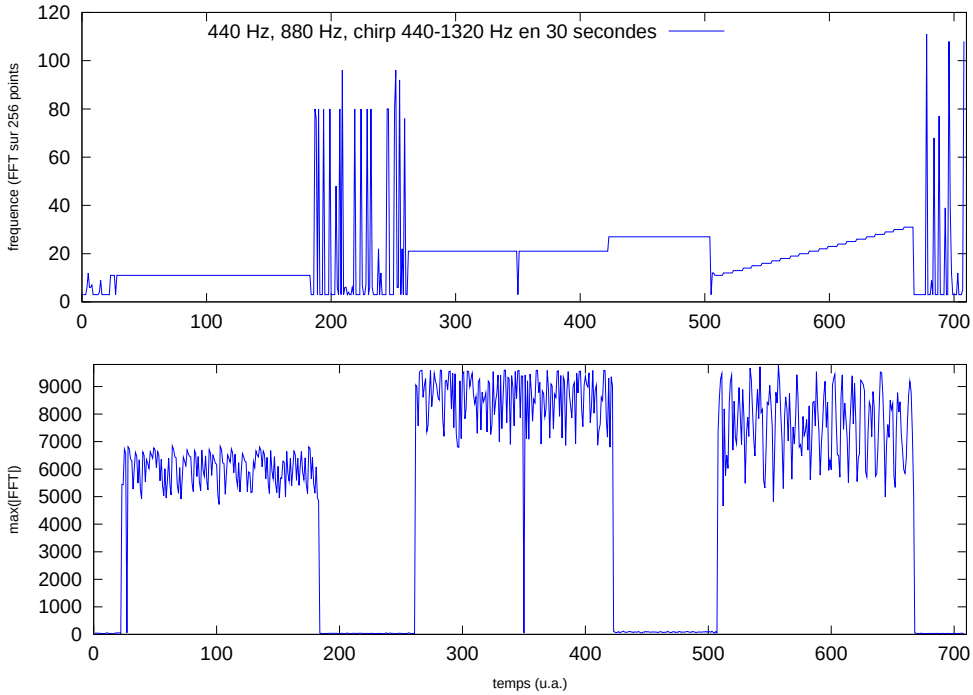
$$= \sum_{k=0..N/2-1} x_{2k} \exp\left(2i\pi 2k \times n/N\right) + \sum_{k=0..N/2-1} x_{2k+1} \exp\left(2i\pi(2k+1) \times n/N\right)$$

$$= \sum_{m=0..M-1} x_{2m} \exp\left(2i\pi m \times n/N\right) + \sum_{m=0..M-1} x_{2m+1} \exp\left(2i\pi m \times n/N\right)$$

$$= P_n + \exp(-2i\pi n/N) I_n$$

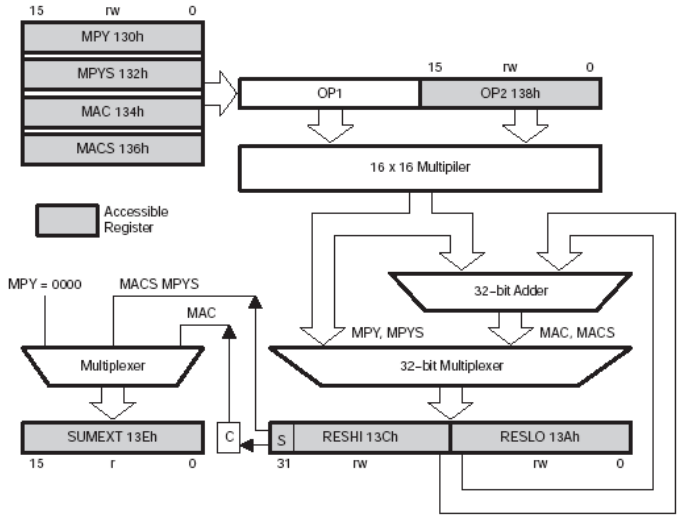
en posant $M = N/2$, et avec P_n le calcul sur les indices pairs et I_n le calcul sur les indices impairs. Chaque calcul d'un terme de FFT nécessite donc N opérations, mais ceci le long d'un arbre binaire de longueur $\log_2(N)$ au lieu de N. La complexité du calcul est donc passée de N^2 à $N \log_2(N)$.

L'implémentation de cet algorithme est décrit en détail dans la note d'application 3722 de Maxim [1].



indice (point)	fréquence (Hz)
11	440
21	880
31	1320

On a bien le point 256 qui correspond à la fréquence 10900 qui est égale à 32768/3.



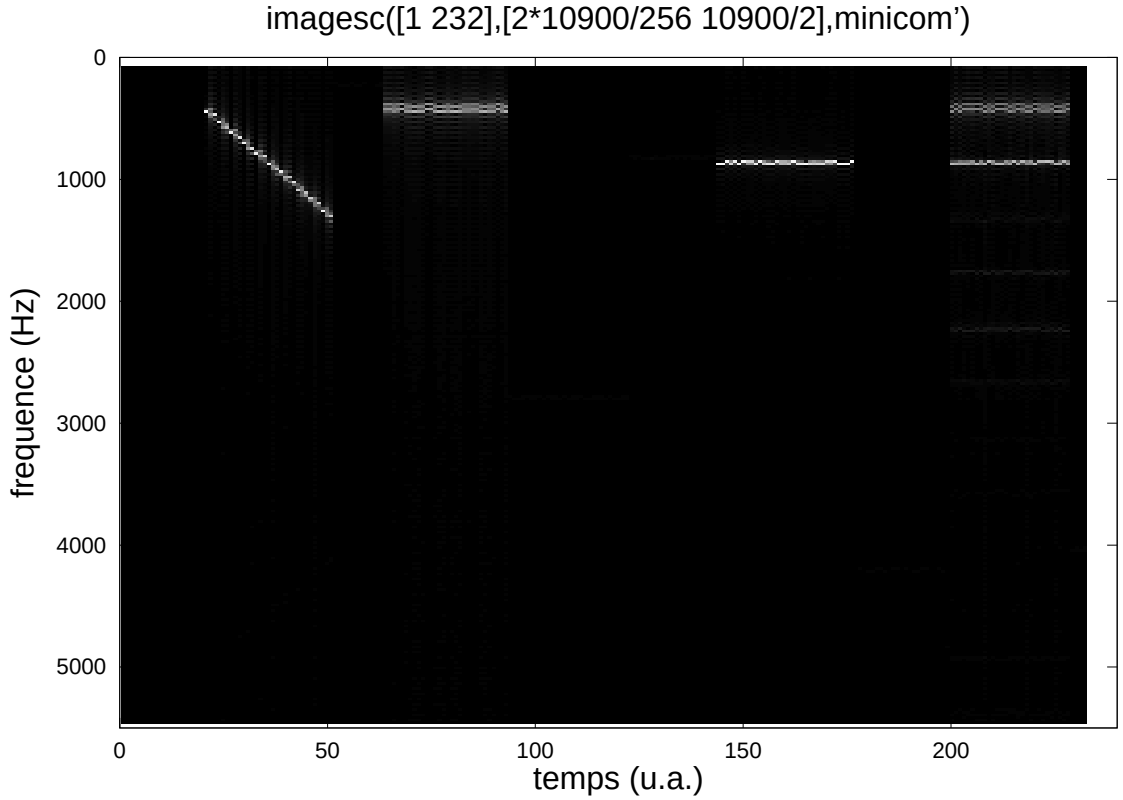
La principale optimisation reste ensuite l'exploitation du multiplieur matériel disponible dans le MSP430. L'étude du code assembleur généré par `msp430-gcc -S` démontre que le compilateur est capable de convertir le symbole `*` en la séquence d'opérations exploitant au mieux le matériel.

Un autre aspect intéressant de l'étude du code de la note d'application AN3722 est de constater la génération automatique de code pour précalculer un certain nombre de constantes. Ainsi, la réorganisation des données pour passer de la transformée de Fourier classique à la FFT est générée automatiquement, ainsi que les tables contenant les fonctions trigonométriques pré-calculées. Par ailleurs, cette génération de code automatique est très sensible à la nature des données manipulées. Les casts successifs (`long` sur 4 octets et `short` sur 2 octets) sont nécessaires pour le bon fonctionnement de ce code :

```
resultMulReCos=((long)cosLUT[tf_index]*(long)x_n_re[b_index])>>7;
resultMulReSin=((long)sinLUT[tf_index]*(long)x_n_re[b_index])>>7;
resultMulImCos=((long)cosLUT[tf_index]*(long)x_n_im[b_index])>>7;
```

[1] `http://www.maxim-ic.com/appnotes.cfm/an_pk/3722`

```
resultMulImSin=((long)sinLUT[tf_index]*(long)x_n_im[b_index])>>7;

x_n_re[b_index] = x_n_re[a_index]-(short)resultMulReCos+(short)resultMulImSin;
x_n_im[b_index] = x_n_im[a_index]-(short)resultMulReSin-(short)resultMulImCos;
x_n_re[a_index] = x_n_re[a_index]+(short)resultMulReCos-(short)resultMulImSin;
x_n_im[a_index] = x_n_im[a_index]+(short)resultMulReSin+(short)resultMulImCos;
```



Spectrogramme obtenu par émission d'un signal à 440 Hz, puis 880 Hz, puis un chirp et finalement les deux fréquences initiales générées simultanément.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "maxqfft.h"

#define msp430

#ifdef msp430
#include <msp430x14x.h>
#include <string.h>
#include <signal.h>
#include <io.h>
#endif
// msp430-gcc -mmcu=msp430x149 -c msp430.c
// msp430-gcc -mmcu=msp430x149 -o maxqfft msp430.o maxqfft.c
// sinon gcc -o maxqfft maxqfft.c -lm

short x_n_re[Nfft]; // FFT input samples, and real part of the spectrum

void getSamplesFromADC()
{unsigned short i;
 short *ptr_x_n_re = x_n_re;
 for(i=0; i<256; i++) {
#ifdef msp430
     LPM1; x_n_re[i]=mesure()/32; P1OUT^=0xff;
#else
     *(ptr_x_n_re++)=(short)(100.*(sin((double)i/3+1.)));
#endif
   }

   for(i=0; i<256; i++)
   {if (x_n_re[i]&0x0080)
       x_n_re[i] += 0xFF00; // Convert to 2's comp
   }
}

int main()
```

```
{
#ifdef msp430
   unsigned char buffer[20];
   init_PORT();
   init_ADC();
   init_rs1_9600();
   init_TIMER();
   eint();

   TACTL|= MC_1 + TACLR;
#endif

#ifdef msp430
   while (1) // FFT Loop
#endif
   {
      unsigned short i;      // Misc index

      short  n_of_b      = N_DIV_2; // Number of butterflies
      short  s_of_b      = 1;       // Size   of butterflies
      short  a_index     = 0;       // fft data index
      short  a_index_ref = 0;       // fft data index reference
      char stage       = 0;       // Stage of the fft, 0 to (Log2(N)-1)

      short  nb_index;    // Number of butterflies index
      short  sb_index;    // Size   of butterflies index

      short x_n_im[Nfft] = {0x0000};   // Imaginary part of x(n) and X(n),

  bzero(x_n_re,sizeof(short)*Nfft);
  bzero(x_n_im,sizeof(short)*Nfft);
        getSamplesFromADC();

      /* Perform Bit-Reversal: Uses an unrolled loop that was created
         with the following C code:

            #include <stdio.h>
            #define  N               256
            #define  LOG_2_N           8

            short bitRev(short a, short nBits)
            {short rev_a = 0;
              for (short i=0; i<nBits; i++)
               {rev_a=(rev_a<<1)|(a&1);a=a>>1;}
               return rev_a;
            }

            short main(short argc, char* argv[])
            {printf("   unsigned short i;\n");
               for(short i=0; i<N; i++)
                  if (bitRev(i,LOG_2_N) > i)
                  {printf("   i=x_n_re[%3d]; "       ,i);
                   printf("x_n_re[%3d]=x_n_re[%3d]; ",i,bitRev(i,LOG_2_N));
                   printf("x_n_re[%3d]=i;\n"          ,bitRev(i,LOG_2_N));
                  }

               return 0;
            }
      */

      i=x_n_re[  1]; x_n_re[  1]=x_n_re[128]; x_n_re[128]=i;
      i=x_n_re[  2]; x_n_re[  2]=x_n_re[ 64]; x_n_re[ 64]=i;
      ...
      i=x_n_re[223]; x_n_re[223]=x_n_re[251]; x_n_re[251]=i;
      i=x_n_re[239]; x_n_re[239]=x_n_re[247]; x_n_re[247]=i;

      for(stage=0; stage<LOG_2_N; stage++)
      {for(nb_index=0; nb_index<n_of_b; nb_index++)
         {short tf_index = 0; // The twiddle factor index
           for(sb_index=0; sb_index<s_of_b; sb_index++)
            {long resultMulReCos;  // jmfriedt : short -> long ***
             long resultMulImCos;
             long resultMulReSin;
             long resultMulImSin;
             short b_index = a_index+s_of_b; // 2nd fft data index

             resultMulReCos=((long)cosLUT[tf_index]*(long)x_n_re[b_index])>>7;
             resultMulReSin=((long)sinLUT[tf_index]*(long)x_n_re[b_index])>>7;
             resultMulImCos=((long)cosLUT[tf_index]*(long)x_n_im[b_index])>>7;
```

```
            resultMulImSin=((long)sinLUT[tf_index]*(long)x_n_im[b_index])>>7;

            x_n_re[b_index] = x_n_re[a_index]-(short)(resultMulReCos+resultMulImSin);
            x_n_im[b_index] = x_n_im[a_index]-(short)(resultMulReSin-resultMulImCos);
            x_n_re[a_index] = x_n_re[a_index]+(short)(resultMulReCos-resultMulImSin);
            x_n_im[a_index] = x_n_im[a_index]+(short)(resultMulReSin+resultMulImCos);

            if (((sb_index+1) & (s_of_b-1)) == 0)
                 a_index = a_index_ref;
              else
                 a_index++;

            tf_index += n_of_b;
           }
           a_index     = ((s_of_b<<1) + a_index) & N_MINUS_1;
           a_index_ref = a_index;
         }
         n_of_b >>= 1;
         s_of_b <<= 1;
      }

      for(i=0; i<N_DIV_2_PLUS_1; i++)   // valeur absolue
       {if ((x_n_re[i] & 0x8000)!=0x0000) x_n_re[i]=-x_n_re[i];
        if ((x_n_im[i] & 0x8000)!=0x0000) x_n_im[i]=-x_n_im[i];
       }


#ifdef msp430
  n_of_b = 0; // pos max
  s_of_b = 0; // val max
  for(i=3; i<N_DIV_2_PLUS_1; i++)
    { sprintf(buffer,"%d %d ",x_n_re[i],x_n_im[i]);
      str_rs1(buffer);
//      if ((x_n_re[i]+x_n_im[i])>s_of_b)
//         {s_of_b=x_n_re[i]+x_n_im[i];n_of_b=i;}
    }
//  sprintf(buffer,"%d %d\r\n",n_of_b,s_of_b);
  sprintf(buffer,"\r\n");
  str_rs1(buffer);
#else
  for(i=0; i<N_DIV_2_PLUS_1; i++) printf("%d %d\n",x_n_re[i],x_n_im[i]);
#endif
   }
return(0);
}
```

Listing 1 – Calcul de la FFT tel que implémenté dans l'AN 3722 de Maxim. Noter la génération automatique de code pour l'organisation des données en mémoire.

```
#ifndef __MAXQ_FFT_H__
#define __MAXQ_FFT_H__

#define Nfft              256
#define N_DIV_2           128
#define N_DIV_2_PLUS_1    129
#define N_MINUS_1         255
#define LOG_2_N             8

/*
   cosine Look-Up Table: An LUT for the cosine function
   created with the following program:

     #include <stdio.h>
     #include <math.h>
     #define  N 256
     void main(int argc, char* argv[])
     {
     printf("const int cosLUT[%d] =\n{\n",N/2);
        for(long i=0; i<N/2; i++)
        {
           printf("%+4d",(int)(128*cos(2*M_PI*i/N)));
           if (i<(N/2-1))     printf(",");
           if ((i+1)%16 == 0) printf("\n");
        }
        printf("};\n");
     }
*/
const short cosLUT[N_DIV_2] =
```

```
{
+128,+127,+127,+127,+127,+127,+126,+126,+125,+124,+124,+123,+122,+121,+120,+119,
+118,+117,+115,+114,+112,+111,+109,+108,+106,+104,+102,+100, +98, +96, +94, +92,
 +90, +88, +85, +83, +81, +78, +76, +73, +71, +68, +65, +63, +60, +57, +54, +51,
 +48, +46, +43, +40, +37, +34, +31, +28, +24, +21, +18, +15, +12,  +9,  +6,  +3,
  +0,  -3,  -6,  -9, -12, -15, -18, -21, -24, -28, -31, -34, -37, -40, -43, -46,
 -48, -51, -54, -57, -60, -63, -65, -68, -71, -73, -76, -78, -81, -83, -85, -88,
 -90, -92, -94, -96, -98,-100,-102,-104,-106,-108,-109,-111,-112,-114,-115,-117,
-118,-119,-120,-121,-122,-123,-124,-124,-125,-126,-126,-127,-127,-127,-127,-127
};

/*
   sine Look-Up Table: An LUT for the sine function
   created with the following program:

     #include <stdio.h>
     #include <math.h>
     #define  N 256
     void main(int argc, char* argv[])
     {
        printf("const long sinLUT[%d] =\n{\n",N/2);
        for(long i=0; i<N/2; i++)
        {
           printf("%+4d",(int)(128*sin(2*M_PI*i/N)));
           if (i<(N/2-1))     printf(",");
           if ((i+1)%16 == 0) printf("\n");
        }
        printf("};\n");
     }
*/
const short sinLUT[N_DIV_2] =
{
  +0,  +3,  +6,  +9, +12, +15, +18, +21, +24, +28, +31, +34, +37, +40, +43, +46,
 +48, +51, +54, +57, +60, +63, +65, +68, +71, +73, +76, +78, +81, +83, +85, +88,
 +90, +92, +94, +96, +98,+100,+102,+104,+106,+108,+109,+111,+112,+114,+115,+117,
+118,+119,+120,+121,+122,+123,+124,+124,+125,+126,+126,+127,+127,+127,+127,+127,
+128,+127,+127,+127,+127,+127,+126,+126,+125,+124,+124,+123,+122,+121,+120,+119,
+118,+117,+115,+114,+112,+111,+109,+108,+106,+104,+102,+100, +98, +96, +94, +92,
 +90, +88, +85, +83, +81, +78, +76, +73, +71, +68, +65, +63, +60, +57, +54, +51,
 +48, +46, +43, +40, +37, +34, +31, +28, +24, +21, +18, +15, +12,  +9,  +6,  +3
};
#endif
```

Listing 2 – Tableaux associés à la FFT : noter la génération automatique de code pour les tables contenant les valeurs de cosinus et sinus

```
#include  <msp430x14x.h>
#include  <string.h>
#include  <signal.h>
#include <io.h>
#include <stdio.h>

void init_rs1_9600(){
 UCTL1=SWRST;    // SetupUART1: mov.b #SWRST,&UCTL1
 UCTL1|=CHAR;    //       bis.b #CHAR,&UCTL1
 UTCTL1=SSEL0;   //       mov.b #SSEL0,&UTCTL1
 UBR01=0x03;     //       mov.b #0x03,&UBR01
 UBR11=0x00;     //       mov.b #0x00,&UBR11
 UMCTL1=0x4A;    //       mov.b #0x4A,&UMCTL1
 UCTL1&=~SWRST; //       bic.b #SWRST,&UCTL1
 ME2|=UTXE1+URXE1;  //   bis.b #UTXE1+URXE1,&ME2
}

void snd_rs1(unsigned char cara)
{while ((IFG2 & UTXIFG1)==0);    // jz    rs_tx1
 TXBUF1=cara;
}

void str_rs1(unsigned char* cara)
{int k=0;
 do {snd_rs1(cara[k]);k++;} while (cara[k]!=0);
}

void init_PORT(void){
  P1SEL = 0x00;
  P1DIR = 0xFF;
  P1OUT = 0x00;
```

```
  P3SEL=0xFE;    // ; P3.6,7 = USART select
  P3DIR=0x5B;    // ; P3.6 = output direction
  P3OUT=0x01;    // ; P3.6 = output val
}

void init_ADC(void){
  ADC12CTL0=SHT0_6+REFON+ADC12ON; // V_REF=1.5 V
  ADC12CTL1=SHP;
  ADC12MCTL0=INCH_0+SREF_1; // p.17-11: single conversion
  ADC12CTL0|=ENC;
}

unsigned short mesure()
{ADC12CTL0|=ADC12SC; //  start conversion
 do {} while ((ADC12CTL1 & 1)!=0);
 return(ADC12MEM0);
}

void init_TIMER(void){
 BCSCTL1 = 0x87;                  // XT2 OFF
 BCSCTL2 = SELM1 + SELS;          // LFXT2: use HF quartz (XT2) if available
 WDTCTL = WDTPW + WDTHOLD;        // Stop WDT

 TACTL=TASSEL0 + TACLR;
 TACCTL0|=CCIE; // CCR0 interrupt enabled
 TACCR0=2;
 TACTL|= MC_0;
}

interrupt(TIMERA0_VECTOR ) wakeup Vector_TIMERA0(void)
{ }
```

Listing 3 – Quelques fonctions pour la communication et initialisation du MSP430

Lors du traitement de N données réelles, nous exploitons des tableaux de N éléments réels et N éléments imaginaires ($2N$ cases mémoire) pour n'exploiter à la fin que $N/2$ éléments de la transformée de Fourier X_n puisque $X_{N-n} = X_n^*$. Il est possible d'optimiser l'occupation mémoire en plaçant la moitié des données initiales dans le tableau de la partie imaginaire des donnés en entrée : alors l'intégralité de la transformée de Fourier est exploitée, et les relations de symétrie entre transformée de données purement réelles ($X_{N-n} = X_n^*$) et purement imaginaires ($X_{N-n} = -X_n^*$) permettent d'identifier quelle partie de la transformée de Fourier vient de quelle donnée initiale.

Pour s'en convaincre, sous GNU/Octave :

```
clear
close
x=sin(2*pi*[0:0.01:10.23]*3)+i*0;
f1=fft(x(1:512));     % 512 complexes
f2=fft(x(513:1024)); % 512 complexes

xx=x(1:512)+i*x(513:1024); % 256 vals
ff=fft(xx);           % 256 complexes

re1(1)=real(ff(1));
re2(1)=imag(ff(1));

for jj=1:255
  re1(jj+1)=real(0.5*(ff(jj+1)+ff(512-jj+1)));
  im1(jj+1)=imag(0.5*(ff(jj+1)-ff(512-jj+1)));
  re2(jj+1)=imag(0.5*(ff(jj+1)+ff(512-jj+1)));
  im2(jj+1)=-real(0.5*(ff(jj+1)-ff(512-jj+1)));
end
plot(real(f1(1:256)-re1))
hold on
plot(imag(f1(1:256))-im1,'g')
plot(real(f2(1:256))-re2,'r')
plot(imag(f2(1:256))-im2,'m')
```

Listing 4 – Comparaison entre le résultat d'une FFT sur N points, et le résultat du calcul exploitant les parties réelles et imaginaires d'un tableau de $N/2$ points.

Références

[1] A.V. Oppenheim & R.W. Schafer, *Discrete-Time Signal Processing*, Prentice Hall (1989)

[2] W.H. Press, B.P. Flannery, S.A. Teukolsky, & W.T. Vertterling, *Numerical recipies in Pascal*, Cambridge University Press (1994), ou sur le web `http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm`