DSM/DAPNIA/SACM
91191 Gif sur Yvette Cedex
France

CEA Saclay

Eléments de Conception d'un Système LLRF Numérique

Michel LUONG - Michel DESMONS

Eléments de Conception d'un Système LLRF Numérique

Michel LUONG – Michel DESMONS

Résumé

Les systèmes LLRF (Low Level RadioFrequency) assurent l'asservissement en amplitude et phase du champ RF dans les cavités accélératrices. Une implémentation numérique apporte une fiabilité et une flexibilité plus grande en comparaison avec une implémentation analogique plus classique. Cependant les performances en terme de stabilisation de champ, i.e. précision et temps de réponse, dépendent fortement du choix de la technologie numérique mise en œuvre pour le traitement du signal nécessaire au fonctionnement du système. Une étude de faisabilité a été réalisée sur la base d'une carte DSP (Digital Signal Processor) industrielle à laquelle était adjoint un module d'acquisition RF dans le cadre du projet IPHI (Injecteur de Protons Haute Intensité). Elle consiste en la synthèse des fonctions clefs d'un système LLRF telles que la démodulation, le filtrage et la compensation, ainsi qu'en l'évaluation des retards des signaux à travers ces blocs fonctionnels. Sa méthodologie et ses résultats sont présentés dans ce rapport.

Introduction

La stabilité en amplitude et en phase du champ accélérateur dans les structures radiofréquence détermine la qualité du transport du faisceau dans un accélérateur. En présence de sources de perturbation externes, elle ne peut être assurée que par le biais de deux boucles d'asservissement, en amplitude et en phase, qui constituent un système LLRF (Low Level RadioFrequency). Les perturbations résultent de déformations mécaniques consécutives aux contraintes thermiques importantes dans les cavités accélératrices en cuivre, aux vibrations microphoniques et à la pression de radiation dans les cavités supraconductrices aux parois minces. Le quadripôle radiofréquence (RFQ) d'IPHI (Injecteur de Protons à Haute Intensité) correspond au premier cas. Un système LLRF peut être réalisé entièrement réalisé avec des composants analogiques : diode pour la détection d'amplitude, mélangeur pour la comparaison de phase, modulateur d'amplitude et modulateur de phase, filtres et compensateurs analogiques. Ces sous-ensembles présentent des temps de propagation faibles inférieurs à la microseconde et offrent la possibilité de produire des boucles aux performances remarquables en termes de précision et de rapidité. Néanmoins, les caractéristiques de ces boucles ne peuvent être modifiées qu'au prix d'une intervention difficile sur les cartes électroniques.

Pour un système LLRF numérique, la représentation vectorielle du champ RF par ses composantes I et Q dans un repère cartésien se substitue à la représentation en amplitude et phase dans un repère polaire car la démodulation I-Q est réalisée aisément par la technique numérique avec une grande précision et dynamique. Toutes les fonctions des sous-ensembles, à l'exception de la modulation I-Q, sont implémentées numériquement. Leurs caractéristiques se définissent par la programmation, d'où la grande flexibilité et adaptabilité qui en résulte. La difficulté pour la conception d'un système numérique réside essentiellement le choix de la plateforme matérielle : FPGA (Field Programmable Gate Array) ou DSP (Digital Signal Processor), et des outils de développement. Ce choix conditionne les performances finales du système, les compétences nécessaires à la conception du système, ainsi que la durée de celle-ci. Si une implémentation sur un FPGA garantit des retards plus courts par un parallélisme plus poussé des processus, sa mise en

œuvre est également plus longue et requiert des compétences plus larges. Par contre, l'utilisation d'une carte DSP industrielle et commerciale permet de minimiser les aspects de conception matérielle. L'implémentation se limite à la programmation en langage C des fonctions clefs grâce un environnement de développement approprié.

Description du Matériel et de l'Outil de Développement

Le matériel est constitué d'un ensemble de deux cartes électroniques produites par la société INNOVATIVE INTEGRATION. Une carte mezzanine appelée module RF permet les conversions analogiques et numériques des signaux. Elle se monte sur une carte M6x au format PCI, équipée d'un DSP Texas Instruments TMS320C6701 à virgules flottantes. Celle-ci réalise le traitement numérique des signaux requis par les fonctions clefs d'un système LLRF. L'environnement de développement utilisé « Code Composer Studio » est commercialisé par TEXAS INSTRUMENTS. Il permet de générer rapidement le code à implanter sur le DSP à partir du langage C grâce à un grand nombre d'aides à la conception et à la vérification.

Module RF

Le module RF comporte 2 entrées ADC pour la conversion analogique/numérique et 2 sorties DAC pour la conversion numérique/analogique. La résolution de ces 4 canaux est de 12 bits qui correspond à la résolution minimale pour les applications LLRF. Une résolution de 14 bits serait préférable pour les collisionneurs linéaires à très haute énergie qui exigent une stabilisation du champ accélérateur beaucoup plus précis. Les circuits ADC (Analog Devices AD9226) permettent un taux d'échantillonnage maximal de 65 Ms/s. Cependant, les filtres anti-repliements qui précèdent ces entrées limitent leur bande passante à 12 MHz. L'accès aux signaux en entrée comme en sortie passe nécessairement par un registre tampon de 1024 mots de 32 bits chacun. Ce registre peut être configuré avec une taille choisie par l'utilisateur. Le module RF communique avec la carte M6x (Figure 1) via un bus propriétaire OMNIBUS. La synchronisation des ADC et DAC est réalisé soit à travers ce bus, soit par une entrée dédiée sur le module RF.

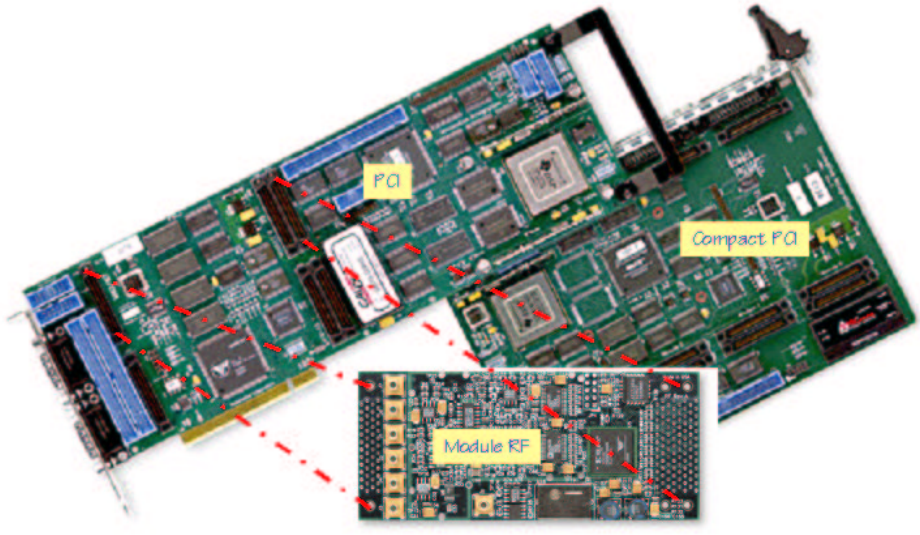


Figure 1 : Module RF et Carte M6x (Innovative Integration)

Carte M6x

Le processeur TMS320C6701 qui équipe la carte M6x est un DSP aux performances élevées cadencé à 160 MHz. Il comporte 4 unités arithmétiques et logiques (ALU), deux multiplicateurs en virgules flottantes, 2 timers 32 bits, une interface JTAG pour la vérification in-situ. Les périphériques supplémentaires, ajoutés par INNOVATIVE INTEGRATION sont pour l'essentiel : une mémoire ASRAM (512 Ko) pour le transfert rapide des données, une mémoire SDRAM (16 Mo), 3 timers 16 bits, une horloge à synthèse numérique directe de grande précision (DDS), et le bus de communication OMNIBUS capable de supporter deux et trois modules d'extension pour la carte en format PCI et Compact PCI respectivement. Le temps de calcul d'une division est de 175 ns, celui d'une transformée de Fourier rapide (FFT) sur 1024 points est de 108 µs.

Code Composer Studio

Code Composer Studio est un environnement de développement intégré et rapide, très similaire à « Borland Builder C++ » pour les applications usuelles sur les plateformes Windows, à la différence près que le fichier de sortie avec l'extension « out » se télécharge dans directement sur le DSP. Le flot de développement à partir du fichier source en langage C est représenté à la Figure 2. Code Composer Studio propose également des utilitaires d'entrée/sortie tels que l'émulateur de terminal qui permet une communication entre le programme DSP et l'ordinateur hôte, c'est à dire l'utilisateur. La Figure 3 montre l'interface graphique du Code Composer Studio pour le développement.

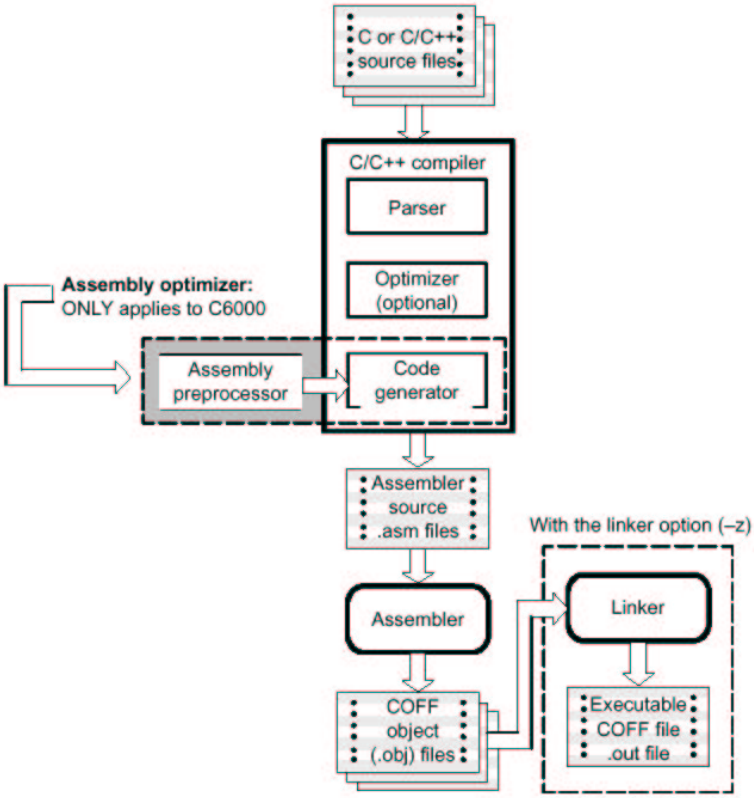


Figure 2 : Flot de développement avec Code Composer Studio

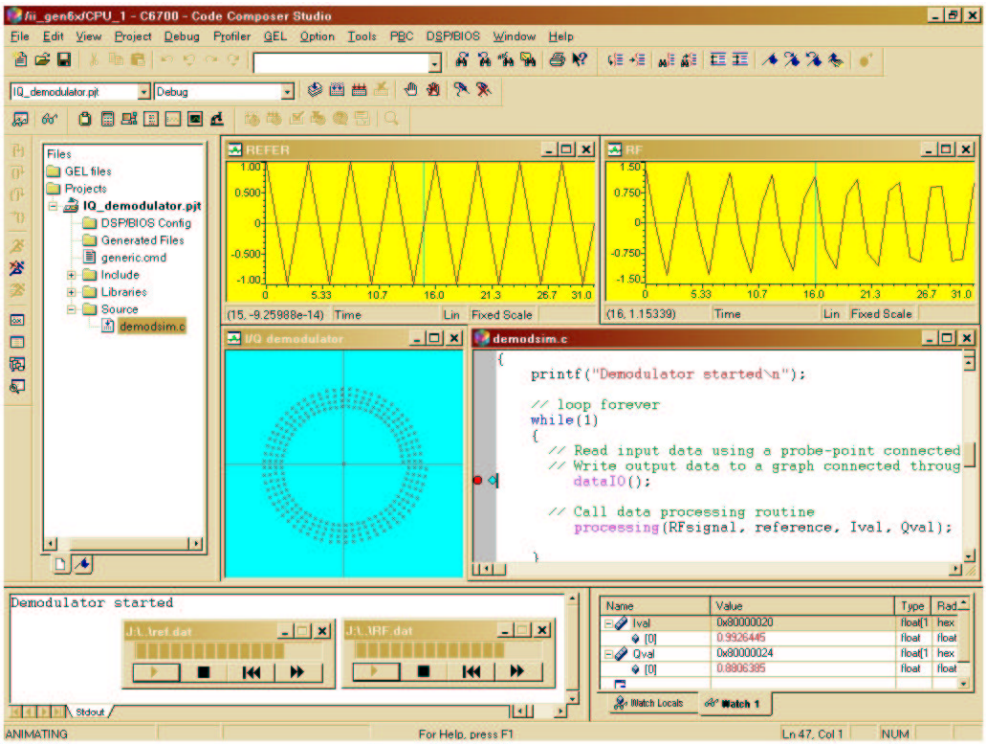


Figure 3 : Interface graphique du Code Composer Studio (exemple : démodulation I-Q)

Dans cet environnement, des fichiers peuvent être importés pour représenter des stimuli et vérifier le fonctionnement de l'algorithme DSP. De même, les données issues de ce dernier peuvent être exportées vers un fichier. Le contrôle des fonctions de la carte M6x et du module RF fait appel à des routines en C disponibles dans les bibliothèques fournies par le fabricant. Par exemple, l'accès aux valeurs acquises par le convertisseur ADC se fait par la fonction, *RF_bleed_adc_fifo(module, (uint32*)&buffer[0], Taille_FIFO)* et l'actualisation de la sortie du convertisseur DAC par la fonction, *RF_fill_dac_fifo(module, (uint32*)&buffer[0], Taille_FIFO),* où *module* désigne un module RF parmi les deux que supporte la carte M6x, *buffer* le registre tampon et *Taille_FIFO* sa taille.

Système LLRF Numérique

Malgré les progrès technologiques spectaculaires de ces dernières années dans le domaine des convertisseur ADC et celui des DSPs, il n'est toujours pas possible d'envisager d'échantillonner directement le signal RF d'une cavité accélératrice dont la fréquence est supérieure à une dizaine de mégahertz. Le traitement du signal dans un système LLRF numérique passe donc nécessairement par une translation de fréquence grâce à un mélangeur. Un schéma générique d'un tel système est présenté à la Figure 4. Le signal à la fréquence IF issu du mélangeur contient une information en amplitude et phase représentative du signal RF. Le choix cette fréquence IF dépend de la dynamique des perturbations et de celle de la cavité accélératrice, en particulier de sa bande passante. Dans le cas des cavités supraconductrices, celle-ci se limite généralement à quelques centaines de hertz, compte tenu du couplage externe. Par contre, les cavités en cuivre se caractérisent par une bande passante d'un ou de deux ordres de grandeur supérieurs. Ainsi, une fréquence IF de 200 kHz à 5 MHz constitue un compromis judicieux pour compenser les perturbations relativement rapides sans dépasser les capacités de calcul du DSP. Puisque le signal de référence est aussi à la fréquence IF, plus celle-ci est basse, plus il sera aisé et économique de le distribuer sur une grande distance, comme dans le cas d'un linac. Le signal de référence assure la synchronisation de l'injection des paquets de particules dans un accélérateur. Les signaux internes qui alimentent le modulateur I-Q et le mélangeur sont régénérés à partir du signal de référence, grâce à des boucles à

verrouillage de phase. Les consignes d'asservissement en I et Q sont introduits sur la carte M6x par l'intermédiaire de l'ordinateur hôte, via l'OMNIBUS. Toutes les autres fonctions clefs du système LLRF sont implémentées sur l'ensemble : module RF + carte M6x. Compte tenu des limitations liées à l'architecture de cet ensemble, qui seront analysées plus tard, le taux d'échantillonnage ne devrait pas dépasser 1 Ms/s pour minimiser les retards de propagation.

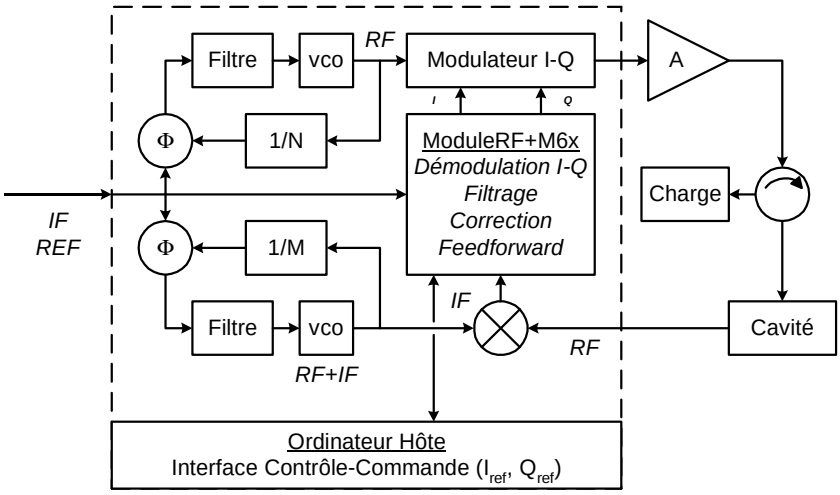


Figure 4 : Schéma générique d'un système LLRF numérique

Démodulation I-Q

L'avantage de la démodulation numérique en I et Q réside dans la grande dynamique et linéarité qu'elle apporte en comparaison à une démodulation en amplitude et phase réalisée de manière analogique. En effet, la détection de phase sur 360 degrés présente généralement une dynamique très limitée en raison de la technologie des mélangeurs utilisés. L'algorithme de la démodulation numérique est en outre très simple. Il consiste d'abord à échantillonner, simultanément le signal de référence et celui issu de la translation de fréquence, tous deux à la fréquence IF, à une fréquence quadruple de cette dernière. Par exemple, les signaux seront échantillonnés à 1 Ms/s pour une fréquence IF de 250 kHz. Ensuite, des opérations élémentaires d'addition et de multiplication entre ces échantillons permettent de calculer les valeurs de I et Q sur une période du signal IF. Comme quatre échantillons sont produits par période pour chaque signal, il s'ensuit une redondance qui peut être exploitée pour améliorer la précision par une moyenne. L'algorithme s'écrit comme suit :

$I_1 = IFrefer[0]*IFsignal[0]+IFrefer[1]*IFsignal[1]$,
$Q_1 = IFrefer[1]*IFsignal[0]-IFrefer[0]*IFsignal[1]$,

$I_2 = IFrefer[2]*IFsignal[2]+IFrefer[3]*IFsignal[3]$,
$Q_2 = IFrefer[3]*IFsignal[2]-IFrefer[2]*IFsignal[3]$,

$I = (I_1+I_2)/2/Aref$,
$Q = (Q_1+Q_2)/2/Aref$,

où *Aref* représente une terme de normalisation qui peut être l'amplitude du signal de référence. Cet algorithme réalise une démodulation à quatre quadrants dont la fonction est équivalente à celle combinée d'un détecteur d'amplitude et d'un détecteur de phase à 360 degrés. Le nombre réduit d'opérations garantit aussi un temps de calcul DSP très court. La précision de la démodulation numérique dépend de la résolution des ADCs et DACs pour la conversion, de leur linéarité, généralement excellente sur une grande dynamique. Elle dépend également de la qualité du mélangeur utilisé pour la translation en fréquence, et qu'il est relativement facile de sélectionner avec soin, alors qu'un détecteur de phase analogique à 360 degrés implique au moins deux mélangeurs avec la difficulté de les appairer avec précaution. Un résultat de démodulation numérique sur quatre quadrants est montré à la Figure 5.

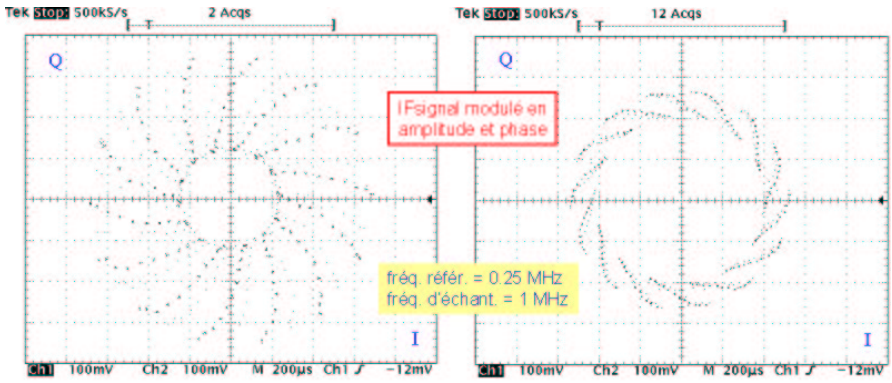


Figure 5 : Démodulation numérique 4-quadrants

Filtrage

Pour réaliser un système LLRF qui assure une parfaite stabilisation du champ accélérateur, l'asservissement doit uniquement prendre en compte le signal provenant du mode fondamental ou mode accélérateur. En pratique, les structures accélératrices peuvent présenter des modes parasites au voisinage du mode fondamental : mode dipolaire pour les RFQs ou mode harmonique pour les cavités multi-cellules. Par exemple, sur une cavité 9 cellules dont le mode fondamental en $\pi/2$ est à 1.3 GHz, le mode harmonique en $8\pi/9$ se situe à 800 kHz au-dessus du premier. La proximité de ces modes rend le filtrage directement sur le signal RF insuffisant. Un filtre passe-bande autour du mode fondamental permet d'atténuer la contribution du mode parasite. Mais il est généralement impossible de réaliser un filtre qui transmet à la fois correctement le signal dans la bande passante et qui présente une atténuation forte en dehors d'une bande relative de 1 pour mille. Pour améliorer la précision du système, la solution consiste alors à filtrer le signal démodulé par un filtre coupe-bande. La Figure 6 montre les fonctions de transfert en boucle ouverte d'un système analogique qui adopte cette approche.

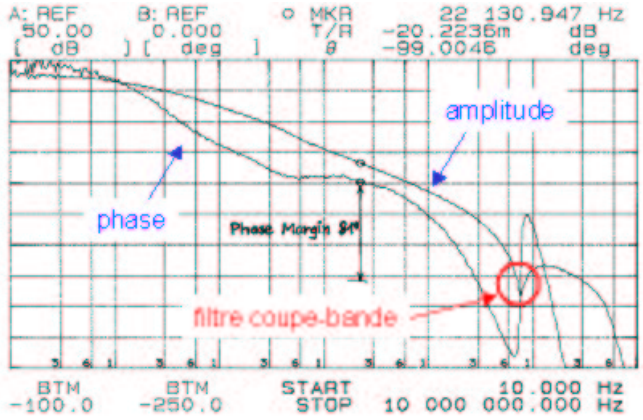


Figure 6 : Filtrage coupe-bande nécessaire à un système LLRF analogique pour une cavité 9 cellules dont le mode parasite se situe à environ 800 kHz du fondamental

Cependant, l'insertion d'un filtre analogique de ce type n'est pas sans conséquence sur la marge de phase du système asservi. Elle tend généralement à la réduire, d'autant plus que le réglage du filtre analogique reste une opération très délicate et difficilement prédictible. Un système LLRF numérique offre la possibilité d'incorporer cette fonction de filtrage dans le traitement du signal de manière parfaitement prédictible grâce aux outils de synthèse de filtre disponibles par exemple dans les produits MATHWORKS (Matlab et Toolboxes). Les coefficients du filtre sont obtenus directement après renseignement de son gabarit et de son ordre, qu'il soit de type non récursif (FIR) ou récursif (IIR). Ainsi, la Figure 7 montre un filtre coupe-bande FIR, centré sur 250 kHz dont les coefficients sont :

float h[9] = {-0.1781455278, 5.231015748e-017, 0.2106440514, 9.325256767e-016, 0.7777373791, 9.325256767e-016, 0.2106440514, 5.231015748e-017, -0.1781455278}

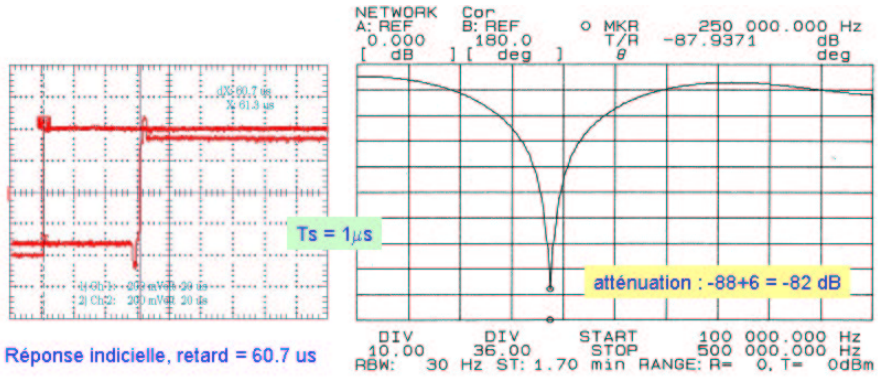


Figure 7 : Filtre coupe-bande numérique FIR

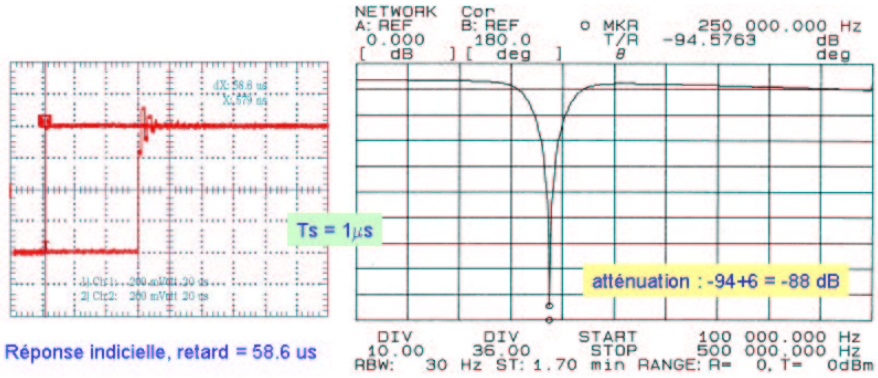


Figure 8 : Filtre coupe-bande numérique IIR

De même, les coefficients d'un filtre IIR (Figure 8) sont donnés :

float b[5] = {0.8005924225, -1.960885905e-016, 1.601184845, -1.960885905e-016, 0.8005924225}
float a[5] = {0.641351521, -4.163336342e-017, 1.561018109, -1.110223025e-016, 1.0}

Pour un retard comparable de 60 µs, la synthèse des deux filtres démontre une très grande précision sur la fréquence caractéristique de 250 kHz comme en témoigne les courbes de mesures. Le filtre IIR est davantage sélectif, mais présente aussi un régime transitoire plus oscillant comme attendu. La seule difficulté de ce type de synthèse réside dans l'écriture de l'algorithme en C qui gère les registres tampons en entrée et en sortie, une fois les coefficient du filtre connus (voir listing en Annexe).

Correction ou compensation

En l'absence de tout filtre RF ou IF, et de tout composant ayant une réponse fréquentielle particulière, une boucle d'asservissement, en amplitude par exemple, comportant une cavité et des amplificateurs dont la bande passante est nettement supérieure à celle de la cavité, possède toujours une réponse en boucle fermée satisfaisante, c'est à dire sans sur-oscillations et dont le comportement dépend uniquement du gain de boucle. Mais les systèmes LLRF réels satisfont rarement cette condition car le filtrage autour de la fréquence de la cavité est généralement nécessaire. La Figure 9 présente le cas d'une cavité à 352 MHz. La bande passante naturelle est déterminée par le facteur de qualité en charge Q_L de 4.4 10^6. Un filtre passe-bande de 280 Hz a été inséré dans la boucle pour atténuer un éventuel mode parasite. Le gain en boucle ouverte est de 50. Il s'agit bien entendu d'un cas d'école dont les caractéristiques n'ont d'autres objectifs que de montrer l'intérêt de la compensation dans un système asservi. Sans la compensation, la marge de phase de ce système ne dépasse pas 20 degrés,

ce qui se traduit par une réponse impulsionnelle et indicielle très oscillante avec un dépassement important. Pour préserver une réponse rapide tout en diminuant ces oscillations, une compensation également appelée correction est nécessaire. Typiquement, un correcteur avec avance ou retard de phase, voire une combinaison des deux, peut être utilisé. La Figure 9 illustre l'action d'un correcteur par avance de phase dont les paramètres ont été déterminés grâce à des simulations avec une application MATLAB dédiée. L'insertion de ce correcteur permet de récupérer une marge de phase plus importante, soit environ 60 degrés. La réponse indicielle correspondante (en rouge) devient alors satisfaisante. Cette compensation revient aussi à modifier l'emplacement des pôles du système en boucle fermée, dans le lieu des zéros et des pôles.

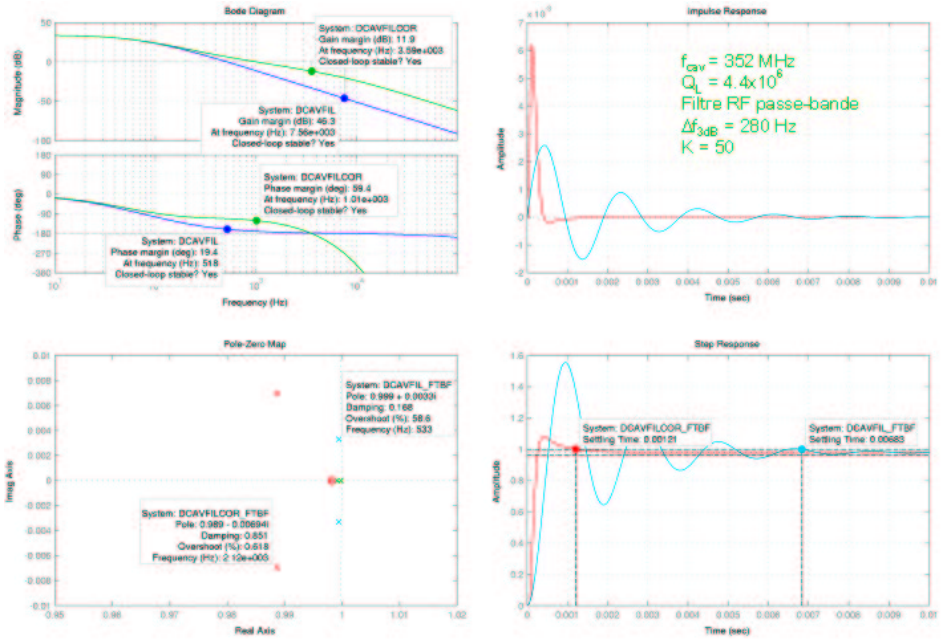


Figure 9 : Intérêt de la compensation sur la dynamique d'un système LLRF

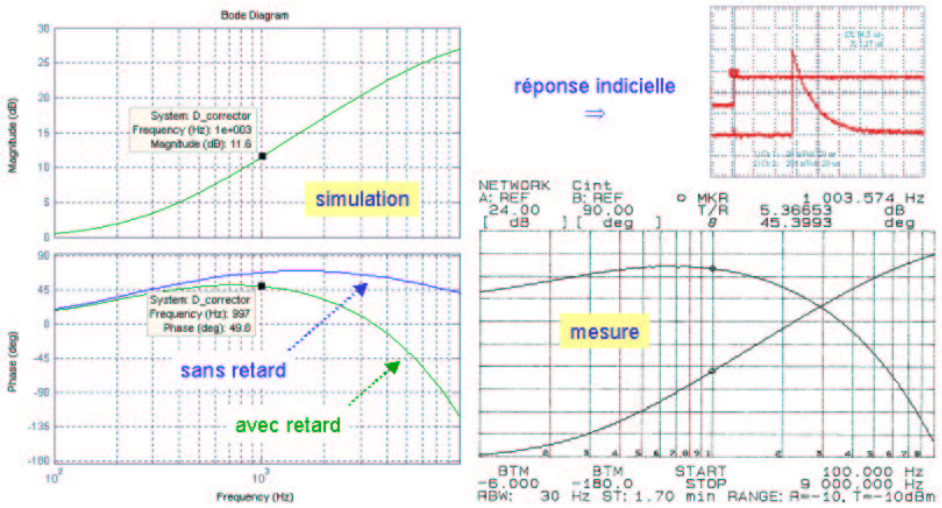


Figure 10 : Synthèse d'un compensateur par avance de phase

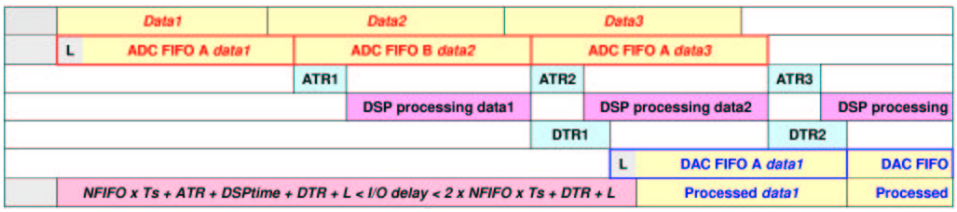
La simulation de l'action du correcteur présentée à la Figure 9 ne prenait pas en compte le retard supplémentaire lié au fonctionnement du module RF et de la carte M6x. La fonction de transfert du correcteur est alors donnée par les coefficients d'un filtre IIR :

float bcor[5] = {-2.9948590e+001, 3.0000000e+001}
float acor[5] = {-9.4858986e-001, 1.0000000e+000}.

Après implémentation, la réponse fréquentielle du correcteur diffère très nettement de celle de la réponse simulée pour ce qui concerne la phase : courbe de mesure à comparer avec la courbe sans retard (bleue) sur la Figure 10. Par contre, si l'on ajoute au filtre IIR un retard de propagation de 50 µs, tel qu'il apparaît sur la réponse indicielle du correcteur, la phase calculée du correcteur devient absolument identifiable à celle mesurée. On peut également noter en comparant la courbe sans retard (bleue) et celle avec retard (verte) que la présence du retard annule quasiment l'avance de phase que le correcteur était censé apporter. Cet exemple montre encore une fois la précision de la synthèse du correcteur (filtre IIR), mais également l'impact critique du retard dans les implémentations réelles. D'une manière générale, on démontre que le gain de boucle qui garantit la rapidité et la précision d'un système LLRF, diminue lorsque les retards dans les éléments constituants de celui-ci augmentent.

Analyse du retard dans le flot de traitement

La taille du registre tampon FIFO, le temps de transfert des données numériques à travers l'OMNIBUS et le traitement DSP conditionnent le retard total de la propagation du signal pour une fonction particulière. La Figure 11 analyse la succession des étapes dans le flot de traitement des données. Les valeurs numérisées du signal par l'ADC ne sont accessibles que lorsque la partie *A* du registre est remplie. Elles sont ensuite transférées via l'OMNIBUS avec une bande passante maximale de 8 MHz sur 32 bits. Le temps dont dispose le DSP pour le calcul est donc borné par la fin du remplissage de la partie complémentaire *B* du registre, soit NFIFO×Ts-ATR. Les limites du retard total sont données à la Figure 11. Il existe donc une valeur optimale de la taille du tampon pour une période d'échantillonnage donnée et un algorithme de traitement donné. Une taille trop grande augmente le retard total tandis qu'une taille trop petite ne laisserait pas suffisamment de temps pour le traitement DSP. Pour les filtres IIR implémentés dans cette étude, la taille était de 8 mots, soit 16 échantillons temporels compte tenu du mode 0 utilisé (voir Annexe). Les retards d'environ 50 µs sont parfaitement compatibles avec cette analyse, sachant que les durées ATR et DTR comprennent en plus du transfert des valeurs des signaux, les octets de contrôle du protocole OMNIBUS.



Ts	*période d'échantillonnage*
NFIFO	*nombre d'échantillons dans la FIFO*
L	*latence de l'ADC = 8 x Ts*
ATR	*temps de transfert de la FIFO ADC vers le DSP*
DTR	*temps de transfert du DSP vers la FIPO DAC*
DSPtime	*temps de calcul de l'algorithme dans le DSP*

Figure 11 : Analyse du retard dans le traitement module RF + carte M6x

Conclusion

Le protocole de transfert des données de la FIFO du module RF vers la carte DSP (M6x) semble être la principale limitation des performances. Pour l'implémentation de filtre ou correcteur numérique, le retard total est de l'ordre de 50 µs et la fréquence d'échantillonnage limitée à 1 Mech/s généralement. La résolution des ADCs et DACs du module RF étant de 12 bits, elle se situe à la limite de ce qui est acceptable pour le système LLRF d'une cavité accélératrice. Le retard de 50 µs confine l'utilisation de cet ensemble (module RF+carte M6x) à des systèmes asservis avec une très faible bande passante (une centaine de Hz), ce qui n'est pas le cas de la cavité RFQ d'IPHI pour laquelle cette étude de faisabilité a été entreprise. Le format PCI de la carte DSP

supportant le module RF, sans un blindage CEM suffisant dégrade également les performances de bruit. La réalisation d'un système plus intégré à base de FPGA sera plus appropriée pour le contrôle LLRF des cavités accélératrices. Le matériel commercialisé par INNOVATIVE INTEGRATION et évalué dans cette étude convient surtout aux applications de traitement massif de données en boucle ouverte, où le retard de propagation n'est pas critique. Le traitement des images peut être l'une de ces applications.

Annexe 1 – Démodulation I-Q Numérique

```
//-------------------------------------------------------------------------------
// Digital I/Q demodulator
// Mode 1 (12 bits resolution, LSW channel 0, MSW channel 1)
// ADC clock from baseboard DDS
//
// User select ADC/DAC buffer size
//-------------------------------------------------------------------------------

#include <periph.h>
#include <unitermio.h>
#include <stdlib.h>
#include <math.h>
#include <TrgStruct.h>

#pragma CODE_SECTION(analog_isr, ".onchip");
#pragma DATA_SECTION(buffer, ".tables");
#pragma DATA_SECTION(reference, ".tables");
#pragma DATA_SECTION(RFsignal, ".tables");

#pragma INTERRUPT(analog_isr);

// General macros
#define module 0
#define gain 0

// General variables
RF_DATA_FORMAT buffer[4];
RF_ID id_struct;
volatile char proc;
int adc_dac_buffer_size;
uint32 nb_samples;
float freq;
volatile float RFsignal[4], reference[4];

// functions prototype
void analog_isr();                    // Processing the data
void stop_timer();                       // Stop internal timer0
void start_timer(uint32 nb_samples);  // Start internal timer0

//-------------------------------------------------------------------------------
//  main()
//-------------------------------------------------------------------------------
main()
{
    char inchar = 'y';
    int key;

    EnableInterrupts();
    DisableInterrupt(EINT0_INTERRUPT); // Disable RF module interrupt
    enable_monitor();

    clrscr();
    gotoxy(10, 0);
    printf("ADC-DAC calibration for I/Q demodulator implementation");
    gotoxy(10, 1);
    printf("Clock from DSP board DDS\n\n");

    // Disable ADC interrupt
    RF_adc_interrupt_disable(module);

    // make sure A/D's and DACs are shut down
    RF_stop(module);

    cursor(OFF);
    printf("\nreading IDROM...\n");
    RF_read_idrom(module, &id_struct);
    cursor(ON);

//----------------------Initializations-------------------------------------------
    while(inchar == 'y')
        {
        printf("\nSampling frequency in MHz ? ");
        scanf("%f", &freq);
        freq = freq*1.0e6;

        adc_dac_buffer_size = 4;
        nb_samples = 4;

       // ADC gain and offset
        RF_set_adc_offset(module, 0, -34);
        RF_set_adc_offset(module, 1, -9);
```

```
        RF_set_adc_gain(module, 0, 1.0);
        RF_set_adc_gain(module, 1, 1.0);

        RF_set_adc_amp_gain(module, 0, gain);
        RF_set_adc_amp_gain(module, 1, gain);

       // DAC gain and offset
        RF_set_dac_offset(module, 0, -40);
        RF_set_dac_offset(module, 1, -20);
        RF_set_dac_gain(module, 0, 1.036);
        RF_set_dac_gain(module, 1, 1.041);

        cursor(OFF);

        // set M6x_DDS as clock source
        timebase(3, 2*freq, 160.0);
        RF_set_adc_clock_source(module, RF_BBDDSC);
        RF_set_dac_clock_source(module, RF_BBDDSC);

        // set mode 1 (2 channels, 12-bit resolution)
        RF_set_adc_mode(module, RF_MODE1);
        RF_set_dac_mode(module, RF_MODE1);

        // clear DAC FIFO
        RF_dac_reset_fifo(module);

        //  Initialize interrupts
        ClearInterrupt(TCLK0_INTERRUPT);
        ActivateInterrupt(TCLK0_INTERRUPT);
        InstallIntVector(analog_isr, TCLK0_INTERRUPT);
        EnableInterrupt(TCLK0_INTERRUPT);

//----------------------Start calibration-------------------------------------------

       cursor(ON);

        RF_start(module);
        start_timer(nb_samples);

        // Echo until "ESC" is pressed
        do
        {
        key = getchar();
        if(key != ESC)     // Send character to UniTerminal (except Esc
            putchar(key);  // because UniTerminal doesn't like Esc)
        if(key == 13) putchar(10);  // also send linefeed if Enter is pressed
        }
       while(key != ESC);

         // disable interrupt on DSP
        stop_timer();
        DisableInterrupt(TCLK0_INTERRUPT);

        // disable A/Ds & DACs
        RF_stop(module);

         // disable ADC amp
        RF_disable_adc_amp(module, 0);
        RF_disable_adc_amp(module, 1);

        printf("\n\nRe-run test? ");
        inchar = tolower(getchar());
        cursor(OFF);

        if(inchar == 'y')
            {
           clrscr();
           gotoxy(10, 0);
           printf("ADC-DAC calibration for I/Q demodulator implementation");
           gotoxy(10, 1);
           printf("Clock from DSP board DDS\n\n");
            normal();
            }
        } // end while
    DisableInterrupt(TCLK0_INTERRUPT);
    DeinstallIntVector(TCLK0_INTERRUPT);

    monitor();
} // end main

//---------------------------------------------------------------------------
//  analog_isr() --  Analog I/O interrupt routine
//---------------------------------------------------------------------------
void analog_isr()
{
    int i;
    float I1, I2, Q1, Q2, Ival, Qval;
```

```
    RF_bleed_adc_fifo(module, (uint32*)&buffer[0], adc_dac_buffer_size);
    for (i = 0; i < 4; i++)
        {
        reference[i] = (float)(buffer[i].Mode1.ch0);
        RFsignal[i] = (float)(buffer[i].Mode1.ch1);
        };

        I1 = reference[0]*RFsignal[0]+reference[1]*RFsignal[1];
        I2 = reference[2]*RFsignal[2]+reference[3]*RFsignal[3];
        Ival = (I1+I2)/2/1024;
        Q1 = reference[1]*RFsignal[0]-reference[0]*RFsignal[1];
        Q2 = reference[3]*RFsignal[2]-reference[2]*RFsignal[3];
        Qval = (Q1+Q2)/2/1024;

    for (i = 0; i < 4; i++)
        {
        buffer[i].Mode1.ch0 = (int)Ival;
        buffer[i].Mode1.ch1 = (int)Qval;
        };

    RF_fill_dac_fifo(module, (uint32*)&buffer[0], adc_dac_buffer_size);
   // clear possible spurious FIFO level interrupt
    ClearInterrupt(TCLK0_INTERRUPT);
}
// End of analog_isr()

//-------------------------------------------------------------------------------------
//  start_timer() --  Timer control
//-------------------------------------------------------------------------------------

void start_timer(uint32 nb_samples)
    {
    Timer *pit = (Timer*)&Periph->Timer[0];
    pit->control.all = 0x001;            // Stop timer 0
    pit->period = nb_samples;
    pit->counter = 0;
    pit->control.all = 0x0C1;            // Run counter
    }

//-------------------------------------------------------------------------------------
//  stop_timer() --  Stop the timer
//-------------------------------------------------------------------------------------
void stop_timer()
    {
    Timer *pit = (Timer*)&Periph->Timer[0];
    pit->control.all = 0x001;
    }

//-------------------------------------------------------------------------------------
// End of PERF_v1.c
//-------------------------------------------------------------------------------------

//-------------------------------------------------------------------------------------
// note : Header file "rf.h" should be modidified as follow :
//
// typedef union
// {
//    struct
//    {
//        volatile int    NA_1          : 4;   // modified by M. Luong
//        volatile int    ch0_2         : 12;
//        volatile int    NA_2          : 4;   // modified by M. Luong
//        volatile int    ch0_1         : 12;
//    } Mode0;
//-------------------------------------------------------------------------------------
```

Annexe 2 – Filtrage ou Compensation

```
//-------------------------------------------------------------------------------------
// ADC Mode 0 (12 bits, 2 samples/FIFO cell)
// ADC clock from baseboard DDS
// Arithmetic operations and digital filter,
// FIR : Fs = 1 MHz, Fstop = 250 kHz
// IIR : Fs = 700 kHz, Fstop = 250 kHz
// User select ADC/DAC buffer size
// Source      : PERF_v1.c
// Executable  : PERF.out
//-------------------------------------------------------------------------------------

#include <periph.h>
#include <unitermio.h>
#include <stdlib.h>
#include <math.h>
#include <TrgStruct.h>

#pragma CODE_SECTION(analog_isr, ".onchip");
#pragma DATA_SECTION(buffer, ".tables");
#pragma DATA_SECTION(h, ".tables");
#pragma DATA_SECTION(x, ".tables");
#pragma DATA_SECTION(y, ".tables");
#pragma DATA_SECTION(a, ".tables");
#pragma DATA_SECTION(b, ".tables");

#pragma INTERRUPT(analog_isr);

// General macros
#define module 0
#define gain 0

// General variables
RF_DATA_FORMAT buffer[1024];
RF_ID id_struct;
volatile char proc;
int adc_dac_buffer_size, clamp_thr;
uint32 nb_samples;
float freq, clamp_level;

// Digital filter variables
int numX, numY, iir_order;
float x[2200], y[2200];
volatile float b[5], a[5];

// FIR coefficients
float h[9] = {
-0.1781455278, 5.231015748e-017, 0.2106440514, 9.325256767e-016,
0.7777373791, 9.325256767e-016, 0.2106440514, 5.231015748e-017, -0.1781455278
};
int numH = 9; // numH is the number of coefficients

// IIR coefficients

// FS = 1.0 MHz
const float biir[5] = {
0.8005924225, -1.960885905e-016, 1.601184845, -1.960885905e-016, 0.8005924225};
const float aiir[5] = {
0.641351521, -4.163336342e-017, 1.561018109, -1.110223025e-016, 1.0};
int filter_iir_order = 4; // number of coef is iir_order+1

// Digital corrector
const float bcor[5] = {
-2.9948590e+001,  3.0000000e+001};
const float acor[5] = {
-9.4858986e-001,  1.0000000e+000};
int corrector_iir_order = 1;

// functions prototype
void analog_isr();                    // Processing the data
void stop_timer();                       // Stop internal timer0
void start_timer(uint32 nb_samples);  // Start internal timer0

//-------------------------------------------------------------------------------------
//  main()
//-------------------------------------------------------------------------------------
main()
{
    int i;
    char inchar = 'y';
    int key;
```

```
    EnableInterrupts();
    DisableInterrupt(EINT0_INTERRUPT); // Disable RF module interrupt
    enable_monitor();

    clrscr();
    gotoxy(10, 0);
    printf("M6x Performance test -- Mode 0, (1 Chan. 12-bit. x2)\n");
    gotoxy(10, 1);
    printf("Clock from DSP board DDS\n\n");

    // Disable ADC interrupt
    RF_adc_interrupt_disable(module);

    // make sure A/D's and DACs are shut down
    RF_stop(module);

    cursor(OFF);
    printf("\nreading IDROM...\n");
    RF_read_idrom(module, &id_struct);
    cursor(ON);

//----------------------Initializations-----------------------------------------
    while(inchar == 'y')
        {
        printf("\nEnter sampling frequency (0.001 to 9.8 MHz): ");
        scanf("%f", &freq);
        freq = freq*1.0e6;

        printf("\nEnter ADC/DAC buffer size (1 to 1024): ");
        scanf("%d", &adc_dac_buffer_size);
        nb_samples = 2*adc_dac_buffer_size;

        printf("\n\nUse IDROM h for gain/offset correction (y/n) ? ");

        if(tolower(getchar()) != 'n')
            {
            printf("Y");
            // init gain-scale factor values to IDROM contents, channel 0 only
            RF_set_adc_gain(module, 0, RF_from_fixed(id_struct.adc_gain_coeff[0]));
            RF_set_adc_offset(module, 0, id_struct.adc_offset_coeff[0]);

            RF_set_dac_gain(module, 0, RF_from_fixed(id_struct.dac_gain_coeff[0]));
            RF_set_dac_offset(module, 0, id_struct.dac_offset_coeff[0]);
            }
        else
            {
            printf("N");
            // init calibration factor values to gain = 1.0, offset = 0.0
            RF_set_adc_gain(module, 0, 1.0);
            RF_set_adc_offset(module, 0, -35);

            RF_set_dac_gain(module, 0, 1.05);
            RF_set_dac_offset(module, 0, -10);
            }
        cursor(OFF);

        // set ADC gain amp for channel 0
        RF_set_adc_amp_gain(module, 0, gain);

        // set M6x_DDS as clock source
        timebase(3, 2*freq, 160.0);
        RF_set_adc_clock_source(module, RF_BBDDSC);
        RF_set_dac_clock_source(module, RF_BBDDSC);

        // set mode 0 (1 channel, 12-bit resolution, stacked x2)
        RF_set_adc_mode(module, RF_MODE0);
        RF_set_dac_mode(module, RF_MODE0);

        // clear DAC FIFO
        RF_dac_reset_fifo(module);

        //  Initialize interrupts
        ClearInterrupt(TCLK0_INTERRUPT);
        ActivateInterrupt(TCLK0_INTERRUPT);
        InstallIntVector(analog_isr, TCLK0_INTERRUPT);
        EnableInterrupt(TCLK0_INTERRUPT);

//----------------------Processing choice-----------------------------------------

       cursor(ON);
       printf("\nSignal processing options :\n\n");
       printf("- 'r' for applying a rectifier,\n");
       printf("- 'c' for clamping the sine wave,\n");
       printf("- 'f' for FIR filtering.\n");
       printf("- 'i' for IIR filtering.\n");
       printf("- 'p' for Phase Advance-Delay corrector.\n");
       printf("- 'return' for echoing.\n");
```

```
        printf("\nPlease make your choice : ");
        proc = tolower(getchar());
        if (proc == 'c') {
            printf("\n\nEnter the clamping level (V) : ");
            scanf("%f", &clamp_level);
            printf("\nPress (+) or (-) to change clamp level by step !\n");
            };
        printf("\n\nEchoing A/D channel 0 to DAC channel 0...\n");
         printf("\nPress (ESC) to stop\n");

         // Clamp variables
         clamp_thr = clamp_level*2048;

         switch(proc)
         {
         // FIR variables and init
         case 'f':
         numX = (int)nb_samples+numH-1;
         numY = (int)nb_samples;
        for (i = 0; i < numX; i++) {x[i] = 0.0;};
        break;
        // IIR variables and init
        case 'i':
        numX = numY = (int)nb_samples+filter_iir_order;
        for (i = 0; i < numX; i++)
        {
        x[i] = 0.0;
        y[i] = 0.0;
        };
        for (i = 0; i <= filter_iir_order; i++)
        {
        b[i] = biir[i];
        a[i] = aiir[i];
        };
        iir_order = filter_iir_order;
        break;
        case 'p':
        numX = numY = (int)nb_samples+corrector_iir_order;
        for (i = 0; i < numX; i++)
        {
        x[i] = 0.0;
        y[i] = 0.0;
        };
        for (i = 0; i <= corrector_iir_order; i++)
        {
        b[i] = bcor[i];
        a[i] = acor[i];
        };
        iir_order = corrector_iir_order;
        break;
        default:;
        };

//----------------------Start acquisition-----------------------------------------

         RF_start(module);
         start_timer(nb_samples);

         // Echo until "ESC" is pressed
         do
         {
         key = getchar();
         if(key != ESC)     // Send character to UniTerminal (except Esc
             putchar(key);  // because UniTerminal doesn't like Esc)
        if (key == '+') {
            clamp_level = clamp_level+0.05;
            clamp_thr = clamp_level*2048;
            };
        if (key == '-') {
            clamp_level = clamp_level-0.05;
            clamp_thr = clamp_level*2048;
            };
         if(key == 13) putchar(10);  // also send linefeed if Enter is pressed
         }
        while(key != ESC);

          // disable interrupt on DSP
         stop_timer();
         DisableInterrupt(TCLK0_INTERRUPT);

         // disable A/Ds & DACs
         RF_stop(module);

          // disable ADC amp
         RF_disable_adc_amp(module, 0);

         printf("\n\nRe-run test? ");
```

```
        inchar = tolower(getchar());
        cursor(OFF);

        if(inchar == 'y')
            {
            clrscr();
            gotoxy(10, 0);
            bold();
            printf("RF Echo - Mode 0, (1 Channel, 12-bit, stacked x2)\n\n");
            normal();
            }
        } // end while
    DisableInterrupt(TCLK0_INTERRUPT);
    DeinstallIntVector(TCLK0_INTERRUPT);

    monitor();
} // end main

//---------------------------------------------------------------------------
//  analog_isr() --  Analog I/O interrupt routine
//---------------------------------------------------------------------------
void analog_isr()
{   int buf_temp;
    int i, j;
    float sum;
    RF_bleed_adc_fifo(module, (uint32*)&buffer[0], adc_dac_buffer_size);

    // Select the processing to be applied to the input signal
    switch(proc) {

//----------------------Take the absolute value-----------------------------------
        case 'r' :
            for (i = 0; i < adc_dac_buffer_size; i++)
                {
                buffer[i].Mode0.ch0_1 = abs(buffer[i].Mode0.ch0_1);
                buffer[i].Mode0.ch0_2 = abs(buffer[i].Mode0.ch0_2);
                };
                break;
//----------------------Clamp the sine wave---------------------------------------

        case 'c' :
            for (i = 0; i < adc_dac_buffer_size; i++)
                {buf_temp = (int)buffer[i].Mode0.ch0_1;
                if (buf_temp > clamp_thr)
                    {buffer[i].Mode0.ch0_1 = (int)(clamp_level*2047);};
                if (buf_temp < -clamp_thr)
                    {buffer[i].Mode0.ch0_1 = (int)(-clamp_level*2047);};
                buf_temp = buffer[i].Mode0.ch0_2;
                if (buf_temp > clamp_thr)
                    {buffer[i].Mode0.ch0_2 = (int)(clamp_level*2047);};
                if (buf_temp < -clamp_thr)
                    {buffer[i].Mode0.ch0_2 = (int)(-clamp_level*2047);};
                };
                break;
//----------------------FIR filter------------------------------------------------
        case 'f' :
                // construction of the x[]
                for (i = 0; i < adc_dac_buffer_size; i++)
                {
                x[2*i+numH-1] = (float)(buffer[i].Mode0.ch0_1);
                x[2*i+numH] = (float)(buffer[i].Mode0.ch0_2);
                }
                // FIR algorithm
                for(j=0; j < numY; j++)
                {
                sum = 0.0;
                    for(i=0; i < numH; i++)
                    {
                    sum += x[i+j] * h[i];
                    }
                y[j] = sum;
                };
                // construction of structure "buffer" for DAC outputs
                for (i = 0; i < adc_dac_buffer_size; i++)
                {
                buffer[i].Mode0.ch0_1 = (int)(y[2*i]);
                buffer[i].Mode0.ch0_2 = (int)(y[2*i+1]);
                };
                // recontruction of overlap
                for (i = 0; i < (numH-1); i++)
                {
                x[i] = x[i+numY];
                };
                break;
//----------------------IIR Filter------------------------------------------------
           case 'p' :
           case 'i' :
```

```
                // construction of the x[]
                for (i = 0; i < adc_dac_buffer_size; i++)
                {
                x[2*i+iir_order] = (float)buffer[i].Mode0.ch0_1;
                x[2*i+iir_order+1] = (float)buffer[i].Mode0.ch0_2;
                }
                // IIR algorithm
                 for(i=0; i<(int)nb_samples; i++)
                 {
                  sum = 0.0;
                  for(j=0; j<iir_order; j++)
                  {
                    sum += x[i+j]  *b[j]   - y[i+j]  *a[j];
                  }
                  y[i+iir_order] = sum + b[iir_order]*x[i+iir_order];
                 }
                // construction of structure "buffer" for DAC outputs
                for (i = 0; i < adc_dac_buffer_size; i++)
                {
                buffer[i].Mode0.ch0_1 = (int)y[2*i];
                buffer[i].Mode0.ch0_2 = (int)y[2*i+1];
                };
                // recontruction of overlap
                for (i = 0; i < iir_order; i++)
                {
                x[i] = x[i+(int)nb_samples];
                y[i] = y[i+(int)nb_samples];
                };
                break;
        default : ;
    }; // end switch

    RF_fill_dac_fifo(module, (uint32*)&buffer[0], adc_dac_buffer_size);

   // clear possible spurious FIFO level interrupt
    ClearInterrupt(TCLK0_INTERRUPT);
}
// End of analog_isr()

//-------------------------------------------------------------------------------
//  start_timer() --  Timer control
//-------------------------------------------------------------------------------

void start_timer(uint32 nb_samples)
    {
    Timer *pit = (Timer*)&Periph->Timer[0];
    pit->control.all = 0x001;             // Stop timer 0
    pit->period = nb_samples;
    pit->counter = 0;
    pit->control.all = 0x0C1;             // Run counter
    }

//-------------------------------------------------------------------------------
//  stop_timer() --  Stop the timer
//-------------------------------------------------------------------------------
void stop_timer()
    {
    Timer *pit = (Timer*)&Periph->Timer[0];
    pit->control.all = 0x001;
    }

//-------------------------------------------------------------------------------
// End of PERF_v1.c
//-------------------------------------------------------------------------------

//-------------------------------------------------------------------------------
// note : Header file "rf.h" should be modidified as follow :
//
// typedef union
// {
//    struct
//    {
//        volatile int    NA_1           : 4;   // modified by M. Luong
//        volatile int    ch0_2          : 12;
//        volatile int    NA_2           : 4;   // modified by M. Luong
//        volatile int    ch0_1          : 12;
//    } Mode0;
//-------------------------------------------------------------------------------
```