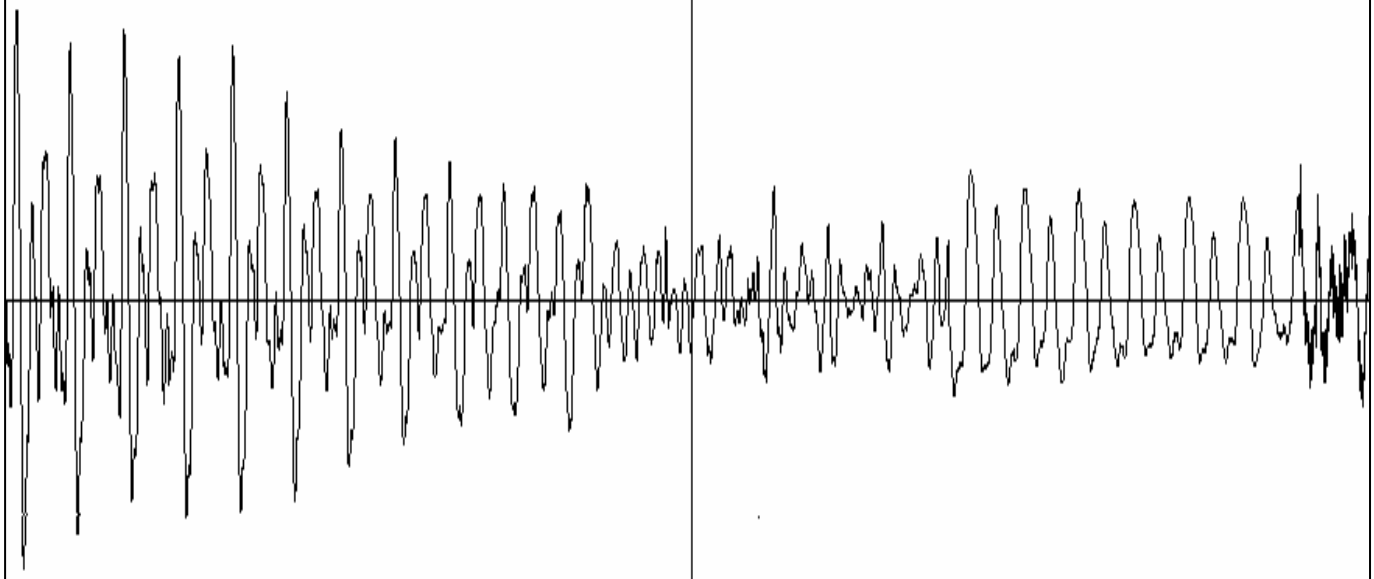
Cédric ESCALLIER
Thierry BAUD

1° Année

Projet d'architecture des ordinateurs



PCSCOPE

Enseignant responsable :
Mr F. MONTANET

Sujet proposé :

Titre du projet: PC Scope

Descriptif du projet:

Définition

Etude et utilisation d'un kit à base de DSP (Digital Signal Processor).

Ce kit est équipé d'un convertisseur CODEC dans la gamme AUDIO et d'une liaison RS232 au PC . Le sujet consiste à acquérir un signal AUDIO injecté à l'entrée du kit et à l'afficher sur le PC à l'aide d'une interface graphique de votre facture. Le pilotage des paramètres de ce traitement (fréquence d'échantillonnage, position du trigger, etc ...) se fait à partir du PC.

Le traitement consistera en:

- Acquisition paramétrée du signal d'entrée
- Définition d'un format pour les données à afficher (trame).
- Transmission de données au PC.
- Contrôle de la trame reçue.
- Affichage des données sur un écran de votre composition.
- Interface de contrôle des paramètres de l'acquisition et de l'affichage 1.2

Domaines abordés

Les principaux domaines abordés sont:

- Programmation DSP en assembleur
- Gestion liaison série avec PC
- Logiciel de contrôle et d'affichage sur PC sous Windows 3.11, Win95 ou DOS

Qui n'a jamais rêvé de transformer son ordinateur en oscilloscope ? Avec un budget minimum, un ordinateur, vous allez pouvoir visualiser des signaux sur votre ordinateur en pouvant changer les principaux paramètres contenus dans un oscilloscope.

Ce projet PcScope a été réalisé dans le cadre du cours d'Architecture des Ordinateurs. Il nous a permis de tester nos connaissances en logique, programmation, dans un but éducatif très attrayant.

Aussi ferons-nous dans une première partie l'analyse du sujet qui a été proposé, afin de voir ses difficultés, et les objectifs que nous nous sommes fixés. Puis dans une seconde partie, nous verrons les moyens mis en œuvre pour la réalisation de PcScope, comment tout s'agence. Enfin, nous expliquerons quelles seraient les améliorations possibles d'un tel projet, et comment réaliser celles-ci.

Réalisation du projet :

I) Analyse du travail demandé

II) Explications approfondies

III) Les réalisations et explications, commentaires

IV) Limitations, quelques améliorations possibles

I) Analyse du projet

Ce projet d'architecture des ordinateurs se divise en plusieurs parties qui touchent à la fois à la programmation en assembleur sur un DSP (Digital Sound Processor), plus particulièrement un kit utilisant ce processeur, et à la communication avec un ordinateur sur lequel des commandes peuvent être envoyées sur ce kit.

A) Analyse des problèmes posés

Le premier problème posé a été de bien délimiter les fonctionnalités que devront posséder un tel projet : aussi nous avons-nous défini les objectifs à atteindre pour ce projet :

- réalisation de la communication entre kit et ordinateur par liaison série
- réalisation de programmes permettant de recevoir et d'envoyer des commandes à partir / sur chacun des terminaux (PC – Kit), avec toutes les implémentations possibles définies dans le sujet
- se mettre d'accord sur le format des programmes réalisés : assembleur pour le Kit et Visual Basic 5.0 pour l'ordinateur.
- savoir comment se déroule un programme sur le kit proposé.

Aussi, se sont posés à nous les problèmes qui se posent à tout débutant en cette matière : comment fonctionnent les programmes pour assembler le programme créé pour le kit, comment fonctionne Visual Basic, comment réaliser telle ou telle fonction. Cela a été le problème le plus long à résoudre.

Puis c'est sur la mise en forme que s'est posée notre attention. Comment faire une interface graphique utilisateur conviviale qui ne trouble pas l'utilisateur et lui laisse une certaine marge de maniabilité.

Ces problèmes mis à plat nous pouvions maintenant nous intéresser au matériel fourni :

B) L'ordinateur

Un ordinateur a été mis gracieusement à notre disposition pour mener à bien ce projet. Mais cet ordinateur assez vétuste ne répondait pas à nos exigences : en effet, cela prenait parfois beaucoup trop de temps pour lancer les applications, voire parfois les ports de communications. Aussi pour éviter de perdre trop de temps sur de tels détails, nous avons eu recours à notre machine personnelle dont ces types d'application était bien supportés.

C) Analyse « externe » du Kit

1) Première ouverture de la boîte : contenu :

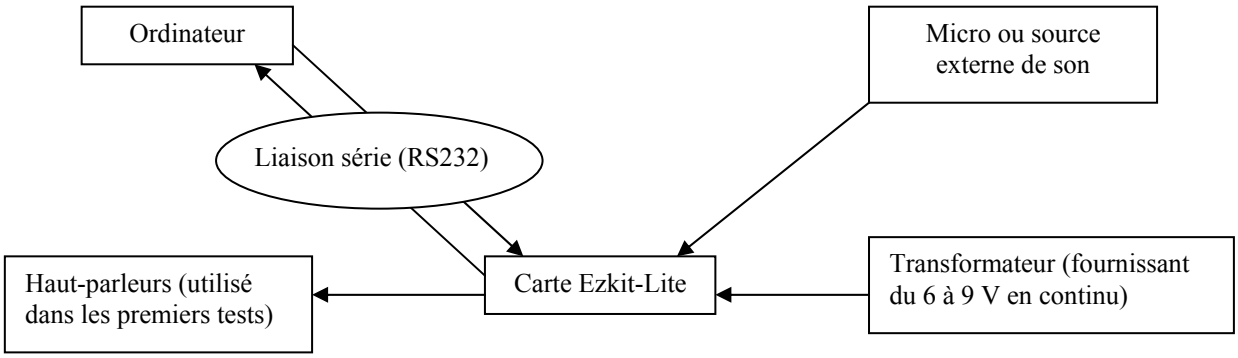
Ce kit Ezkit-Lite est livré dans un carton dans lequel sont contenus plusieurs objets :

- une carte imprimée : LA carte sur laquelle se trouve le DSP, une EPROM … fabriquée par ANALOG DEVICES (voir par la suite la description complète des composants de la carte).
- Un câble série 9 broches.
- 2 manuels de référence : l'un traitant du kit plus spécialement, l'autre de la famille DSP 21XX.
- 2 disquettes contenant un logiciel de test pour la carte et des exemples.
- Un transformateur 220V~ 6V=.
- Diverses documentations relatives au processeur DSP 2181.

2) Mise en place du matériel fourni :

La phase suivante consistait à mettre en place tout ce matériel de manière à mieux comprendre comment agencer le travail en fonction des fonctionnalités requises.

Schéma explicatif :



Voici donc présentées dans le schéma ci-dessus les principales sources externes interagissant sur le kit DSP, et les différentes sorties.

II) Explications approfondies

Suite à l'exposition des objectifs, des tests à la fois sur l'ordinateur et sur le kit on été mis en place. Il s'agissait pour nous de comprendre aussi bien au travers des exemples que des documentations fournies et trouvées sur Internet comment allait s'organiser ce projet PcScope.
Aussi, dans un premier temps avons-nous regardé les programmes exemples fournis avec le kit. Ensuite, afin de mieux comprendre comment ces programmes étaient agencés, nous travaillions les documentations utiles. Enfin, lorsque nos idées étaient claires sur les objectifs, sur les moyens à mettre en œuvre nous programmions le kit (plus particulièrement le processeur), et l'interface graphique sur l'ordinateur.

A) Les programmes exemples testés, base du travail réalisé :

Les programmes testés sont ceux créés lors de l'installation du logiciel EzkitApp, un freeware proposé avec le kit. Ce n'est que plus tard que nous décidons d'utiliser un autre programme (donné par Mr Olivetto) qui est EzKitSDE. En effet, ce programme est une interface graphique de développement de logiciels pour les processeurs de la famille 21XX de ANALOG DEVICES : ainsi, il inclut une compilateur, assembleur et éditeur de lien pour l'architecture des processeurs donnés.
Il nous a permis de mieux comprendre les différentes étapes nécessaires à la création et la « mise en route » d'un programme sur le kit.

Les programmes testés sont des programmes qui utilisent certaines propriétés du Kit, aussi bien le processeur (calcul de la Transformée de Fourier Discrète d'un signal donné grâce au DSP), que le codec (Codeur – décodeur, c'est la « puce » qui gère les arrivées de son, sorties) pour la compression de données…
Ces programmes nous ont permis de comprendre la structure d'un programme assembleur, et nous a donné de plus amples idées sur la réalisation du programme qui sera implémenté sur le kit.
Aussi, pour la réalisation finale du programme pour le DSP, nous sommes-nous basés sur les programmes « echo.dsp » et « delay.dsp », car ceux-ci implémentent beaucoup des fonctionnalités qui vont être demandées par le projet : changement d'état ou de traitement des données en fonction d'un clic sur l'ordinateur (on change les valeurs de l'écho …) d'où l'utilisation de la liaison série, utilisation d'un buffer temporaire pour stocker des données qui ne seront rendues qu'au bout d'un temps donné, sur demande ou automatiquement par exemple.

Bien que cela ait constitué une base pour notre travail, ces programmes n'ont pas répondu à toutes nos exigences, plus spécifiques. Aussi a-t-il fallu créer un programme inspiré de ces sources, mais plus complet ; pour cela il a été nécessaire pour nous de comprendre comment fonctionnait cette carte : pour cela nous avons eu recours à la documentation, aussi longue soit-elle.

B) Le kit EzKit-LITE d'ANALOG DEVICES

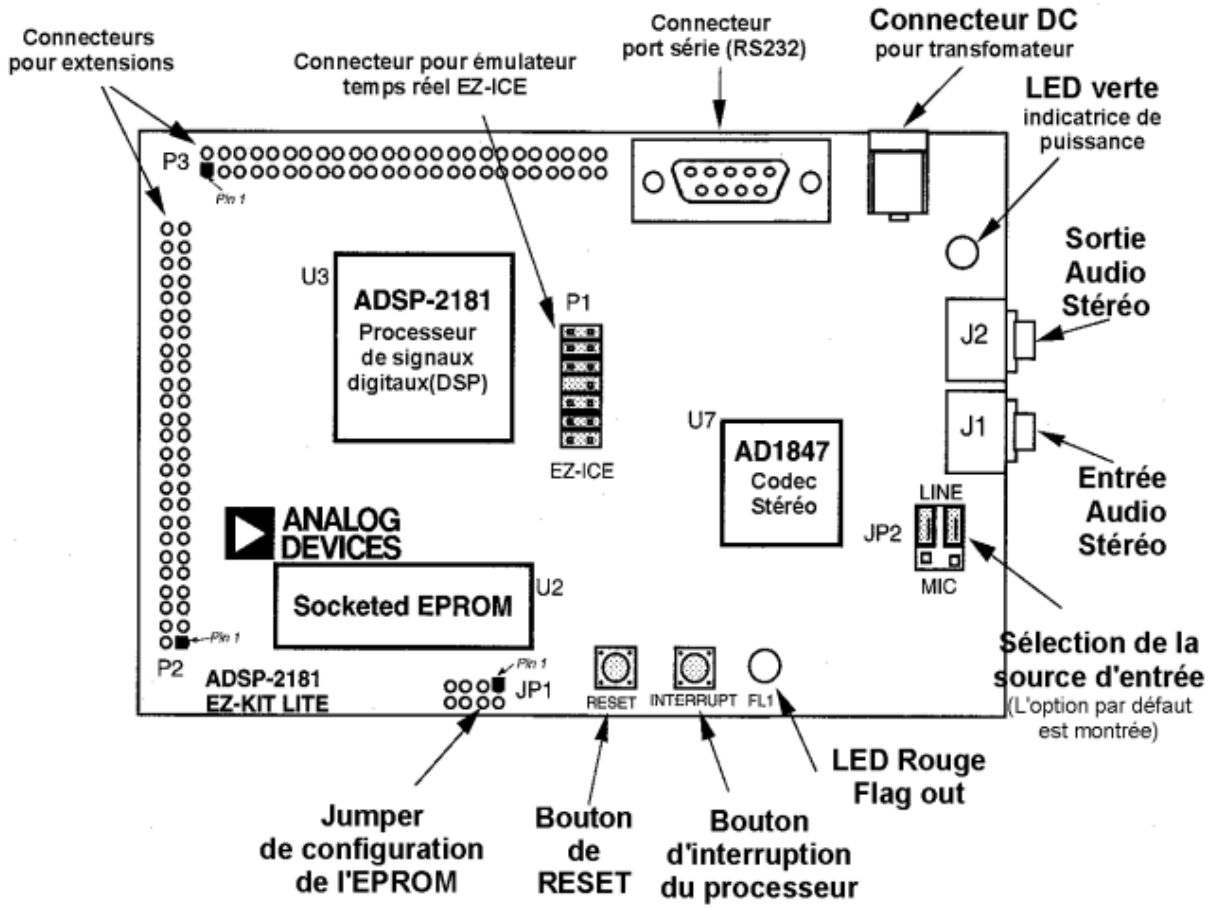


Fig 1.) Le kit Ezkit-LITE

La figure 1. Nous montre les principaux composants de cette carte : Nous allons voir plus en détails dans la suite de ce rapport comment fonctionne cette carte, les divers circuits intégrés, leur fonctionnement et leurs interconnexions.

1) Le principal élément : le DSP

Note : DSP signifie Digital Sound Processor

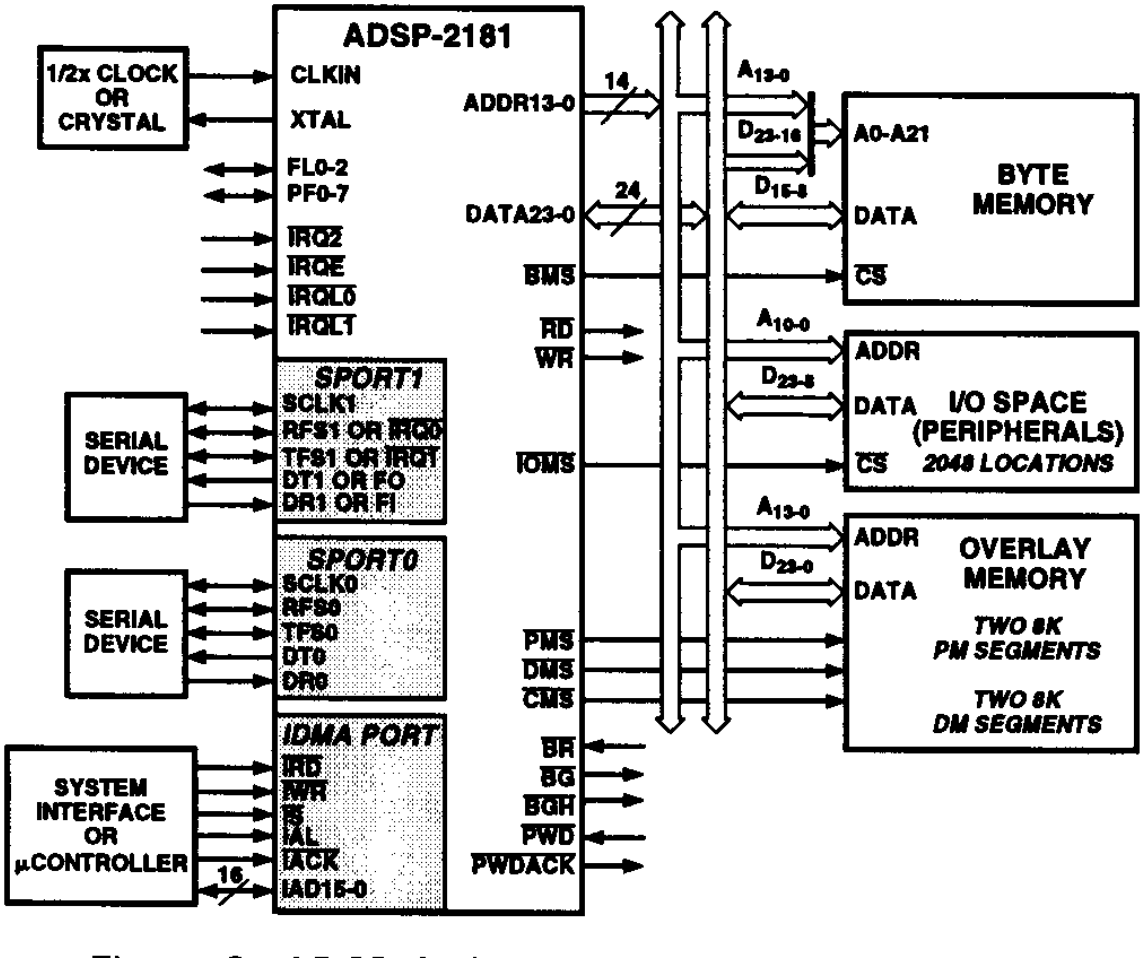


Figure 2. les interfaces au processeur 2181

Ce processeur ADSP 2181 propose beaucoup de fonctionnalités pour des traitements arithmétiques / numériques :

L'unité arithmétique / logique (ALU) :

Celle-ci propose des fonctions arithmétiques et logiques standardisées. Ces opérations sont les addition, soustraction, négation, incrémentation, décrémentation et valeur absolue. Les fonctions assembleurs associées sont AND, OR, XOR NOT.
Cette unité logique utilise des registres 16 bits , X et Y et un registre 32 bits, R. Des variables sont utilisées lors de dépassements de capacité (retenue ou overflow par exemple), le signe (AN), quotient, …

Les registres utilisés sont :

Source pour le port X	Source pour le port Y	Destination pour le port R de sortie
AX0,AX1	AY0,AY1	AR
AR	AF	AF
MR0,MR1,MR2		
SR0,SR1		

MR0, MR1 et MR2 sont des registres résultats de multiplication ou accumulation, SR0 et SR1 sont des registres de décalage (bit à bit).

Par exemple, pour effecteur une addition, il faudra utiliser un registre X et un registre Y et envoyer le résultat dans le registre AR : on a en langage d'assemblage :

AR = AX0 + AY0 ;

Il est très fréquent d'utiliser de telles expressions, mais il faut bien se rappeler les problèmes que l'addition, la multiplication de registres sur 16 bits donne : en effet, pour des opération telles que le « et » logique ou le « ou » logique (etc …) ne donne pas lieu à des dépassement de capacité ou des positionnement de bit de signe ou de contrôle de donnée, seules les additions, soustraction trouvent leurs résultats codés sur 32 bits pour une addition de 2 registres 16 bits. Parfois même, lorsque par exemple on veut ajouter deux valeurs codées sur 16 bits, on a des problèmes de dépassement de capacité : dans ces cas le registre AV est positionné :
Exemple : 0x7000 + 0x9000 = 0x10000
qui est codé sur 17 bits …

L'unité de multiplication / d'accumulation (MAC) :

Cette unité réalise la multiplication, la multiplication avec addition cumulative, saturation, … de manière très rapide.
Les registres utilisés dans cette unité sont les MY ou MX ou MR qui sont définis comme pour l'ALU :

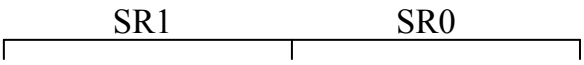
Source pour le port X	Source pour le port Y	Destination pour le port R
MX0, MX1	MY0, MY1	MR (MR2, MR1, MR0)
AR	MF	
MR0, MR1, MR2		
SR0, SR1		

Les décalages :

Ils sont réalisés de manière « naturelle » par le processeur :
On utilise pour cela des fonctions assembleur : LSHIFT, ASHIFT avec en paramètres les nombres de bits à décaler, et le sens du décalage.
Les registres SR (qui se décompose en SR0 et SR1 pour les octets de poids faible et de poids fort de ce registre de décalage), SI.
Exemple de décalage :
On donne SI = 0xB6A3 = 10110110 10100011 et on veut faire un décalage de 5 bits sur la droite de SI. On tape alors la commande assembleur :

SR=LSHIFT SI BY –5 (HI)

Qui aura pour effet de mettre dans SR la valeur décalée :



SR = 00000**101 10110101 00011**000 00000000

Le « HI » a pour effet de dire de mettre les octets de poids forts de SI dans SR et de ensuite effectuer un décalage sur la gauche de –5 bits c'est à dire un décalage de 5bit sur la droite. Il suffit ensuite d'utiliser les registres SR0 et SR1 pour traiter les résultats.

De plus, ce processeur propose des fonctions autres qu'arithmétiques : il permet de créer un programme à part entière, avec des comparaisons, des sauts conditionnels, inconditionnels, des systèmes de boucle, …
Pour cela, tout un jeu d'instructions en langage d'assemblage trouve leur place (ces fonctions ne seront pas détaillées les unes après les autres cela prendrait trop de temps et n'aurait aucun rapport avec le sujet traité donc pour avis voici quelques unes des fonctions les plus utilisées :
JUMP, LOOP, JE (JUMP IF EQUAL), EQ, IF, … dont la plupart se basent sur certains registres positionnés ou non.

En outre, ce type de processeur gère des interruptions logicielles ou matérielles : qu'est ce qu'une interruption ?
Les interruptions sont des phénomènes qui surviennent dans le cours d'une tâche présente à un instant non prévisible (c'est à dire qu'il n'y a aucune synchronisation entre le processus courant et le processus qui demande l'attention du processeur). Chaque processeur exécute une tâche courante. Cependant, pour tenir compte des évènements qui peuvent survenir, mais à des instants qu'on ne peut prévoir (on a prévu ces évènements, ce que l'on ne sait pas c'est le moment où ils vont apparaître, un peu comme l'existence de la foudre dans une région donnée, mais quand et où exactement, on ne le sait pas), chaque processeur est accompagné d'un circuit périphérique de gestion des interruptions.

Nous utiliserons ces interruptions pour gérer les entrées /sorties du codec et de la liaison série PC / kit. Le fonctionnement plus en détail des interruptions qui vont survenir dans le programme créé seront expliqué en même temps que le programme lui-même.

Ensuite, le processeur ADSP 2181 possède intégré sur son chip de la RAM c'est à dire de la mémoire vive. Ceci permet de stocker des variables, buffers, ou autres résultats d'opération. C'est précisément ce buffer que nous utilisons pour stocker les valeurs échantillonnées.
Pour accéder à un segment mémoire (16 bits), il suffit de mettre dans un registre la valeur rendue par la commande : DM (pour Direct Memory)
Exemple : AX0 = DM (i2, 16) ;
Met dans AX0 le contenu de l'adresse (i2 + 16).

D'autres fonctions et fonctionnalités existent sur un tel processeur, certaines d'entre elles seront vues lors des commentaires et explications du programme assembleur, au fur et à mesure de leur apparition.
Nous pouvons maintenant nous intéresser au composant qui permet l'échantillonnage, la compression de données et surtout de transformer un signal analogique en signal numérique et inversement, le codec (Codeur – Décodeur).

2) Le CODEC :

Ce codec est un circuit intégré intitulé « ANALOG DEVICES, AD1847, SOUNDPORT ».
Pour donner un exemple, on peut trouver des codecs comme celui-ci sur la carte son d'un ordinateur : c'est eux qui gèrent les transformations analogique – numérique et numérique – analogique lorsque l'on joue un fichier son sur l'ordinateur. Cela se présente comme un puce de 1 à 2 cm². Les paramètres de celle-ci sont configurés par le programme envoyé au processeur : cela se passe par des paramètres donnés sous forme de bits mis à 1 ou à 0 à une certaine adresse mémoire : par exemple si l'on veut changer la fréquence d'échantillonnage du codec, il suffira de changer à l'adresse 0xc85k (cette dite fréquence d'échantillonnage varie en fonction de la valeur de k) (voir programme DSP fourni en annexe).
Comme exprimé précédemment, le codec permet donc des transformations analogiques numériques, mais aussi des compressions de donnée :
En effet, il existe plusieurs formats de codage des données : ce sont les lois A-Law et µ-Law.
Le codec es capable de générer deux types non compressés de format : le 8 bits non signé linéaire PCM (Pulse Com Modulation), le 16 bits complément à deux linéaire PCM et 2 types compressés sur 8 bits selon la loi en µ ou en A.

Entre les 2 données linéaires PCM, c'est la précision de la valeur de la donnée qui varie. En effet, le type 16 bits permet de digitaliser le son dans une plage dynamique de 96 dB contre seulement 48 dB pour le 8 bits.
Aussi, utiliserons-nous le 16 bits linéaire PCM pour la représentation des échantillons.
Les formats de données compressés en μ ou en A utilisent un codage non linéaire avec une moindre précision pour les signaux de large amplitude.
Toutefois, cette perte de précision est compensée par une augmentation de la plage dynamique à respectivement 64 dB et 72 dB. Le fait que le format de données compressées en μ ne soit pas linéaire signifie que sa représentation graphique telle quelle n'a aucun sens.

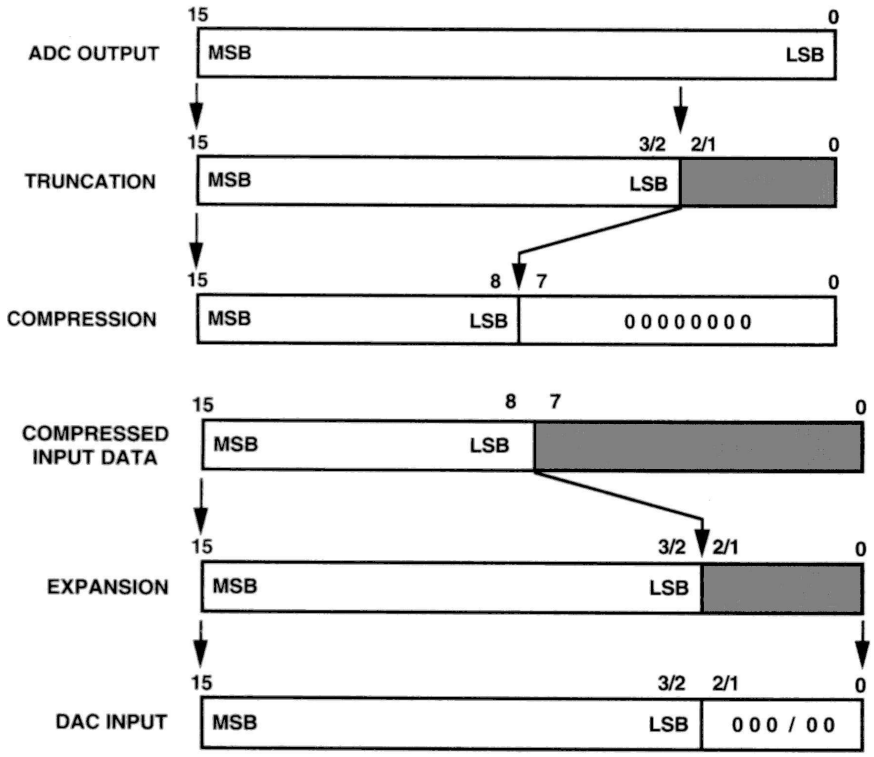


Figure 3. A-Law or μ-Law Expansion

Le problème lors de ces compressions c'est la perte d'informations le signal perd de son amplitude, harmoniques ; par contre, c'est un gain précieux en temps de transferts des données (on envoie 1 octet au lieu de deux).

On peut maintenant se poser la question de savoir comment le processeur communique avec le codec, le câblage, comment est ce que cela fonctionne t il ?

3) Les ports série

C'est sur les liaisons séries que reposent toutes les transmissions codec-DSP de ce kit.
Ces liaisons série sont au nombre de deux sur ce kit : une entre le codec et le DSP et un autre (plus visible) qui relie l'ordinateur au kit.
Il s'agit grâce à ces liaisons série de transmettre numériquement des données. Pour se faire un standard de communication a été établi : la liaison RS232 (date de 1962).
La communication série, consiste à transmettre sur une seule ligne, les données sous forme de variations électriques pour chaque bit ou chaque groupe de bits à transmettre (variations binaire). Les vitesses de transfert sur une communication série s'exprime en bits / seconde ou en bauds.
La communication sur un port série peut s'effectuer soit d'un sens vers l'autre uniquement à la fois, c'est ce que l'on appelle le halfduplex, si la transmission peut s'effectuer dans les deux sens en même temps on parle alors de fullduplex.

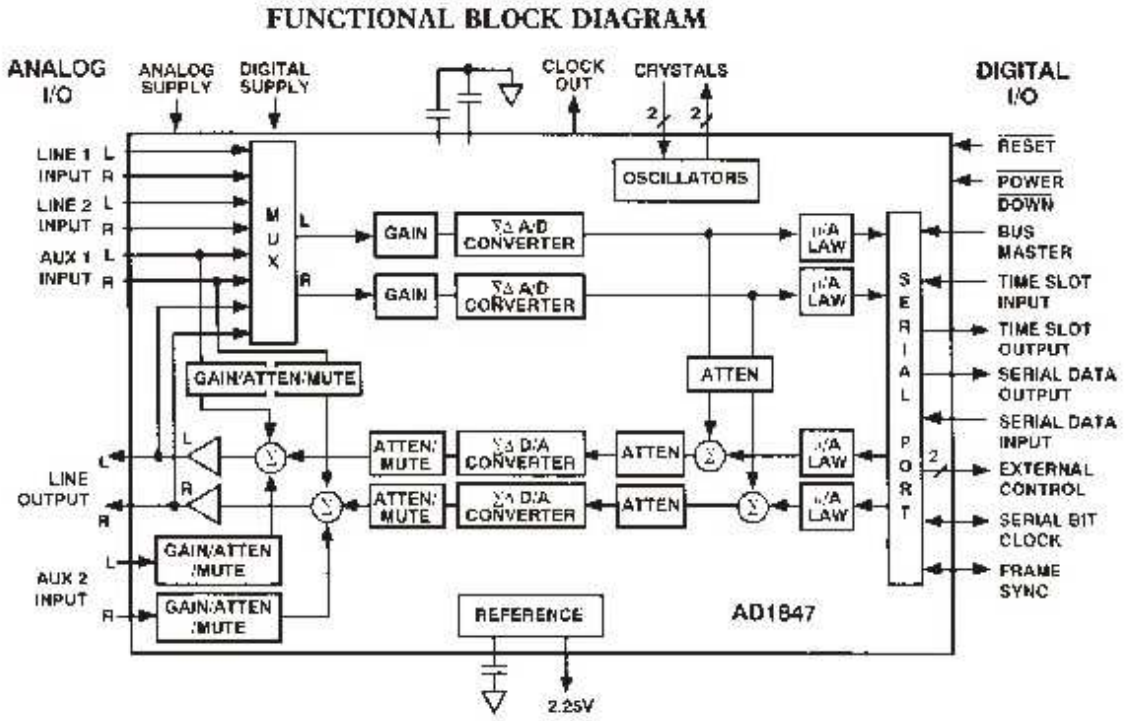


Fig 4. Interfaçage Codec AD1847 et liaison série

Ce schéma indique comment se fais la transmission des échantillons : dès l'assertion du "Frame Sync", le codec AD1847 met les mots (Status+L_sample+R_sample) sur le fil Tx, et par la suite "**Time Slot Output**" indique la fin de transmission des mots à l'intérieur de ce paquet.

On peut choisir la vitesse de transfert du port en changeant les valeur de SCLCK à l'adresse 0x852 SPORT0_Rfsdiv (nous verrons cela plus en détails dans la présentation du programme).

Le second port présent sur la carte est le port de communication vers un connecteur 9 broches mâle. Celui-ci est utilisé pour le transfert de données sériel entre ordinateur et carte. Là encore nous verrons plus en détail la configuration de ce port dans le programme créé par nos soins.

4) L'EPROM

EPROM est le sigle pour « Erasable Programmable Read-Only Memory » ce qui signifie que c'est une mémoire en lecture seule qui peut être effacée et reprogrammée (des rayons ultraviolets permettent l'inscription brute de données sur ce chip qui devient Read-Only).
Ainsi, c'est dans cette EPROM que se trouve le « MONITOR », c'est le programme de boot (de démarrage) pour la carte : elle initialise les ports, codec …

Nous avons donc vu en détails comment se déroulaient les échanges en processeur et codec, comment tous ces composants fonctionnaient / interagissaient; Aussi allons-nous voir comment nous avons implémenté dans des programmes assembleur les possibilités et fonctionnalités de ce kit.

III) Réalisations, explications et commentaires

Une fois les documentations bien dépouillées, les tests réalisés, des programmes réalisant nos attentes ont dû être mis en place. A chaque nouvelle ré implémentation, il fallait changer quelque chose soit au niveau du DSP soit au niveau de l'interface sur l'ordinateur. Cela supposait donc de chaque fois remettre en commun les travaux effectués de manière à ce que tout soit bien mis à jour.

Aussi un code source pour le kit (qui peut être amélioré) a été créé, ainsi qu'une interface graphique pour l'ordinateur. Nous verrons donc quelles sont les particularités et problèmes du programme et les difficultés que nous ont posées leur implémentation.

A) Le programme en langage d'assemblage pour le kit

Ce programme regroupe toutes les diverses particularités énoncées lors de la présentation du processeur. En effet, nous avons dû utiliser beaucoup des fonctions du DSP pour mener à bien ce projet. Voyons maintenant comment doit s'organiser un programme en langage d'assemblage.

- Déclaration des variables, des constantes, des buffers.
- Déclaration des commandes d'initialisation.
- Table des vecteurs d'interruption
- Début des commandes du programme (drapeau start).
- Initialisation des ports sériels, du codec, maskage des interruptions.
- Programme a proprement dit : boucle principale avec tous les tests voulus
- Drapeaux pour les sauts défini dans la table des vecteurs d'interruption.

C'est cette même structure que l'on retrouve dans le programme que nous avons créé.

Celui-ci débute par la déclaration de constantes, de variables, de macros (c'est à dire de fonctions-types) (les commentaires ont été supprimés): on retrouve les fonctions nécessaires aux déclaration :

.include <system.k>;

Ce type de déclaration permet d'inclure des déclarations ou des initialisations définies dans le fichier donné entre <>.

On peut aussi définir :

des fonctions (comme dans les langages de haut niveau) :

```
macro zero(%0, %1, %2);
.local loop;
cntr = %2;  %1 = 1;
     do loop until ce;
loop:       dm(%0, %1) = 0;
.endmacro;
```

L'appel de fonctions :

Zero (i2,m2,L2) ;

Définition de constantes :

.const fs = 0x852 ;

Définition de variables, de différents types:

```
.var/dm         stat_flag ;       {variable en mémoire}
.var/dm/circ    valeur[10] ;    {variable en mémoire, circulaire}
```

circulaire signifie que lorsque le pointeur courant dans le tableau valeur arrive à la fin du tableau, celui-ci est automatiquement repositionné au début.

On initialise :

Initialisation de tableaux :

.init tx_buf: 0xc000, 0x0000, 0x0000;

ce qui signifie : tx_buf [0] = 0xc000 , tx_buf [1] = 0x0000 et tx_buf [2] = 0x0000.

Initialisation de variables :

est réalisé de la même manière.

Une fois les variables déclarées et initialisées, on va définir la table des vecteurs d'interruptions : celle-ci permet de définir l'action à effectuer en fonction du type d'interruption reçu. On voit que les seules interruptions menant à une action particulières sont le RESET, une communication avec le codec, et une interruption envoyée lors de l'envoi de données par le port série 1.
La commande « rti » veut dire return to interrupt, c'est-à-dire « retourne à l'endroit où tu t'es arrêté avant de recevoir l'interruption, en restaurant l'état précédent des registres ».

Aussi, lorsque l'on aura l'interruption reset, le programme sautera à start, si le DSP doit recevoir une valeur de la part du codec, il faudra exécuter la portion de code nommée input_samples (ceci est donc exécuté chaque fois que le codec créé un nouvel échantillon).

La seconde phase consiste en le début réel du programme : cela passe donc par l'initialisation des ports série et du codec.
En résumé nous avons :

- initialisation des pointeurs sur tableaux, donc des principaux buffers :

```
i5 = ^rx_buf;
l5 = %rx_buf;
```

- configuration du port série 0 (fais le lien entre le codec et le DSP) :

```
ax0 = b#0000110011010111;   dm (SPORT0_Autobuf) = ax0;
ax0 = 0;    dm (SPORT0_RFSDIV) = ax0;
ax0 = 0;    dm (SPORT0_SCLKDIV) = ax0;
ax0 = b#100001100001111;   dm (SPORT0_Control_Reg) = ax0;
ax0 = b#0000000000000111;   dm (SPORT0_TX_Channels0) = ax0;
{   ^15         00^   transmit word enables: channel # == bit # }
ax0 = b#0000000000000000;   dm (SPORT0_TX_Channels1) = ax0;
{   ^31         16^   transmit word enables: channel # == bit # }
ax0 = b#0000000000000111;   dm (SPORT0_RX_Channels0) = ax0;
{   ^15         00^   receive word enables: channel # == bit # }
ax0 = b#0000000000000000;   dm (SPORT0_RX_Channels1) = ax0;
{   ^31         16^   receive word enables: channel # == bit # }
```

pour une signification plus précise de ceci se reporter au code source, commenté.
cette portion de code permet donc d'initialiser le port série 0 : vitesse de transmission , autobuffering, codage (loi A, µ ou simple), …
Cette partie du code a été créée à partir du fichier « echo.dsp ».
On rend ensuite disponible les ports sériels :

```
ax0 = b#0001100000000000;   dm (System_Control_Reg) = ax0;
```

puis on force et enlevons les interruptions « edge-sensitive » logiciellement :

```
ifc = b#00000111111111;
```

on règle ensuite la sensibilité de toutes les interruptions :

```
icntl = b#00010;
(0 = sensible au niveau, 1 = sensible au « pont »)
toutes les interruptions sont donc sensibles au niveau sauf l'IRQ1
```

Note : la vitesse de la communication est fixée à 9600 bauds pour les deux ports série.

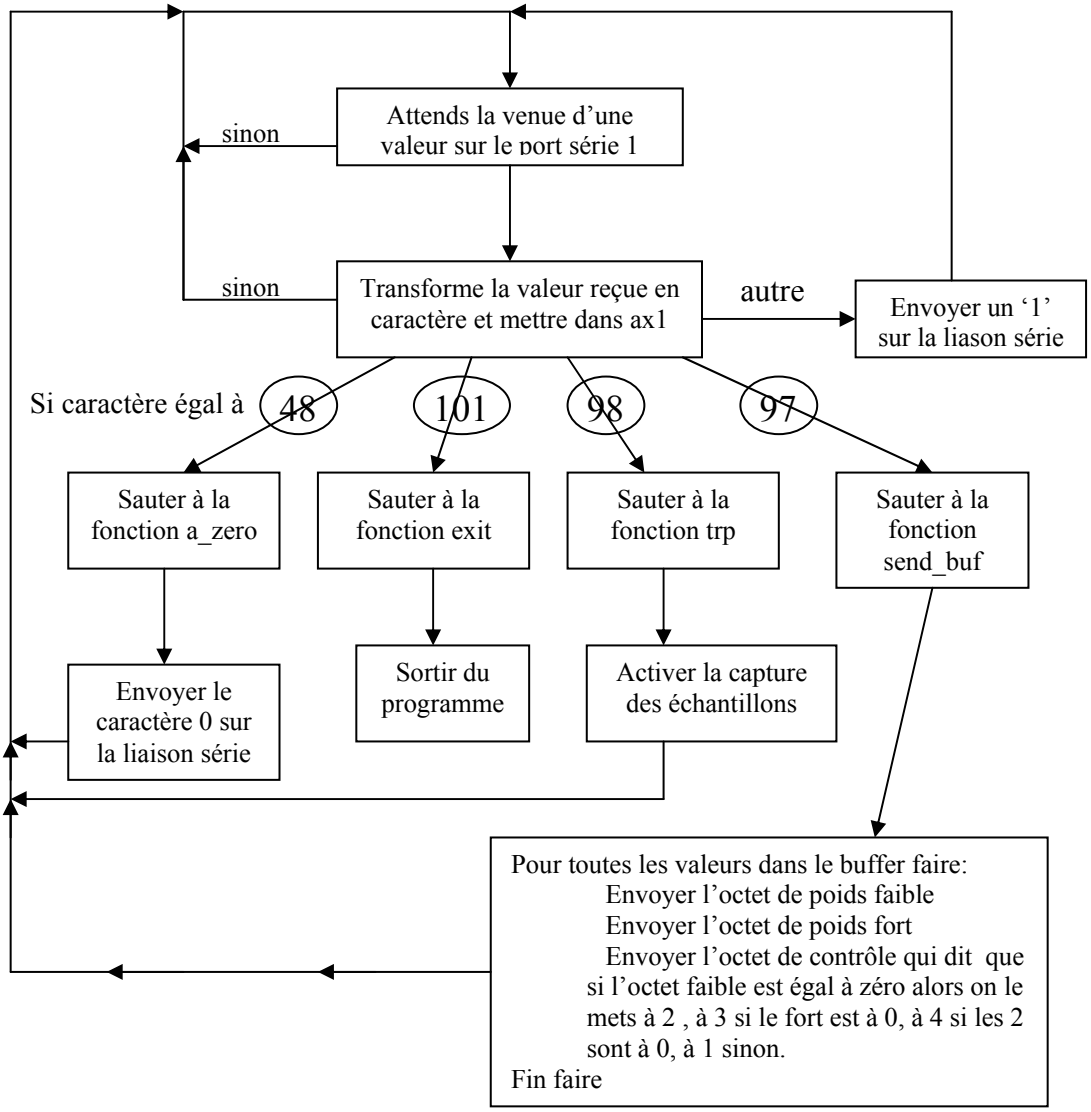
puis le mode opératoire du processeur est fixé :

mstat = b# 1 0 0 0 0 0 0;

{ | | | | | | +- | sélection du banc de données
| | | | | +-- | mode renversé de la FFT (DAG1)
| | | | +--- | mode de dépassement de capacité de l'unité arithmétique
| | | +---- | mode de saturation de AR, 1=saturé, 0=wrap
| | +----- | résultat de l'unité MAC, 0=fractionel, 1=entier
|+------ | mise en route du timer
+------- | mode GO
}

On effectue en suite l'initialisation du codec 1847.

C'est maintenant que commence l'algorithme que nous avons mis en place : celui-ci se résume :



Rien de bien étonnant dans cette algorithme, la seule particularité est que l'on envoie le buffer sans faire un « reset » du kit : cela permet pouvoir refaire une capture sans pour autant renvoyer le programme code sur le kit.
Une question se pose alors : quelle est donc l'utilité de l'octet de contrôle ?
Nous avons remarqué, après beaucoup d'essais consécutifs, que lorsque l'on demandait au kit de nous envoyer un 0 (la valeur 0, et non pas le caractère), on ne recevait rien sur l'ordinateur ; aussi une parade à ce très gros problème (une valeur 16 bits est envoyée en 2 fois, si on « manque » une valeur, l'interprétation du résultat sera fausse à partir de ce moment.
Aussi, plutôt que d'envoyer un 0 qui ne sera jamais reçu on envoie à la place un 1 et on positionne l'octet de contrôle.

Si l'octet de poids faible est à 0 alors l'octet de contrôle est à 2,
si l'octet de poids fort est à 0 alors l'octet de contrôle est à 3,
si les octets de poids fort et de poids faible sont à 0 alors l'octet de contrôle est à 4,
sinon l'octet de contrôle est à 1.

Aussi pour transmettre un échantillon codé sur 16 bits, il faudra envoyer 3 octets !
Un moyen pour contourner ce problème serait de faire de la compression, c'est à dire coder sur 8 bits un échantillon de 16 bits (voir Lois A et μ).

L'algorithme ainsi donné, il s'agit maintenant de le traduire en langage d'assemblage :

Tester l'arrivée d'une valeur sur le port série 1 :

Une adresse mémoire se charge de récupérer les valeurs arrivant, et positionne nu flag lorsqu'une valeur est détectée. Cette adresse est CHAR_WAITING_FLAG qui correspond à 0x3e09 en mémoire.

```
ar = dm (CHAR_WAITING_FLAG);
none = pass ar;
if ne jump debut;
```

Transformer la valeur reçue en caractère :

Là encore l'adresse PTR_TO_GET_CHAR, 0x3e08 va nous aider en mettant la valeur reçue dans le registre ax1 (valeur sur 8 bits !).

```
i4 = dm (PTR_TO_GET_CHAR);
call (i4);
if lt jump debut;
```

Tester la valeur de la valeur reçue :

Il s'agit d'utiliser les fonctionnalités qu'offre l'unité arithmétique :
On met dans AY0 la valeur à tester, dans AX0 la valeur référence, et on effectue la soustraction et on regarde la valeur de la retenue :

```
ay0 = 48;
none = ax1 - ay0;
if eq jump a_zero;
```

Envoyer une valeur quelconque sur le port série :

Il suffit de mettre la valeur voulue dans le registre AX1, et d'appeler l'adresse PTR_TO_OUT_CHAR qui va prendre l'octet de poids faible de AX1 et l'envoyer sur la liaison série. On fait donc :

```
ax1= dm(check);
i4 = dm (PTR_TO_OUT_CHAR);
call (i4);
```

Sortir du programme :

Ceci est réalisé par un soft reboot : il s'agit d'envoyer soit la commande rts (return to subroutine) soit de placer un 1 à l'adresse 0x3fe7 sur le sixième bit : on a alors le jeu d'instructions :

```
ax0=dm(0x3fe7);
ay0 = 0x20;
ar=ax0 and ay0;
dm(0x3fe7) = ar;
```

Envoyer les échantillons qui ont été entrés dans le buffer :

Ceci a été une partie de ce projet sur laquelle nous avons passé du temps : cela comprend le protocole expliqué ci dessus mais aussi le programme en langage d'assemblage qu'il a fallu implémenter pour répondre à l'algorithme :
Il s'agit de faire une boucle allant de 0 à la taille du tableau alloué aussi commençons nous par mettre ces valeurs dans des registres auxquels nous ne toucherons pas :

```
mx0 = 0;
ay0 = D-1;
```

puis on initialise le pointeur sur le tableau alloué et le pas de la boucle :

```
i2=^w;
m0=1;
```

on débute alors le corps de boucle :

on positionne check à 1 (c'est l'octet de contrôle qui sera envoyé à la fin)

on lit le nouvel échantillon du buffer : il faut savoir que chaque fois que l'on lit un nouvel échantillon pour le buffer, le pointeur (i2 dans notre cas) se déplace du pas donné en argument (M0 = 1 ici).

On met dans une variable temporaire la valeur de AX1 (pour ne pas avoir à relire plus tard la valeur à l'adresse mémoire (i2 –1).

On fait un et logique de AX1 avec 0xFF : cela va donner un résultat dont l'octet de poids fort sera nul. Aussi, nous allons pouvoir regarder aisément (c'est la même chose que précédemment en remplaçant la valeur référence de AX0 par 0) si AX1 est égal à 0 :

```
ay1 = 0xff;
ar=ax1 and ay1;
ay1 = ar;
ax0 =0;
none = ay1 - ax0;
if ne jump zero_1;
```

si le programme saute à zero_1 c'est que AX1 n'était pas à 0, sinon, il met la variable check à 2 (signifiant octet de poids faible à 0) et met AX1 à 1.

On envoie l'octet de poids faible de AX1 (de la manière vue précédemment).

-on attend que le caractère soit bien parti-

On remet dans AX1 et SI la valeur de la variable temp sauvegardée précédemment.

On effectue ensuite un décalage logique de –8 bits sur la gauche, ce qui revient donc à faire un décalage à droite de l'octet de poids fort.

sr=lshift si by -8 (lo);

On teste ensuite la valeur de l'octet de poids faible du registre SR0. S'il est égal à 0, cela signifie que l'octet de poids fort de la valeur échantillonnée est nul. Aussi, nous allons tester la valeur de check afin de savoir si check doit à son tour être positionné à 3 si check était égal à 1, 4 sinon.

On envoie l'octet décalé.

-on attend que le caractère soit bien parti-

On envoie check, l'octet de contrôle.

-on attend que le caractère soit bien parti-

On incrémente la valeur du compteur de boucle.

On retourne au début de boucle si le compteur de boucle est inférieur au maximum fixé, valeur contenue dans AY0.

Se pose maintenant la question de connaître comment nous allons gérer les échantillons nouvellement arrivés :

Dans la table des vecteurs d'interruption nous avons vu que lorsque le codec tentait d'envoyer un échantillon au DSP, une interruption se déclenchait ayant pour effet de faire sauter le programme au label input_sample :

Là encore un algorithme a été mis sur pied pour éviter tous les problèmes liés au fait que le codec envoie en permanence des échantillons au DSP. Notre but est de démarrer la saisie des échantillons au moment où l'ON VEUT.

Pour cela, une variable « trompeur » sert de témoin :

Cette variable est originellement positionnée à 0 : cela signifie que l'on ne veut pas remplir le buffer des échantillons.

Lorsqu'une portion de programme change la valeur de cette variable, en une valeur supérieure à 0, le bout de programme input_samples prend l'échantillon courant (de la voie gauche) et le met dans le buffer, et incrémente la valeur de la variable « trompeur » et fait un « return to interrupt ». Si la valeur de trompeur est supérieur au nombre maximal d'échantillons, la saisie des échantillons est terminée. On n'incrémente plus trompeur, on ne met plus d'échantillons dans le buffer, on fait un « rti ».

(Voir le programme complet proposé en annexe pour avoir une réelle image de ce qu'est un tel programme).

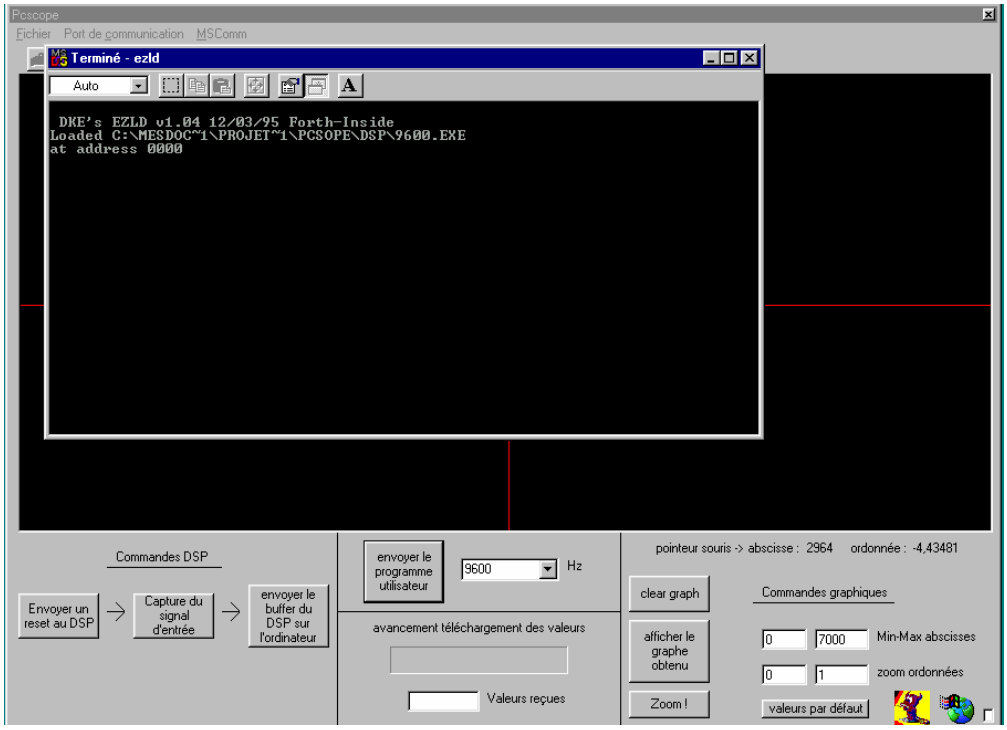
Nous avons vu tout ce qui concernait la partie programmation du Processeur ADSP 2181, ainsi que les problèmes que nous avons dû surmonter. Nous pouvons nous intéresser maintenant à la manière dont le programme a été implémenté côté ordinateur afin de répondre aux limitations de ce programme assembleur, mieux comprendre comment est codée le son.

B) L'interface implémentant les facilités voulues :

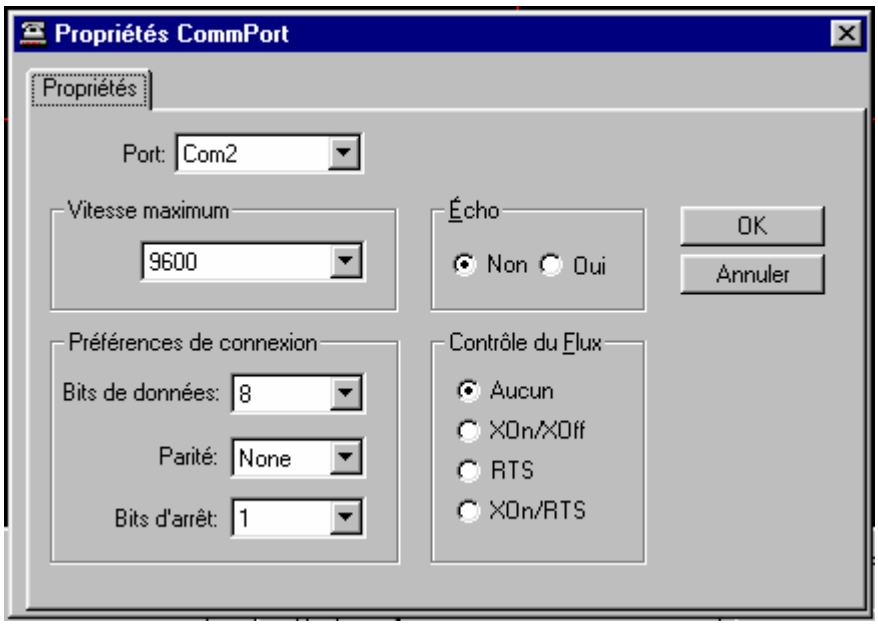
Nous avons vu que notre programme réalisé en langage d'assemblage proposait certaines fonctionnalités qui ne pouvait être utilisables que si l'on envoyait sur la liaison série la chaîne de caractère spécifique. Aussi, a-t-il fallu trouver un logiciel qui réponde aux facilités de communications avec le port série et d'interfaçage.

1) Choix du logiciel d'implémentation de l'interface graphique :

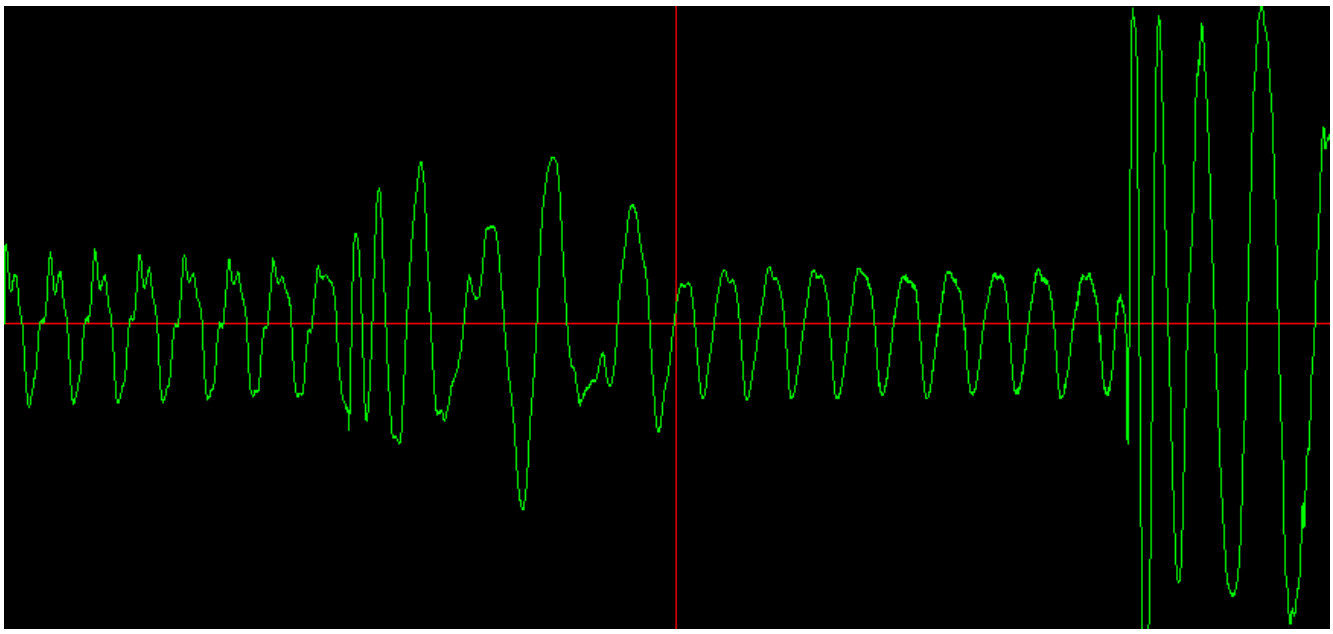
Dès le début du projet, tous les binômes étaient d'accord pour utiliser Microsoft Visual Basic 5.0 ®, parce que certains avaient soit déjà utilisé ou déjà vu manipuler et connaissaient les facilités extrêmes pour la création d'une interface graphique. Néanmoins, pour une personne novice, cet apprentissage est assez long (on passe trop de temps à chercher des fonctions par exemple). Aussi, avons-nous utilisé ce logiciel pour réaliser la partie graphique du projet PcScope. Voici quelques « screenshots » permettant de se donner une idée sur les fonctionnalités du programme :



Chargement du programme utilisateur sur le Kit



Menu de configuration du port de communication


Un exemple de résultat obtenu

2) Les facilités à implémenter :

Nous avons vu par exemple que pour déclencher l'échantillonnage, il fallait envoyer le caractère 'b', aussi a-t-on créé des boutons spécifiques qui envoient, lorsque l'on clique dessus, la chaîne voulue sur le processeur.
Il s'agissait ensuite de récupérer les échantillons envoyés par le Kit, lorsque nous le voulions, puis les traiter, et enfin les afficher.

Nous voulions que notre interface graphique puisse gérer l'envoi d'un programme utilisateur sur le processeur par un simple clic de souris, que l'on puisse travailler sur différentes parties du graphe obtenu, d'où l'implémentation d'une fonction de zoom, aussi bien en fonction du temps qu'en amplitude de signal reçu. Mais aussi et surtout rendre celle-ci conviviale et simple d'usage.

3) l'implémentation de ces fonctionnalités :

a) l'envoi d'un commande spécifique au Kit:

Pour cela nous utilisons des boutons, qui envoie une chaîne de caractères par un simple clic. Il suffit d'écrire une procédure Visual Basic (« langage » orienté objet) :

```
Private Sub Command1_Click()
'on veut que le kit nous envoie le contenu de son buffer
   If MSComm1.PortOpen Then
      MSComm1.Output = "b"
   Else
      MsgBox "vous devez tout d'abord ouvrir le port", 48
   End If
End Sub
```

Ce morceau de programme a pour effet que lorsque l'on clique sur le bouton associé, regarde si le port de communication est ouvert, et la chaîne « b » sur le port, et le cas échéance envoie un petit message à l'utilisateur lui disant d'ouvrir le port de communication.

Ainsi, pour envoyer autant de commandes que l'on veut, il suffit de créer autant de boutons que de commandes.

b) le décodage des valeurs reçues :

Lorsque l'on demande au kit de nous envoyer le buffer (plein d'échantillons), celui-ci nous envoie ces échantillons (codés sur 16 bits à l'origine) par paquets de 8 bits (2 par 2 + un octet de contrôle). Les valeurs brutes envoyées par le kit sont stockées dans un tableau de « long » au fur et à mesure de leur arrivée. Aussi, va-t-il falloir décoder les valeurs reçues par paquets de 3 octets dans ce tableau.

Toujours en intégrant les propriétés du protocole établit précédemment (octet de poids faible, octet de poids fort, octet de contrôle) nous recréons les valeurs numériques sortant du codec AD1847 grâce à la fonction decode1() :

```
Private Sub decode1()
'permet de décoder les valeurs reçues (8bits) en 16 bits
   Dim i As Long
   Dim t As Long
   Dim o As Long
   t = 0
   For i = 1 To j + 1 Step 3
```

```
    If buffer_affichage(i + 2) = 1 Then
       buffer_final(t) = buffer_affichage(i)
       buffer_final(t + 1) = buffer_affichage(i + 1)
    ElseIf buffer_affichage(i + 2) = 2 Then
       buffer_final(t) = 0
       buffer_final(t + 1) = buffer_affichage(i + 1)
    ElseIf buffer_affichage(i + 2) = 3 Then
       buffer_final(t) = buffer_affichage(i)
       buffer_final(t + 1) = 0
    ElseIf buffer_affichage(i + 2) = 4 Then
       buffer_final(t) = 0
       buffer_final(t + 1) = 0
    End If
    t = t + 2
  Next i
  o = 0
  For i = 0 To j + 1 Step 2
    buffer_decode(o) = buffer_final(i) + buffer_final(i + 1) * 256
    o = o + 1
  Next i

End Sub
```

La première boucle a pour but de déterminer si oui ou non un octet donné est à 0, la seconde permet tout simplement de mettre dans le tableau buffer_decode la valeur originelle numérique en sortie du codec : la multiplication par 256 d'un nombre codé sur 8 bits a pour effet de le décaler de 8 bits sur la gauche. Ainsi, en réalisant la seconde boucle pour 2 octets donnés, on rend un échantillon sur 16 bits.

Un fois cette fonction terminée, le tableau de valeurs buffer_decode contient des valeurs numériques, mais dans un format SPECIFIQUE : le format 1.15, c'est à dire que sur un échantillon donné codé sur 16 bits, le premier échantillon est le signe de l'échantillon, et les 15 autres bits la valeur entière.
Ce n'est qu'au moment de l'affichage que ces valeurs seront traduites, pour des raisons d'économies en espace mémoire.

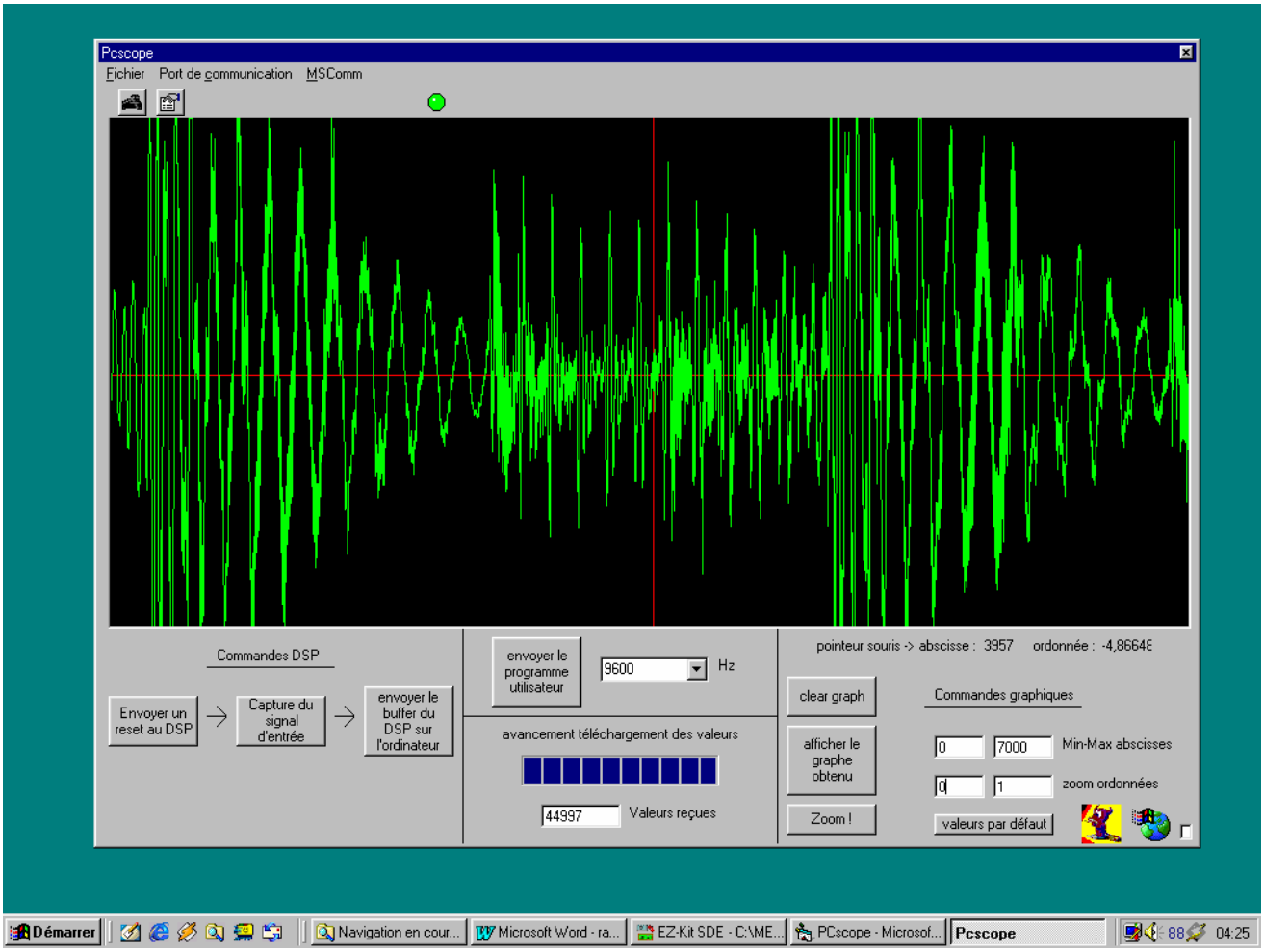
c) l'affichage des valeurs décodées :

C'est la fonction Command8_Click() qui s'occupe de réaliser cette fonctionnalité : elle puise ses valeurs dans le tableau buffer_decode() et transforme les valeurs 16 bits en valeurs au format 1.15 :
Les valeurs résultat sont comprises entre –1 et 0,999969. Donc peu de problèmes d'affichage : il suffit de multiplier cette valeur par la hauteur du graphique divisé par 2, et d'y ajouter une fois la hauteur du graphe divisé par 2 pour avoir un graphe centré. Tracer un ligne entre ces deux points, et le tour est joué.
(Voir le programme donné en annexe pour le programme source et l'implémentation de cet algorithme).

d) les autres fonctionnalités considérables comme « superflues »

Ces fonctionnalités sont par exemple l'affichage du numéro de d'échantillon que l'on est en train de regarder, ainsi que sa valeur en volt. Le zoom peut paraître lui aussi fastidieux mais permet des facilités accrues.

A préciser tout de même que ce programme est une adaptation de vbterm (un logiciel tel que hyperterminal permettant la connexion sur un modem aux services de l'Internet ou du minitel), une application exemple livrée avec le logiciel, nous y avons rajouter les fonctionnalités voulues et supprimées celles qui étaient superflues.
Un exemple de résultat obtenu :



P.S. : l'envoi sur le kit d'un programme utilisateur se fait par l'appel d'un autre petit freeware, ezld distribué gratuitement sur le site de Analog Devices.

Aussi voyons-nous que l'interface graphique n'a été un problème que dans le sens où il fallait l'adapter à nos exigences, et savoir la manipuler aisément.

Le projet ainsi réalisé nous nous sommes intéressés aux améliorations possibles d'un tel projet, quels pourraient être ses ouvertures, à la manière de les réaliser.
C'est ce que nous allons étudier dans l'ultime partie de ce rapport.

IV) Limitations, améliorations

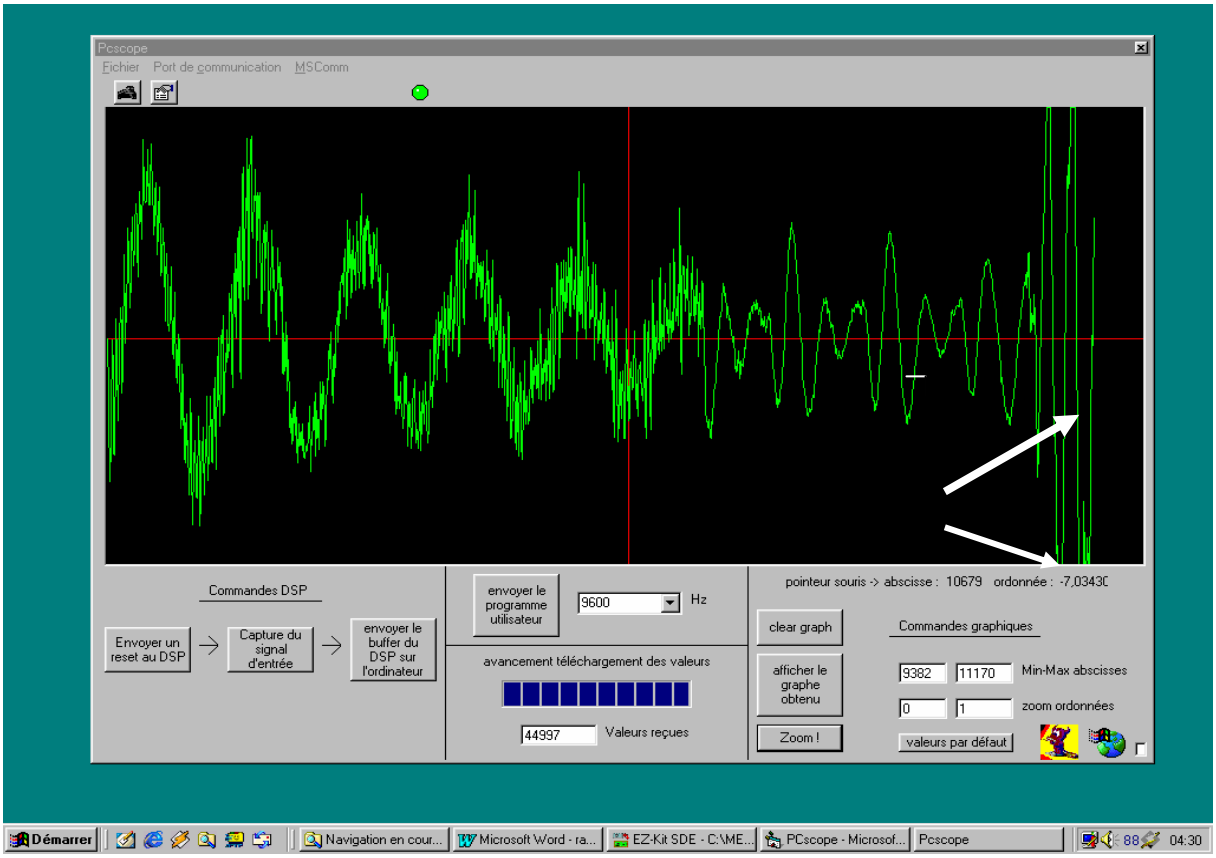
Comme tout autre projet, le projet réalisé trouve aussi ses limitations que ses ouvertures, et c'est en cela qu'un tel projet reste intéressant car nous n'avons jamais fini de l'améliorer, de lui implémenter des fonctionnalités aussi riches soient-elles.

A) Les limitations du travail réalisé :

Le travail réalisé propose des fonctionnalités de base, mais suffisantes pour les objectifs proposés ; cependant, on pourrait trouver long le temps utile au téléchargement des valeurs échantillonnées (44 s en moyenne …).

Mais les limitations ne sont pas seulement qu'au niveau de la liaison série ; l'interface graphique pose encore quelques problèmes : si l'on approche la souris trop près de la ligne d'abscisse, les valeurs du voltage obtenu sont fausses. Parfois quelques bugs apparaissent lors d'un zoom horizontal, la courbe n'est pas tracée d'un bout à l'autre du graphe. (voir exemple page suivante)

Exemple flagrant :



Ce sont les seuls bugs de l'interface graphique que pour l'instant nous avons recensé.

B) améliorations, ouvertures possibles et autres implémentations

Le gros avantage d'un tel projet est qu'il est ouvert :
En effet, nous avons utilisé une infime partie de toutes les fonctionnalités que propose un tel kit. Nous allons dans la suite de cette sous-partie voir quelles seraient de nouvelles implémentations, comment les réaliser.

1) Compression de données :

Comme exprimé précédemment, une compression de données peut être effectuée en temps réel sur le kit : cela aurait pour avantages de permettre l'allongement de la durée du buffer, de temps de transfert de données plus rapide, et surtout nous n'aurions pas à utiliser des octets de contrôle. Les inconvénients de cette compression de données ont été exprimés ci-dessus (voir page 12).

L'implémentation de tels algorithmes est assez facilement réalisable, et bien géré par le DSP (un programme implémentant la compression μ ne fait que 2ko, alors que celui pour la compression avec la loi A, quelques 30ko).

2) Pouvoir Zoomer sur la partie voulue :

En effet, dans son implémentation actuelle, le zoom n'est possible que sur l'axe des abscisses sans aucune limitation, et un zoom vertical mais seulement en gardant 0 comme référence : cela pourrait être changé afin de pouvoir zoomer sur n'importe quelle partie du graphique obtenu. Il suffirait pour l'affichage de ne garder que les valeurs qui sont comprises dans la plage définie par l'utilisateur.

3) Réalisation de calculs sur les signaux : FFT

En effet, la puissance de calcul du processeur ADSP 2181 (33 MHz, 31 Mflops) permet de calculer en temps réel la Transformée de Fourier Discrète du signal donné en entrée du codec. Il suffit pour cela d'implémenter cette fonction dans le code source du DSP. On demandera au kit –à volonté- de nous envoyer le buffer dans lequel le processeur aura évalué les échantillons donnés.

4) Pouvoir enregistrer la courbe obtenue, soit dans un fichier son, soit dans un fichier normal :

Si lors de l'utilisation de l'interface graphique, on arrive à avoir une courbe représentative qui nous intéresse pour pouvoir y travailler dessus, cela est impossible. Il faut donc sauvegarder les échantillons transmis par le kit dans un fichier, puis pouvoir les restaurer lorsque nous en avons besoin.

En fait, il ne serait pas plus difficile de rajouter dans ce fichier quelques entêtes pour qu'un player (un logiciel sachant « jouer » un fichier audio) puisse reconnaître le type de format du fichier et nous jouer la musique sur l'ordinateur …

Il suffirait au niveau de l'interface graphique de rajouter un bouton qui permette de choisir l'endroit de l'enregistrement du fichier, son format.
Puis, lorsque nous voudrions revoir ce fichier, il suffirait d'enlever les entêtes de fichier .wav, et d'interpréter la suite des octets comme un signal venant du DSP (même traitement donc même fonction).

5) Envoyer des échantillons numérisés sur l'ordinateur vers le Kit :

Cela aurait pour but que le kit joue la musique qu'on lui fait passer. Pour cela, et dans l'état actuel du programme assembleur, il faudrait donc envoyer tout d'abord une commande au kit pour qu'il se mette en mode d'attente des échantillons. Ensuite il s'agit d'envoyer tous les échantillons avec le « protocole » réalisé au-dessus, mais en sens inverse. Cette fois c'est à l'ordinateur de prendre un échantillon 16 bits , d'envoyer l'octet de poids faible, l'octet de poids fort et l'octet de contrôle. Ensuite il suffira de rajouter un test dans input_sample pour montrer que si une variable est positionnée alors on écrit sur tx_buf la valeur échantillonnée reçue.

6) Réaliser une interface temps réel :

La meilleure ouverture de ce projet est de réaliser une interface qui affiche en temps réel les échantillons reçus du Kit : pour cela, il faudrait que la communication PC - KIT soit plus rapide de manière à pouvoir envoyer au moins tous les échantillons créés.

L'envoi de ces échantillons sera réalisé dans la fonction input_samples : un échantillon est créé, il est aussitôt transmis sur la liaison série.

Aussi pouvons nous dire que les extensions ce projet sont multiples et touchent parfois à des domaines très variés.

Conclusion :

Malgré les limitations du travail réalisé, ce projet a été très bénéfique en lui-même car nous a permis de réaliser un travail de groupe.

Ce projet a été pour nous le moyen de tester nos connaissances au delà de l'enseignement pratiqué à l'ESIL. Tous les domaines ont été travaillés : aussi algorithmique que architecture des ordinateurs. Il nous a permis de mieux comprendre comment fonctionne un processeur de type ADSP, mieux cerner les problèmes liés à la mise en place d'un kit comme celui-ci, mais aussi de nous ouvrir aux possibilités incroyables qu'est capable de réaliser une carte comme celle-ci.