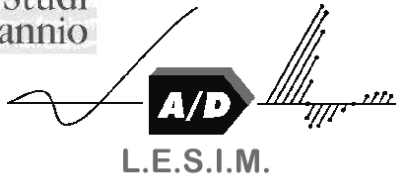
UNIVERSITÀ DEGLI STUDI DEL SANNIO

ELABORAZIONE DEI SEGNALI E DELLE INFORMAZIONI DI MISURA
2007/2008

CHORUS EFFECT

SVILUPPO DI SOFTWARE PER L'ACQUISIZIONE E L'ELABORAZIONE
IN TEMPO REALE DI SEGNALI SU DSP.



BORRIELLO ALESSANDRO
mat.195000713

LAMONICA MICHELE
mat.195001051

PORFIDO MICHELE
mat.195001039

RUBINO MARCELLO
mat.195001092

CHORUS EFFECT *PRESENTAZIONE*

Esattamente come un coro è un insieme di voci, il Chorus Effect applicato ad uno strumento musicale è in grado di riprodurre il segnale non facendo più percepire il singolo strumento, ma dando l'impressione del suono contemporaneo (o quasi) di diversi dispositivi.

Il Chorus viene spesso utilizzato per potenziare una traccia vocale o aggiungere spaziosità stereo a un audio mono, difatti l'effetto aggiunge spessore al suono e lo rende più ricco, é per tale motivo che l'applicazione del Coro è anche definita "arricchimento".

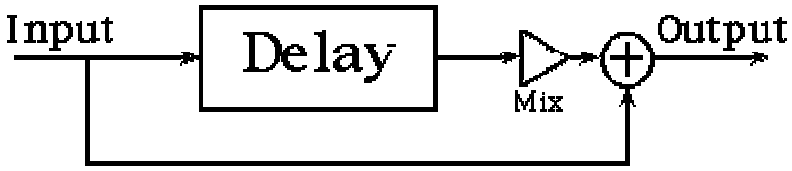
❖ COME FUNZIONA

Cosa accade quando due persone suonano uno strumento all'unisono?

I due suoni non sono perfettamente sincronizzati e tra essi c'è un ritardo, che per giunta non è mai lo stesso. È esattamente questo il risultato che attendiamo dopo l'applicazione dell'effetto su un suono singolo.

Il ritardo desiderato tra i suoni può essere ottenuto mediante una linea di ritardo a lunghezza variabile; dove con lunghezza variabile s'intende che il ritardo non è mai lo stesso ma cambia col passare del tempo (questo è un concetto che ora non può essere tanto chiaro, ma che capiremo meglio poi).

Per meglio rendere la situazione possiamo rappresentarla mediante un diagramma a blocchi:

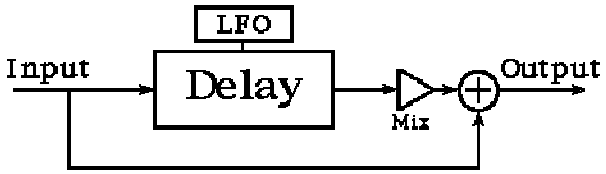


Dal diagramma si più intuire il funzionamento del Chorus Effect, in particolare, saltano all'occhio, tre elementi: il delay (ritardo) che ritarda il segnale in ingresso con una dilatazione temporale che oscilla tra i 20 ed i 30 ms, un guadagno, ed un nodo sommatore che sovrappone il segnale ritardato al segnale in ingresso dando l'impressione che il suono non sia più singolo ma doppio (è il caso del diagramma in questione, è possibile aggiungere un numero indefinito di ritardi, aumentando in tal modo il numero dei suoni percepiti in uscita).

L'unico punto ancora da discutere è in che modo oscilli la dilatazione temporale del ritardo.

In generale, abbiamo bisogno di una forma d'onda periodica (come ad esempio un segnale sinusoidale) che abbia una frequenza lentamente variabile, con frequenze pari o inferiori ai 3Hz. Possiamo utilizzare, per produrre una forma d'onda con frequenza lentamente variabile, un ***Low Frequency Oscillator*** (**LFO** – oscillatore a bassa frequenza).

È quindi possibile controllare il suono del coro cambiando la forma d'onda del ritardo in frequenza, in ampiezza o in forma.

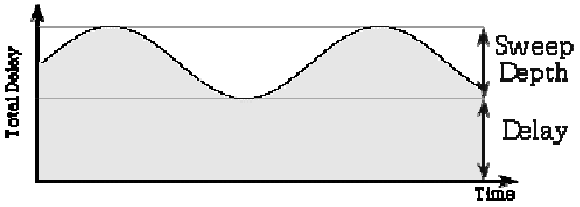


❖ PARAMETRI

Delay (*ritardo*): Il ritardo controlla semplicemente la dilatazione temporale utilizzata, come già detto, essa oscilla tra i 20 e i 30 ms.

Sweep Depth: controlla di quanto il ritardo totale varia nel tempo, usualmente viene espresso in millisecondi, e di fatti la somma di Sweep Depth e Delay ci da il massimo ritardo utilizzato nell'elaborazione del segnale. In alternativa si può pensare allo Sweep Depth come all'ampiezza dell'LFO.

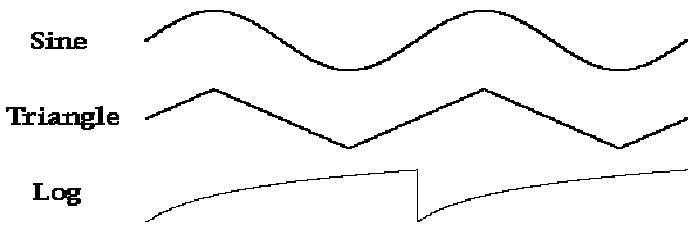
La relazione tra Sweep Depth e Delay può meglio essere apprezzata nell'immagine seguente:



Lo Sweep Depth inoltre aumenta anche la modulazione del tono introdotta dal ritardo tempo variabile; quando lo Sweep Depth è ampio accade che il suono in uscita presenti un effetto simile ad un "gorgheggio".

LFO Waveform (*forma d'onda*): ci mostra il modo in cui il ritardo varia nel tempo. Quando la forma d'onda raggiunge un massimo anche il ritardo raggiungerà il suo picco. Quando la forma d'onda è crescente il tono risultante tende ad essere più basso.

Le tre forme d'onda, generalmente, più utilizzate sono l'onda sinusoidale, quella triangolare e la logaritmica (tutte e tre riportate in figura).



La modulazione del tono introdotta dall'effetto coro, è relativa a quanto velocemente vari la forma d'onda utilizzata, in corrispondenza di una variazione più grande della forma d'onda viene prodotta una più grande modulazione del tono, mentre le posizioni

relativamente piatte lungo la forma d'onda producono una modulazione del tono molto piccola o addirittura nulla.

Da quanto appena detto, possiamo meglio capire come lo Sweep Depth incida sulla modulazione del tono: più si allunga verticalmente la forma d'onda più il tono viene alterato.

La sinusoide è una funzione molto fluida che varia sempre, per cui anche il tono cambierà costantemente in modo fluido, l'onda triangolare ha soltanto due inclinazioni che producono altrettanti toni, che tra loro si alternano in modo repentino, in fine la logaritmica cresce ciclicamente in modo fluido con un brusco salto alla fine di ogni periodo : il tono varia sempre lungo tutto il periodo dell'onda fino ad una consistente variazione in coincidenza del salto.

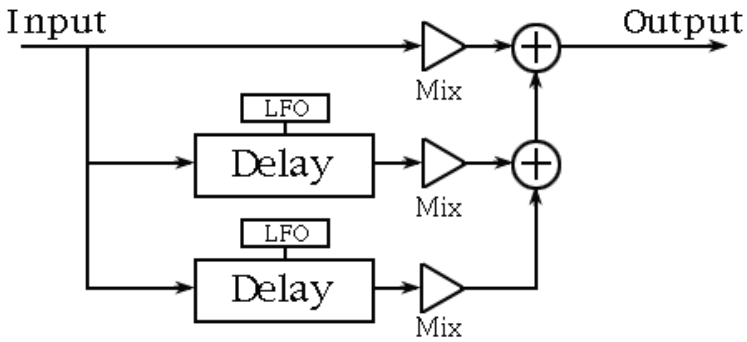
Speed/Rate (*velocità*): il controllo della velocità è abbastanza lineare, si riferisce a quanto in fretta la forma d'onda si ripeta, ed inoltre incide notevolmente sulla modulazione del tono. Aumentare la velocità equivale a comprimere in durata il periodo dell'onda.

Numero di voci: fino a questo punto, abbiamo visto come, da un singolo suono in ingresso, è possibile ottenere un suono doppio (è presente un singolo ritardo); ma non c'è ragione per cui non sia possibile desiderare che l'output riproduca un numero di suoni sovrapposti, superiore a due, ossia che invece di essere soltanto due strumenti a suonare ce ne sia un numero imprecisato.

Tipicamente per implementare un coro multi-voce si utilizza una singola forma d'onda che però assume una fase diversa per ogni ritardo, questo significa che in uno stesso istante di tempo ogni voce è in un punto diverso della forma d'onda, in modo che tutte le voci abbiano diverso ritardo.

Se tutte le voci fossero in fase il ritardo sarebbe lo stesso, come se si utilizzasse un singolo ritardo con un accresciuto guadagno (le singole voci si sovrappongono aumentando il livello della voce in uscita).

Chiaramente è anche possibile creare un coro in cui ogni voce utilizzi una sua forma d'onda personalizzata, logicamente il risultato finale è leggermente diverso.



❖ IMPLEMENTAZIONE

Analogica: per implementare il ritardo vengono utilizzati circuiti puramente elettrici, come ad esempio il *Sample-and-Hold* oppure il *Bucket-Brigade*, questi dispositivi assumono come input una tensione (proveniente dallo stesso segnale in ingresso) per un periodo di tempo limitato.

Più circuiti possono essere concatenati in serie per poter ottenere il ritardo desiderato.

Digitale: i ritardi vengono semplicemente implementati con buffer circolari (presa una porzione di memoria vi si scrivono i valori campionati della forma d'onda in ingresso, per poi andarli a leggere nel corso dell'elaborazione).

È anche possibile implementare il ritardo mediante tecniche di interpolazione, che risultano concettualmente semplici ma possono introdurre una certa quantità di rumore.

Dopo aver brevemente presentato il risultato atteso, ripercorrendo tutti i passaggi, passiamo alla nostra realizzazione del Chorus Effect.

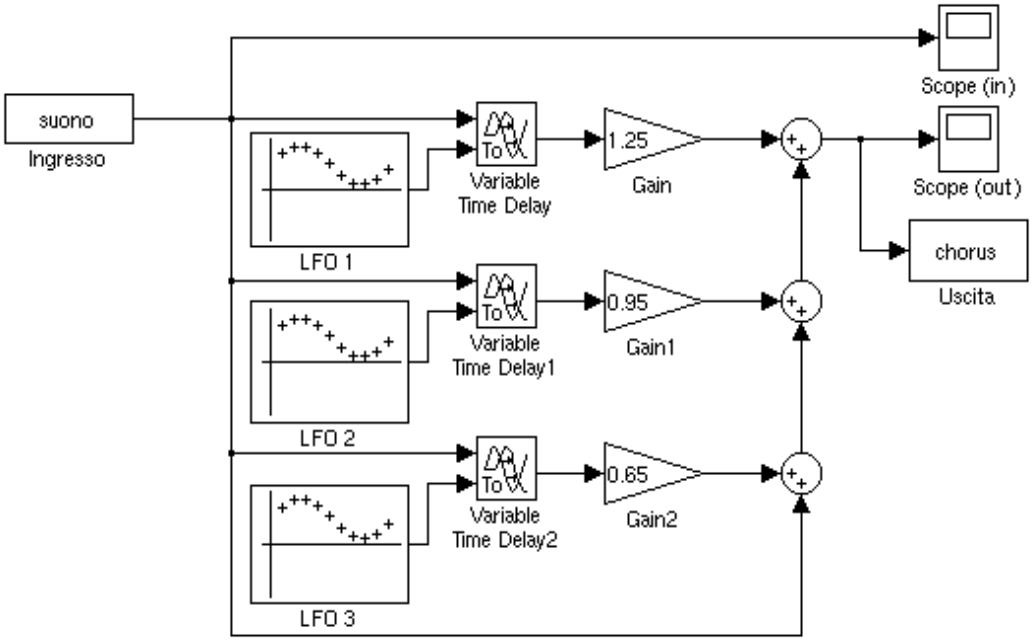
❖ SIMULAZIONE & DEFINIZIONE DEI PARAMETRI

Avvalendoci del supporto Matlab, abbiamo accuratamente definito la specifica del progetto e grazie ad una serie di simulazioni iterative abbiamo attribuito ai parametri i valori che meglio assolvono i nostri scopi.

Come in precedenza detto bastano due voci per ricreare il Chorus Effect, da parte nostra siamo partiti con l'intenzione di riprodurre un coro col massimo numero di voci possibile: ben presto abbiamo constatato che oltre le quattro voci l'effetto risultava alquanto confuso e quindi di nessuna utilità, di fatti non arricchiva il suono ma lo distorceva rendendolo a tratti irriconoscibile e sgradevole all'ascolto.

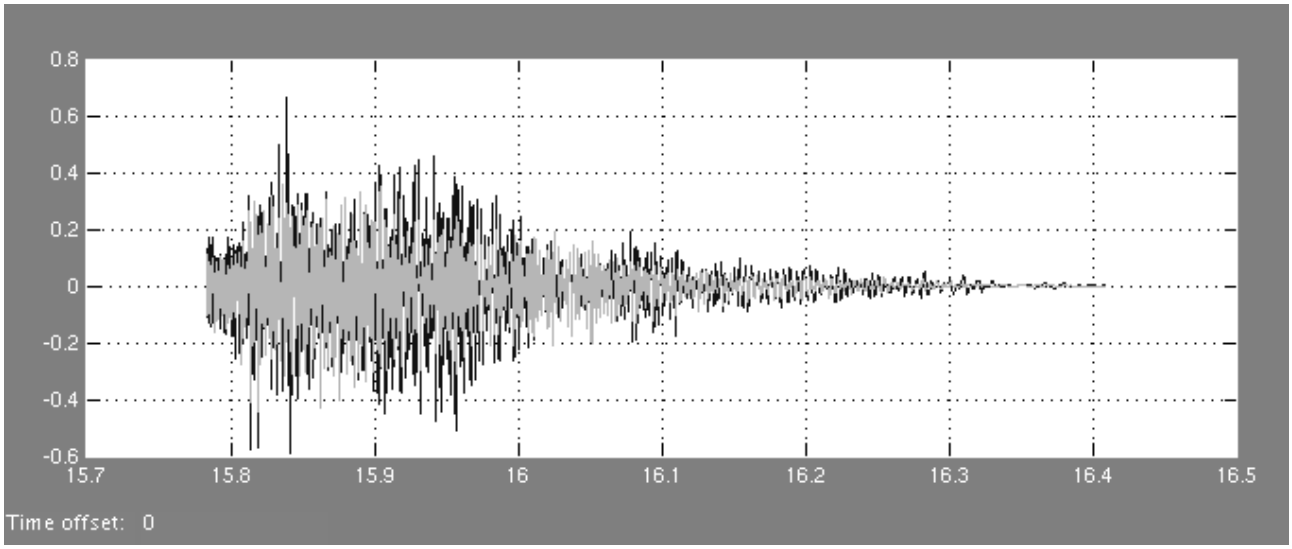
Delle quattro voci la principale è senza dubbio quella non ritardata, in effetti fa da "guida" al segnale in uscita, le altre tre vengono ottenute dalla prima con ritardi variabili e guadagni differenti.

Lo schema seguente riassume in modo snello e diretto le scelte fatte:



Come ultimo dettaglio da non dimenticare la frequenza della forma d'onda fissata sui 3Hz.

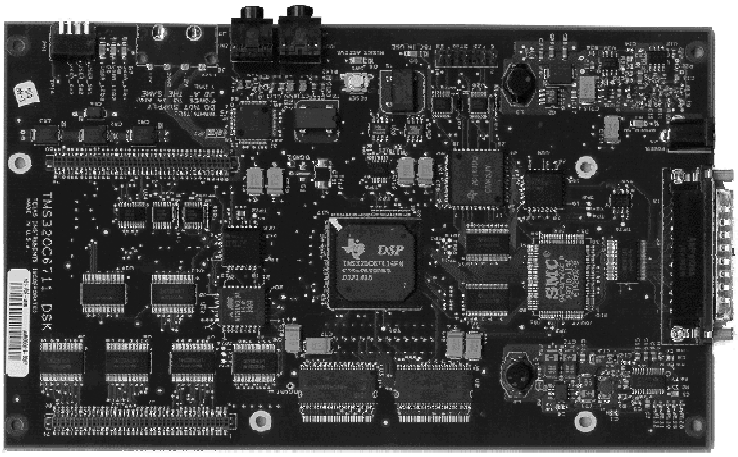
Il seguente grafico illustra la differenza tra un suono in ingresso (pochi secondi di un brano suonato esclusivamente con chitarra acustica) e l'uscita a cui è applicato il Chorus Effect:



In chiaro l'ingresso e con gradazioni più scure l'uscita.

❖ HARDWARE UTILIZZATO

Per la concreta realizzazione del progetto è stato utilizzata una scheda DSK con processore DSP e più precisamente: *TMS320C6711 DSK* n° 0247001347 prodotto dalla *Texas Instruments*, di seguito è possibile osservarne un'immagine:



Prima di procedere nell'implementazione del Chorus Effect diamo un'occhiata alla nostra scheda DSK per notare alcune delle sue principali caratteristiche.

Caratteristiche principali del TMS320C6711:

- processore DSP progettato per l'elaborazione real time e ottimizzato per operazioni di streaming dati, con 8 unità funzionali (due moltiplicatori e 6 unità aritmetiche, logiche e di controllo) distribuite su 2 datapath;
- frequenza di clock 150 MHz;
- nessun supporto hardware per funzionamento in multitasking;
- 16 MB SDRAM;
- 128 KB Flash ROM;
- possibilità di effettuare l'accesso diretto in memoria (DMA);
- esegue fino a 8 istruzioni in parallelo in un ciclo;
- esecuzione condizionale di tutte le istruzioni;
- possibilità di calcoli a 40 bit;
- supporto hardware per istruzioni floating-point IEEE single-precision (32 bit) e double-precision (64 bit);
- moltiplicazione intera 32x32 bit con risultato a 32 o 64 bit.

Un breve accenno sulle periferiche di cui la scheda DSK è dotata:

EDMA (**E**nhanced **D**irect **M**emory **A**ccess) **Controller**:

scambia dati tra blocchi di memoria senza l'intervento della CPU. Ha 16 canali indipendentemente programmabili e uno spazio in memoria per le configurazioni;

HPI (**H**ost **P**ort **I**nterface):

una porta parallela attraverso cui una CPU host può accedere direttamente allo spazio di memoria del DSP ed ha accesso alle periferiche mappate in memoria;

Bus d'espansione:

ha due diverse funzioni: la porta host può operare come porta slave asincrona (come HPI) oppure come master/slave sincrona utile per interfacciarsi con vari host bus protocols, la I/O port può interfacciarsi con dispositivi periferici di I/O FIFO sincroni o asincroni;

EMIF (**E**xternal **M**emory **I**nter**F**ace):

è una particolare interfaccia per diversi dispositivi esterni come memorie esterne, SDRAM, ROM, ecc…

McBSP (**M**ulti**c**hannel **B**uffered **S**erial **P**ort):

un'interfaccia seriale che può trasferire un dato da e per la memoria con l'uso del DMA o attraverso interrupts alla CPU. Consente comunicazioni full-duplex con registri doppiamente bufferizzati fino a 128 canali, trasferisce dati da 8, 12, 16, 20, 24 e 32 bit ed ha companders A-low e μ-low. Si può facilmente interfacciare con vari dispositivi (es. codecs, bus seriali);

Timers e GPIO (**G**eneral **P**urpose **I**/**O**):

Il DSP ha timers a 32 bit per uso generale come temporizzatori, contatori, generatori di impulsi e interrupts per CPU e DMA. La periferica GPIO fornisce un insieme dedicato di segnali di I/O per il DSP; le transizioni di questi segnali possono essere usati per generare interrupts e sincronizzare CPU e DMA.

Tutte le periferiche elencate generano interrupts ed eventi per il DSP e per il DMA ed hanno registri di controllo mappati in memoria. Inoltre, il DSP può effettuare la fase di boot a partire anche dalle periferiche esterne (ROM o flash esterne, CPU host, ecc…).

❖ *DETTAGLI IMPLEMENTATIVI*

Conclusa la fase di analisi e simulazione è possibile passare alla parte concreta del progetto ossia la scrittura del codice.

Avendo analizzato l'hardware è immediatamente intuibile che le risorse a nostra disposizione sono di gran lunga superiori a quelli che sono gli scopi del progetto, in effetti la scheda DSK è in grado di compiere elaborazioni real time a frequenze molto elevate, risultando molto performante e spropositata per la realizzazione di un "semplice" effetto coro. Dal punto di vista strettamente ingegneristico, l'utilizzo di un siffatto hardware rappresenta un vero e proprio spreco, ma allo stesso tempo facilita altamente la progettazione non imponendo grossi vincoli e lasciando ampia flessibilità, eliminando in tal modo il problema dei vincoli strutturali che generalmente costituiscono il lavoro più consistente nella realizzazione di un progetto.

La stesura del suddetto codice ha avuto come riferimento la simulazione effettuata col supporto Matlab.

Il cerchio delle questioni da risolvere si è in tal modo ristretto a soli due problemi:

- In primo luogo come gestire l'input e l'output della scheda DSK, in verità non si è trattato di un vero e proprio problema, visto e considerato il grosso supporto fornito dalle librerie preesistenti.

  Per prelevare i campioni dalla scheda audio è stato configurato l'interrupt HWI9, questo interrupt viene generato ogni qual volta è disponibile in ingresso sulla porta McBSP0 un nuovo campione da processare.

  Per la lettura dei campioni in ingresso è stata usata la funzione *AD535_HWI_read*, e allo stesso modo per la scrittura del campione risultante dall'elaborazione si è utilizzata la funzione *AD535_HWI_write,* entrambe contenute nella libreria *BSL.*

- Sicuramente di maggiore rilievo la generazione dei ritardi grazie ai quali è possibile ottenere il Chorus Effect.

  E' stato necessario creare un buffer circolare di dimensione pari a 256 campioni; la scelta di questo numero non è stata completamente arbitraria, per realizzare l'effetto sarebbe bastato un buffer di 240 campioni (valore calcolato in base al massimo ritardo richiesto, ed al periodo di campionamento) ma utilizzando 256 campioni la realizzazione risulta estremamente più semplice poiché richiede il solo utilizzo di una *AND* logica col valore 255 sull'indice di accesso.

  ```
  ...

  delayed_sample = (CHORUS_GAIN*input_buf[(input_idx-samples_delay) & 255]);

  ...
  ```

  Come detto, l'implementazione prevede 3 voci ritardate sovrapposte all'ingresso corrente. Il ritardo viene realizzato spostandosi all'indietro nel buffer di un valore variabile nel tempo, tale valore viene generato da un LFO software basato su una sinusoide discreta con frequenza di 3Hz i cui valori estremi sono il numero massimo e minimo di campioni da ritardare (240-160 campioni, corrispondenti a 30-20 ms).

  La sinusoide discreta è la Z-trasformata della corrispondente in analogico con una frequenza di campionamento a 8KHz.

❖ IL CODICE

Di seguito il codice che implementa l'effetto.

FILE: **chorus.h**

```
#define VOLUME 3
#define CHORUS_GAIN 1.25
#define SAMPLE_FREQ_KHZ 8
#define MIN_DELAY_MS 20
#define MAX_SWEEP_DEPTH_MS 10

//   Costanti per il calcolo del numero di campioni necessari per il ritardo
//   variabile sinusoidale

#define MIN_DELAY_SAMPLES SAMPLE_FREQ_KHZ*MIN_DELAY_MS
#define MAX_SWEEP_DEPTH_SAMPLES SAMPLE_FREQ_KHZ*MAX_SWEEP_DEPTH_MS

//   Costanti per il calcolo della dimensione del buffer minima

#define MAX_DELAY_MS MIN_DELAY_MS+MAX_SWEEP_DEPTH
#define BUFFER_SIZE 256
```

FILE: **sine_float.c**

```
#include "chorus.h"             // definizione delle costanti

//   Il coefficiente A ed i tre valori iniziali generano un LFO (Low Frequency
//   Oscillator) a 3Hz quando si lavora ad una frequenza di campionamento di 8KHz

float y[3] = {0,0.002356192,0};
float A = 1.999994448;

//   Restituisce il ritardo variabile generato dall'LFO espresso in numero di
//   campioni

short sineDelayGen(void) {
     y[0] = y[1] * A - y[2];
     y[2] = y[1];
     y[1] = y[0];

     return((short)((MAX_SWEEP_DEPTH_SAMPLES/2)*y[0]))+
                       +(MAX_SWEEP_DEPTH_SAMPLES/2)+ MIN_DELAY_SAMPLES;
```

FILE: **chorus.c**

```
/*   Prototipi   */

void codec_out(short sample);
void codec_init(void);
void init_HWI(void);
void emif_init(void);
short sineDelayGen(void);


/*   Include   */

#include <c6x.h>                // definizioni del compilatore C6000
#include <csl.h>                // header CSL
#include <csl_irq.h>
#include <csl_mcbsp.h>
#include <bsl.h>                // header BSL
#include <bsl_ad535.h>
#include <bsl_suggestions.h>

#include "chorus.h"             // definizione delle costanti


/*   Variabili globali   */

//   AD535 Handle e Config structure

AD535_Handle hAD535;
     AD535_Config my_AD535_Config = {
     AD535_LOOPBACK_DISABLE,
     AD535_MICGAIN_OFF,
     AD535_GAIN_0DB,
     AD535_GAIN_0DB
};


//   Buffer circolare

short input_buf[BUFFER_SIZE];   // minimo 240
short input_idx = 0;            // indice del buffer



/*    Main Routine   */

void main(){
     int i;
     CSL_init();                // Inizializza la libreria CSL
     BSL_init();                // Inizializza la libreria BSL
     codec_init();
     init_HWI();

//   Viene scritto qualcosa per far iniziare McBSP a trasmettere interrupt

     AD535_write(hAD535, 0);

//   Il buffer viene inizializzato prima di procedere (azzerandolo)

     for (i=0; i<BUFFER_SIZE; i++)
     input_buf[i]=0;

     while(1);
}
```

```
/*   Interrupt hardware – inizializzazione   */

//   Il codec AD535 è collegato fisicamente all'McBSP0. Quindi, il sistema utilizzerà
//   l'interrupt McBSP0 di trasmissione (XINT0) per segnalare quando scrivere il
//   prossimo campione in uscita
//   XINT0 è stato mappato sull'interrupt 9 (HWI9) tramite il tool di configurazione


void init_HWI(void){            // Attiva tutti gli interrupt

     IRQ_globalEnable();        // Attiva l'interrupt McBSP0 di trasmissione
     IRQ_enable(IRQ_EVT_XINT0);
}


/*   Codec Functions   */

void codec_init(){


//   Vengono utilizzate le routine BSL per l'open, il reset ed il config dell'AD535
//   parte della routine AD535_open() apre e configura l'McBSP0, visto che tutte le
//   comunicazioni col codec AD535 sono gestite via McBSP0.

     hAD535 = AD535_open(AD535_localId);    // Apre l' AD535 e ne prende l'handle
     AD535_reset(hAD535);                   // Resetta l' AD535
     AD535_config(hAD535,&my_AD535_Config); // Configura l' AD535


//   Viene imposto un bit nel registro McBSP, in modo tale da farlo funzionare anche
//   quando cessa l’attività del debugger; ciò evita la corruzione del codec

     MCBSP_setfree(0);
}



/*   Mette in uscita al codec AD535 il campione passato   */

void codec_out(short sample) {
     AD535_HWI_write(hAD535, sample);
}


/*   Preleva il campione disponibile dal codec AD535   */

short codec_in() {
     return AD535_HWI_read(hAD535);
}


/*   Si é verificato l'interrupt (é disponibile un nuovo campione)   */

void XINT0_HWI() {

     float delayed_sample;      // Contiene i campioni ritardati

     short samples_delay;       // Contiene il ritardo in numero di campioni

     short current_sample=codec_in();
                                // contiene il campione corrente

     input_buf[input_idx] = current_sample;
                                //Il campione corrente viene inserito nel buffer
```

```
//   In questo istante l'LFO viene ritardato (in numero di campioni)

     samples_delay = sineDelayGen();


//   Il campione ritardato viene prelevato dal buffer circolare, per poi essere
//   amplificato di CHORUS_GAIN
//   Viene utilizzata una AND per realizzare un indice circolare, per valori negativi
//   (es. : 0 - 160 = 255 - 159 = 94 )

     delayed_sample=(CHORUS_GAIN*input_buf[(input_idx-samples_delay) & 255]);



//   Al campione ritardato, viene sommato un altro campione ritardato di 1 ms
//   (8 campioni), con un guadagno attenuato di 0,3

     delayed_sample+=((CHORUS_GAIN-0.3) *
                              input_buf[(input_idx-(samples_delay-8)) & 255]);



//   Ai due campioni ritardati, viene sommato un ulteriore campione ritardato di 2 ms
//   (16 campioni), con un guadagno attenuato di 0,6

     delayed_sample+=((CHORUS_GAIN-0.6) *
                             input_buf[(input_idx-(samples_delay-16)) & 255]);



//   Si avrà in uscita il campione corrente sommato alle tre componenti ritardate,
//   il tutto amplificato di VOLUME (ciò rende più udibili i suoni)

     codec_out(VOLUME * (current_sample+((short) delayed_sample)));



//   Indice circolare che realizza l'accesso al buffer

     input_idx = (input_idx + 1) & 255;
}
```

❖ TEST & ULTIMI ACCORGIMENTI

La fase finale del progetto prevede la verifica dei risultati ottenuti; da subito, nonostante qualche capriccio da parte della scheda DSK e alcune correzioni di rito, i test hanno avuto risultati positivi.

Il riscontro finale, nel complesso, è apparso soddisfacente.

CHORUS EFFECT *MANUALE D'UTILIZZO*

Di seguito le istruzioni per chi, avvalendosi del progetto da noi realizzato, decidesse di applicare in tempo reale ad un qualsiasi suono il Chorus Effect:

- collegare la scheda *DSK* al PC tramite la porta parallela;
- collegare l'uscita audio della scheda (con il contrassegno *OUT*) ai diffusori;
- collegare all'ingresso audio della scheda (con il contrassegno *IN*) il microfono o un qualsiasi apparecchio (come uno stereo o lo stesso PC);
  *[**N.B.** Tenere presente che l'ingresso della scheda DSK è MONO]*
- collegare alla scheda l'apposito alimentatore;
- sul PC in uso avviare il kit di sviluppo software *CCS* (Code Composer Studio DSK v2.10.00);
- aprire il file di progetto presente nella cartella *progettoChorus* (*Project->Open* dal menù del CCS);
- compilare il progetto (premere il pulsante *Build All*);
- programmare la scheda caricando in memoria il codice binario risultato dalla compilazione appena effettuata (*File->Load Program);*
- far eseguire dalla scheda il programma caricato (il nostro *Chorus Effect*) selezionando sempre dal menù *Debug->Run;*
- da questo momento tutto quello che verrà acquisito dal *DSP* verrà riprodotto in uscita simulando un coro;
- è possibile terminare l'effetto dal menù *Debug->Halt.*