Pedaleira Digital de Efeitos Para Guitarras

Gabriel da Silva Soares
Faculdade do Gama
Universidade de Brasília
Gama - DF, Brasil

Ricardo Vieira Borges
Faculdade do Gama
Universidade de Brasília
Gama - DF, Brasil

I. RESUMO

O projeto proposto consiste na implementação de uma pedaleira na placa Raspberry Pi 3B que adicione efeitos em tempo real ao som de uma guitarra, buscando a inovação no que se refere a efeitos já existentes no mercado.

II. INTRODUÇÃO

Um pedal ou pedaleira digital é um equipamento capaz de modificar o sinal natural proveniente de uma guitarra, baixo, violão e etc. (cada equipamento voltado para o instrumento utilizado). A ressonância das cordas do instrumento é captada pelos captadores que geram um sinal elétrico que é apenas amplificado ou modificado pela peladeira para depois ser amplificado dependendo da necessidade. Estes são bastantes utilizados em apresentações ou estúdios para gravação e basicamente adicionam efeitos ao sinal por meio de um microprocessador embarcado no equipamento, dependendo da modificação desejada, são adicionados os tipos de filtros aplicados no sinal, que são representados por equações implementadas digitalmente no computador, após a passagem do sinal por estes filtros, é disponibilizada uma saída com o sinal modificado para o amplificador.

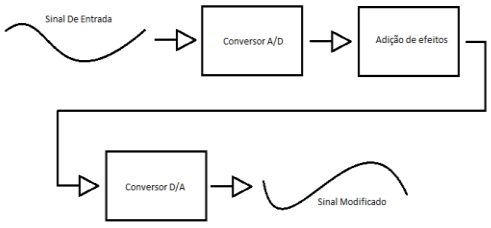

Figura 1 – caminho do sinal.

III. DESENVOLVIMENTO

Primeiramente optou-se pela realização de diversos testes com sinais genéricos de guitarra (formato .WAV) em softwares consolidados como Matlab e Scilab, tentando adicionar edições simples ao sinal como multiplicar por funções, aplicar filtros como o de média móvel, funções de atraso e etc, gerando outro arquivo de mesmo formato, reproduzindo e observando se o resultado sonoro é satisfatório.

No anexo 1 há um código que demonstra a modificação de um sinal por modulação em amplitude e aplicação de raiz quadrada, dessa forma, a saída Y é a combinação do sinal atual e sua cópia atrasada seguindo a equação:

$$Y = \sqrt{x_1 * \cos(\omega_1 t)} + \sqrt{x_1 Z^{-n} * \cos(\omega_2 t)}$$

Alguns efeitos sonoros foram estudados, tais como chorus, delay, flanger, vibrato e os mais comuns [2] para que os efeitos criados tenham alguma distinção dos já existentes.

Nos testes de efeitos sonoros propostos foi utilizado apenas um notebook contendo o software Scilab para modificação dos sinais em formato .WAV.

Para a aquisição do sinal sonoro pela raspberry é necessário um conversor analógico-digital com uma resolução mínima de 12 bits e frequência de amostragem mínima de 10KHz, já que os sinais de guitarra não ultrapassam os 5KHz e a amostragem foi feita seguindo o critério de Nyquist. No mercado não foi encontrado algo que cumprisse a necessidade e que fosse de fácil acesso, então optou-se por utilizar o conversor AD de uma launchpad Tiva-C (microcontrolador tm4c123), que atende às necessidades.

A conversão AD foi feita utilizando um pino da placa Tiva-C, lendo os valores analógicos e os

convertendo para uma faixa de 0 até 4095 unidades em binário, com taxa de amostragem de 125kHz, totalizando os 12 bits e os comunicando com o Raspberry-Pi.

Por ter utilizado o AD de um microcontrolador, foi necessário optar por uma forma de comunicação. Foi escolhido o protocolo SPI por conseguir manter altas velocidades de transmissão com baixas taxas de erros, pelo fato de ser uma comunicação síncrona. O anexo III mostra o código de implementação da conversão AD e da comunicação SPI na placa Tiva-C. O anexo II mostra o código utilizado na raspberry para ler os valores recebidos pelo SPI e acender ou apagar dois LEDs de acordo com o valor lido no AD, para testar o funcionamento da comunicação.

Outra preocupação foi também a saída de áudio. A intenção é de se usar as próprias ferramentas da placa para fazer a conversão de bits para áudio novamente. Dessa forma, optou-se pelo uso da ALSA – acrônimo para Advanced Linux Sound Architecture, que possui uma série de bibliotecas e aparatos que possibilitam o controle da saída de áudio da placa.

IV. RESULTADOS

É evidente que o processamento em tempo real envolve outras dificuldades que não foram transpassadas com o teste em arquivo pronto, porém, o intuito da simulação computacional é descobrir efeitos úteis e interessantes para posterior aplicação no equipamento.

Depois de diversos testes de aplicações de funções matemáticas ao sinal, percebeu-se que este só começou a demonstrar diferenças sonoras significativas com a adição de funções trigonométricas ou atrasos.

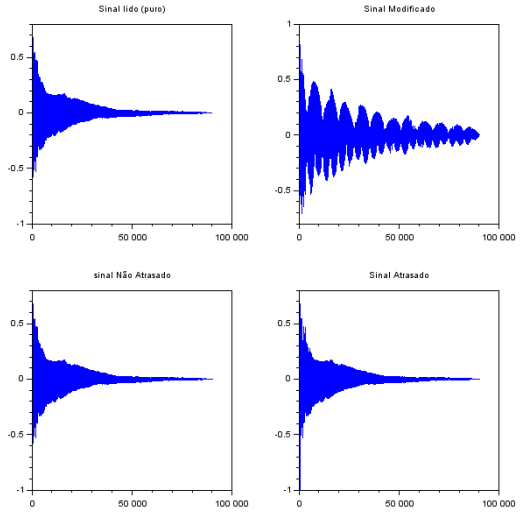

Figura 2: Demonstração gráfica do sinal modificado pela expressão Y.

A leitura analógica foi testada utilizando apenas um potenciômetro, em uma ferramenta de debug para verificação dos resultados. Os resultados mensurados foram bastante satisfatórios quando comparados a tensão lida no pino analógico, comprovando um bom funcionamento do conversor analógico, com taxa de amostragem apropriada para o sinal final utilizado no projeto (cerca de 5kHz), restando apenas a comunicação SPI em modo escravo que também apresentou bom funcionamento quando testada.

O SPI possui um módulo no kernel de fácil manuseio na placa Raspberry Pi, porém, ele só suporta palavras de 8 bits na transmissão e recepção e o projeto necessita de palavras de 12 bits. Dessa forma, para ler os 12 bits executa-se duas leituras, e então estas são armazenadas num mesmo buffer. Porém, a segunda leitura é deslocada 8 vezes para que os bits mais significativos sejam dispostos no lugar certo.

A biblioteca asoundlib.h da ALSA foi utilizada para escrever um programa que lê um arquivo .WAV, necessitando da taxa de amostragem, número de canais e tempo a ser tocado. O resultado foi bem positivo, mas ainda falta conseguir utilizar a biblioteca para áudio em tempo real.

V. REVISÃO BIBLIOGRÁFICA

[1] Mitch Gallagher. *Guitar Tone: Pursuing the Ultimate Guitar Sound*. Cengage Learning; p. 81.

[2] A. W. FRANÇA. *Uso de Processamento Digital de Áudio na Implementação de Efeitos em Instrumentos Musicais*. Julho de 2015.

http://bdm.unb.br/bitstream/10483/13268/1/2015_AndreWagnerFranca.pdf

[3] T. Jeff. *Introduction to sound programming with ALSA.* Setembro de 2004 *https://www.linuxjournal.com/article/6735?page=0,1*

[4] Alsa. *PCM (Digital Audio) Interface*. http://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html

[5] Documentação da spidev.

https://www.kernel.org/doc/Documentation/spi/spidev

[6] A. Paterniani. *spidev code example.*

https://elixir.bootlin.com/linux/v3.5/source/include/linux/spi/spidev.h#L114

[7] Jonatham W. Valvano. Design de Sistemas Embarcados. Disponível em: http://users.ece.utexas.edu/~valvano/

ANEXO I

```
n = 94803; //quantidade de amostras
T = n - 1; //tamanho do vetor
step = 1000; //define a precisão
t_x = 0:(1/step):(T/step); //tempo para o som normal
N = n/10; //atraso
t_z = 0:(1/step):((T/step)+(N/step)); //tempo para a escala atrasada em N


s = loadwave("C:\Faculdade\Proc. Sinais\a2.wav");
//playsnd(s);

x1=[s, zeros(1,N)];
x2=[zeros(1,N), s];
z = sqrt(x1.*cos(0.4*t_z))+ sqrt(x2.*sin(0.8*t_z));
playsnd(z);

clf();
subplot(221);
plot(s);
xtitle("Sinal lido (puro)");
subplot(222);
plot(z);
xtitle("Sinal Modificado");
subplot(223);
plot(x1);
xtitle("sinal Não Atrasado");
subplot(224);
plot(x2);
xtitle("Sinal Atrasado");
```

Figura 3: Código gerado para testar a expressão matemática no Scilab.

ANEXO II

```
int main()
{
        wiringPiSetup();
        pinMode(SL, OUTPUT);
        pinMode(led1, OUTPUT);
        pinMode(led2, OUTPUT);
        unsigned int receive1=0, receive2=0, c=0;
        unsigned long receive;
        int mode;
        struct WAVdef header;
        FILE *wav_fp;
        digitalWrite(SL, HIGH); //inicia SL em 1
        spi_fd = open("/dev/spidev0.0", O_RDWR);
        wav_fp = fopen("som.wav", "wb");
        SPI_Config();
        for(c=0; c<1010; c++){
                digitalWrite(SL, HIGH);
                usleep(100);
                digitalWrite(SL, LOW); // SL -> 0; data in
                SPI_Read_Write(spi_fd, &receive1, 1);
                digitalWrite(SL, HIGH);
                usleep(100);
                digitalWrite(SL, LOW);
                SPI_Read_Write(spi_fd, &receive2, 1);
                receive = (receive1 + (receive2<<8));
                if(receive>2000){
                        digitalWrite(led1, HIGH);
                        digitalWrite(led2, LOW);
                }
                else{
                        digitalWrite(led1, LOW);
                        digitalWrite(led2, HIGH);
```

Figura 4: Função principal do programa que lê os dados do SPI. O arquivo .WAV é apenas para salvar os valores da conversão em um arquivo de áudio, mas aqui não está sendo utilizado.

Notar que no código acima a função SPI_Read_Write() foi chamada duas vezes devido ao fato de estarmos transferindo 12 bits, porém o buffer só recebe 8 bits por vez. Então a variável receive2 sofre um deslocamento e se torna o nibble mais significativo.

```
void SPI_Read_Write(int fd, unsigned int *data, int length){

        int ret;
        struct spi_ioc_transfer spi;
        memset(&spi, 0, sizeof(spi));
        spi.tx_buf = (unsigned long)data;
        spi.rx_buf = (unsigned long)data;
        spi.len    = length;
        spi.delay_usecs = 0;
        spi.speed_hz = 500000;
        spi.bits_per_word = 8;
        //Transferencia full duplex
        ret = ioctl(fd, SPI_IOC_MESSAGE(1), &spi);

}
```

Figura 5: Função SPI_Read_Write(), baseada na função do Github do Professor.

AXENO III

```
#include <stdint.h>
#include <stdlib.h>
#include "TM4C123.h"

//PE2 -> entrada analógica

//PA2 -> SSIClk
//PA3 -> slave select
//PA4 -> MOSI (master output / slave input)
//PA5 -> MISO (master input / slave output)

void ADC_config()
{

  //configurando ADC
  SYSCTL->RCGCGPIO |= 0x0010;                   //habilitando clk em PE
  GPIOE->DIR &= ~(0x0004);                      //porta PE2(ADCin) como entrada

  GPIOE->AFSEL |= 0x0004;                       //função alternativa no pino PE2
  GPIOE->DEN &= ~(0x0004);                      //desabilita função digital no pino PE2
  GPIOE->AMSEL |= 0x0004;                       //habilita função analógica no pino PE2

  SYSCTL->RCGCADC |= 0x0001;                    //habilita ADC0
  SYSCTL->RCGC0 |= 0x10000;                     //habilita clk para ADC0, mantém MAXADC0SPD em 00 p/ taxa de amostragem de 125kHz

  ADC0->SSPRI = 0x0123;                         //define SS3 como sequenciador com alta prioridade
  ADC0->ACTSS &= ~(0x0008);                     //desabilitar ASEN3 para configurar
  ADC0->EMUX |= 0xF000;                         //start do sequenciador (modo contínuo)
 ADC0->SSMUX3 &= 0x000F;                                //clear
 ADC0->SSMUX3 += 1;                                     //set Ain1 (PE2)
 ADC0->SSCTL3 = 0x0006;                                 //habilitando IE0 e END0
 ADC0->ACTSS |= 0x0008;                                 //finalmente habilita ASEN3 (sequenciador)



oid SSI_config()


 //configurando SSI
 SYSCTL->RCGC1 |= (1<<4);
 SYSCTL->RCGC2 |= (1<<0);
 SYSCTL->RCGCGPIO |= (1<<0);                            //habilitando clk em PA
 SYSCTL->RCGCSSI |= (1<<0);                             //ativando SSI0
 while((SYSCTL->PRGPIO & 0x0001)==0){};                 //aguarda PortA ser ativada
 //GPIOA->DIR |= 0x0008;                                  //PA3 como saída (master)
 //GPIOA->DIR &= ~(0x0008);                             //PA3 como entrada (slave)
 GPIOA->AFSEL |= 0x003C;                                //função alternativa nos pinos SSI
 //GPIOA->AFSEL &= ~(0x0008);
 GPIOA->PCTL &= ~0x222200;                              //clear
 GPIOA->PCTL |= 0x222200;                               //especificar função dos pinos (tabela 23-5)
 GPIOA->AMSEL &= ~(0x003C);                             //desabilita função analógica
 GPIOA->DEN |= (1<<2)|(1<<3)|(1<<4)|(1<<5);             //habilitando função digital em todos os pinos SSI
 GPIOA->PUR |= (1<<2)|(1<<3)|(1<<4)|(1<<5);
```

```
  SSI0->CR1 &= ~(0x0002);                       //desabilitando SSE p/ configurar
  SSI0->CR1 |= 0x0004;                          //define como escravo c/ saída habilitada
  //SSI0->CR1 |= 0x0000;                          //master
  SSI0->CC = 0x0005;                            //16MHz
  SSI0->CPSR &= ~(0x00FF);
  SSI0->CPSR |= 0x00A0;                         //pré-escala divisora do clk, nº de 2 a 254 (16MHz/160 = 100kHz)
  SSI0->CR0 |= 0x0007;                          //definir SCR=0, SPH/SPO = 0/0, freescale SPI, 8bit data.
  SSI0->CR1 |= 0x0002;                          //ativar SSI

}

void send_data(int data)
{

  SSI0->DR = data;
  while((SSI0->SR & (1<<0))==0){};            //aguarda transmissão

}

void send_byte(int bits12)
{
  int i=0;
  int bits8;
   while(i<2){
     if(i==0){
       bits8 = bits12 & 0xFF;
       send_data(bits8);
     }
     if(i==1){
       bits8 = (bits12>>8);
       send_data(bits8);
     }
     i++;
   }

}

int convert_write(void)
{

  int ADC;

  ADC0->PSSI = 0x0008;                                  //inicia sequenciador SS3

  while((ADC0->RIS & 0x0008)==0){};                     //aguada conversão
```

```
  ADC = ADC0->SSFIFO3;                    //salva conversão em variável
  ADC0->ISC = 0x0008;

  send_byte(ADC);

  return ADC;

}

int main()
{

  ADC_config();
  SSI_config();

  while(1)
  {

    convert_write();
    //send_data(0xEE);

  }
```