



INSTITUTO SUPERIOR TÉCNICO
MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE
COMPUTADORES

SISTEMAS ELECTRÓNICOS DE PROCESSAMENTO DE SINAL

Modem BPSK

Maria Margarida Dias dos Reis n.º 73099

David Gonçalo C. C. de Deus Oliveira n.º 73722

Nuno Miguel Rodrigues Machado n.º 74236

Grupo n.º 5 de segunda-feira das 15h30 - 18h30

Lisboa, 1 de Junho de 2015

Índice

1	Introdução	1
2	Projecto de Demonstração	1
3	Projecto #1 - NCO	1
4	Projecto #2 - Transmissor BPSK	10
5	Projecto #3 - Receptor BPSK	15
6	Conclusões	17
7	Anexos	18

1 Introdução

Com este trabalho laboratorial o objectivo é a familiarização com o sistema de desenvolvimento de *software* e *kit* de processamento digital de sinal DSK TMS320C6713. O processador em causa é de 32 bits, com um relógio de 225 MHz, sendo capaz de fazer o *fetching* e execução de 8 instruções por ciclo de relógio. Relativamente ao *software*, a ferramenta utilizada para programar o DSK é o CCS v5.5.

Na primeira fase do projecto pretende-se implementar um oscilador numericamente controlado (NCO) e de seguida um transmissor *binary phase-shift keying* (BPSK).

teddy - falar das demonstrações, ganho é
 $i = 33$

2 Projecto de Demonstração

3 Projecto #1 - NCO

Um oscilador numericamente controlado permite gerar uma frequência instantânea proporcional à amplitude do sinal de entrada. É um gerador digital de sinal que cria uma representação síncrona, discreta no tempo e discreta em amplitude de uma forma de onda.

As características do NCO são apresentadas na seguinte tabela.

Tabela 1: Características do NCO.

Parâmetro	Símbolo	Valor	Descrição
frequência de amostragem	f_s	16 kHz	
frequência mínima	f_{min}	2 kHz	frequência a que a amplitude do sinal de entrada é mínima
frequência máxima	f_{max}	6 kHz	frequência a que a amplitude do sinal de entrada é máxima

Pretende-se primeiramente desenvolver um oscilador de relaxação utilizando uma variável inteira com sinal de 16 bits e a circularidade da representação em complemento para dois. Na figura abaixo encontra-se uma representação do sinal a obter.

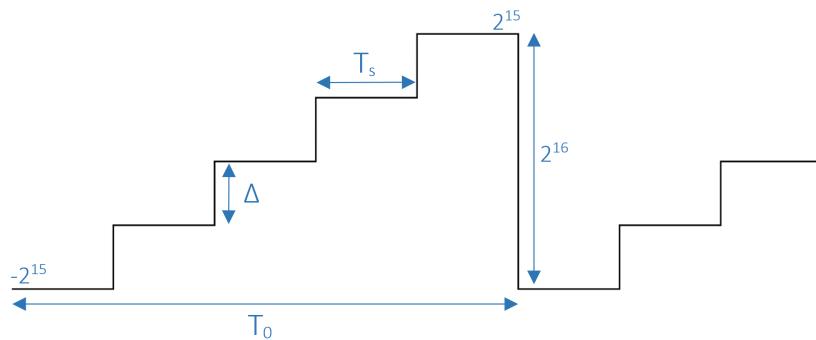


Figura 1: Esquema do oscilador de relaxação.

Com recurso à Figura 1 pode-se deduzir que

$$f_0 = \frac{\Delta}{2^{16}} \times f_s \leftrightarrow \Delta = \frac{f_0}{f_s} \times 2^{16}. \quad (3.1)$$

Existe uma variável de estado da rampa que a cada T_s , período de amostragem, é incrementada de Δ , como se pode ver na Figura 1. A variável de estado da rampa é de 16 *bits* com representação em Q_{15} e, sabendo que o maior número positivo que se pode representar em Q_{15} é $2^{15} - 1 = 32767$ e o menor número negativo que se pode representar é $-(2^{15} - 1) = -32767$, a variável de estado começa com o valor -32767 e vai até um máximo de 32767. Quando é atingido o valor máximo, 32767, entra em efeito a circularidade da representação em complemento para dois e, assim, a variável de estado não atinge o valor de 2^{15} , “dando a volta” para -32767.

Relativamente à variável Δ esta encontra-se também representada em Q_{15} . O NCO tem como característica uma frequência f_0 que varia entre 2 kHz e 6 kHz. Estes valores são controlados a partir da amplitude do sinal de entrada. Quando esta for mínima, a frequência f_0 é de 2 kHz e quando for máxima, a frequência f_0 é de 6 kHz. Com estas especificações pode-se calcular três valores de Δ com recurso à equação (2.1), para a frequência mínima, a frequência média e a frequência máxima.

Tabela 2: Valores de Δ para as três frequências especificadas.

f_0	Δ
2 kHz	8192
4 kHz	16384
6 kHz	24576

Em código, a variável de estado da rampa é **status** e a variável que representa os incrementos é **delta**. No código abaixo está a criação da rampa para um frequência de 4 kHz.

```

1 void main(){
2
3     short delta = 16384;
4     short status = -32767;
5
6     while(1){
7         ...
8         //criacao da rampa
9         status = status + delta;
10        ...
11    }
12 }
```

Nas figuras da próxima página pode-se ver o sinal obtido experimentalmente que representa a rampa para dois valores diferentes de frequência f_0 .

Como se pode ver na Figura 2(c), por comparação com o esperado teoricamente da Figura 1, o oscilador de relaxação implementado funciona de acordo com o previsto. De notar que o sinal que se observa está invertido relativamente ao teórico, o que já é esperado, quando se considera que antes de ser observado no osciloscópio passa por um inversor.

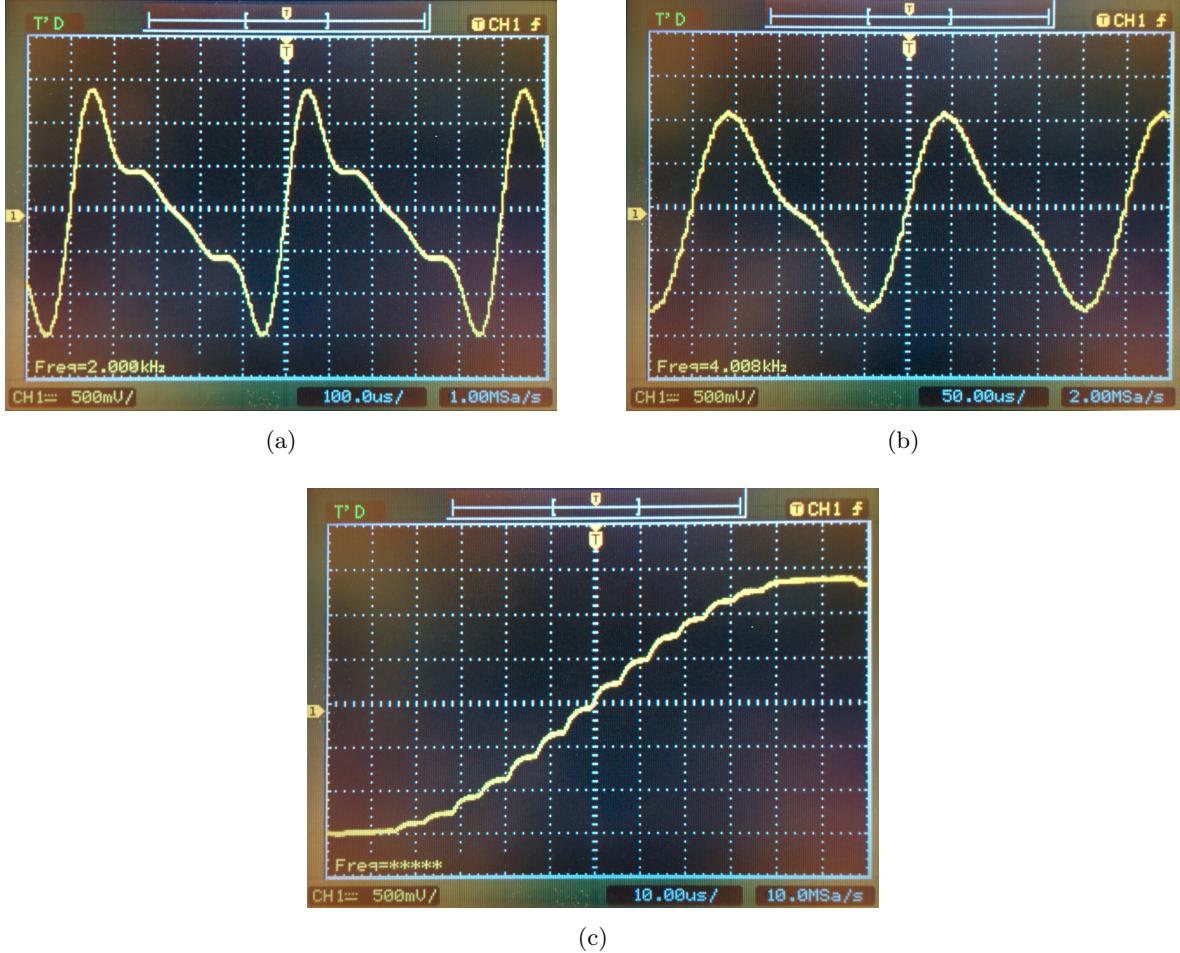


Figura 2: Oscilador de relaxação para $f_0 = 2$ kHz (a), oscilador de relaxação para $f_0 = 4$ kHz (b) e pormenor da rampa criada (c).

Com o oscilador implementado pretende-se agora criar uma *look-up-table* (LUT) com 32 valores positivos de meio período da função seno. É necessário começar por determinar esses valores, para que, posteriormente, os mesmos sejam convertidos para o formato mais preciso de representação, Q_{15} , uma vez que se encontram no intervalo $[-1, 1]$. Assim, tendo em conta que meio período da função seno é π , podemos calcular os valores da seguinte maneira:

$$a_k = \sin\left(\frac{\pi}{32}k\right), k = 0, 1, \dots, 31. \quad (3.2)$$

Os 32 valores determinados são então convertidos para o formato Q_{15} , recorrendo a:

$$a_{k15} = \text{round}\left(a_k \times 2^{15}\right), \quad (3.3)$$

sendo assim criada a LUT pretendida.

Apresenta-se de seguida o excerto de código onde é declarada a LUT com os valores de meio período da função seno, no vector **sine** que tem 33 posições, cada uma de 16 bits.

```

1 ...
2 // LUT do seno

```

```

3     short sine[33] =
4     {0,3212,6393,9512,12540,15447,18205,20788,23170,25330,27246,28899,30274,31357,
5     32138,32610,32767,32610,32138,31357,30274,28899,27246,25330,23170,20788,18205,
6     15447,12540,9512,6393,3212,0};
7     ...

```

Como se pode constatar, a LUT é declarada com 33 valores, o que se deve ao facto de ser necessário garantir que, quando o valor de *i* for igual a 32, seja possível aceder ao valor da função seno correspondente, o que não seria possível caso a LUT fosse apenas declarada com 32 valores.

É possível implementar uma solução diferente, em que é realizada a operação lógica AND de *i* e *i+1* para o valor de *y1* e *y2* (necessário para o caso da interpolação), respectivamente, com a máscara 31, de acordo com as seguintes linhas de código:

```

1     ...
2     y1 = sine[i&31];
3     y2 = sine[(i+1)&31];
4     ...

```

Esta solução faz com que não seja necessário mais memória para criar a LUT, embora seja realizado um maior número de operações.

david - rever
este excerto de
código acima, o
prof diz "o i ja
vem limitado a
3"

Para que se possa agora aceder aos valores da função seno, é utilizada a variável de estado do oscilador, **status**, como índice da LUT. Apenas 5 *bits* da variável são utilizados para endereçar a LUT, sendo criada a variável *i*, tal como especificado na Figura 3, onde a variável de estado do oscilador é representada por *x*.

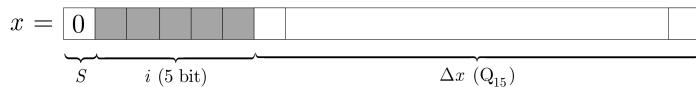


Figura 3: Representação da variável do estado do oscilador.

De modo a aceder aos 5 *bits* pretendidos da variável de estado, é realizado um deslocamento de 10 *bits* para a direita, sendo, de seguida, utilizada a função lógica AND com a máscara 31 (5 *bits* menos significativos com o valor lógico 1). É apresentado o excerto de código que realiza o procedimento especificado.

```

1     ...
2     //indexar a LUT e obter os valores do seno
3     i = (status>>10)&31;
4     y1 = sine[i];
5     ...

```

david - aqui o
prof tambem se
queixou do and
com 31

Foram depois criadas duas variáveis com o objectivo de controlar a amplitude e frequência do sinal sinusoidal. A variável **delta** representa o controlo da frequência e a variável **amp** representa o controlo da amplitude. O código que permite implementar este controlo é apresentado de seguida.

```

1 void main(){
2 ...
3 //variavel de controlo de frequencia
4 short delta = 0
5 //variavel de controlo da amplitude: define um ganho de 1/2
6 short amp = 16384;
7 short yf = 0;
8 ...
9 while(1){
10 if(intflag != FALSE){
11 ...
12 //obtencao do valor para a frequencia
13 delta = 16384 + (inbuf>>2);
14 ...
15 //controlo da amplitude e frequencia
16 yf = (y1*delta<<1)>>16;
17 y = (yf*amp<<1)>>16;
18 ...
19 if(status < 0)
20 y = -y;
21 ...
22 AIC_buffer.channel[LEFT] = y;
23 }
24 }
25 }
```

Analizando a Tabela 2 verifica-se que o valor de `delta` oscila com uma amplitude de 8192 em torno de 16384, Δ_0 . Ou seja, f_0 tem uma frequência central em 4 kHz, oscilando com uma amplitude de 2 kHz. O incremento do oscilador é obtido de acordo com a seguinte equação, onde x é a amplitude do sinal de entrada:

$$\Delta = \Delta_0 + kx. \quad (3.4)$$

Com esta conclusão, teve de se garantir que o valor da amplitude do sinal de entrada não ultrapassa 8192, mantendo a relação entre cada amostra. Optou-se por dividir o valor de cada amostra por 4, $k = 1/4$, pois a amplitude máxima é de 32767, o equivalente a um *shift* de 2 bits para a direita.

Em baixo está o código referente ao cálculo para obter o valor de `delta`, sendo que todas as variáveis definidas neste excerto são de 16 bits, `short`, em formato Q_{15} .

```

1 ...
2 //obtencao do valor para a frequencia
3 delta = 16384 + (inbuf>>2);
4 ...
```

Tendo o valor de `delta`, é simples obter a amplitude de cada amostra do sinal de saída, multiplicando `delta` por `y1`, valor obtido da LUT especificada anteriormente. Em baixo está representado um excerto do código que demonstra a obtenção da amplitude do sinal de saída. Todas as variáveis são de

16 bits, tendo `y1` e `delta` o formato de Q_{15} , como também `yf`. Isto deve-se ao facto de o formato do resultado da multiplicação com duas variáveis em Q_{15} ser Q_{30} com replicação do bit de sinal. Assim, é necessário efectuar um *shift* para a esquerda para remover o bit de sinal replicado, resultando num formato final de Q_{31} , para 32 bits. Para se poder armazenar numa variável de 16 bits, no formato Q_{15} , é necessário efectuar um *shift* de 16 posições para a direita, permitindo armazenar os 16 bits mais significativos do resultado de 32 bits.

O código apresentado de seguida demonstra a explicação referida.

```

1 ...
2 // controlo da amplitude e frequencia
3 yf = (y1*delta<<1>>16);
4 ...

```

teddy - neste paragrafo de cima, na 1a frase o prof colocou 2 ?

Para o controlo da amplitude do sinal de saída, multiplica-se o resultado final obtido anteriormente por uma constante de 16 bits em formato Q_{15} . Está representado um excerto de código que demonstra a alteração da amplitude do sinal de saída. Neste caso todas as variáveis são também de 16 bits, tendo `yf` e `amp` o formato de Q_{15} , como também `y`. Para armazenar a variável `y` em Q_{15} recorre-se à mesma lógica explicada anteriormente de fazer 15 *shifts* para a direita ao resultado da multiplicação.

O código apresentado de seguida demonstra a explicação referida.

```

1 ...
2 // controlo da amplitude
3 y = (y_f*amp<<1>>16);
4 ...

```

teddy - neste excerto de código afinal e uma soma? o prof diz "para que? nao percebo?"

Sabendo que os valores da LUT permitem aceder às arcadas positivas do seno, quando a variável de estado da rampa é negativa, `status < 0`, o sinal de saída tem que ser negado. Em seguida está representado o código referente à explicação anterior:

```

1 ...
2 if(status < 0){
3     y = -y;
4 }
5 ...

```

O sinal de saída pode ser observado no canal esquerdo da DSP.

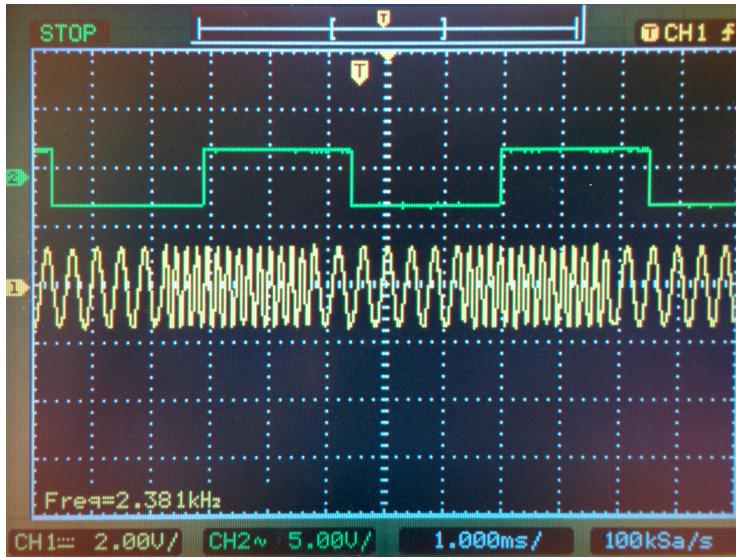
```

1 ...
2 AIC_buffer.channel[LEFT] = y;
3 ...

```

Com o oscilador controlado já implementado procede-se à fase de testes. A verificação de funcionamento pode ser vista na Figura 4, onde se utilizou um sinal de entrada, a verde, definido como uma onda quadrada com uma frequência de 200 Hz e uma amplitude próxima de 1 V, no intuito de poder verificar o controlo da frequência do sinal de saída, a amarelo, a partir da amplitude do sinal de entrada.

Quando a amplitude do sinal de entrada é máxima pode-se visualizar um aumento da frequência do sinal de saída, ou seja, para um mesmo intervalo de tempo há um maior número de ciclos. Quando a amplitude do sinal de entrada é mínima, a frequência do sinal de saída diminui, ou seja, há um menor número de ciclos para o mesmo intervalo de tempo. Com estes resultados, pode-se concluir que o método de controlo utilizado funciona de modo adequado.



na parte final
do parágrafo
o prof diz que
temos de ter
cuidado com
esta conclu-
são pq ha um
atraso de pro-
cessamento

Figura 4: Saída modulada na frequência do oscilador controlado (a amarelo) com *input* de uma onda quadrada de 200 Hz (a verde).

Pretende-se agora melhorar a qualidade do oscilador sinusoidal utilizando interpolação linear. Esta interpolação é feita lendo dois valores consecutivos, y_1 e y_2 , da LUT do seno e depois obtém-se o valor sinusoidal interpolado com recurso à seguinte equação

$$y = y_1 + (y_2 - y_1)\Delta x. \quad (3.5)$$

Na Figura 3, onde se encontra representada a variável de estado da rampa, pode-se ver que os 10 bits menos significativos desta correspondem à variável Δx da equação (3.5), que está representada no formato Q_{15} .

Para se obter o valor de Δx é utilizada a função lógica AND com a máscara 1023 (10 bits menos significativos com o valor lógico 1), sendo de seguida necessário efectuar um *shift* de 5 posições para a esquerda para que a variável seja representada em Q_{15} .

Com acesso a esse parâmetro, falta ler dois valores consecutivos da função seno, o que pode ser feito endereçando a LUT com recurso à variável *i* que foi anteriormente definida.

O excerto de código seguinte demonstra a obtenção do valor de Δx , armazenado na variável *delta_x* e dois valores consecutivos da função seno, *y1* e *y2*.

```

1 ...
2 //obtencao do valor de delta_x
3 delta_x = (status&1023)<<5;
4 i = (status >> 10) & 31;
5 y1 = sine[i];
6 y2 = sine[i+1];
7 ...

```

Pode-se agora computar y de acordo com a equação (3.5). Quando se faz a subtracção entre y_2 (Q_{15}) e y_1 (Q_{15}), o resultado ficaria no formato Q_{14} . No entanto, dado que se subtraem dois valores positivos em Q_{15} , tem-se a garantia de que o resultado é sempre correctamente armazenado em Q_{15} , o que é preferível face à opção de Q_{14} , pois garante mais resolução.

O resultado desta subtracção é então multiplicado com Δx , que toma sempre valores entre 0 e 1, ou seja, está-se a multiplicar dois valores no formato Q_{15} , que origina um valor no formato Q_{30} com replicação do bit de sinal. Assim, é necessário efectuar um *shift* para a esquerda para remover o bit de sinal replicado, resultando num formato final de Q_{31} , para 32 bits. Para se poder armazenar numa variável de 16 bits, no formato Q_{15} , é necessário efectuar um *shift* de 16 posições para a direita, permitindo armazenar os 16 bits mais significativos do resultado de 32 bits.

O valor da subtracção que é seguida de uma multiplicação é agora somado ao valor de y_1 , ou seja, está-se a somar dois números no formato Q_{15} , o que dá um resultado que seria no formato Q_{14} . No entanto, quando se analisa o pior caso em que $y_1 = 32610$, $y_2 = 32767$ e $\Delta x = 32767$, o resultado da subtracção de y_2 com y_1 multiplicado por Δx é igual a 5144419 em Q_{31} ou 78 em Q_{15} , dividindo por 2^{16} . De seguida soma-se y_1 com o resultado anterior, tendo um valor final de 32688, pelo que se pode guardar em Q_{15} o resultado da equação (2.5).

O valor do sinal interpolado é agora multiplicado pelo sinal *amp*, ou seja, é efectuada mais uma multiplicação entre dois números no formato Q_{15} , sendo necessário efectuar o *shift* para esquerda e os 16 *shifts* para a direita explicados anteriormente.

O excerto de código apresentado de seguida demonstra a obtenção do valor interpolado, que é armazenado na variável *y*.

```

1 ...
2 //obtencao do valor sinusoidal interpolado
3 y = ( ( amp*( y1 + ( (y2-y1)*delta_x << 1 ) >> 16) ) << 1 ) >> 16);
4 ...

```

Como se viu, foram desenvolvidos dois osciladores sinusoidais - com e sem interpolação - sendo agora importante compará-los, comparação feita inicialmente ao nível dos espectros. De referir que para comparar os sinais fixou-se o valor de Δ , ou seja, não se incluiu a modulação, optando por $\Delta = 16384$, ou seja, uma frequência de 4 kHz.

Quando se analisou os espectros das sinusoides não se verificou qualquer diferença entre eles e não foi possível concluir sobre qual seria o melhor método, como se pode ver nas figuras da próxima página, em que ambos os espectros apresentam uma risca na frequência central, 4 kHz, tal como esperado.

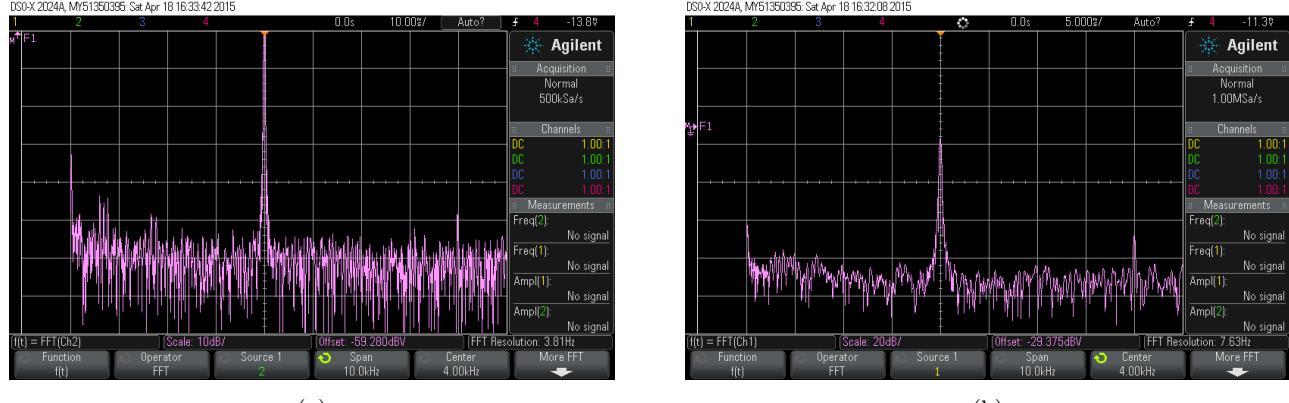


Figura 5: Espectro do sinal de saída sem interpolação (a) e com interpolação (b).

Assim, para se poder obter resultados mais conclusivos sobre qual o melhor método recorreu-se ao modo persistência do osciloscópio. Este modo sobrepõe múltiplas formas de onda no mesmo *display*, com as formas de onda mais recentes a serem enfatizadas com uma saturação mais profunda. De referir que para comparar os sinais fixou-se agora o valor de Δ a 16380, ou seja, uma frequência que não é um múltiplo da frequência de amostragem. Isto é feito porque, quando a frequência é múltiplo da frequência de amostragem, não se ganha nada em implementar o método da interpolação. No entanto, na maioria das vezes a frequência não será múltiplo de f_s e assim, tem-se a ganhar quase sempre.

Na Figura 6 apresenta-se as formas de onda obtidas para as sinusoides geradas de acordo com os dois métodos, com Δ a 16380.



Figura 6: Onda sinusoidal obtida sem interpolação (a verde) e com interpolação (a amarelo).

Como se pode ver, o sinal representado a verde, implementação sem interpolação, apresenta maior dispersão em torno dos valores pretendidos para a forma de onda. O sinal representado a amarelo, implementação com interpolação, não tem tanta dispersão, estando a gama de valores mais próxima do pretendido.

Assim se pode concluir que o método da interpolação compensa na maioria das vezes, pois fornece um oscilador de melhor qualidade.

4 Projecto #2 - Transmissor BPSK

Neste projecto, pretende-se implementar o codificador de um transmissor BPSK (*binary phase-shift keying*), representado na Figura 7.

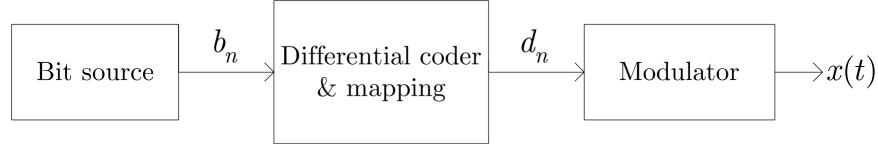


Figura 7: Esquema de um transmissor BPSK.

A modulação BPSK faz o mapeamento de uma mudança na fase de 0° ou 180° para um valor de *bit* de 0 ou 1, respectivamente. Para que não haja ambiguidade na fase, que ocorre quando se utilizam fases absolutas, aplica-se codificação diferencial. Assim, quando o canal introduz uma mudança de fase desconhecida é possível recuperar a sequência de *bits*.

O codificador tem como entrada os *bits* b_n , uma sequência que vai alternando entre o valor lógico 0 e o valor lógico 1 ($b_n = 1, 0, 1, 0, \dots$) e como saída os valores d_n , que podem ser -1 ou +1. O codificador realiza a operação $c_n = c_{n-1} \oplus b_n$, considerando $c_0 = 0$, para que depois possam ser determinados os *bits* da sequência d_n , de acordo com o mapeamento:

$$c_n = '0' \rightarrow d_n = -1; \quad (4.1)$$

$$c_n = '1' \rightarrow d_n = +1. \quad (4.2)$$

Assim, o sinal modulado BPSK é dado por:

$$s(t) = \sin(2\pi f_0 t + \pi c_n) = d_n \sin(2\pi f_0 t). \quad (4.3)$$

Ou, a tempo discreto:

$$s_n = s(nT_s) = d_n \sin(2\pi f_0 T_s n). \quad (4.4)$$

É utilizada uma frequência de amostragem, $f_s = 1/T_s$, de 16 kHz e uma frequência da portadora, f_0 , de 4 kHz. A taxa de bits é de $f_b = 1$ kbps, por isso, por cada *bit*, há 4 períodos da portadora.

Os *bits* da sequência b_n são implementados recorrendo a um contador, uma vez que a cada 16 amostras do sinal de entrada, é necessário que haja uma alteração do *bit* seguinte da sequência b_n , passando de 0 para 1 ou vice-versa. Para que seja realizada essa alteração, é utilizada a função XOR do bit b_n anterior com 1. Apresenta-se de seguida o excerto de código que implementa a sequência de bits b_n .

```

1 ...
2 if(b_i>15){
3     b_i=0;
4     b_n=(b_n^1); //xor entre valor anterior de bn e o valor logico 1
5 ...
6 }

```

```

7     b_i++;
8 ...

```

Foi criada outra solução para a obtenção da sequência b_n com objectivo de não utilizar a instrução condicional, *if*. Seguiu-se o conselho do enunciado de usar um contador que quando ocorre *overflow* gera um novo *bit* alternado. Está representado de seguida no excerto de código:

```

1 ...
2 b_i++;
3 b=(b_i&16)>>4; //mascara para obter o bit de overflow
4 b_n=(b_n^b); //xor para alternar o bit bn
5 b_i=(b_i&15); //obtencao dos 4 bits menos significativos
6 ...

```

Como se pode observar, ocorre *overflow* quando o contador, *b_i*, atinge o valor 16, sendo utilizado este valor porque a frequência de amostragem, de 16 kHz, é dividida por esse valor de forma a obter o resultado desejado de uma taxa de transmissão de 1 kbps. Para detectar esta ocorrência aplica-se a máscara 16 (o quinto *bit* menos significativo com o valor lógico 1) e efectua-se um *shift* de 4 posições para a direita.

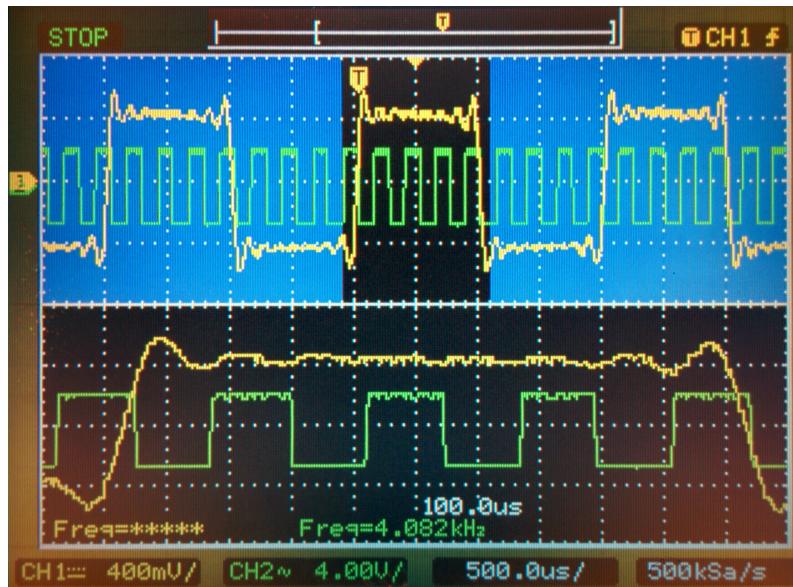


Figura 8: Sinal b_n (a amarelo) sobreposto com uma onda quadrada com frequência de 4 kHz (a verde).

Na figura anterior pode-se analisar o resultado do sinal b_n , verificando-se que é um sinal binário, uma vez que só toma o valor 0 ou 1, com um ritmo de transmissão de 1 kbps. Cada *bit* (quando o sinal representado a amarelo toma o valor máximo ou mínimo) contém 4 ciclos da onda quadrada (onda representada a verde que tem uma frequência de 4 kHz), o que resulta numa frequência de *bit* de $4 \text{ kHz}/4 = 1 \text{ kHz}$. Sabendo que um período do sinal b_n é composto por dois *bits* (valor máximo e mínimo da onda representada a amarelo) conclui-se que se tem um sinal com frequência de 500 Hz, ou seja, $4 \text{ kHz}/8 = 500 \text{ Hz}$.

o prof diz que
nao sabe o que
e aquela onda
de 4k

De seguida, aplica-se a função `XOR` entre o resultado anterior e a variável `b_n`, de forma a obter uma sequência b_n que varia alternadamente entre 0 e 1, ou vice-versa, a uma taxa de $f_b = 1$ kbps. Com o *bit-rate*, b_n , criado aplica-se o codificador diferencial de forma a gerar o sinal c_n , de acordo com seguinte equação:

$$c_n = c_{n-1} \oplus b_n \quad (4.5)$$

A equação anterior foi implementada no ciclo de criação do *bit-rate*, b_n , aplicando a função `XOR` entre o bit c_{n-1} e b_n , como se mostra no seguinte excerto de código:

```

1   ...
2   c_n = 0;
3   ...
4   if(b_i>15){
5     b_i=0;
6     b_n=(b_n^1);
7     c_n=c_n^b_n; //codificador diferencial
8     ...
9   }
10  b_i++;
11  ...

```

De seguida, aplica-se o mapeamento aos *bits* do sinal c_n na constelação BPSK, de forma a obter o sinal d_n , ou seja, os dois símbolos da constelação: 1 e -1 . O mapeamento segue as expressões da equação (4.1) e (4.2).

A solução computacional mais eficiente que se encontrou consiste em três fases. Na primeira, é efectuado um *shift* de 15 posições para a esquerda, no intuito de o *bit* mais significativo ter o valor do sinal c_n . Em segundo, nega-se o vector. E em último, é efectuado um *shift* de 15 posições para a direita. De seguida encontra-se um esquema que demonstra a solução encontrada para a obtenção do d_n como também o código.

```

1   ...
2   if(b_i>15){
3     ...
4     c_n = c_n^b_n;
5     shift15_cn = c_n<<15;      // shift de 15 posicoes para a esquerda
6     not_shift15 = ~shift15_cn; // negacao do sinal anterior
7     d_n = not_shift15 >>14;    // shift de 14 posicoes para a direita
8   }
9   ...

```

Pretende-se agora gerar a portadora com frequência f_0 a 4 kHz. De forma a implementar a portadora, que vai ser multiplicada pela sequência d_n , originando o sinal modulado, é criada uma LUT idêntica à do Projecto #1. Neste caso, uma vez que só há 4 amostras por período, já que a frequência de amostragem é $f_0 = 16$ kHz, basta especificar 4 valores da onda sinusoidal. Assim, é declarada a LUT com os valores 0, 1, 0 e -1, representados no formato mais preciso, Q_{15} :

rever se compensa o if ou nao - profiling do codigo?

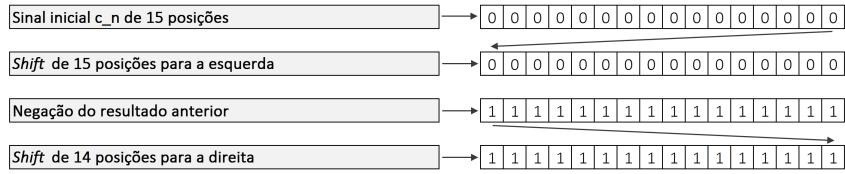


Figura 9: Representação da situação em que $c_n = 0$, resultando num mapeamento de $d_n = -1$.

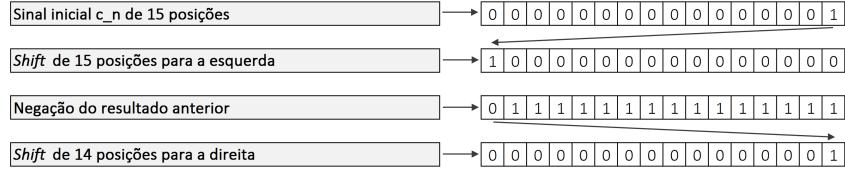


Figura 10: Representação da situação em que $c_n = 1$, resultando num mapeamento de $d_n = +1$.

```

1 ...
2 //LUT do seno com 4 amostras
3 short sine[4] = {0,32767,0,-32767};
4 ...

```

Com a LUT declarada, usou-se o excerto de código seguinte para poder gerar a portadora de frequência f_0 a 4 kHz. O código encontra-se no *loop* da rotina principal, de forma a obter um valor da LUT a cada 16 kHz, incrementando a variável de indexação da *look-up-table*, `sine_i`, a cada passagem do *loop*. É necessário aplicar a máscara 3, os dois *bits* menos significativos a 1, para poder aceder às posições 0 a 3 da LUT.

```

1 while(1{
2 ...
3     sine_i= sine_i&3; // mascara para obter os 3 bits menos significativos
4     y= sine[sine_i]; // indexacao da LUT
5     sine_i++; // incrementador do index
6 ...
7 }

```

Com a primeira parte do modulador criado, geração da portadora, pode-se aplicar a fase final do seu desenvolvimento, a multiplicação do sinal d_n com a portadora, gerando assim um sinal modelado do tipo BPSK. O código seguinte demonstra esta última fase.

```

1 ...
2     y = d_n*y; // multiplicacao do sinal com a portadora
3     AIC_buffer.channel[LEFT] = y;// sinal observado no canal esquerdo
4 ...

```

Com o modulador criado procede-se à fase de testes. Sabe-se que o sinal d_n tem uma frequência de 500 Hz com um ritmo de transmissão de 1 kbps, e é modulado por uma portadora de 4 kHz. Assim, existe uma inversão de fase a cada 4000/500 ciclos, ou seja, a cada 8 ciclos.

Analizando a figura seguinte verifica-se que o modulador está a funcionar correctamente, invertendo a fase no zero da portadora e a cada 8 ciclos.

Na figura anterior, pode-se verificar que a inversão de fase faz-se correctamente - quando o sinal d_n

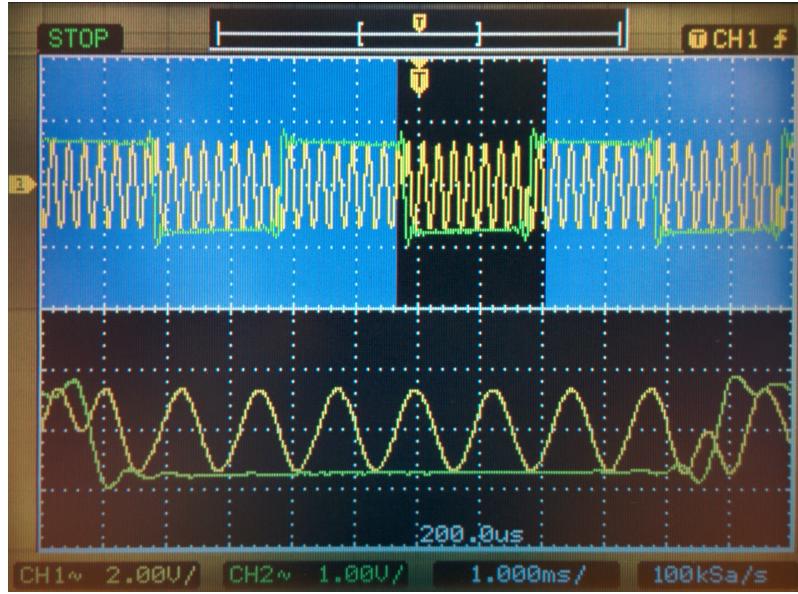


Figura 11: Sobreposição do sinal modulado (a amarelo) com o sinal d_n (a verde).

passa de $+1$ para -1 a inversão é feita para a arcada positiva, como se pode verificar do lado esquerdo da figura. Quando o sinal d_n passa de -1 para $+1$ a inversão é feita para a arcada negativa, como se pode verificar na inversão do lado direito da figura.

Apresenta-se também uma imagem em que se encontra apenas representado o sinal modulado, com um *zoom* sobre uma altura em que ocorrem duas inversões de fase.

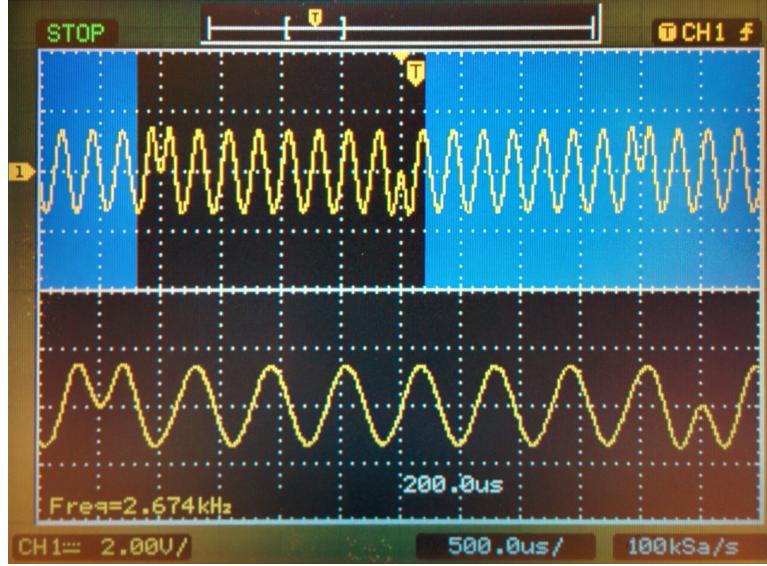


Figura 12: Sinal modulado (em cima) e pormenor obtido com recurso a uma janela temporal (em baixo).

Efectuou-se também um registo ao nível de espectros para vários sinais, como se pode ver nas figuras da próxima página.

Relativamente ao espectro do sinal b_n verifica-se que existe uma risca na frequência de 500 Hz, e também em 1500 Hz, 2500 Hz, e por aí adiante, com um intervalo entre riscas de 1 kHz. Isto deve-se ao facto de o ritmo de transmissão dos *bits* ser de 1 kbps e a sequência ser alternada. Perante estas

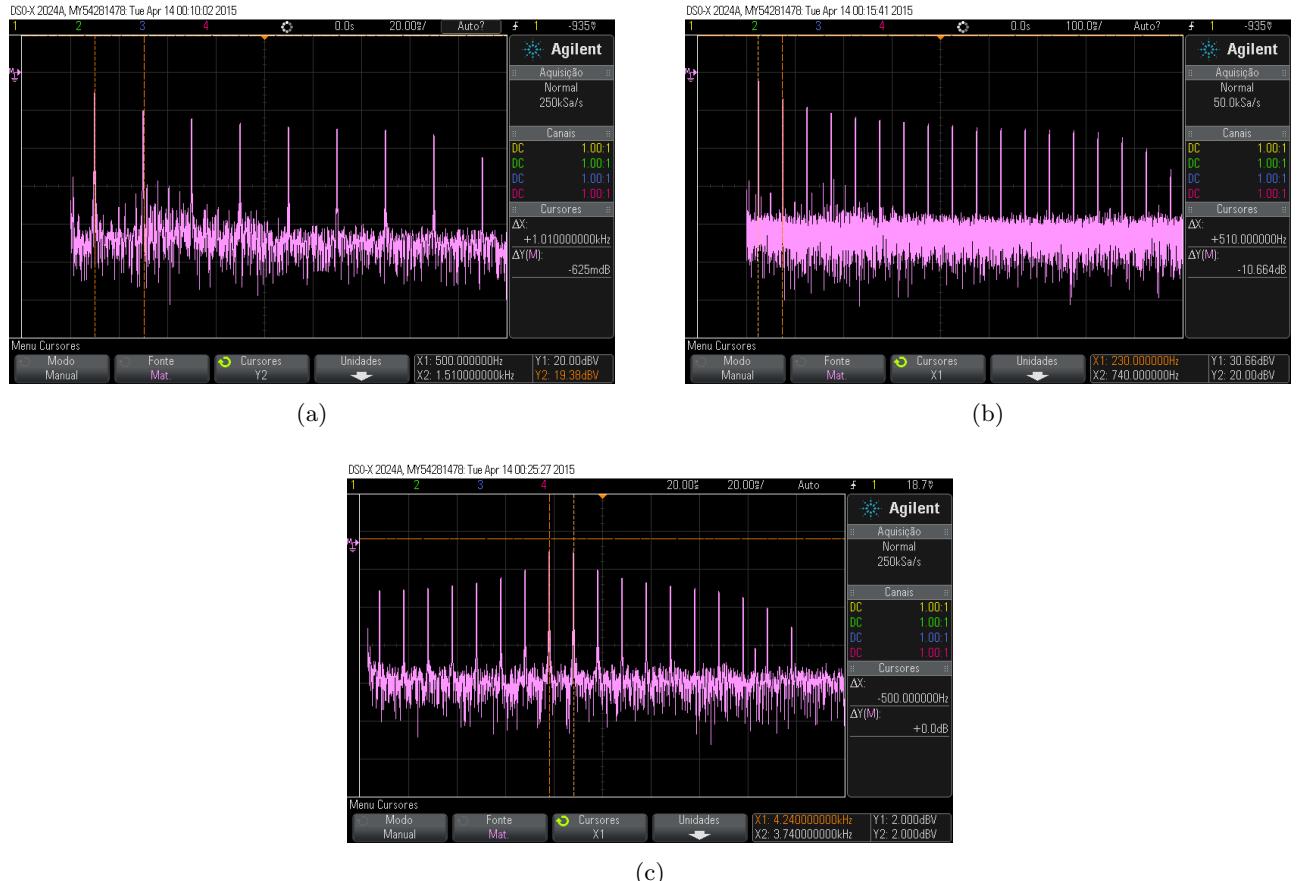


Figura 13: Espectro do sinal b_n (a), espectro do sinal d_n (b) e espectro do sinal y_n (c).

condições tem-se uma onda quadrada com 500 Hz de frequência. O espectro do sinal b_n representa a informação do bit com valor lógico 1, enquanto o espectro de d_n representa a informação resultante do mapeamento do bit com valor lógico 1 e -1.

O espectro de y_n permite verificar o fenómeno de modulação, na medida em que se verifica que o espectro de d_n foi deslocado para altas frequências, ou seja, encontra-se em torno do espectro da portadora que corresponde a uma risca na frequência f_0 , 4 kHz. De facto, ocorre a convolução do espectro do sinal d_n com o espectro da portadora.

imagem da sequencia aleatoria de bits

scrambler -
margarida

5 Projecto #3 - Receptor BPSK

COSTAS LOOP

descrambler -
teddy

1 - teddy

era bom testar que no filtro com beta=0 deixar passar para comonentes maior que ws/2, fazer um teste e vemos que falha

2 e 3 - david
- nas imagens
e bom tirar a
LB, para com-
provar que esta
a funcionar

4 - nao nos
lembramos do
que era isto

5 - margarida

depois desta
parte comenta-
mos as imagens
que tiramos no
ultimo lab

6 Conclusões

Relativamente ao Projecto #1, pode-se concluir que um oscilador controlado feito com recurso ao método da interpolação apresenta resultados com mais qualidade, isto para casos em que o valor de Δ representa frequências que não são múltiplos da frequência de amostragem.

Já em relação ao Projecto #2, este correu de acordo com o esperado, sendo que se procedeu a uma melhoria ao nível da eficiência do código para que apenas houvesse o *if* do contador. Verificou-se que retirar este último *if* não compensava pois era necessário replicar o código e a eficiência computacional seria equiparável.

7 Anexos

7.1 Anexo I - Código do Projecto #1

```
1 #include "dsk6713_aic23.h"
2 #include "C6713dskinit.h"
3
4 Uint32 fs = DSK6713_AIC23_FREQ_16KHZ;
5
6 char intflag = FALSE;
7 union {Uint32 samples; short channel[2];} AIC_buffer;
8
9 short sine[33] =
10 {0,3212,6393,9512,12540,15447,18205,20788,23170,25330,27246,28899,30274,31357,
11 32138,32610,32767,32610,32138,31357,30274,28899,27246,25330,23170,20788,18205,
12 15447,12540,9512,6393,3212,0};
13
14 interrupt void c_int11(){
15     output_sample(AIC_buffer.samples);
16     AIC_buffer.samples= input_sample();
17     intflag = TRUE;
18     return;
19 }
20
21 void main(){
22     short inbuf = 0;
23     short delta = 0 ;
24     short status = -32767;
25     short y1 = 0, y2 = 0, y_n = 0, y_s = 0;
26     short i = 0, delta_x = 0;
27     short amp = 16384;
28
29     comm_intr();
30     while(1){
31         if(intflag != FALSE){
32             intflag = FALSE;
33
34             inbuf = AIC_buffer.channel[LEFT];
35
36             delta = 16384 + (inbuf>>2);
37
38             status = status + delta;
39             delta_x = (status & 1023)<<5;
40
41             i = (status>>10)&31;
42             y1 = sine[i];
43             y2 = sine[i+1];
44             y_s = amp*(y1 + delta)>>15;
```

```

42     y_n = (amp*(y1 + ((y2-y1)*delta_x>>15))>>15);
43
44     if(status < 0){
45         y_s = -y_s;
46         y_n = -y_n;
47     }
48     AIC_buffer.channel[LEFT] = y_n; //saida com interpolacao
49     AIC_buffer.channel[RIGHT] = y_s; //saida sem interpolacao
50 }
51 }
52 }
```

7.2 Anexo II - Código do Projecto #2

```

1 #include "dsk6713_aic23.h"
2 #include "C6713dskinit.h"
3
4 UInt32 fs = DSK6713_AIC23_FREQ_16KHZ;
5
6 char intflag = FALSE;
7 union {UInt32 samples; short channel[2];} AIC_buffer;
8
9 short sine[4] = {0,32767,0,-32767};
10
11 interrupt void c_int11(){
12     output_sample(AIC_buffer.samples);
13     AIC_buffer.samples= input_sample();
14     intflag = TRUE;
15     return;
16 }
17
18 void main(){
19     short d_n, y;
20     short b_i = 1, sine_i = 0, b = 0;
21     short c_n = 0;
22     short b_n = 1, shift15_cn = 0, not_shift15 = 0;
23
24     comm_intr();
25     while(1){
26         if(intflag != FALSE){
27             intflag = FALSE;
28
29             sine_i= sine_i&3;
30             y = sine[sine_i];
31             sine_i++;
32             y = (32767*y)>>15 ; //esta linha e para tirar?
33
34 }
```

```

35     /* implementacao do contador sem recurso a instruccao condicional if */
36     //b_i++;
37     //b=(b_i&16)>>4;
38     //b_n=(b_n^b);
39     //b_i=(b_i&15);*/
40     if(b_i>15){
41         b_i=0;
42         b_n=(b_n^1);
43         c_n=c_n^b_n;
44         shift15_cn=c_n<<15;
45         not_shift15=~shift15_cn;
46         d_n=not_shift15>>14;
47     }
48     b_i++;
49     y=d_n*y;
50
51     AIC_buffer.channel[LEFT] = y;
52 }
53 }
54 }
```