



MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE  
COMPUTADORES

## SISTEMAS ELECTRÓNICOS DE PROCESSAMENTO DE SINAL

### BPSK Modem

Grupo n.º 2/3

André Filipe Barroso Cerqueira      n.º 65144

Guilherme Branco Teixeira      n.º 70214

João André Catarino Pereira      n.º 73527

segunda-feira 15h30 - 18h30, LE1

Lisboa, 17 de Abril de 2015

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
<b>2</b>	<b>Projecto</b>	<b>1</b>
2.1	Projectos de Demonstração . . . . .	1
2.1.1	sine8_buf . . . . .	1
2.1.2	loop_intr . . . . .	1
2.2	BPSK . . . . .	2
2.2.1	P1. Oscilador controlado numericamente . . . . .	2
2.2.2	P2. Transmissor BPSK . . . . .	9
<b>3</b>	<b>Conclusão</b>	<b>14</b>
<b>4</b>	<b>Anexos</b>	<b>14</b>
4.1	Anexo A . . . . .	14
4.2	Anexo B . . . . .	16

---

<sup>1</sup>As linhas de código apresentadas durante o relatório têm como objectivo demonstrar a maneira de raciocinar na resolução de problemas, não representando uma cópia exata do código usado em laboratório, podendo até, serem consideradas *pseudo-código*

# 1 Introdução

Este trabalho consiste na primeira parte do projeto de laboratório da cadeira: desenvolvimento de um modem de *Binary Phase Shift Keying* (BPSK).

Teve como objectivo a familiarização com o ambiente de desenvolvimento integrado de DSP que consiste nas placas de desenvolvimento DSK TMS320C6416 e DSK TMS320C6713 da *Texas Instruments* e no software de desenvolvimento *CodeComposerStudio v5.5*. Para tal foram estudados dois projetos exemplo (`sine8_buf` e `loop_intr`) e, usando as ferramentas de debug, alteraram-se certos parâmetros de forma a observar os efeitos nos sinais resultantes. Também se consolidaram os conhecimentos adquiridos desenvolvendo dois mini projetos: um oscilador sinusoidal controlado numericamente e um transmissor BPSK. Os projetos desenvolvidos foram incorporados nos anexos A e B, podendo analisar os mesmos com maior detalhe nos anexos.

## 2 Projecto

### 2.1 Projectos de Demonstração

Foram analisados dois projetos de demonstração com o intuito de familiarizar com os equipamentos e as principais rotinas do projeto.

#### 2.1.1 sine8\_buf

O objectivo deste projeto é representar a função sinusoidal, multiplicada por um determinado ganho, através de um conjunto de amostras que equivalem a um período da mesma, repetindo nos períodos seguintes esse mesmo conjunto. Este procedimento é realizado através da rotina de interrupção presente no programa.

Ao analisar o código deste projeto, à primeira vista podemos concluir logo que este usa uma frequência de amostragem de 8 kHz, tem um ganho  $G = 10$  predefinido e usa 8 amostras para representar a sinusoide. Depois de observar a sinusoide no osciloscópio variou-se o ganho a fim de perceber a sua influência e também o seu limite.

Para compreender o limite desta sinusoide é necessário ter em conta que se usa o formato de vírgula fixa Q15 para as suas amostras. Este formato tem como limite o valor  $(2^{15} - 1) = 32767$ . Considerando o valor máximo da sinusoide, se multiplicarmos a mesma por um ganho  $G = 33$  obtemos um valor superior ao permitido pelo formato Q15 (*overflow*), fazendo com que nesses pontos o valor da sinusoide "caia".

#### 2.1.2 loop\_intr

Este projeto fornece-nos um template para os próximos projetos, em termos de comunicação com a placa e rotina de interrupção. Pode-se observar nas últimas linhas de código como se liga os sinais de entrada e saída aos canais da placa.

No projeto anterior observou-se os efeitos de *overflow* de uma variável. Neste observam-se os efeitos de aliasing devido ao sinal de line-in ter a mesma frequência que a frequência de amostragem. A partir dos 4khz deixamos de ver uma sinusóide com os 3.3V. Não foi feita a experiência de mudar para sinal quadrado e variar a frequência, mas provavelmente não se iria ver um sinal quadrado pois o espectro (infinito) deste teria que ser filtrado pelo anti-aliasing filter.

## 2.2 BPSK

Este projeto é constituído por duas partes, a primeira é um oscilador controlado numericamente e a segunda é um transmissor BPSK. Devido à existência de um inversor na placa utilizada deu-se, ao longo deste projeto, atenção ao efeito do inversor nos sinais, como se poderá observar nas próximas secções.

### 2.2.1 P1. Oscilador controlado numericamente

Usou-se o projeto descrito na secção 2.1.2 como base para a construção de um oscilador controlado numericamente (*NCO*). Um *NCO* é um gerador de sinal digital que cria uma forma de onda discretamente representada no tempo e na amplitude. O *NCO* tem as seguintes características:

Tabela 1: Características do *NCO* a construir.

Parâmetro	Símbolo	Valor
Frequência de amostragem	$f_s$	16kHz
Frequência mínima	$f_{min}$	2kHz
Frequência máxima	$f_{max}$	6kHz

O projeto de construir o *NCO* seguiu os seguintes passos que estão posteriormente explicados:

1. Oscilador de Relaxamento (Integrador de Rampa).
2. Look-up-table (LUT).
3. Obtenção do sinal sinusoidal através da LUT.
4. Criação de duas variáveis para controlar a frequência e amplitude do sinal sinusoidal.
5. Utilização do sinal de entrada para controlar a frequência do sinal.
6. Teste do oscilador com uma onda quadrada à entrada para observar o sinal modulado.
7. Melhoria da qualidade do oscilador com interpolação linear.
8. Comparaçao dos espectros com e sem interpolação.

## P1-1

Visto que a placa usada no laboratório (DSK TMS320C6713) tem 1V de amplitude máxima na saída, foi usado o formato Q15 para representar o sinal de saída, pois um sinal representado em Q15 tem amplitude máxima de  $2^{15} - 1 = 32767$ .

Para construir um Oscilador de Relaxamento foi criada uma rotina que, em cada ciclo, incrementa a variável de amplitude do sinal de saída de modo a que este tenha o comportamento de uma rampa. Este incremento tem de tomar um valor, que por agora se vai considerar constante, tal que 32767 seja seu múltiplo. Tendo em conta as frequências da tabela 1, podemos chegar aos seguintes valores de incremento com as suas frequências correspondentes:

Tabela 2: Valores a atribuir ao incremento do sinal na saída.

Frequência de saída( $f_0$ )	Incremento( $\Delta$ )
$2kHz$ ( $f_{min}$ )	8192
$4kHz$	16384
$6kHz$ ( $f_{max}$ )	24576

Os valores na tabela 2 foram calculados através da seguinte fórmula:

$$f_0 = \frac{\Delta}{2A} f_s \Leftrightarrow \Delta = 2A \frac{f_0}{f_s} , A = 32767 \quad (1)$$

Foi então possível obter as seguintes rampas com diferentes frequências:

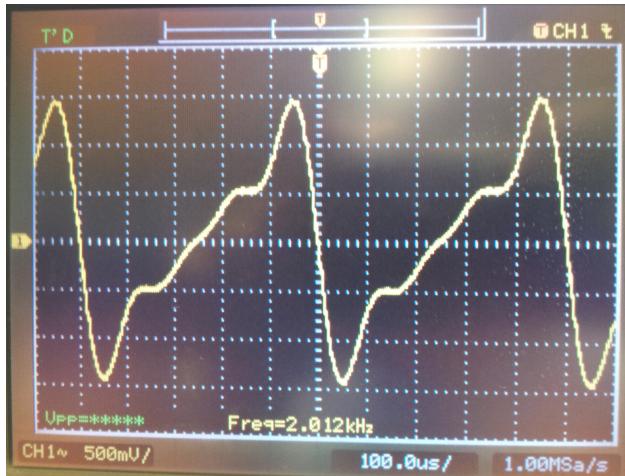


Figura 1: Rampa com frequência de  $2kHz$

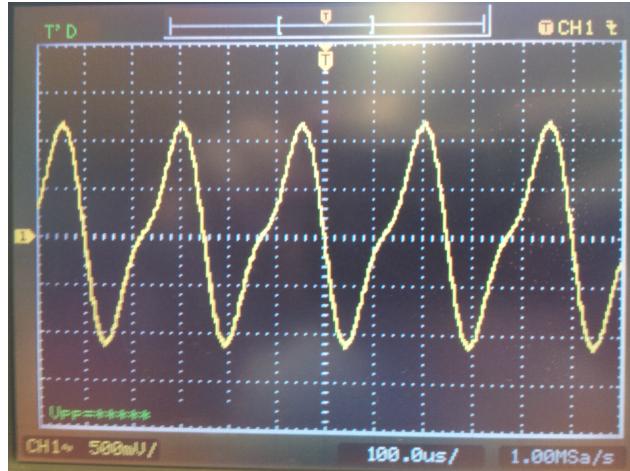


Figura 2: Rampa com frequência de 4kHz

Ao observar as duas rampas pode-se constatar que para uma frequência mais baixa, ou seja, um incremento menor, a rampa apresenta mais "degraus" e um menor declive. Um incremento menor na rampa significa um maior número de amostras da mesma dentro do intervalo possível, o que reduz o declive da rampa. É de salientar que devido à presença do inversor da placa foi introduzido o simétrico da rampa calculada no canal de modo a observar esta no sentido correto.

### P1-2

Foi criada uma tabela (LUT) com os valores em Q15 de meio ciclo de um seno de modo a que seja possível ir retirar os seus valores para criar um sinal sinusoidal. Esta tabela tem 32 valores e foi construída com a seguinte equação:

$$32767 * \sin \frac{n\pi}{32}, \quad n = 0, 1, 2, \dots, 31 \quad (2)$$

Para calcular apenas meio ciclo do seno, é necessário ter 32 valores pertencentes ao intervalo  $[0, \pi]$  na fase. Assim, é necessário restringir a fase a esse intervalo através da divisão observada na expressão,  $\frac{n\pi}{32}$ . O produto do seno pelo valor 32767 é a conversão do seno para Q15.

Não se multiplica por  $2^{15} = 32768$  pois os sinais estão em complemento para dois, o que significa que para o máximo do seno, esta LUT ultrapassaria o limite do formato, o que causaria o corte dessa amostra no sinal.

### P1-3

Usando como base a rampa criada na secção P1-1 para indexação da LUT criada na secção P1-2, foi possível criar um sinal sinusoidal.

De acordo com o projeto, a rampa é representada num formato Q10, com 1 bit sinal, 5 bits inteiros e 10 bits fracionários. Para a indexação são usados os 5 bits inteiros, ou seja, os 5 mais significativos (excluindo o bit de sinal) do sinal da rampa. Esses 5 bits irão apontar para o valor da tabela de LUT a usar para criar a sinusoide.

Este processo foi efetuado com o seguinte pedaço de código dentro do ciclo:

```

rampa=rampa+delta;      // Criar a rampa
index=rampa>>10;
index=31 & index;          // Usar apenas 5 bits
sinusoide=LUT[ index ];

```

Como se pode observar, para isolar os 5 bits inteiros fez-se dez *shifts* para a direita de maneira a ter um sinal apenas com zeros, o bit sinal e esses 5 bits encostados à direita. A seguir aplicou-se uma máscara constituída por uma AND com 31 bits, de maneira a retirar o bit sinal e finalmente isolar num sinal o índice pretendido da LUT. Com o índice é uma questão de aceder à tabela e obter  $y_1$ , neste caso representado pelo sinal *sinusoide* (Figura 3).

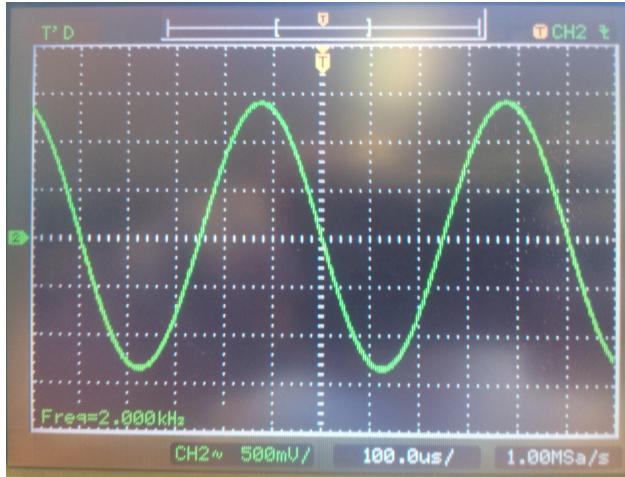


Figura 3: Sinal sinusoidal criado de 2kHz.

Como se pode observar na figura foi possível criar um sinal aproximadamente sinusoidal através de uma LUT indexada por uma rampa. Nesta figura não é possível notar mas quando se aproxima mais o sinal no osciloscópio verifica-se pequenos degraus ao longo do sinal, que simbolizam a aproximação obtida através das amostras.

#### P1-4

Foram criadas duas variáveis para fosse possível controlar a frequência e a amplitude do sinal à saída, *delta* e *ampl*, respectivamente.

A primeira variável foi já antes referida, nas secções P1-1 e P1-3. Esta, caso alterada irá alterar a frequência da rampa, e por consequência, a frequência do sinal de saída. Como se pode observar na tabela 2, esta variável terá como limite máximo o valor 24576 e como limite mínimo 8192 devido às frequências mínima e máxima. Podemos também concluir que quanto maior for esta variável, maior a frequência do sinal de saída, tal como se o seu valor diminuir, a frequência de saída irá diminuir também.

A segunda variável (*ampl*) foi criada com o propósito de modelar a amplitude do sinal de saída. Esta causou mudanças mais significativas no código, tal como se pode verificar:

```
rampa=rampa+delta;
```

```

index=rampa>>10;
index=31 & index ;
aux=ampl*LUT[ index ] ;
aux=aux<<1;           // Retirar bit de sinal extra
sinusoide=-aux>>16;   // Colocar o sinal com 16 bits em Q15

```

Esta variável tem como valor máximo 32767, pois os valores afixados na tabela LUT já apresentam um valor com a amplitude máxima de Q15 de maneira a não ter problemas com o formato de representação nem com a amplitude do sinal transmitido à placa. Como se pode ver no código a variável *ampl* multiplica diretamente pela LUT.

Quando se multiplica dois sinais, o sinal resultante fica com um bit sinal replicado que é eliminado através de um *shift* esquerdo. Mas depois ainda é necessário ir buscar os 16 bits mais significativos através de 16 *shifts* para a direita pois pretende-se um sinal com 16 bits em Q15.

Na equação seguinte pode-se observar como obter o formato resultante num produto:

$$Q_m * Q_k = Q_{m+k-n+1} \quad (3)$$

Neste caso como se multiplica dois sinais Q15 e  $n=16$  bits representa-se o resultado com formato Q15 como foi dito anteriormente. Com a criação destas variáveis passou-se a ter um oscilador numericamente controlado.

### P1-5

Para que a frequência do sinal de saída seja modelado através da amplitude do sinal de entrada, é necessário criar uma relação linear entre o sinal de entrada *inbuf* e o incremento da rampa *delta*, pois este é que define a frequência do sinal sinusoidal com a rampa. Como tal a relação entre o *inbuf* e o *delta* é ilustrada pela seguinte equação:

$$\Delta = \Delta_0 + (K * inbuf) \quad \Delta_0 = 16384, K = \frac{1}{4} \quad (4)$$

Tendo em conta a tabela 2, delta varia entre 8192 e 24576 com  $\Delta_0$  correspondente à frequência central, 4 kHz. Para restringir delta a esse intervalo foi necessário introduzir  $K = \frac{1}{4}$ . Assim, obtém-se uma relação proporcionalmente direta entre delta e o sinal de entrada, o que significa que se aumentar ou diminuir a amplitude do sinal de entrada também aumenta ou diminui o delta.

Pode-se observar a implementação da equação explicada anteriormente, no código seguinte:

```

delta=16384-(inbuf>>2);
rampa=rampa+delta ;           // rampa

```

O duplo *shift* para a direita é equivalente a dividir por 4 e mais eficiente pois evita uma passagem do programa pela ALU, também representa o efeito de K. Embora na equação (4) se some o efeito do *inbuf*, no programa tem que se ter em conta o efeito do inversor na placa, por isso se subtraí esse efeito.

### P1-6

Para testar o funcionamento do NCO, pôs-se na entrada do DSP um sinal sinusoidal com frequência igual a 2 Hz, como sinal modulante, em vez da onda quadrada pedida no enunciado pois seria mais fácil observar a variação de frequência do sinal de saída se a variação de amplitude do sinal de entrada fosse gradual em vez de ser constante com descontinuidades.

Esperou-se que, no máximo e mínimo de amplitude da sinusoide de entrada, a onda modulada tivesse máxima e mínima frequência respetivamente, e que uma variação na amplitude do sinal modulante provocasse um efeito linear na frequência do sinal modulado. Tal observou-se no osciloscópio, contudo não é possível representar numa figura para referência uma vez que seria necessário uma gravação de vídeo do processo para poder observar a variação pretendida.

### P1-7

Com o objectivo de melhorar a qualidade do sinal criado vai ser usado o método de interpolação linear descrito na seguinte equação:

$$y = y_1 + (y_2 - y_1) * \Delta_x \quad (5)$$

Sendo que  $\Delta_x$  refere-se aos 10 bits menos significativos ignorados na altura em que se retirou da rampa o índice da LUT. O processo de recolher o valor de  $\Delta_x$  e do cálculo da interpolação estão descritos nas seguintes linhas de código:

```
rampa=rampa+delta ;
deltaX=1023 & rampa;      // Retirar os 10 bits menos significativos
deltaX=deltaX<<5;        // Converter deltax em Q15
( . . . )
Y1=-((amplitude*LUT[index])<<1)>>16;    // Y1=LUT(i) convertido para Q15
Y2=-((amplitude*LUT[index+1])<<1)>>16;    // Y2=LUT(i+1) convertido para Q15
aux=(Y2-Y1)*deltaX ;
aux=aux<<1;                  // Retirar bit de sinal extra
Y=Y1+(aux>>16);           // interpolacao
```

Como se pode observar nas linhas de código, a interpolação é realizada com dois valores que são retirados da tabela LUT, usando como índice *index* e (*index*+1) respectivamente. Neste método com interpolação estaremos a usar 10 bits, que no método anterior (sem interpolação) foram descartados fazendo um sacrifício na precisão, estes bits constituem a variável  $\Delta_x$  que será usada na equação da interpolação.

Para isolar a mesma foi usada uma máscara primeiro para retirar os 10 bits que interessam e depois efetuaram-se 5 *shifts* para a esquerda de modo a converter para Q15.

Ambos os sinais (com e sem interpolação) podem ser observados na Figura 4, embora a sua comparação não possa ser efetuada por mera observação.

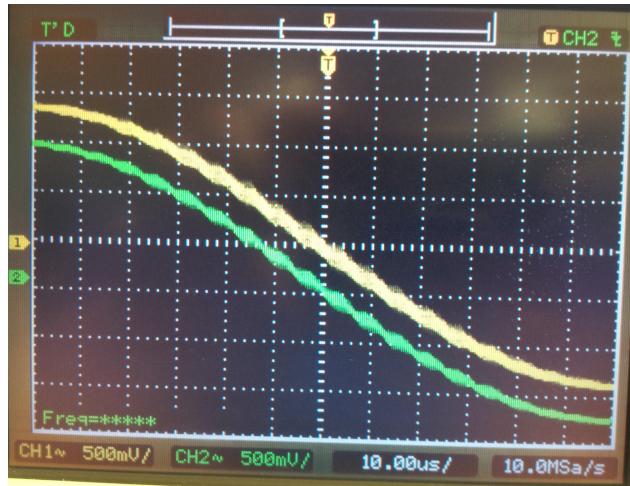


Figura 4: Sinal sinusode com interpolação e sem interpolação.

Nesta figura é possível observar os "degraus" mencionados anteriormente, tanto na onda interpolada como na onda não interpolada.

#### P1-8

Não sendo possível estabelecer uma diferença entre os sinais com interpolação e sem interpolação foi necessário recorrer à observação dos espectros deles mesmos (Figuras 5 e 6).

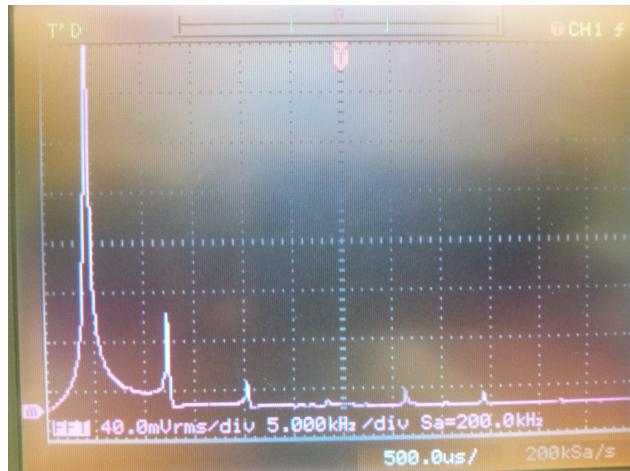


Figura 5: Espectro do sinal sinusode sem interpolação.

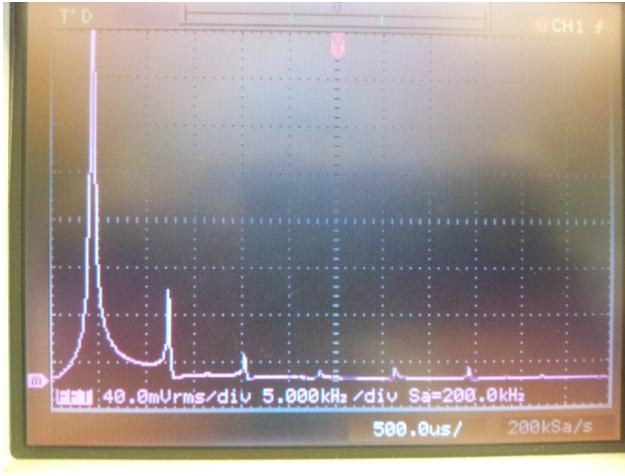


Figura 6: Espectro do sinal sinusoide com interpolação.

De novo não foi possível observar nas figuras diferenças visíveis entre os dois sinais, desta vez entre os seus espectros. Apresentam aparentemente a mesma amplitude para cada frequência entre eles mesmos.

A conclusão que se pode tirar com estas observações é que o método de interpolação, embora apresente uma melhor precisão dos valores, não apresenta uma melhoria nos resultados, sendo assim um método que apenas irá exaustar mais os recursos do processador sem obter qualquer tipo de retorno notável. Podemos então concluir que caso seja preciso usar um sinal digital de uma sinusoide podemos usar apenas o método que não usa interpolação.

É de referir que não se está a excluir a hipótese da interpolação melhorar o sinal. É afirmado que os pontos têm uma melhor precisão, mas as melhorias não são observáveis pelos métodos e equipamentos usados.

### 2.2.2 P2. Transmissor BPSK

O objectivo deste projeto é criar um transmissor BPSK com recurso a três elementos principais, uma fonte de bits, um codificador diferencial e mapeador, e um modulador. Neste projeto foi utilizada uma frequência de amostragem  $f_s = 16$  kHz e uma frequência portadora  $f_0 = 4$  kHz.

#### P2-1

Para ter uma fonte de bits no transmissor usa-se um "bit-rate clock" cuja função vai ser criar uma sequência de bits  $b_n$  com  $f_b = 1$  kbps. Isto significa que, considerando  $f_s$ , a cada 16 ciclos é gerado um novo bit, alternado em relação ao anterior. Assim, implementou-se um contador que é incrementado em cada ciclo e que tem uma condição para verificar quando chegar ao valor 16. Ao entrar nessa condição é implementada a lógica para cálculo do novo bit da sequência e o contador é reiniciado.

Para calcular o novo bit basta negar o bit anteriormente obtido de modo a obter uma sequência

de bits alternada, tendo sido concretizado este raciocínio com recurso uma simples XOR:

$$b_n = b_{n-1} \oplus 1 \quad (6)$$

Com esta operação, o bit resultante será sempre alternado em relação ao anterior. Assim, obtém-se uma onda quadrada, observada no osciloscópio, que varia entre "0"e "1"e representa  $b_n$  com uma frequência de 500 Hz(Figura 7). Esta frequência deve-se ao fato de dividir-se  $f_b$  por dois pois cada meio ciclo da onda quadrada corresponde a um bit.

Após obter a fonte de bits passou-se ao segundo elemento do transmissor: o codificador diferencial e mapeador. Começando pelo codificador diferencial, este serve para evitar ambiguidades na fase de maneira a poder sempre recuperar uma sequência de bits num canal que tenha sofrido uma variação na fase. A codificação de  $b_n$  resulta também de uma operação lógica XOR, como se pode observar:

$$c_n = c_{n-1} \oplus b_n , c_0 = 0 \quad (7)$$

Como um bit novo só é calculado quando o contador chega ao valor 16, o mesmo também só é codificado nessa condição, ou seja, a cada 16 ciclos é codificado um bit.

Tal como em  $b_n$  também  $c_n$  é representado por uma onda quadrada que varia entre "0"e "1"só que com o dobro do período, devido a só ter dois resultados para os quatro casos possíveis da XOR que realiza a codificação(Figura 7).

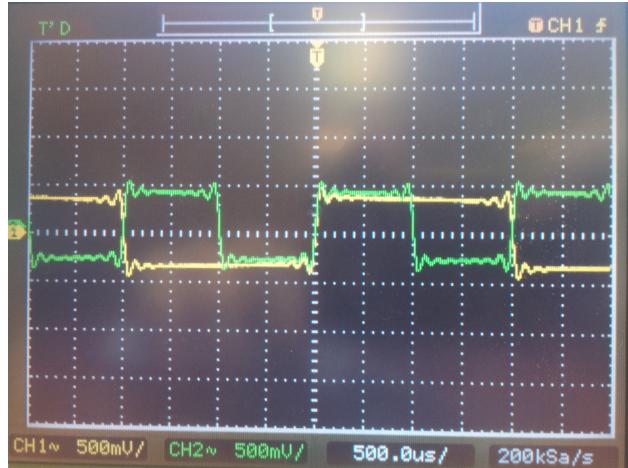


Figura 7:  $b_n$ (verde) e  $c_n$ (amarelo)

Depois de obter  $c_n$  passa-se ao mapeamento do mesmo. O mapeamento baseia-se em duas condições:

$$c_n = '1' \rightarrow d_n = +1 \quad c_n = '0' \rightarrow d_n = -1 \quad (8)$$

Esta lógica podia ser facilmente implementada com recurso a duas condições "if", mas optou-se por evitar essa lógica para tornar o programa mais eficiente. Assim recorreu-se a um *shift* e a uma subtração. Atenção que esta não é a maneira mais eficiente pois a subtração faz com que o

programa tenha de passar pela ALU. Pode-se observar então o mapeamento efetuado através da seguinte expressão:

$$d_n = 32767 * ((c_n << 1) - 1) \quad (9)$$

Antes de mais, o ganho que está a ser multiplicado é utilizado para converter o resultado em Q15, como já foi feito anteriormente. Considerando a expressão sem esse ganho, vê-se que para  $c_n = 0$ , como o *shift* não influencia o resultado, o mesmo só depende da subtração e é sempre o pretendido,  $d_n = -1$ . Se  $c_n = 1$ , o *shift* duplica esse valor, e depois ao subtrair obtém-se  $d_n = 1$ .

Tal como em  $b_n$  e  $c_n$  esta operação só é executada a cada 16 ciclos pois depende diretamente de  $c_n$  e só se mapeia um novo bit depois de ele ser codificado. Concluído o mapeamento obtém-se mais uma vez uma onda quadrada mas desta vez varia entre -1"e "1"(figura 8).

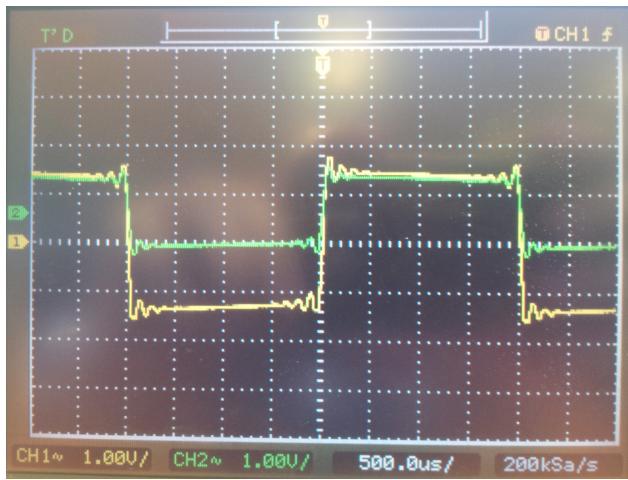


Figura 8: Formas de onda de  $d_n$  (amarelo) e  $c_n$  (verde).

## P2-2

Falta agora gerar a onda portadora a modular. Esta foi implementada de forma mais simples face ao projeto anterior uma vez que a frequência é estática (4 kHz) e este valor consiste numa fração inteira da frequência de amostragem (16kHz). Em primeiro lugar criou-se uma tabela com quatro valores dum período da sinusoida, sendo esta:

Tabela 3: Amostras da onda portadora

contador	seno
0	0
1	32767
2	0
3	-32767

Escolheram-se estes valores uma vez que o período de amostragem coincide com os instantes de máximo, mínimo e zero-crossing da portadora. Para gerar a portadora (Figura 9) recorreu-se a um

contador que, em cada interrupção (ocorrendo em cada instante de amostragem, como explicado no enunciado), aponta para cada entrada da tabela e põe a amostra numa variável que, após se incrementar o contador, irá ser multiplicada por  $d_n$ . Esta tabela não gera uma onda triangular pois das harmónicas destas, abaixo da frequência de amostragem ( $f_0=4\text{kHz}$  e  $3f_0=12\text{kHz}$ ) apenas a primeira harmónica se encontra na banda de passagem do filtro de reconstrução do DAC, que terá frequência de corte  $F_s/2$ .

A onda modulada é então representada pela seguinte expressão:

$$s_n = d_n \sin(2\pi f_0 T_s n) \quad (10)$$

Neste caso os valores do seno são os valores das amostras discutidas anteriormente.

### P2-3

Após implementar o modulador, tem-se o transmissor BPSK completo e para testá-lo, pôs-se nos dois canais do osciloscópio a onda portadora e a onda modulada (Figura 9).

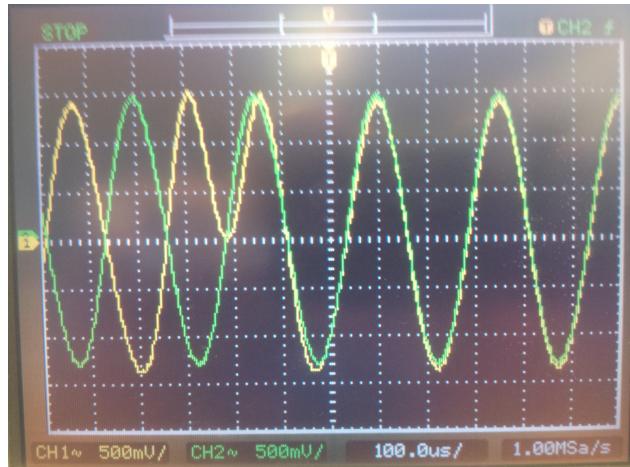


Figura 9: Onda portadora (verde) e onda modulada (amarelo)

Como se pode observar pela imagem e considerando a expressão (8), a onda portadora começa em oposição de fase com a onda modulada. Isto ocorre quando  $d_n = -1$ , o que significa que quando  $d_n = 1$  a onda modulada é igual à portadora. É possível observar quando  $d_n$  muda de valor pois é exatamente quando a onda modulada inverte a sua fase (Figura 10).

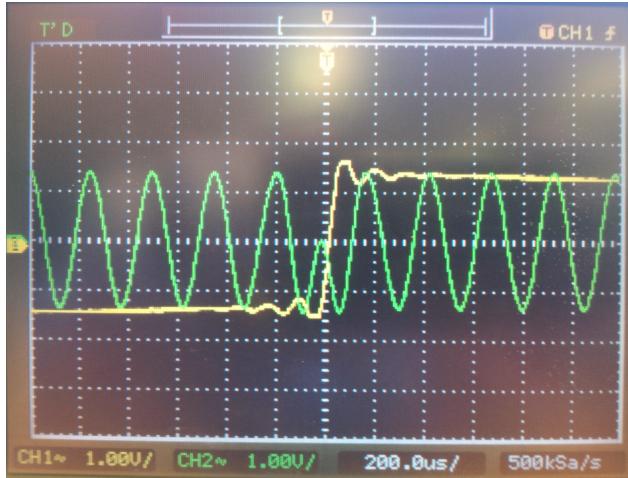


Figura 10: Formas de onda de  $d_n$  (amarelo) e  $s_n$  (verde).

Este fenômeno acontece a cada 8 ciclos pois, como já foi explicado anteriormente, a frequência da onda correspondente à fonte de bits é o bit-rate dividido por dois e uma vez que a codificação divide esta frequência por dois, ou seja, a frequência de  $d_n$  é 250 Hz e por isso cabem oito ciclos da portadora num meio ciclo de  $d_n$ .

Para compreender melhor o sinal modulado, observou-se o espetro do mesmo (Figura 11).

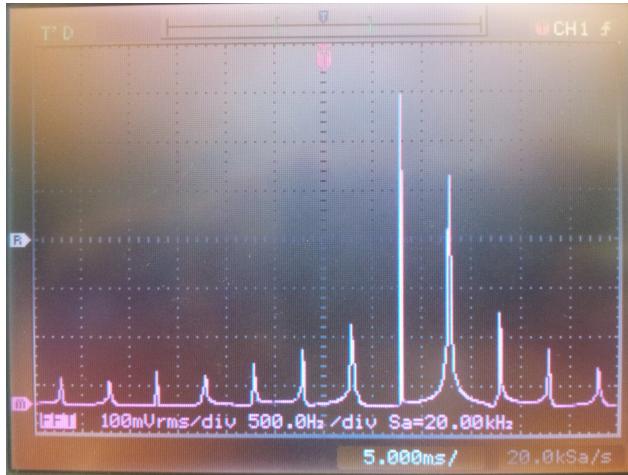


Figura 11: Espectro do sinal  $s_n$ .

A onda modulada no tempo consiste na multiplicação de uma onda quadrada com uma onda sinusoidal, na frequência isto consiste na convolução do espetro da primeira (impulsos nos múltiplos ímpares da frequência fundamental) com o espetro da segunda (um impulso à frequência  $f_0$ ). Isto resultaria numa translação do espetro da onda quadrada, ficando centrado na frequência da onda sinusoidal.

Na imagem acima vê-se o espetro de  $d_n$  centrado em 4kHz como esperado. Como se pode observar, as harmónicas de maior amplitude estão espaçadas 250 Hz em relação a  $f_0$  indicando que a frequência fundamental da onda quadrada é 250 Hz, como esperado.

### 3 Conclusão

-Principais resultados e conclusões sobre eles, erros a corrigir (se houverem), o que melhorar

### 4 Anexos

#### 4.1 Anexo A

Oscilador Controlado Numericamente:

```
#include "dsk6713_aic23.h"                                //codec-DSK support file
#include "C6713dskinit.h"

Uint32 fs=DSK6713_AIC23_FREQ_16KHZ;                      //set sampling rate

short rampa=0;
short delta ;
int index=0;
short LUTresultados=0;
short x[32];
short LUT[32];
short Y1=0;
short Y2=0;
short Y;
short deltaX;
short ampl;
int aux;
char intflag = FALSE;
union {Uint32 samples; short channel[2];} AIC_buffer;

interrupt void c_int11()                                     //interrupt service routine
{
    output_sample(AIC_buffer.samples);                     //output data
    AIC_buffer.samples= input_sample();                   //input data
    intflag = TRUE;
    return;
}

void main()
{
```

```

short          inbuf ;
short LUT[32]={0,3212,6393, 9512, 12540, 15447, 18205, 20788, 23170, 25330, 27246
comm_intr();                                //init DSK, codec, McBSP

ampl=32766;                                //0.5*32767
delta=16384;
inbuf=0;

while(1){                                 //infinite loop
    if(intflag != FALSE){
        intflag = FALSE;
        //deltamin=8192;
        //deltamax=24576;
        delta=16384-(inbuf>>2);
        rampa=rampa+delta;                      //rampa
        deltaX=1023 & rampa;
        deltaX=deltaX<<5;                     //isola deltaX
        index=rampa>>10;
        index=31 & index;                       //isola indice
        aux=ampl*LUT[index];
        aux=aux<<1;                           //elimina sinal replicado
        if(rampa<0){
            LUTresultados==aux>>16;
        } else{
            LUTresultados=aux>>16;
        }
        aux=ampl*LUT[index+1];                  //calculo de Y2
        aux=aux<<1;
        if(rampa<0){
            Y2==aux>>16;
        } else{
            Y2=aux>>16;                      // Y2=LUT(i+1) convertido para Q15
        }
        Y1=LUTresultados;                      // Y1=LUT(i) em Q15
        aux=(Y2-Y1)*deltaX;
        aux=aux<<1;
        Y=Y1+(aux>>16);                    //interpolacao
    }
}

```

---

```

    inbuf = AIC_buffer.channel[LEFT];           // faz loop do canal esquerdo
    AIC_buffer.channel[LEFT] = -LUTresultados;
    //inbuf = AIC_buffer.channel[RIGHT];        // faz loop do canal direito
    AIC_buffer.channel[RIGHT] = -Y;

}
}

}

```

## 4.2 Anexo B

Transmissor BPSK:

```

#include "dsk6713_aic23.h"                      //codec-DSK support file
#include "C6713dskinit.h"

Uint32 fs=DSK6713_AIC23_FREQ_16KHZ;            //set sampling rate

short cont= 1;
short cont2= 0;
short bn= 1;
short cn= 0;
short dn = -32767;
short sin[4]={0,32767,0, -32767};
short mod = 0;
short sn = 0;
char intflag = FALSE;
union {Uint32 samples; short channel[2];} AIC_buffer;

interrupt void c_int11()                         //interrupt service routine
{
    output_sample(AIC_buffer.samples);           //output data
    AIC_buffer.samples= input_sample();          //input data
    intflag = TRUE;
    return;
}

void main()
{
    //short inbuf;

```

```

comm_intr();                                // init DSK, codec, McBSP

//inbuf=0;

while(1){                                     // infinite loop
    if(intflag != FALSE){
        intflag = FALSE;
        if(cont==16){                         // condicao para criacao de novo bit
            cn=bn^cn;                        // codificacao de bn
            bn=bn ^ 1;                       // calculo de novo bit
            cont=0;                           // reset de contador
            dn=32767*((cn<<1)-1);       // mapeamento do bit codificado
        }
        cont=cont+1;
    }

    mod = sin [cont2];
    cont2= cont2+1;
    sn=((dn*mod)<<1)>>16;           // Q15 virgula fixa
    cont2=cont2&3;

    //LUT[bn]= AIC_buffer.channel[LEFT];      // faz loop do canal esquerdo
    AIC_buffer.channel[LEFT] =dn;
    //inbuf = AIC_buffer.channel[RIGHT];      // faz loop do canal direito
    AIC_buffer.channel[RIGHT] = sn;
}

}
}

```