



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	2
3	Projecto #1 - NCO	4
4	Projecto #2 - Transmissor BPSK	13
5	Projecto #3 - Receptor BPSK	22
6	Conclusões	39
7	Anexos	41

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 e receptor *binary phase-shift keying* (BPSK). Na figura seguinte apresenta-se um *modem* BPSK, sendo que os módulos a azul são os que se pretende implementar.

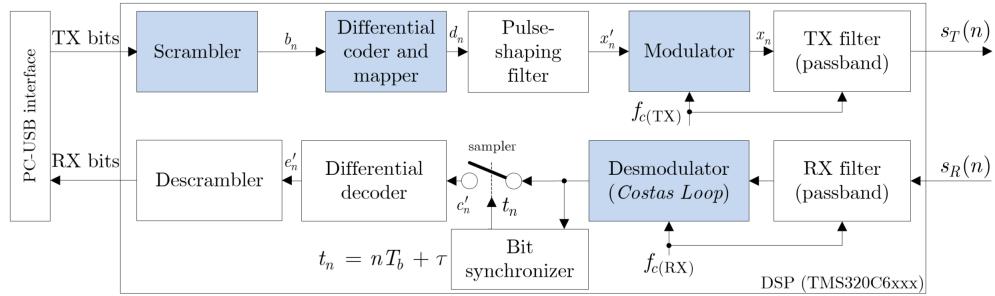


Figura 1: *Modem* DSP-BPSK.

Na Figura 2 está representado o *subset* do *modem* BPSK a implementar.

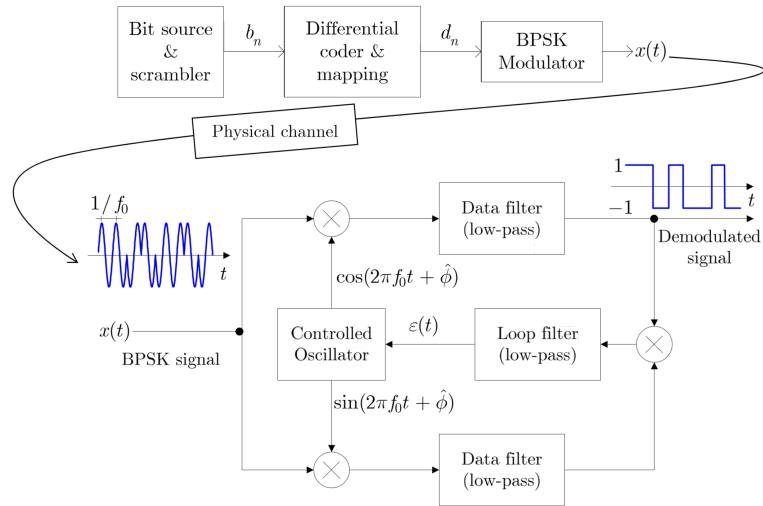


Figura 2: *Subset* do *modem* BPSK a implementar.

2 Projecto de Demonstração

Com o objectivo de fazer uma adaptação às ferramentas de trabalho realiza-se um projeto de demonstração que reproduz um seno através de uma *look-up-table* (LUT) de 8 posições, como se pode ver no excerto de código seguinte.

```
1   ...
2   //LUT do seno
3   short sine_table[8] = {0,707,1000,707,0,-707,-1000,-707};
4   ...
```

Esta LUT é lida num *loop* e os seus valores são multiplicados por um factor de ganho antes de serem exibidos no osciloscópio, ou seja, ocorre uma multiplicação de duas variáveis do tipo **short** com representação em Q_{15} . O formato do resultado da multiplicação de duas variáveis em Q_{15} é 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*.

Em complemento para dois e numa notação do formato Q_{15} pode-se representar valores no intervalo $-2^0 \leq x \leq 2^0 - 2^{-15}$, o que corresponde a valores decimais entre -32767 e 32767. Quando é atingido o valor máximo, 32767, entra em efeito a circularidade da representação em complemento para dois e, assim, para este caso, o valor máximo da amplitude do seno não atinge o valor 2^{15} , “dando a volta” para -32767. Com um valor inicial para o ganho de 10, o valor máximo de amplitude que ocorre no seno gerado é de $1000 \times 10 = 10000$, valor que se encontra compreendido no intervalo referido anteriormente. Também para um factor de ganho de 32 o valor máximo da amplitude do seno pode ser representada no formato Q_{15} . Porém, quando o factor de ganho passa para 33 esse valor máximo passa para $1000 \times 33 = 33000$, valor que gera um *overflow*, como explicado anteriormente.

Para um factor de ganho de 10 os valores do seno para 8 iterações do *loop* que o gera são $\{0, 7070, 10000, 7070, 0, -7070, -10000, -7070\}$ e para um factor de ganho de 33 os valores são $\{0, 23331, -233, 23331, 0, -23331, 233, -23331\}$. O valor de 233 deriva do *overflow* gerado e a amplitude da sinusoide fica com o valor $33000 - 32767 = 233$.

Analizando os valores anteriores verifica-se que a sinusoide com o factor de ganho de 33 tem uma frequência que é o triplo da frequência da sinusoide com o factor de ganho de 10, como se pode ver na figura da próxima página.

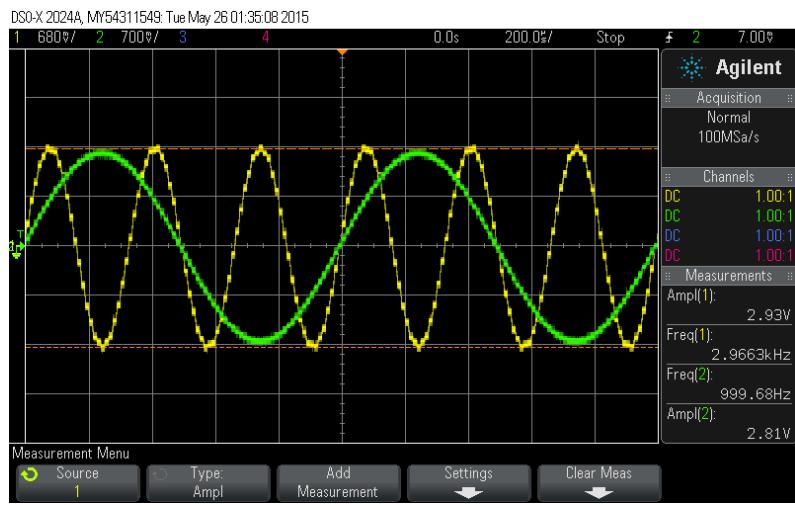


Figura 3: Sinusoide gerada para um factor de ganho de 10 (a verde) e para um factor de ganho de 33 (a amarelo).

Verifica-se que a onda a verde tem uma frequência próxima de 1 kHz e a onda a amarelo tem uma frequência próxima de 3 kHz, ou seja, o triplo da frequência da sinusoide que tem um factor de ganho de 10.

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 tabela seguinte.

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.

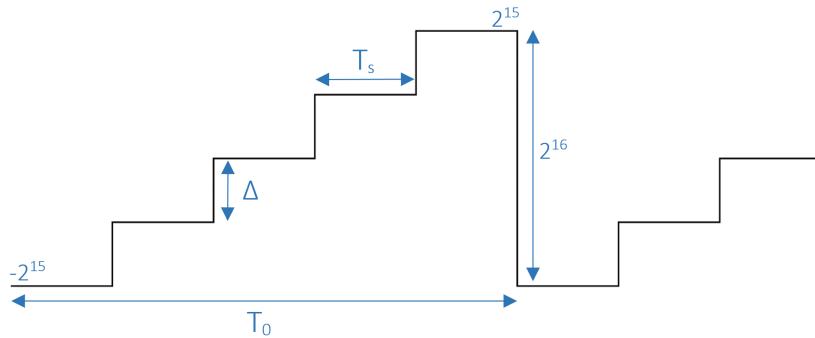


Figura 4: Esquema do oscilador de relaxação.

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 .

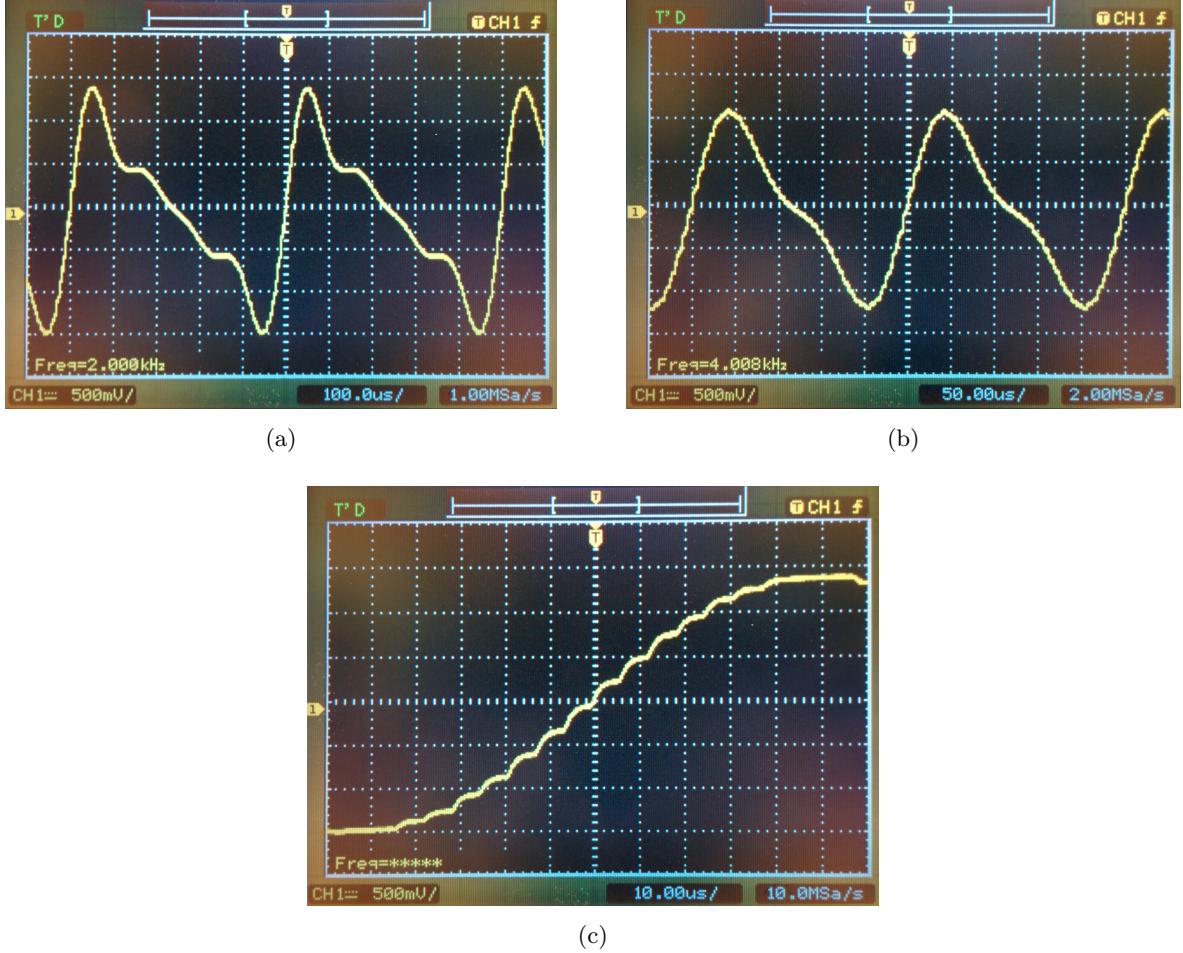


Figura 5: Oscilador de relaxação para $f_0 = 2$ kHz (a), oscilador de relaxação para $f_0 = 4$ kHz (b) e pormenor da rampa criada (c).

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.

Com o oscilador implementado pretende-se agora criar uma 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:

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

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, sendo a variável `i` incrementada, de acordo com as seguintes linhas de código:

```

1   ...
2   i = i&31;
3   y1 = sine[i];
4   y2 = sine[(i+1)];
5   i++;
6   ...

```

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

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 .

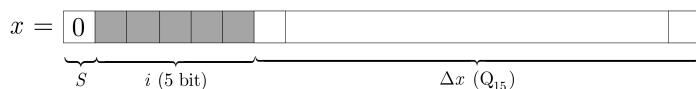


Figura 6: Representação da variável do estado do oscilador.

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   ...

```

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, somando `delta` com `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 de frequencia
3 yf = y1 + delta;
4 ...

```

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 ...

```

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, ressalvando que existe um atraso de processamento como se pode verificar na figura seguinte em que os sinais apresentam um ligeiro desfazamento face ao que se tinha esperado.

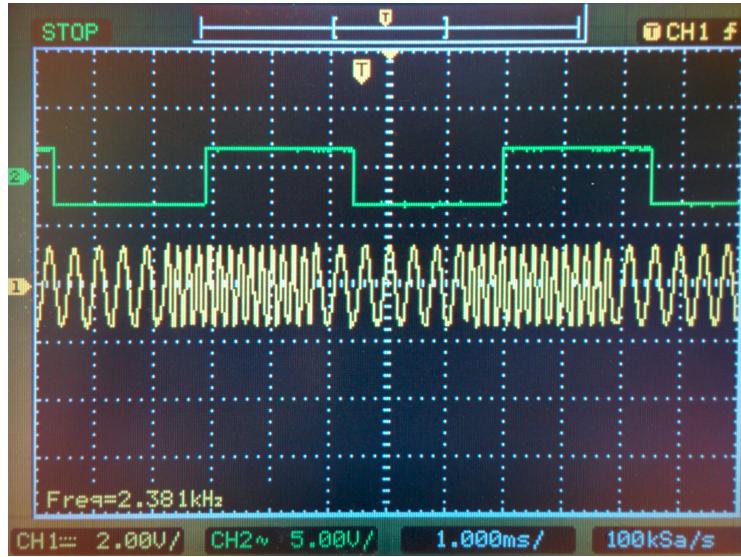


Figura 7: Saída modulada na frequência do oscilador controlado (a amarelo) com *input* de uma onda quadrada de 200 Hz (a verde).

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 6, 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.

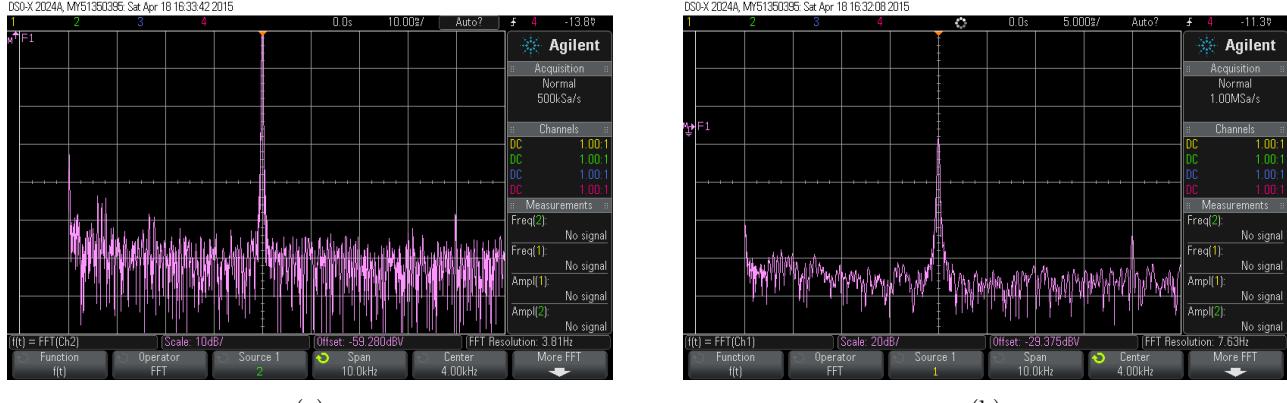


Figura 8: Espectro do sinal de saída sem interpolação (a) e com interpolação (b).

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



Figura 9: 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 10.

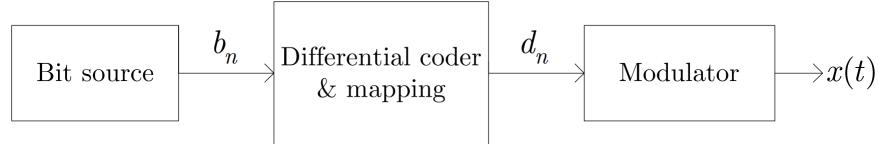


Figura 10: Esquema de um transmissor BPSK.

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.

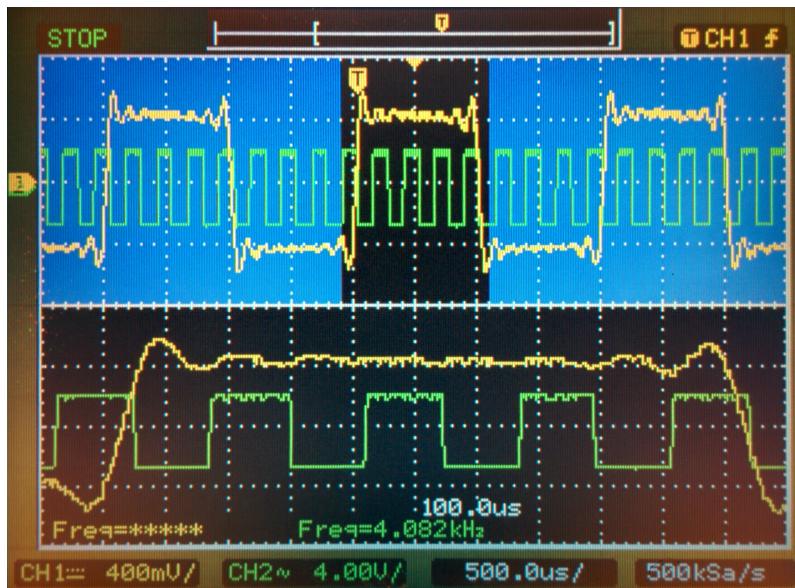


Figura 11: Sinal b_n (a amarelo) sobreposto com uma onda quadrada com frequência de 4 kHz (a verde).

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, a verde, gerada pelo gerador com uma frequência de 4 kHz com o intuito de saber o número de períodos que *bit* tem), 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, $4 \text{ kHz}/8 = 500 \text{ Hz}$.

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 ...

```

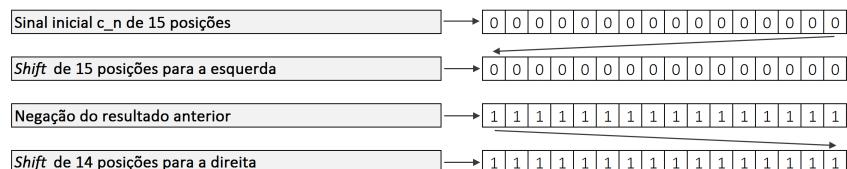
De seguida, aplica-se o mapeamento aos *bits* do sinal c_n na constelação BPSK, de forma a obter o sinal d_n , ou seja, os dois símbolos da constelação: 1 e -1 . O mapeamento segue as expressões da equação (4.1) e (4.2).

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 ...

```



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

Figura 12: Representação da situação em que $c_n = 0$, resultando num mapeamento de $d_n = -1$.

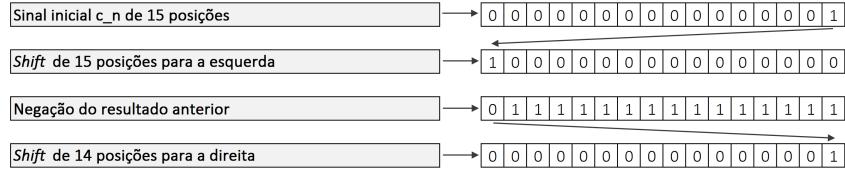


Figura 13: Representação da situação em que $c_n = 1$, resultando num mapeamento de $d_n = +1$.

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} :

```

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.

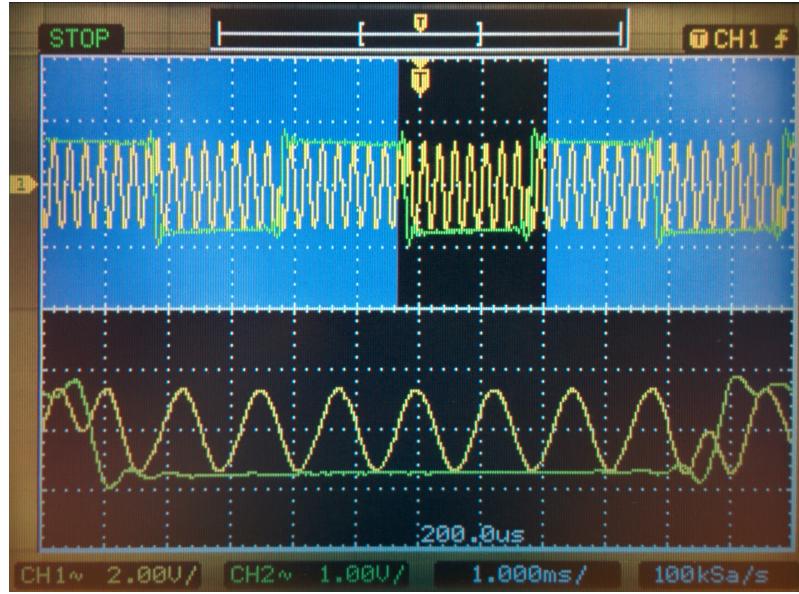


Figura 14: Sobreposição do sinal modulado (a amarelo) com o sinal d_n (a verde).

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

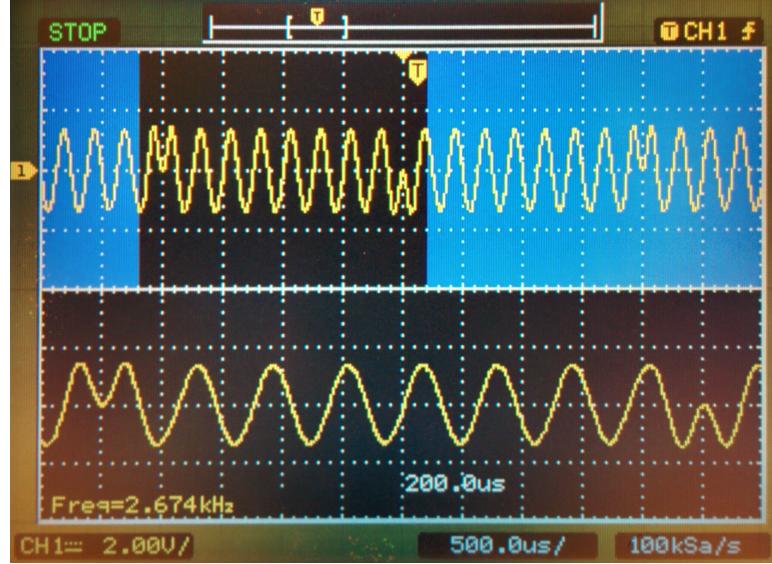


Figura 15: Sinal modulado (em cima) e pormenor obtido com recurso a uma janela temporal (em baixo).

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

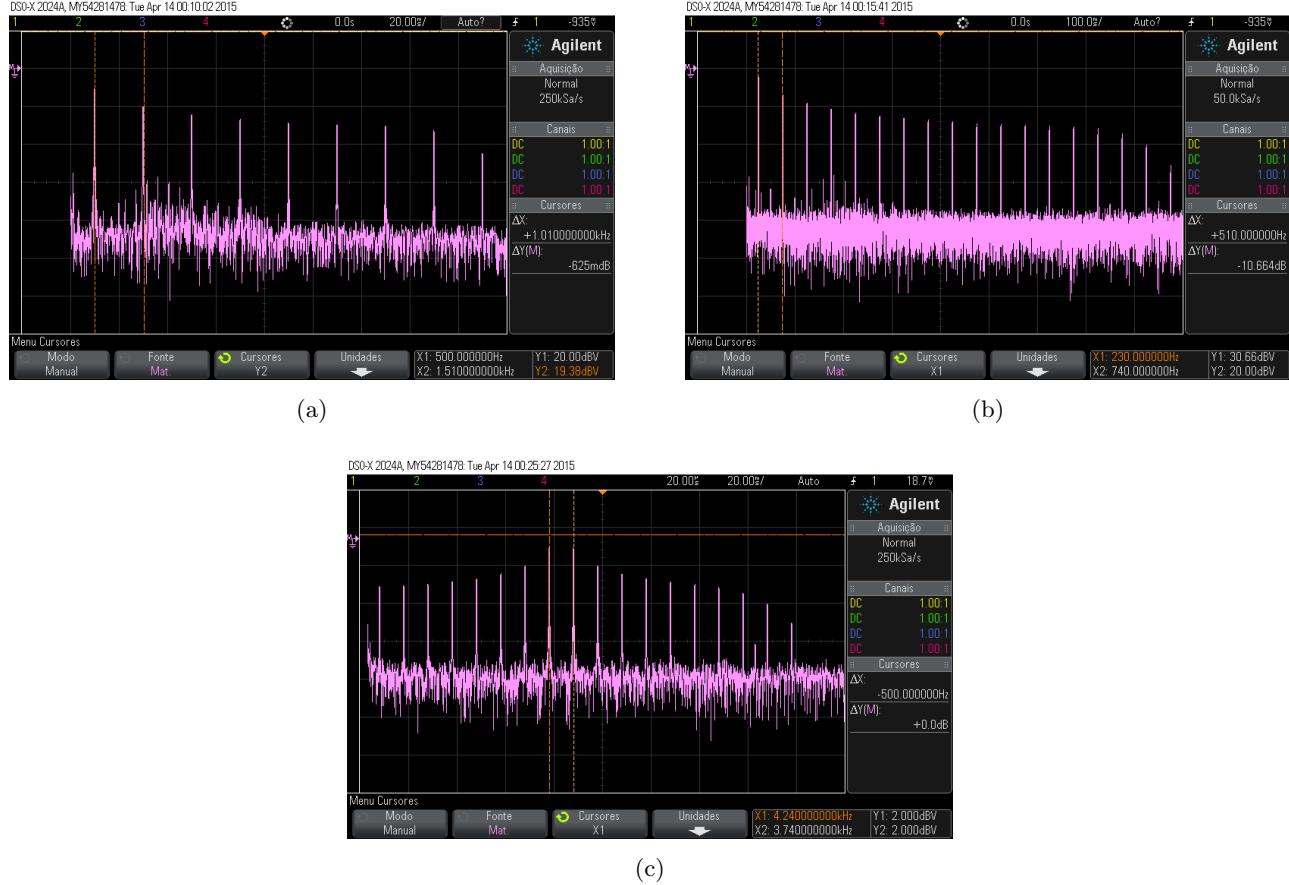


Figura 16: Espectro do sinal b_n (a), espectro do sinal d_n (b) e espectro do sinal y_n (c).

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 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 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.

Analizando agora a Figura 1 verifica-se que do transmissor BPSK ainda falta implementar o módulo de *scrambler*. Nesta altura do desenvolvimento do projecto já estão implementados os módulos de geração de bits, de codificação diferencial e de modulação, estando o sinal BPSK, $x(t)$, disponível.

A geração de bits que se implementou é, no fundo, um simples gerador de uma sequência de bits bem definida ($b_n = 1, 0, 1, 0, \dots$). No entanto, esta sequência não é apropriada para transmissão e, como tal, a sequência referida anteriormente necessita de ser “misturada” de modo a que surja de forma aleatória. Para se efectuar isto recorre-se a um circuito denominado de *scrambler*.

O *scrambler* é um dispositivo que inverte a ordem e polaridade dos bits da mensagem a ser

enviada. A utilização deste dispositivo tem dois principais objectivos - a facilidade de execução dos circuitos de recuperação de relógio, *clock and data recovery*, e outros circuitos adaptativos do receptor. Esta melhoria deve-se ao facto do *scrambler* eliminar longas sequências constantes de 0 ou 1. O outro objectivo deve-se ao facto de eliminar a dependência do sinal a ser transmitido no espectro de potência, ou seja, o sinal transmitido fica mais disperso de forma a maximizar os requisitos do espectro de potência. Se a potência for concentrada numa largura de banda limitada pode haver interferência nos canais adjacentes do receptor devido à intermodulação, este fenómeno é conhecido por *cross-modulation*.

O circuito do dispositivo *scrambler* tem como expressão matemática o seguinte polinómio:

$$P(x) = 1 + x^{-6} + x^{-7} \quad (4.6)$$

Os valores de x^{-6} e x^{-7} representam sinal de saída com um atraso de 6 e 7 períodos, respectivamente. A implementação do circuito segue a seguinte equação e esquema, e_n sinal de saída e b_n sinal de entrada no *scrambler*

$$e_n = b_n \oplus e_{n-6} \oplus e_{n-7}; \quad (4.7)$$

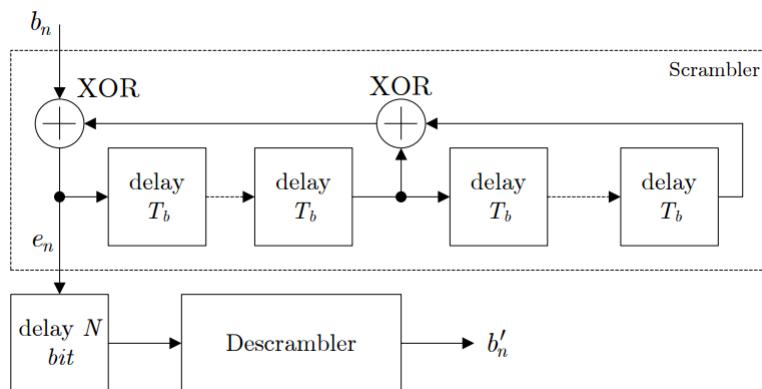


Figura 17: Esquema da implementação do *scrambler*.

Analizando o esquema proposto efectuou-se a implementação do circuito na DSP com o apoio do código C. Começou-se por definir uma variável de 8 bits, `delay_scrambler`, que guarda os atrasos temporais até 7 períodos. O código seguinte demonstra a implementação e obtenção dos atrasos necessários para o *scrambler*. É de referir que o sinal de entrada corresponde à sequência de bits `b_n` gerada anteriormente.

```
1 unsigned char delay_scrambler = 0;
2
3 ...
4 delay_scrambler = (delay_scrambler>>1)&0x7f; // deslocamento temporal
5 delay_scrambler = delay_scrambler|(e_n<<7); //actualizacao do novo valor de e_n
no vector
6 ...
7
```

Assim sendo, o *bit* mais significativo representa o resultado do *scrambler* com atraso 0 e, consequentemente, o *bit* menos significativo representa o atraso 7 na linha temporal. Seguindo esta lógica o vector `delay_scrambler` é actualizado da seguinte forma:

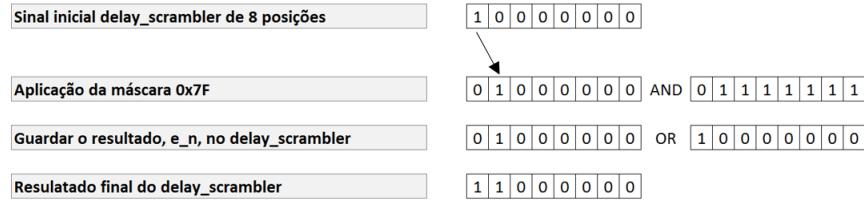


Figura 18: Esquema do cálculo do `delay_scrambler`.

Inicialmente é realizado um *shift* de uma posição para a direita no vector `delay_scrambler`, representando o deslocamento temporal de cada amostra. De seguida, aplica-se a máscara `0x7f` de forma a obter um 0 no *bit* mais significativo e mantendo os valores das amostras deslocadas. Com o deslocamento feito recorre-se à operação lógica OR para guardar o resultado do *scrambler* no atraso temporal correspondente.

Com a actualização do vector `delay_scrambler` realizada, a obtenção do resultado do *scrambler* é obtida da seguinte forma:

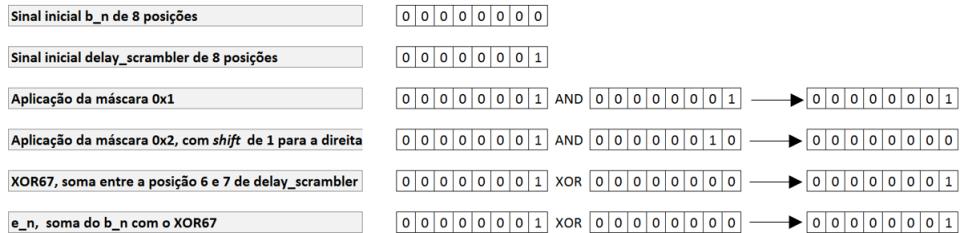


Figura 19: Esquema do cálculo da saída do *scrambler*.

Começa-se por somar os resultados do *scrambler* referente ao atraso 6 e 7, guardando o resultado na variável `xor67`. De seguida, soma-se com `b_n` originando `e_n`. A implementação em código segue no excerto seguinte.

```

1 short    b_i_TX = 1;
2 short    e_n = 0, xor67 = 0;
3 short    b_n = 1;
4 unsigned char delay_scrambler = 0;
5
6 ...
7 if(b_i_TX > 15){
8     b_i_TX = 0;
9     b_n = (b_n^1);
10
11     /* scrambler */
12     xor67 = (delay_scrambler&1)^((delay_scrambler&2)>>1);
13     e_n = b_n^xor67;
14     delay_scrambler = (delay_scrambler>>1)&0x7f; //deslocamento temporal

```

```

15     delay_scrambler = delay_scrambler | (e_n<<7); //atualizacao do novo valor de
16     e_n no vector
17     ...
18 } b_i_TX++;
19 ...

```

A imagem seguinte representa a visualização do sinal de saída do *scrambler* como também do sinal **b_n**.

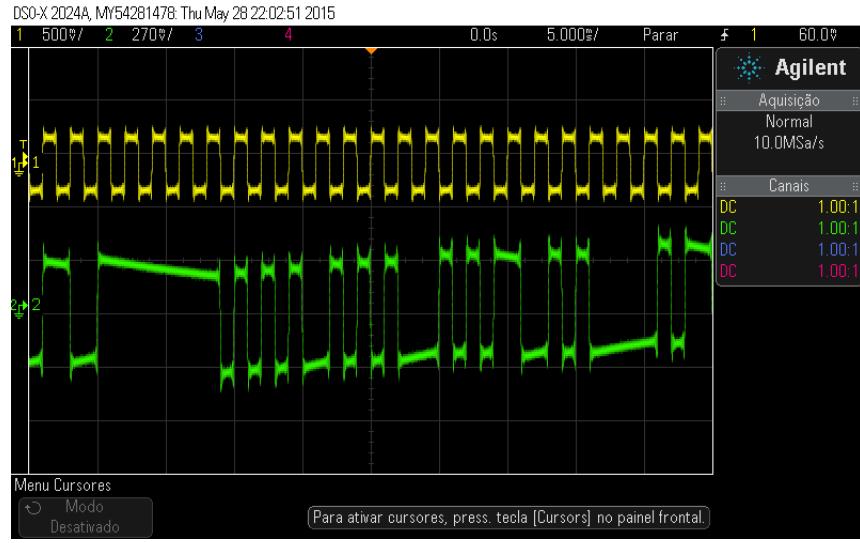


Figura 20: Sinal **b_n** (a amarelo) e sinal **e_n** (a verde).

Como se pode observar, a implementação do *scrambler* foi bem sucedida devido à aleatoriedade da sequência de *bits* obtida, como se pode verificar no sinal a verde.

5 Projecto #3 - Receptor BPSK

A implementação do *descrambler* tem a função de obter a sequência de *bits* correcta para o PC. Assim sendo, o circuito do *descrambler* tem que conseguir obter toda a sequência de *bits* que entra no *scrambler*. Anteriormente foi referido que o *scrambler* segue a seguinte expressão polinomial:

$$P(x) = 1 + x^{-6} + x^{-7}. \quad (5.1)$$

Partindo desta equação consegue obter-se a expressão de recuperação do sinal de entrada do *scrambler*:

$$b'_n = e_n \oplus e_{n-6} \oplus e_{n-7}. \quad (5.2)$$

O sinal b'_n representa a sequência de *bits* recuperada do sinal de saída do *scrambler*, e_n . O esquema seguinte mostra a implementação da equação anterior com maior detalhe.

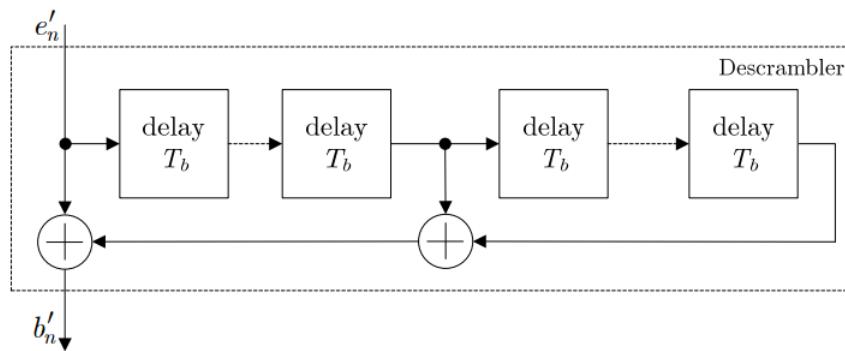


Figura 21: Esquema do cálculo da implementação do *descrambler*.

Analisando o esquema e seguido a lógica de implementação do *scrambler*, começou-se também por definir uma variável de 8 *bits* com o intuito de guardar o sinal de saído do *scrambler* até um máximo de 7 períodos de relógio, `delay_descrambler`. A actualização da variável `delay_descrambler` é feita de forma semelhante à actualização da variável `delay_scrambler`, a diferença ocorre em usar o sinal de entrada a ser guardado em vez do sinal de saída. Isto pode ser verificado no código seguinte, onde o sinal de entrada é a variável `e_n` e o sinal de saída a variável `b_nlinha`.

```

1 short    b_i_RX = 1;
2 short    e_n = 0, xor67 = 0;
3 short    b_nlinha = 1;
4 unsigned char delay_descrambler = 0;
5
6 ...
7 if(b_i_RX > 15){
8     b_i_RX = 0;
9     /* descrambler */
10    xor67 = (delay_descrambler&1)^((delay_descrambler&2)>>1);
11    b_nlinha = e_n^xor67;
12    delay_descrambler = (delay_descrambler>>1)&0x7f;

```

```

13     delay_descrambler = delay_descrambler | (e_n<<7);
14 }
15 b_i_RX++;
16 ...

```

É de salientar que entre o *scrambler* e o *descrambler* existe um *delay* de N bits sendo a saída do *descrambler* não igual à entrada do *scrambler* mas sim igual à entrada deslocada de N.

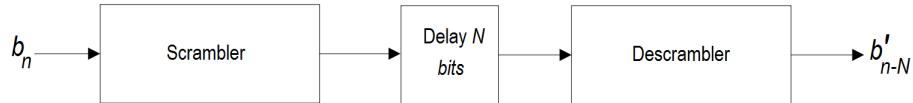


Figura 22: Esquema que representa *delay* existente entre o *scrambler* e o *descrambler*.

Analizando o esquema anterior e as equações do (4.7) e (5.2), consegue-se demonstrar que b'_n é igual a b_{n-N} seguindo os passos:

$$e_n = b_n \oplus e_{n-6} \oplus e_{n-7} \rightarrow \text{delay de } N \rightarrow e_{n-N} = b_{n-N} \oplus e_{n-6-N} \oplus e_{n-7-N} \quad (5.3)$$

$$b'_n = e_{n-N} \oplus e_{n-6-N} \oplus e_{n-7-N}; \quad (5.4)$$

$$\Leftrightarrow b'_n = b_{n-N} \oplus e_{n-6-N} \oplus e_{n-7-N} \oplus e_{n-6-N} \oplus e_{n-7-N}; \quad (5.5)$$

$$\Leftrightarrow b'_n = b_{n-N}; \quad (5.6)$$

De seguida é apresentado o sinal de saída do *descrambler*, b'_n e o sinal de entrada do *scrambler*, b_n . Pode-se verificar que os sinais são iguais pois a implementação da conexão dos dois circuitos foi realizada com um *delay* de N igual 0.

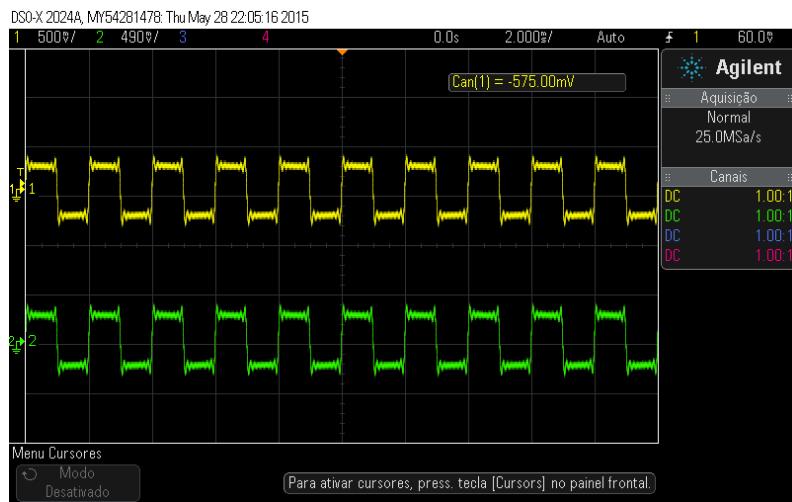


Figura 23: Sinal de entrada do *scrambler* (a amarelo) e sinal de saída do *descrambler* (a verde).

Com o *descrambler* implementado procede-se para a fase seguinte no desenvolvimento do *modem* BPSK - implementar o *Costas Loop*.

Inicialmente, pretende-se implementar um NCO (*numerically controlled oscillator*) controlado pelo sinal de erro, $\varepsilon(t)$, com a seguinte característica:

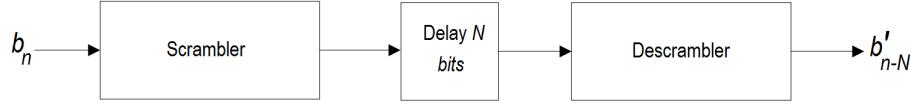


Figura 24: Esquema que representa *delay* existente entre o *scrambler* e o *descrambler*.

$$f_{osc} = f_{ol} + K_0 \varepsilon(n), \quad (5.7)$$

sendo que f_{ol} trata-se da frequência livre do oscilador, que deve ser definida como sendo igual à frequência da portadora do sinal BPSK, ou seja, $f_{ol} = f_0$. O sinal $\varepsilon(n)$ trata-se da saída do filtro de *loop*, `y_loopfilter`. Assim, considerando o gerador de uma onda sinusoidal implementado na primeira parte do projecto, que usa uma LUT de 32 valores de meio período da função seno, é possível implementar este oscilador de forma idêntica, tendo em atenção o valor de f_{ol} e de K_0 , que tal como no oscilador implementado anteriormente, tem o valor de $1/4$, no código esta representado por dois *shifts* para a direita.

```

1 short sine[32] =
2   {0,3212,6393,9512,12540,15447,18205,20788,23170,25330,27246,28899,30274,31357,
3   32138,32610,32767,32610,32138,31357,30274,28899,27246,25330,23170,20788,18205,
4   15447,12540,9512,6393,3212};
5 short delta_erro = 0, y_loop_filter = 0, status_erro = 0, i_erro = 0, y_sin = 0;
6 ...
7   delta_erro = 16384 + (y_loopfilter >> 2);
8   status_erro = status_erro + delta_erro ;
9   i_erro = status_erro >> 10;
10  i_erro = i_erro & 31;
11
12  y_sin = sine[i_erro]; //seno
13
14  if(status_erro < 0){
15    y_sin = -y_sin;
16  }
17 ...

```

É ainda pretendido que seja gerada uma outra saída em quadratura, ou seja, um coseno. Para a sua implementação baseou-se na seguinte figura:

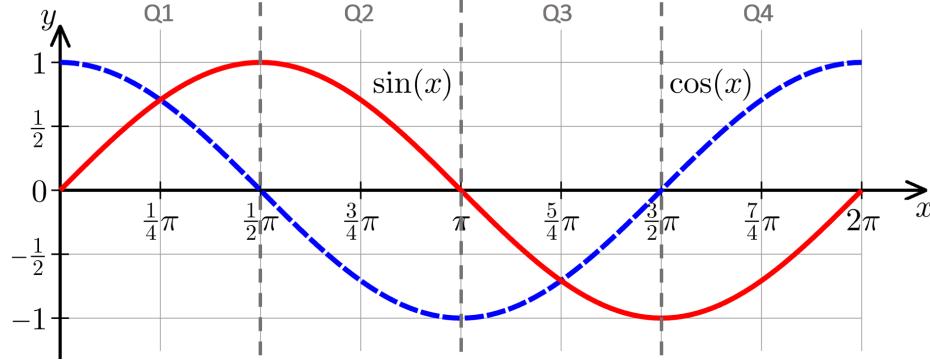


Figura 25: Sobreposição de um seno e de um cosseno.

Sabendo que a LUT tem os valores de meio período da função seno, é possível gerar um cosseno seguindo o esquema anterior. O quadrante 1, Q1, representa um quarto de um período partindo da amplitude máxima até zero, sendo este quadrante facilmente implementado indexando a LUT com um *offset* de 16. Este método também é utilizado para o quadrante 3, Q3, em que se parte da uma amplitude mínima até zero. A máscara de 31 é utilizada para garantir que não haja acessos à memória indevidos.

```
1 short  sine[32] =
2 {0,3212,6393,9512,12540,15447,18205,20788,23170,25330,27246,28899,30274,31357,
3 32138,32610,32767,32610,32138,31357,30274,28899,27246,25330,23170,20788,18205,
4 15447,12540,9512,6393,3212};
5 short delta_errno = 0, y_loop_filter = 0, status_errno = 0, i_errno = 0, y_sin = 0,
6 y_coseno = 0;
7 ...
8 y_sin = sine[i_errno]; //seno
9 y_cos = sine[(i_errno+16)&31]; //coseno
10
11 if(status_errno < 0){
12     y_sin = -y_sin;
13     y_cos = -y_cos; /*Quadrante Q1 e Q3*/
14 }
```

Para implementação dos quadrantes 2 e 4, Q2 e Q4, foi necessário acrescentar uma verificação referente à variável de indexação da LUT. Analisando a figura, verifica-se que os quadrantes pretendidos são referentes à metade superior da LUT, ou seja, quando `i_erro` for maior que 15, sendo necessário inverter o sinal dos valores da LUT. Como referido no código seguinte:

```
1  
2 short sine[32] =  
3 {0,3212,6393,9512,12540,15447,18205,20788,23170,25330,27246,28899,30274,31357,  
4 32138,32610,32767,32610,32138,31357,30274,28899,27246,25330,23170,20788,18205,  
5 15447,12540,9512,6393,3212};
```

```

3   short delta_erro = 0, y_loop_filter = 0, status_erro = 0, i_erro = 0, y_cos = 0,
4     y_sin = 0;
5
6   ...
7
8   y_cos = sine[(i_erro+16)&31]; //coseno
9   y_sin = sine[i_erro]; //seno
10
11
12   if(i_erro > 15){ /*Q2 e Q4*/
13     y_cos = -y_cos;
14   }
15
16   if(status_erro < 0){
17     y_sin = -y_sin;
18     y_cos = -y_cos;
19   }
20
21   ...

```

Na Figura 26, encontra-se representado o sinal de saída do oscilador, que é mostrado aqui apenas para demonstração do correcto funcionamento do oscilador, já que a implementação do filtro de onde sai o sinal de erro apenas será explicada posteriormente.

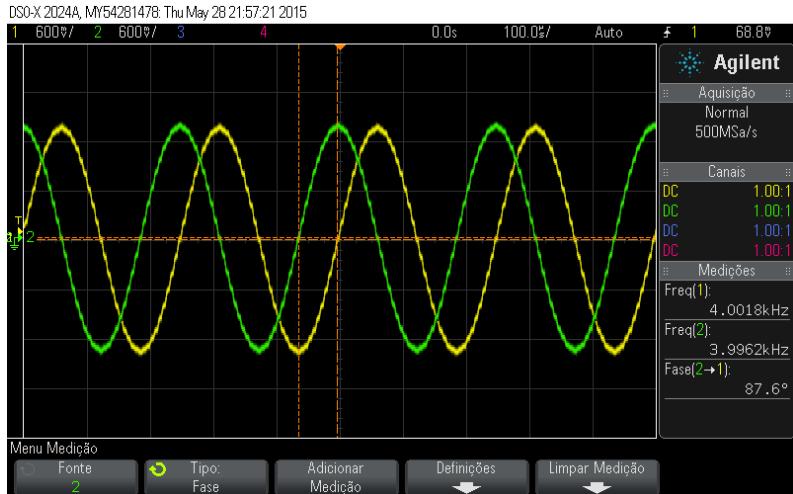


Figura 26: Representação do coseno (a verde) e do seno (a amarelo).

Como se pode ver, a diferença de fase entre as duas formas de onda é de 87.6° , o que comprova que foi possível gerar os dois sinais em quadratura.

O *modem* BPSK apresentado na Figura 2 é constituído por vários módulos, sendo que três desses módulos tratam-se de filtros passa-baixo. Esses filtros apresentam um pólo e um zero, satisfazendo a seguinte equação às diferenças:

$$y(n) = \alpha y(n-1) + \gamma \frac{1-\alpha}{1-\beta} [x(n) - \beta x(n-1)], \quad (5.8)$$

que apresentam um diagrama de fluxo de sinal como o apresentado na Figura 27.

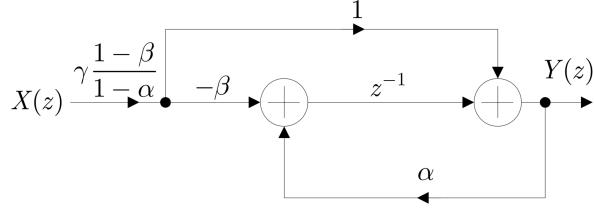


Figura 27: Diagrama de fluxo de sinal de filtro passa-baixo.

Aplicando a transformada Z, é possível determinar a função de transferência do filtro, obtendo-se:

$$H(z) = \frac{\gamma \frac{1-\alpha}{1-\beta}(1-\beta z^{-1})}{1-\alpha z^{-1}}. \quad (5.9)$$

Neste caso, considera-se $\beta = 0$, resultando uma expressão mais simples para a função de transferência:

$$H(z) = \frac{\gamma(1-\alpha)}{1-\alpha z^{-1}}. \quad (5.10)$$

Após a determinação da função de transferência geral, para o caso em que $\beta = 0$, é necessário calcular as constantes α e γ para os dois tipos de filtros presentes no *Costas Loop*, que apesar de serem ambos filtros passa-baixo, apresentam frequências de corte diferentes, sendo que o ganho DC é igual nos dois casos. As especificações dos dois tipos de filtros, *data filter* e *loop filter*, sendo que existem dois filtros do primeiro tipo e apenas um do segundo, são apresentadas na Tabela 3.

Tabela 3: Especificações dos filtros passa-baixo.

	Frequência de corte (f_c)	Ganho DC
<i>Data Filter</i>	1 kHz	1
<i>Loop Filter</i>	10 Hz	1

Sabendo que o ganho DC é igual em todos os filtros, é possível determinar o valor de γ dos mesmos, tendo em conta que em DC se tem $\omega = 0$, portanto:

$$z = e^{j\omega T_s} = 1. \quad (5.11)$$

De onde vem:

$$H(z=1) = \frac{\gamma(1-\alpha)}{1-\alpha} \longrightarrow \gamma = 1. \quad (5.12)$$

Sabe-se, à *priori*, analisando a função de transferência $H(z)$, já com $\gamma = 1$, que o valor de α tem que estar compreendido entre -1 e 1 para o filtro ser estável, sendo que esse valor, para o filtro ser passa-baixo, tem que estar definido entre 0 e 1. Mais ainda, esse valor vai ser diferente para os dois tipos de filtros, visto que o mesmo é determinado em função da frequência de corte, f_c . Recorrendo à expressão do ganho do filtro vem:

$$|H(j\omega)| = \frac{1-\alpha}{|1-\alpha e^{j\omega T_s}|}, \quad (5.13)$$

em que é possível particulizar o valor do ganho do filtro à frequência de corte:

$$|H(j\omega_c)| = \frac{1 - \alpha}{|1 - \alpha e^{j\omega_c T_s}|} = \frac{1 - \alpha}{\sqrt{1 - \alpha^2 \cos^2(\omega_c T_s) + \alpha^2 \sin^2(\omega_c T_s)}}, \quad (5.14)$$

de onde se poderia retirar o valor de α , o que envolveria alguma manipulação algébrica. No entanto, sabendo que a frequência de corte dos filtros é muito inferior ao valor da frequência de amostragem dividido por 2, $f_s/2$, vem que:

$$\omega_c T_s = 2\pi \frac{f_c}{f_s} \ll 1, \quad (5.15)$$

podendo considerar-se a seguinte aproximação:

$$e^{-j\omega T_s} \approx 1 - j\omega_c T_s, \quad (5.16)$$

resultando então uma expressão para o ganho do filtro, à frequência de corte, bastante mais simples:

$$|H(j\omega_c)| = \frac{1 - \alpha}{|1 - \alpha(1 - j\omega_c T_s)|}, \quad (5.17)$$

resultando, finalmente:

$$\omega_c = \frac{1 - \alpha}{\alpha T_s} = \frac{1 - \alpha}{\alpha} f_s \quad (5.18)$$

Assim, tendo em conta que $f_s = 16$ kHz, pode-se calcular o valor de α para os filtros de dados e filtro de *loop*. Sabendo que o valor vai estar contido no intervalo $]0, 1[$, pode-se utilizar a representação numérica mais precisa Q_{15} para representar o valor de α , de forma a efectuar a implementação dos filtros. Na tabela seguinte, são apresentados os valores de α e também a sua conversão em Q_{15} .

Tabela 4: Valores de α dos filtros passa-baixo.

	α	α_{Q15}
<i>Data Filter</i>	0.718030	23528
<i>Loop Filter</i>	0.996088	32640

Assim, conhecendo-se todos os parâmetros das funções de transferência dos filtros, é possível implementar os filtros, segundo a equação às diferenças:

$$y(n) = \alpha y(n-1) + (1 - \alpha)x(n), \quad (5.19)$$

sendo que um dos filtros de dados tem como entrada o sinal resultante da multiplicação entre o sinal BPSK e o sinal $\cos(2\pi f_0 + \phi_0)$ e o outro tem como entrada o sinal resultante da multiplicação entre o sinal BPSK e o sinal $\sin(2\pi f_0 + \phi_0)$. De seguida, é apresentado o excerto de código que permite implementar os filtros.

```

1 short x_cos = 0, x_sine = 0, y_cos_datafilter = 0, y_sin_datafilter = 0;
2 short in_loopfilter = 0, y_loop_filter = 0, alpha_1k = 23528;
3 short alpha_10 = 32640, um_menos_alpha_1k = 9240, um_menos_alpha_10 = 127;
4
5 ...
6 //filtro de dados

```

```

7   x_cos = ((xt_bpsk*y_cos)<<1>>16;
8   x_sine = ((xt_bpsk*y_sin)<<1>>16;
9   y_cos_datafilter = (((alpha_1k*y_cos_datafilter)<<1)
10      + ((x_cos*um_menos_alpha_1k)<<1))>>16;
11   y_sin_datafilter = (((alpha_1k*y_sin_datafilter)<<1)
12      + ((x_sine*um_menos_alpha_1k)<<1))>>16;
13
14 //filtro de loop
15 in_loopfilter = ((y_cos_datafilter*y_sin_datafilter)<<1>>16;
16 y_loopfilter = (((alpha_10*y_loopfilter)<<1)
17      + ((in_loopfilter*um_menos_alpha_10)<<1))>>16;
18 ...

```

No entanto, antes de ligar todo o sistema, é importante testar o filtro que tem frequência de corte de $f_c = 1kHz$. Para efectuar este teste põe-se à entrada do filtro um sinal proveniente do gerador de sinais, de forma a comprovar o correcto funcionamento deste. É de referir que não é possível testar o filtro com frequência de corte de 10 Hz, já que à entrada da placa se encontra um filtro com frequência de corte 20 Hz, o que torna impossível verificar que a implementação do filtro de *loop* está correcta. Assim, para o filtro de dados, visualizou-se a saída do mesmo, apresentada na Figura X, com um sinal de entrada sinusoidal de 250 Hz, de forma a verificar o ganho em baixas frequências.

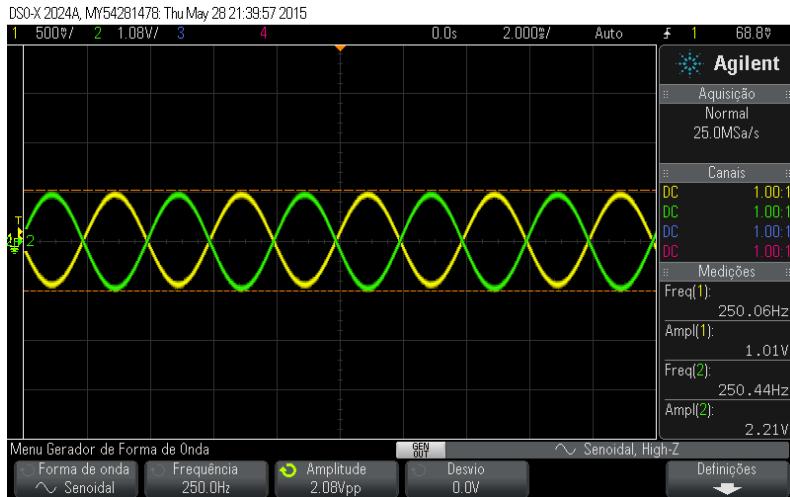


Figura 28: Verificação do ganho em baixas frequências - sinal de entrada (a verde) e sinal de saída (a amarelo) do filtro de dados.

Analizando os valores de amplitude dos sinais, pode verificar-se que o ganho é de, aproximadamente, 1/2, não correspondendo ao ganho unitário pretendido, o que se deve ao facto da placa introduzir um factor de ganho que faz com que o sinal de saída do filtro tenha um ganho inferior a 1.

Testou-se ainda a frequência de corte, sendo que, para isso, aumentou-se a frequência do sinal de entrada até o sinal de saída do filtro apresentar uma queda de 3 dB, ou seja, tendo um ganho em baixa frequência de 1.01 V.

Pretende-se registar a frequência a que o filtro apresenta no sinal de saída um valor de, aproximadamente, 0.7 V.

$$|H(j\omega_{-3 \text{ dB}})| = \frac{|H(j\omega \approx 0)|}{\sqrt{2}} \approx \frac{1}{\sqrt{2}} = 0.7V \quad (5.20)$$

Verificou-se, como se pode observar na Figura X, que a frequência de corte apresenta o valor de 963 Hz, o que se trata de um valor próximo de 1 kHz. A pequena diferença em relação ao valor pretendido pode-se dever a se ter registado uma frequência onde ainda não tinham caído exactamente os 3 dB.

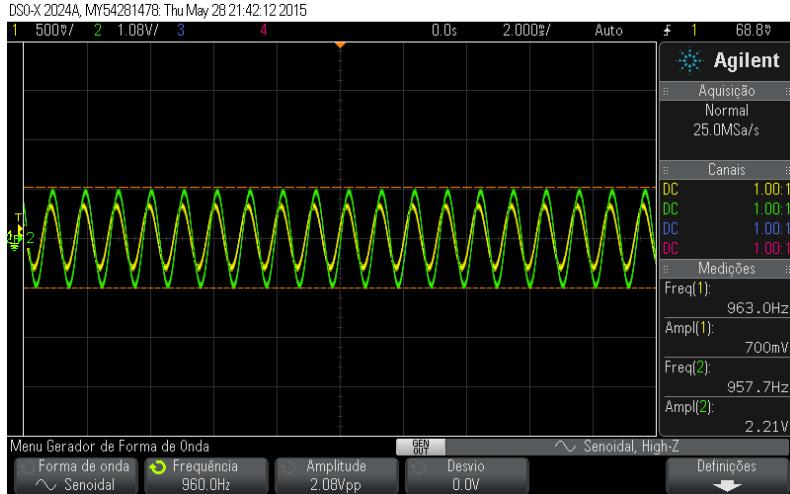
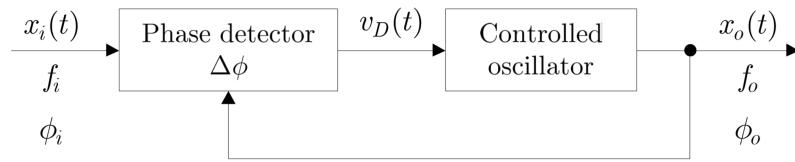


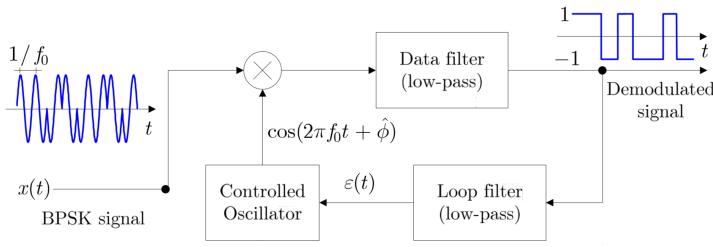
Figura 29: Verificação da frequência de corte - sinal de entrada (a verde) e sinal de saída (a amarelo) do filtro de dados.

Com o *Costas Loop* implementado pretende-se testar o seu funcionamento para um sinal de entrada sinusoidal. Numa primeira fase de testes apenas se considera o *upper arm*, de maneira a que o loop se comporte como um clássico DPLL (*digital phase locked loop*), tal como se pode observar na figura abaixo. Um DPLL é um tipo de PLL que é realizado com recurso a circuitos lógicos e/ou microprocessadores, tendo a capacidade de processar sinais discretos tanto em tempo como em amplitude.

4 - não nos
lembra do
que era isto



(a)



(b)

Figura 30: Diagrama de blocos de um PLL básico (a) e do *upper arm* do *Costas Loop* projectado (b).

As duas fases ϕ_i e ϕ_o são comparadas no detector de fase, o que gera o sinal $v_D(t)$ que é uma função da diferença $\phi_i - \phi_o$ (sinal de erro). Este sinal controla a frequência (e, consequentemente, a fase) do oscilador de maneira a que ϕ_o “segue” a variação em ϕ_i .

Posto isto, o PLL pode estar num de dois estados - *locked* (sincronizado) ou *unlocked* (dessincronizado). Quando no primeiro estado, o oscilador gera um sinal com uma frequência que é, em média, igual à frequência do sinal de entrada $f_o = f_i$ e o *loop* ajusta o oscilador de fase de modo a que este “siga” ϕ_i . Quando o PLL se encontra no segundo estado, a frequência f_o não está relacionada com f_i e se houver variações em ϕ_i estas não serão necessariamente “seguidas” por ϕ_o .

Assim, para testar o correcto funcionamento do PLL projectado é necessário verificar o comportamento do sinal à saída do oscilador controlado e à entrada do detector de fase. No projecto em causa o detector de fase é um multiplicador, que é o adequado para processar sinais do tipo sinusoidal.

À entrada colocou-se um sinal sinusoidal com amplitude de 1 V_{PP}, fazendo-se variar a sua frequência, partindo de um valor inicial de 4 kHz, que é a frequência de funcionamento da malha dessincronizada. Nas figuras seguintes apresenta-se o comportamento do PLL para duas situações distintas, estando representado a amarelo o sinal de entrada e a verde o sinal à saída do oscilador controlado.

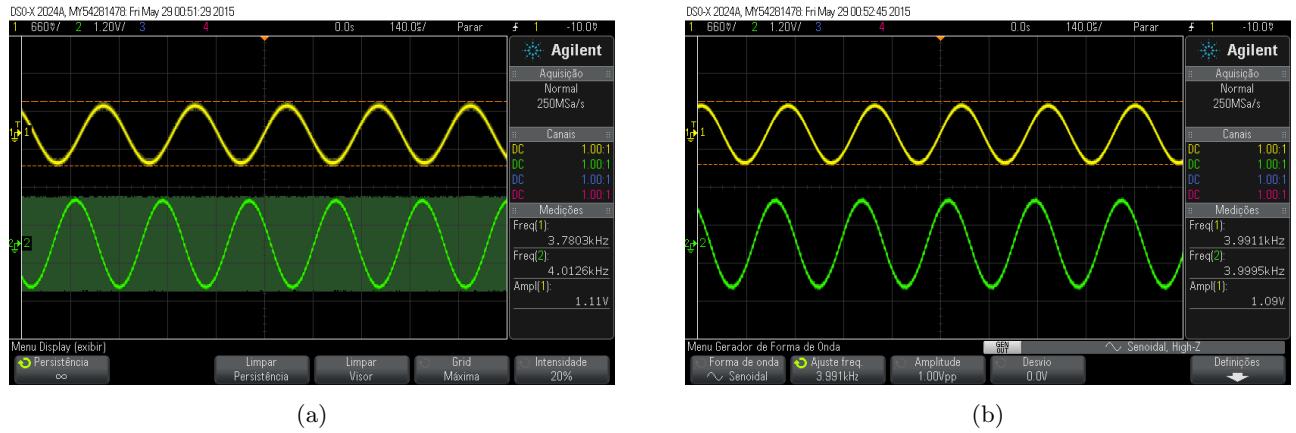


Figura 31: Situação em que o PLL está dessincronizado (a) e sincronizado (b).

Para ambas as situações recorreu-se ao modo persistência do osciloscópio, com o objectivo de ajudar a reconhecer a situação de dessincronismo do PLL. De facto, na Figura 31(a) verifica-se que o sinal à saída do oscilador apresenta bastante dispersão em torno dos valores pretendidos para a forma de onda, tal como esperado. Quando se ajustou a frequência do sinal de entrada e o PLL entrou num estado de sincronismo obtém-se os resultados da Figura 31(b), em que o sinal representado a verde não apresenta dispersão em torno dos valores pretendidos, tal como esperado.

Uma vez verificado o correcto funcionamento do *upper arm* do *Costas Loop*, conectou-se o *lower arm* e testou-se todo o *loop* utilizando também um sinal sinusoidal à entrada. Verificou-se que ambos os *arms* apresentam o comportamento esperado. Assim, procede-se à medição da banda de captura e da banda de manutenção do *loop*.

O sinal de entrada deve estar compreendido na banda de captura (Δw_C) do *loop*. Uma vez que se adquira o sincronismo, o *loop* manter-se-á nesse estado, dentro de certos limites, que definem a banda

de manutenção ($\Delta\omega_L$), sendo que esta banda pode ser diferente da banda de captura.

Para se calcular experimentalmente o valor das bandas, é necessário observar o valor médio do sinal de erro, enquanto se faz variar lentamente a frequência do sinal de entrada crescentemente, partindo de um estado inicial de *loop* dessincronizado, até que ocorre uma captura em w_C^- . De seguida, aumenta-se a frequência do sinal de entrada até que se perde o sincronismo em w_L^+ . O passo seguinte é efectuar um varrimento no sentido contrário, adquirindo sincronismo em w_C^+ , que depois é perdido em w_L^- .

Relativamente ao sinal de erro que se observa nesta medição, considere-se o caso em que o PLL está fora de sincronismo e o oscilador gera uma frequência f_o . Quando o *loop* se fecha (ou quando o sinal de entrada tem uma frequência f_i), se a diferença entre as duas frequências for pequena o suficiente, então o filtro de *loop* irá desenvolver um sinal de erro que leva a frequência do oscilador até f_i . Eventualmente, o *loop* vai adquirir sincronismo e neste estado a frequência média gerada pelo oscilador é igual à frequência média do sinal de entrada.

Na figura seguinte encontra-se uma representação gráfica do valor médio do sinal de erro em função da frequência do sinal de entrada, identificando a banda de captura e de manutenção.

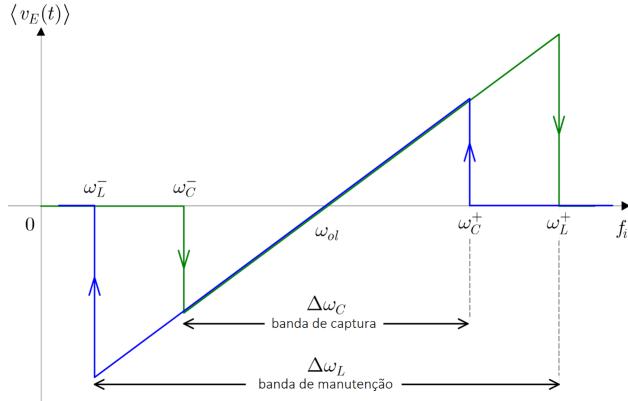


Figura 32: Representação das bandas que caracterizam o PLL.

Da figura anterior verifica-se que, regra geral, $\Delta\omega_L \geq \Delta\omega_C$.

O cálculo das bandas foi feito para situações diferentes da tensão de entrada, $V_{in} = 1 V_{PP}$ e $V_{in} = 3 V_{PP}$, e também para situações diferentes da frequência de corte do filtro de *loop*, $f_c = 10$ Hz e $f_c = 100$ Hz.

Como o filtro de *loop* foi projectado para um frequência de corte de 10 Hz, é necessário calcular o valor de α para 100 Hz. Recorrendo à equação (5.18) tem-se $\alpha = 0.9622$, cuja representação em Q_{15} é 31529.

Tabela 5: Cálculo das bandas de captura e de manutenção.

frequência de corte do filtro	amplitude do sinal de entrada					
	1 V _{PP}			3 V _{PP}		
	ω_C^-	ω_C^+	$\Delta\omega_C$	ω_C^-	ω_C^+	$\Delta\omega_C$
10 Hz	3.946 kHz	4.039 kHz	93 Hz	3.918 kHz	4.067 kHz	149 Hz
	4.159 kHz	3.825 kHz	334 Hz	4.505 kHz	3.488 kHz	1017 Hz
100 Hz	3.869 kHz	4.129 kHz	260 Hz	3.789 kHz	4.218 kHz	437 Hz
	4.167 kHz	3.830 kHz	337 Hz	4.508 kHz	3.499 kHz	1009 Hz

Analizando a tabela anterior, é possível concluir que, para a mesma frequência de corte, o aumento da amplitude do sinal de entrada causa o aumento das bandas de captura e manutenção. Em relação à comparação entre os valores obtidos para as bandas para diferentes frequências, percebe-se que um aumento da frequência de corte, que corresponde a um aumento do ruído, faz com que as bandas de captura e aquisição tenham valores superiores, portanto pode concluir-se que a malha sincroniza para um intervalo maior de frequências com o aumento do ruído. De facto, para que a malha possa sincronizar é necessário que haja ruído.

O passo seguinte corresponde à conexão entre o modulador do transmissor e o desmodulador do receptor, ou seja, o sinal de entrada do *Costas Loop* é o sinal modelado do tipo BPSK. Este sinal corresponde à multiplicação do sinal d_n com a portadora, como se viu já anteriormente.

Para se verificar o correcto funcionamento do *loop* observou-se o sinal à saída do filtro de dados, esperando obter um sinal desmodulado compreendido entre 1 e -1, com valor médio 0.

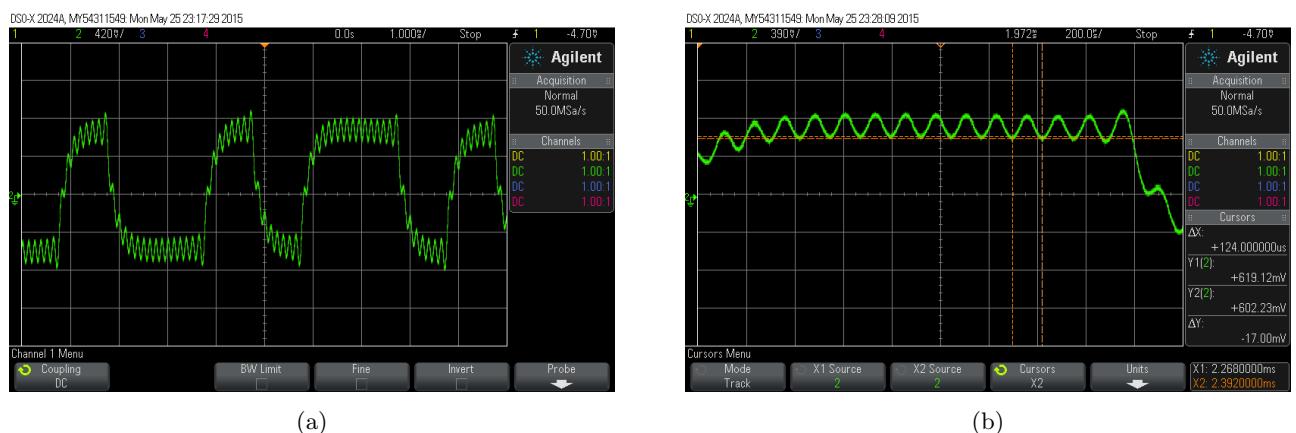


Figura 33: Sinal à saída do filtro da dados (a) e pormenor feito sobre a zona de amplitude máxima (b).

explicar estas imagens

Com a conexão estabelecida entre o modulador e transmissor testou-se novamente o valor das bandas de captura e de manutenção, como se pode ver na tabela seguinte.

Tabela 6: Cálculo das bandas de captura e de manutenção para o *Costas Loop* completo.

frequência de corte do filtro	10 Hz	amplitude do sinal de entrada		
		3 V _{PP}		
		$\omega_C^- = 3.971 \text{ kHz}$	$\omega_C^+ = 4.014 \text{ kHz}$	$\Delta\omega_C = 43 \text{ Hz}$
		$\omega_L^+ = 4.055 \text{ kHz}$	$\omega_L^- = 3.929 \text{ kHz}$	$\Delta\omega_L = 126 \text{ Hz}$

Realizando o teste do filtro de dados com uma entrada sinusoidal de 8 kHz, é possível, observando a Figura X, perceber que o filtro não anula completamente o sinal de entrada, o que causa o problema descrito anteriormente, aquando da Figura 33.

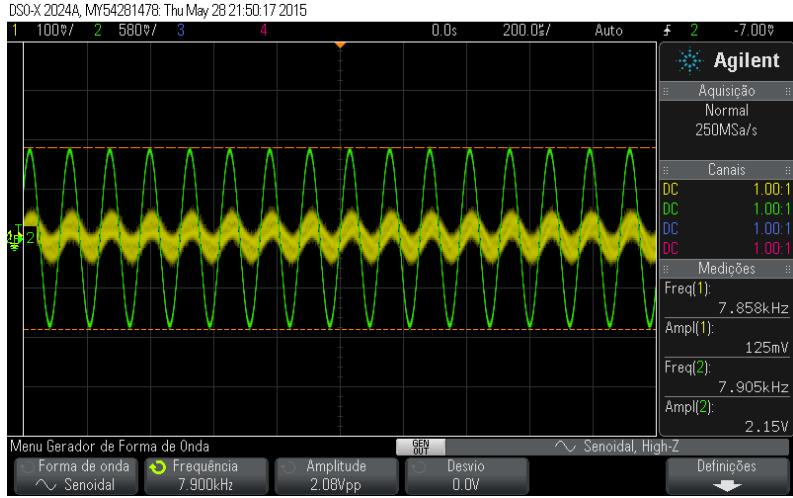


Figura 34: Sinal de entrada do filtro (a verde) e sinal de saída do filtro de dados (a amarelo) para o caso do filtro sem zero.

De forma a solucionar este problema, implementou-se um filtro idêntico ao já implementado, adicionando um zero na frequência de 8 kHz, o que permite anular essa componente no sinal de saída. Na figura seguinte apresenta-se o ganho do filtro para as duas situações de presença e ausência de zero.

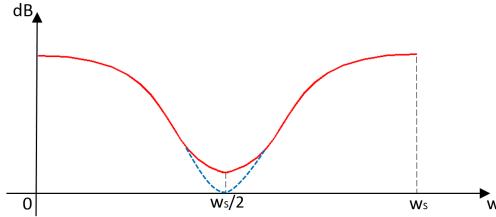


Figura 35: Ganho do filtro para o caso em que não tem um zero (a vermelho) e melhoria na função de transferência aquando da adição de um zero em 8 kHz (a azul).

Assim, de forma a implementar esta solução, é necessário que $\beta = -1$, que vem de:

$$H(z) = \frac{\gamma \frac{1-\alpha}{1-\beta} (1 - \beta z^{-1})}{1 - \alpha z^{-1}} = 0 \Rightarrow \beta z = 1. \quad (5.21)$$

e, para a frequência de 8 kHz, sendo $z = e^{j\omega T_s} = e^{j2\pi \frac{8 \times 10^3}{16 \times 10^3}}$, é possível determinar o valor de β .

Resulta então a função de transferência, mantendo-se os valores de α e γ :

$$H(z) = \frac{\gamma \frac{1-\alpha}{2} (1 + z^{-1})}{1 - \alpha z^{-1}} \quad (5.22)$$

concluindo-se que, em relação à implementação anterior, basta dividir o ganho por 2 e utilizar o valor da amostra anterior, somando-a à actual, respeitando a seguinte equação às diferenças:

$$y(n) = \alpha y(n-1) + (1 - \alpha)[x(n) + x(n-1)] \quad (5.23)$$

cujo excerto de código que permite implementar essa equação é apresentado de seguida.

```

1 short x_cos = 0, x_sine = 0, y_cos_datafilter = 0, y_sin_datafilter = 0;
2 short in_loopfilter = 0, y_loop filter = 0, alpha_1k = 23528;
3 short alpha_10 = 32640, um_menos_alpha_1k = 9240, um_menos_alpha_10 = 127;
4
5 ...
6 //filtro de dados com zero em ws/2
7 y_cos_datafilter = (((alpha_1k*y_cos_datafilter)<<1)
8 + (((x_cos + x_cos_old)*um_menos_alpha_1k>>1)<<1))>>16;
9 y_sin_datafilter = (((alpha_1k*y_sin_datafilter)<<1)
10 + (((x_sine + x_sine_old)*um_menos_alpha_1k)>>1)<<1))>>16;
11
12 x_cos_old = x_cos;
13 x_sine_old = x_sine;
14 ...

```

Na Figura X, pode observar-se a saída do filtro digital implementado com um zero em 8 kHz, para um sinal de entrada sinusoidal com essa frequência.

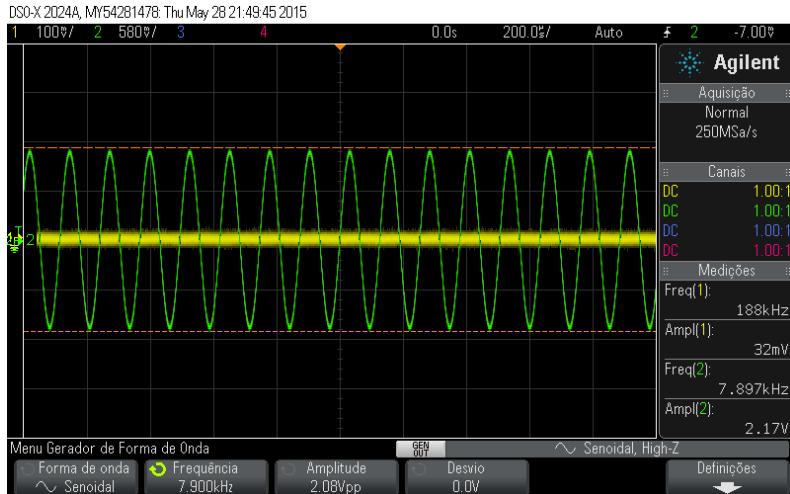


Figura 36: Sinal de entrada do filtro (a verde) e sinal de saída do filtro de dados (a amarelo) para o caso do filtro com zero.

Conclui-se assim que o filtro implementado anula a componente de 8 kHz, o que comprova a implementação de forma correcta do mesmo, permitindo que o sinal de saída do filtro, com o *Costas Loop* completo, apresente de uma forma clara uma diferença entre o nível *high* e *low*, tal como representado na Figura X.

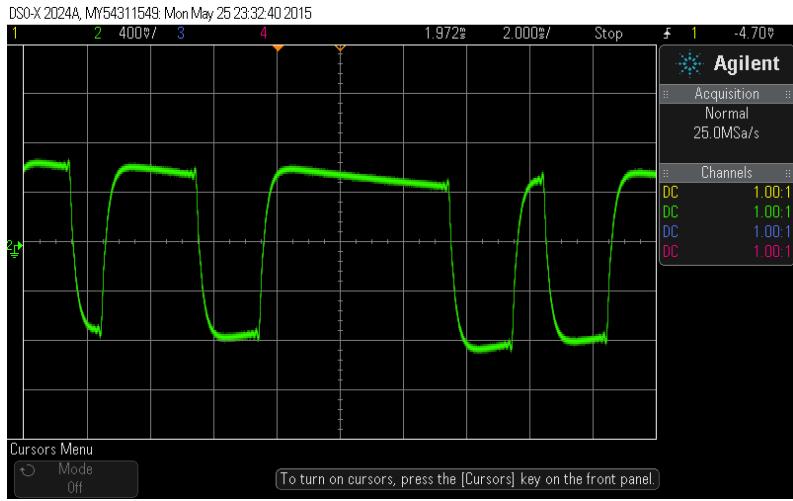


Figura 37: Sinal à saída do filtro da dados com o *Costas Loop* completo.

Depois de testado o *Costas Loop* este vai ser caracterizado através de vários sinais e espectros. De referir que as imagens retiradas daqui para a frente são com o filtro com o zero que elimina interferências em $w_s/2$.

Primeiramente, analisa-se o sinal modulado do tipo BPSK, no domínio do tempo e da frequência.

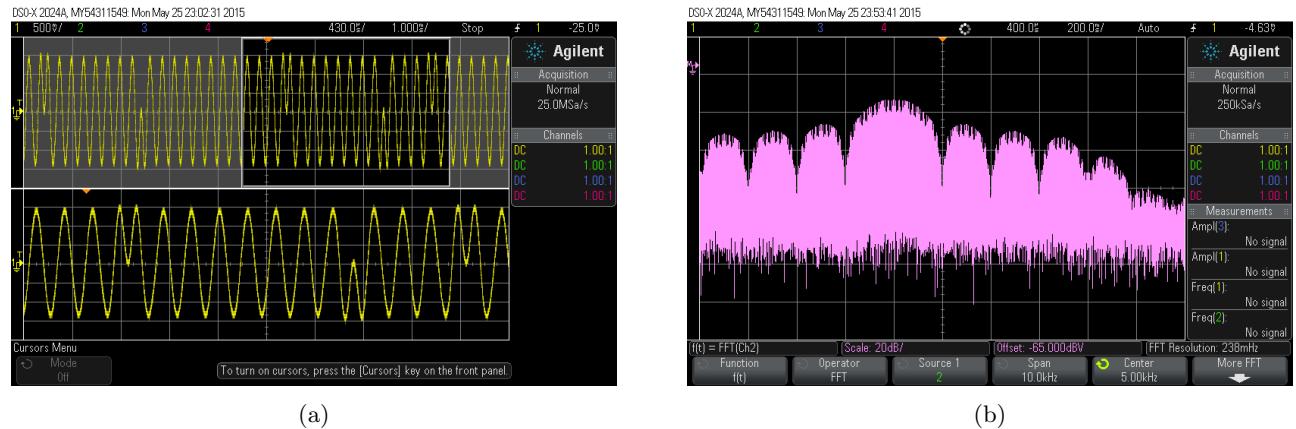


Figura 38: Sinal modulado do tipo BPSK no domínio do tempo (a) e no domínio da frequência (b).

Na Figura 35(a), é possível observar o sinal do tipo BPSK, que apresenta uma forma de onda idêntica à que era suposto obter, tendo em conta que o sinal $x(t)$ é da forma:

$$x(t) = s(t) \cos(2\pi f_0 t + \Delta ft + \phi_0) + noise \quad (5.24)$$

sendo que a componente $\Delta ft + \phi_0$ se trata de perturbações angulares introduzidas pelo canal físico que garante a propagação do sinal. No domínio da frequência, estando o SPAN definido como sendo de 10 kHz, pode visualizar-se na Figura 35(b) a componente da frequência fundamental do sinal BPSK, a 4 kHz, e as suas harmónicas.

De seguida, analisa-se as formas de onda à saída dos filtros do *upper* e *lower arm*.

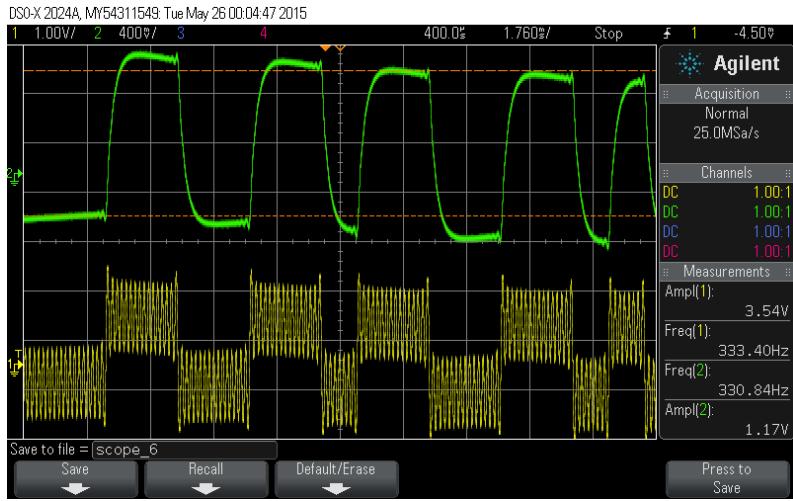


Figura 39: Sinal à entrada do filtro de dados (a amarelo) e à saída do filtro (a verde).

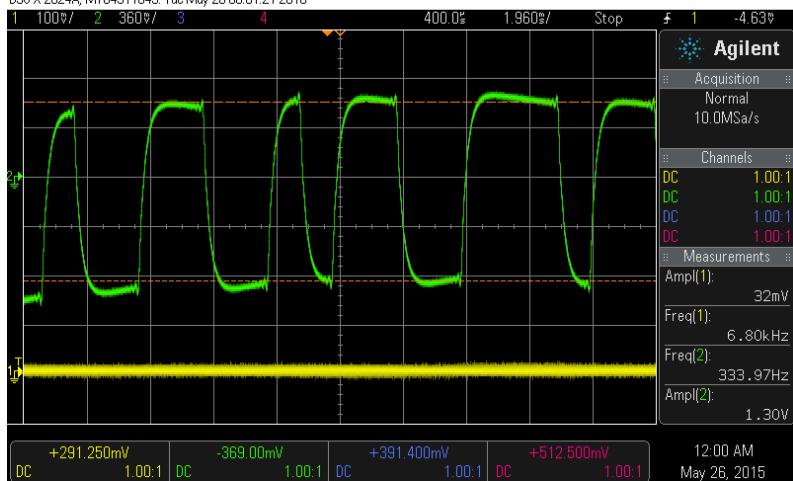


Figura 40: Sinal à saída do filtro de dados do *upper arm* (a verde) e à saída do filtro dados do *lower arm* (a amarelo).

Os sinais representados na figura anterior, que se tratam das saídas dos filtros do *upper* e *lower arm*, estão de acordo com o previsto teoricamente, já que os mesmos são descritos por:

$$E[x_1(t)] \approx \frac{1}{2}s(t) \cos(\Delta\phi) \approx \frac{1}{2}s(t); \quad (5.25)$$

$$E[x_2(t)] \approx \frac{1}{2}s(t) \sin(\Delta\phi) \approx 0. \quad (5.26)$$

correspondendo $E[x_1(t)]$ ao sinal de saída do filtro pertencente ao *upper arm* e $E[x_2(t)]$ à saída do *data filter* do *lower arm*, sendo que o sinal $s(t)$ é o sinal de informação transmitido e $x_1(t)$ e $x_2(t)$ representam os sinais:

$$x_1(t) = x(t) \cos(2\pi f_0 t + \phi_0) = s(t) \cos(2\pi f_0 t + \Delta f t + \phi_i) \cos(2\pi f_0 t + \phi_0); \quad (5.27)$$

$$x_2(t) = x(t) \sin(2\pi f_0 t + \phi_0) = s(t) \cos(2\pi f_0 t + \Delta f t + \phi_i) \sin(2\pi f_0 t + \phi_0). \quad (5.28)$$

Veja-se agora o sinal de saída do filtro de *loop*, ou seja, o sinal de erro que depois controla o oscilador. Foi analisada esta situação para o caso em que o filtro de dados tem o zero na sua função de transferência.

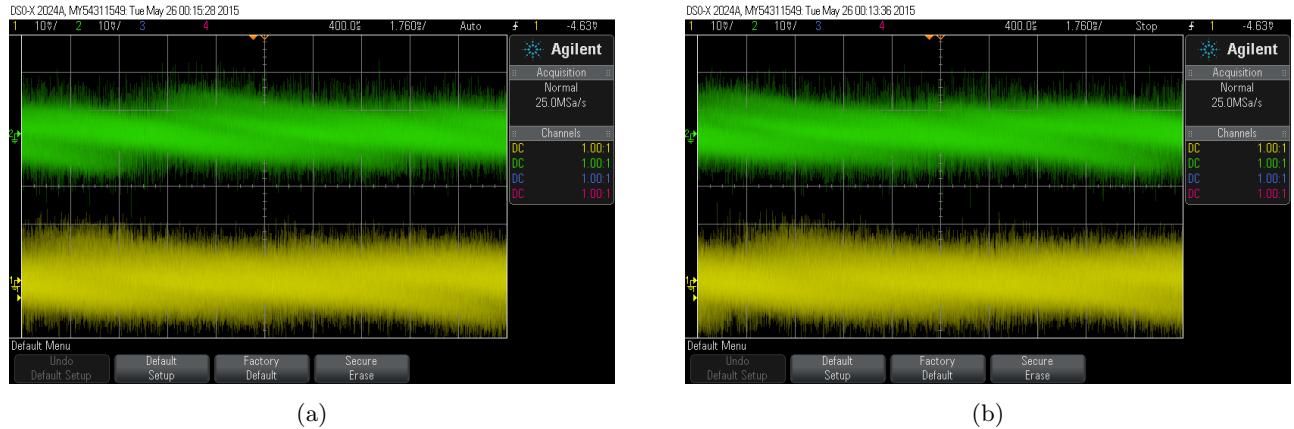


Figura 41: Sinal de erro para o caso do filtro sem zero (a) e com zero (b).

Quando se analisa as figuras anteriores verifica-se que são bastante semelhantes, não havendo diferenças perceptíveis a “olho nu”. De facto, para o caso do filtro que não tem zero, quando se multiplica as duas sinusoides com 8 kHz de frequência que surgem à saída dos filtros de dados (como se pode ver na Figura 33(b)) originam-se duas sinusoides - uma com uma frequência que corresponde à soma das frequências, ou seja, com $2 \times 8 = 16$ kHz e outra com uma frequência que corresponde à subtração das frequências, ou seja, com $8 - 8 = 0$. Considerando que a primeira sinusoide não é observável no osciloscópio porque não passa na placa e que a segunda sinusoide corresponde a uma componente DC, assim se explica que não haja diferenças no sinal de erro entre haver um zero ou não na função de transferência dos filtros de dados.

De seguida, faz-se uma análise do transitório do processo de aquisição do *loop*. Para se poder contar o tempo que demora a aquisição do sistema implementa-se um contador de amostra que, de 1000 em 1000 amostras, introduz no sistema uma situação de não-sincronismo, colocando o oscilador e o sinal de erro que o controla em extremos opostos, forçando o valor do erro ao seu máximo, 32767. O excerto de código seguinte representa este forçar do sinal de erro.

```

1 ...
2     if(trans > 1000){
3         trans = 0;
4         y_loopfilter = 32767; //forçar o valor maximo que o sinal de erro pode tomar
5     }
6     trans++;
7 ...

```

Durante este período de aquisição, o funcionamento do sistema não é linear e, enquanto o sinal de erro não vier a zero, a malha não sincronizará e os *bits* do sinal desmodulado estarão errados.

Calculou-se o tempo de aquisição, com o auxílio dos cursores do osciloscópio, determinando o intervalo de tempo entre o início da situação de não-sincronismo e o instante em que se pode considerar

que o sinal de saída do filtro de dados apresenta os *bits* bem definidos, sendo perceptível a diferença entre 0 e 1, para duas frequências de corte do filtro de *loop*, 10 Hz e 100 Hz, como se pode ver nas figuras seguintes. O sinal a verde representa a saída do filtro de *loop*, o sinal de erro, e o sinal a amarelo representa a saída do filtro de dados do *lower arm* do *Costas Loop*.

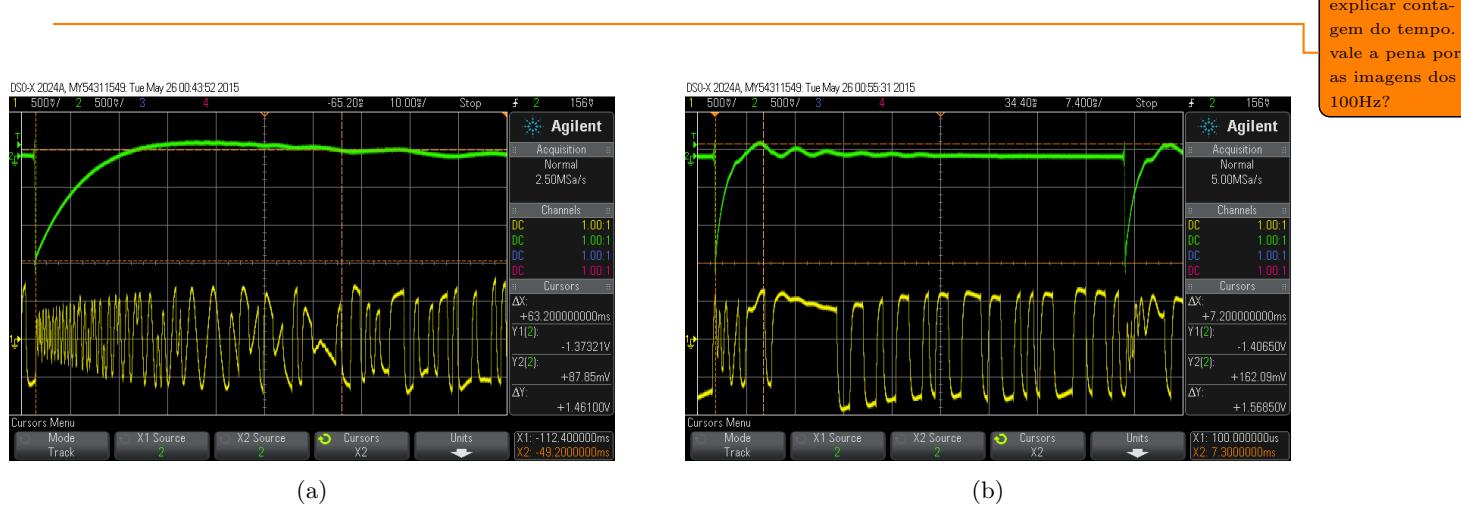


Figura 42: Tempo de aquisição da malha para uma f_c do filtro de *loop* a 10 Hz(a) e a 100 Hz(b).

Como se pode ver nas figuras anteriores, o tempo de transitório para um valor de f_c de 10 Hz é de 63.2 ms e para um valor de f_c de 100 Hz é de 7.2 ms. Assim, verifica-se que, quanto maior a frequência do filtro de *loop*, mais rápida é a malha a sincronizar.

De facto, durante a aquisição quer-se um transitório rápido, para que se atinja rapidamente o sincronismo, mas na manutenção quer-se uma largura de banda mais pequena para que o sistema não tenha tanto ruído. Assim, é frequente utilizar-se duas larguras de banda diferentes - uma para o transitório (aquisição) e uma para a manutenção.

Saber qual o valor deste tempo de transitório é extremamente importante, pois pode ser utilizado para enviar um cabeçalho para o sistema que contém *bits* piloto que foram previamente estudados de acordo com as funcionalidades do sistema em causa. Uma vez que termine o envio do cabeçalho, o sistema encontra-se pronto para adquirir sinais.

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.

Relativamente ao Projecto #3, em que o objectivo era desenvolver um receptor BPSK, o mesmo

foi implementado da forma esperada, tendo sido analisados várias características do receptor. A implementação dos filtros *Data do Costas Loop* com um zero em 8 kHz, o que anula uma componente com essa frequência que estava à entrada dos filtros, garante que à saída do sistema se tenha um sinal em que se podem distinguir claramente os bits '0' e '1', o que é de extrema importância. Em relação ao regime transitório, onde é avaliada a capacidade do sistema adquirir o sincronismo, após o ter perdido, pôde concluir-se que o processo de aquisição do sincronismo apresenta um tempo de transição que varia com o mesmo factor de variação da frequência de corte do *loop filter*.

Vejam o que escrevi e alterem/acrescentem o que acharem necessário

conclusao geral
e conclusao do projecto 3

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
33             /* implementacao do contador sem recurso a instrucao condicional if */
34             //b_i++;
```

```

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

7.3 Anexo III - Código do Projecto #3

```

1 #include "dsk6713_aic23.h"
2 #include "C6713dskinits.h"
3
4 Uint32 fs=DSK6713_AIC23_FREQ_16KHZ;
5
6 char intflag = FALSE;
7 short sine[32] =
8     {0,3212,6393,9512,12540,15447,18205,20788,23170,25330,27246,28899,30274,31357,
9      32138,32610,32767,32610,32138,31357,30274,28899,27246,25330,23170,20788,18205,
10     15447,12540,9512,6393,3212};
11
12 short alpha_10 = 32640, alpha_100 = 31529, alpha_1k = 23528,
13     um_menos_alpha_1k = 9240, um_menos_alpha_10 = 127, um_menos_alpha_100 = 1238;
14 union {Uint32 samples; short channel[2];} AIC_buffer;
15
16 interrupt void c_int11(){
17     output_sample(AIC_buffer.samples);
18     AIC_buffer.samples= input_sample();
19     intflag = TRUE;
20     return;
21 }
22
23 void main(){
24     short d_n = 0, y1 = 0, y2 = 0, y_sin = 0, y_cos = 0, xt_bpsk = 0, y_port = 0;
25     short x_cos = 0, x_sine = 0, y_cos_datafilter = 0, y_sin_datafilter = 0;
26     short x_cos_old = 0, x_sine_old = 0;
27     short in_loopfilter = 0, y_loopfilter;
28     short b_i_TX = 1, b_i_RX = 1, sine_i = 0, inbuf;
```

```

25 short c_n = 0, e_n = 0, xor67 = 0;
26 short i = 0 , delta = 0 ,trans = 0;
27 short status = -32767;
28 short status_erro = -32767, delta_erro = 16384, i_erro = 0;
29 short b_n = 1, shift15_cn = 0, not_shift15 = 0, b_nlinha = 0;
30 unsigned char delay_scrambler = 0, delay_descrambler = 0;
31
32 comm_intr();
33 while(1){
34     if(intflag != FALSE){
35         intflag = FALSE;
36
37         inbuf = AIC_buffer.channel[LEFT];
38
39         /////////////////////////////////
40         ////////////////// TX /////////////////////
41         /////////////////////////////////
42         delta = 16384;
43         status = status + delta;
44         i = status >> 10;
45         i = i & 31;
46
47         y_port = sine[i];
48
49         if(status < 0 ){
50             y_port = -y_port;
51         }
52
53         if(b_i_TX>15){
54             b_i_TX = 0;
55             b_n = (b_n^1);
56             /* scrambler */
57             xor67 = (delay_scrambler&1)^((delay_scrambler&2)>>1);
58             e_n = b_n^xor67;
59             delay_scrambler = (delay_scrambler>>1)&0x7f;
60             delay_scrambler = delay_scrambler | (e_n<<7);
61
62             c_n=c_n^e_n;
63             shift15_cn=c_n<<15;
64             not_shift15=~shift15_cn;
65             d_n=not_shift15>>14;
66         }
67         b_i_TX++;
68
69         xt_bpsk = y_port*d_n;
70
71         /////////////////////////////////

```

```

72 //////////////// RX ///////////////////////
73 /////////////////////////////////
74 delta_erro = 16384 + (y_loopfilter >>2);
75 status_erro = status_erro + delta_erro ;
76 i_erro = status_erro >> 10;
77 i_erro = i_erro & 31;

78

79 y_cos = sine[(i_erro+16)&31]; //coseno
80 y_sin = sine[i_erro]; //seno

81

82 if(i_erro > 15){
83     y_cos = -y_cos;
84 }
85 if(status_erro < 0){
86     y_sin = -y_sin;
87     y_cos = -y_cos;
88 }

89

90 //filtro de dados com fc = 1 kHz
91 x_cos = (xt_bpsk*y_cos)>>15;
92 x_sine= (xt_bpsk*y_sin)>>15;
93 //x_cos = xt_bpsk;
94 // x_sine= xt_bpsk;
95 y_cos_datafilter = (((alpha_1k*y_cos_datafilter)<<1) + ((x_cos*
um_menos_alpha_1k)<<1))>>16; /*Filtro sem zero em ws/2*/
96 y_sin_datafilter = (((alpha_1k*y_sin_datafilter)<<1) + ((x_sine*
um_menos_alpha_1k)<<1))>>16; /*Filtro sem zero em ws/2*/

97

98 // y_cos_datafilter = (((alpha_1k*y_cos_datafilter)<<1) + (((x_cos + x_cos_old)
99 /* *um_menos_alpha_1k)>>1)<<1))>>16; /*Filtro com zero em ws/2*/
100 // y_sin_datafilter = (((alpha_1k*y_sin_datafilter)<<1) + (((x_sine +
101 x_sine_old)*um_menos_alpha_1k)>>1)<<1))>>16; /*Filtro com zero em ws/2*/

102 x_sine_old = x_sine;
103 x_cos_old = x_cos;

104 in_loopfilter = (y_cos_datafilter*y_sin_datafilter)>>15;
105 y_loopfilter = (((alpha_10*y_loopfilter)<<1) + ((in_loopfilter*
106 um_menos_alpha_10)<<1))>>16;
107 //y_loopfilter = (((alpha_100*y_loopfilter)<<1) + ((in_loopfilter*
108 um_menos_alpha_100)<<1))>>16;

109 if(b_i_RX>15){
110     b_i_RX = 0;
111     /* descrambler ^
112     xor67 = (delay_descrambler&1)^((delay_descrambler&2)>>1);
113     b_nlinha = e_n^xor67;

```

```
113     delay_descrambler = (delay_descrambler>>1)&0x7f;
114     delay_descrambler = delay_descrambler | (e_n<<7);
115 }
116 b_i_RX++;
117 /*
118 if(trans>1000){
119     trans=0;
120     y_loopfilter = 32767;
121 }
122 trans++; */
123
124 ///////////////////////////////////////////////////
125 ////////////// OUT //////////////////////////////
126 ///////////////////////////////////////////////////
127 AIC_buffer.channel[LEFT] = y_cos;
128 AIC_buffer.channel[RIGHT] = y_cos_datafilter;
129 }
130 }
131 }
```