

MESTRADO INTEGRADO EM ENGENHARIA ELECTROTÉCNICA E DE
COMPUTADORES

SISTEMAS ELECTRÓNICOS DE PROCESSAMENTO DE SINAL

Desenvolvimento de um Modulador BPSK com uso de *Costas Loop*

Grupo n.^o 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, 1 de Junho de 2015

Conteúdo

1	Introdução	1
2	Projecto	1
2.1	Projectos de Demonstração	1
2.1.1	sine8_buf	1
2.1.2	loop_intr	2
2.2	BPSK Modem	2
2.2.1	P1. Oscilador controlado numericamente	2
2.2.2	P2. Transmissor BPSK	10
2.2.3	Receptor <i>Costas Loop</i>	15
3	Conclusão	27
4	Anexos	27
4.1	Anexo A	27
4.2	Anexo B	30
4.3	Anexo C	31

¹As 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

O presente trabalho trata-se do projeto de laboratório da cadeira: desenvolvimento de um modem de *Binary Phase Shift Keying* (BPSK) usando um *Costas Loop*.

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*, e na implementação de alguns dos blocos de um modem BPSK, nomeadamente o *Scrambler*, *Differential coder and mapper*, *Modulator* e *Demodulator(Costas Loop)*.

No projeto foi utilizada a placa DSK TMS320C6713. Inicialmente, na primeira parte do projeto, 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 projetos: um oscilador sinusoidal controlado numericamente e um transmissor BPSK.

Na segunda parte do projeto foram incluídos os dois blocos mencionados no desenvolvimento dos restantes, dimensionaram-se filtros e testou-se e analisou-se o dispositivo desenvolvido. Os projetos desenvolvidos foram incorporados nos anexos A, B e C podendo analisar os mesmos com maior detalhe.

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 seja o inverso do que deveria ser, ou seja ocorre *wraparound* na variável da sinusóide (como se poderia esperar dado que se programa em C).

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 sinusoide 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 porque o espectro (infinito) deste teria que ser filtrado pelo anti-aliasing filter.

2.2 BPSK Modem

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 Relaxação (Integrador de Rampa).
2. Look-up-table (LUT).
3. Obtenção de um sinal sinusoidal através da LUT.

4. Criação de variáveis para controlo da 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 sinusoidal à entrada.
7. Melhoria da qualidade do oscilador com interpolação linear.
8. Comparação dos espectros com e sem interpolação.

1. Oscilador de Relaxação

Para construir um Oscilador de Relaxação 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. Para que esta tenha precisão máxima e amplitude máxima de 1 V foi usado o formato Q15 para representar o sinal de saída, pois este formato tem amplitude máxima de $2^{15} - 1 = 32767$.

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(Δ)
$2kHz$ (f_{min})	8192
$4kHz$	16384
$6kHz$ (f_{max})	24576

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

$$f_0 = \frac{\Delta}{2A} f_s \Leftrightarrow \Delta = 2A \frac{f_0}{f_s}, A = 32767 \quad (2.1)$$

Esta equação trata-se de determinar quantos incrementos se faz de -32767 a 32767 ($2A$), de modo a que ocorra *wraparound*, sabendo que em cada ciclo da frequência de amostragem $T_s = \frac{1}{f_s}$ se faz um incremento Δ para se ter um período de rampa $T_0 = \frac{1}{f_0}$.

Foi então possível obter as seguintes rampas com diferentes frequências:

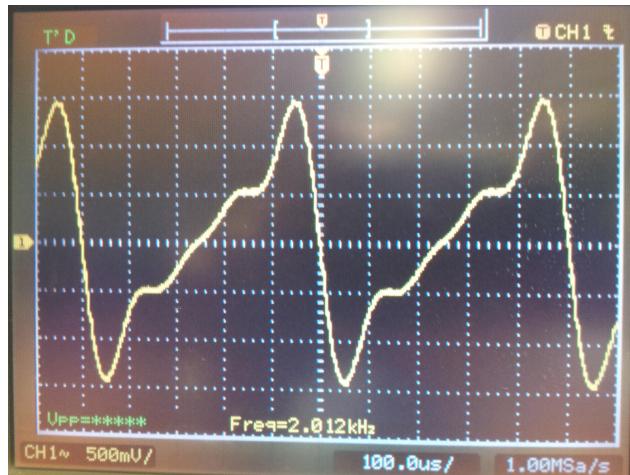


Figura 1: Rampa com frequência de $2kHz$

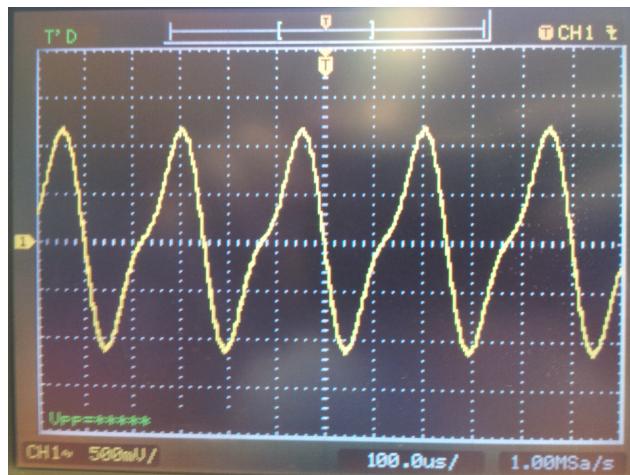


Figura 2: Rampa com frequência de $4kHz$

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.

2. Look-up-table

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

$$\text{round} \left(32767 * \sin \frac{n\pi}{32} \right), \quad n = 0, 1, 2, \dots, 31 \quad (2.2)$$

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 *overflow* nessa amostra do sinal resultando num sinal incorreto.

Como a tabela LUT inclui apenas meio ciclo de seno, ao criar a sinusoide é necessário negar os valores adquiridos ao criar o segundo meio ciclo do seno.

3. Obtenção de um sinal sinusoidal através da LUT

Usando como base a rampa criada na secção P1-1 para indexação da LUT criada na secção 2.2.1, 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 fraccionários. 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
sinusoid=LUT[ index ];
```

Como se pode ver, 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 *sinusoid* (Figura 3).

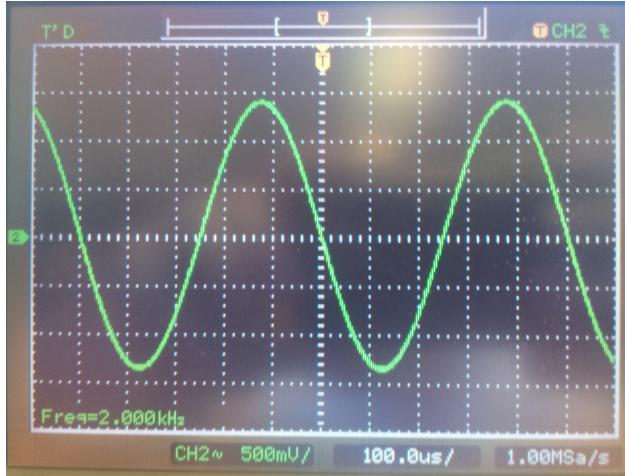


Figura 3: Sinal sinusoidal criado de 2kHz.

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.

4.Criação de variáveis para controlo da frequência e amplitude do sinal sinusoidal

Foram criadas duas variáveis para que 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 2.2.1 e 2.2.1. Esta variável, caso alterada iria 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 e também se o seu valor diminuir, a frequência de saída irá diminuir.

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:

$$Q_m * Q_k = Q_{m+k-n+1} \quad (2.3)$$

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.

5.Utilização do sinal de entrada para controlar a frequência do sinal

Para que a frequência do sinal de saída seja modulada 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:

$$\Delta = \Delta_0 + (K * \text{inbuf}) \quad \Delta_0 = 16384, K = \frac{1}{4} \quad (2.4)$$

Tendo em conta a tabela 2, delta varia entre 8192 e 24576 com Δ_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, uma vez que o DSP não possui divisor, evita uma passagem do programa pela ALU, também representando o efeito de K. Embora na equação 2.4 se some o valor do *inbuf*, no programa tem que se ter em conta o efeito do inversor na placa e por isso efectua-se uma subtração.

6.Teste do oscilador com uma onda sinusoidal à entrada

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. Para isso ser possível era necessário usar a tal onda quadrada como sinal de entrada.

7. Melhoria da qualidade do oscilador com interpolação linear

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

$$y = y_1 + (y_2 - y_1) * \Delta_x \quad (2.5)$$

Sendo que Δ_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 Δ_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) & 31])<<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 Δ_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. De seguida multiplica-se Δ_x com a subtração de dois valores em Q15, que resultam em Q15, pois não ultrapassam 1. Feito isto faz-se um shift para a esquerda para retirar o sinal duplicado e shift 16 para a direita para converter o resultado do multiplicador, variável aux , no formato Q31 para

completei
este pa-
ragrafo a
explicar
a arit-
metica,
acho...

Q15 para se somar de seguida com a variável $y1$ também em Q15 e guardar na variável y que será o sinal de interpolação, também em Q15 pois a soma não excede 1.

Ambos os sinais (com e sem interpolação) podem ser observados na Figura 4, embora a sua comparação não possa ser efetuada por mera observação. Para estes sinais utilizou-se $\Delta_0 = 20479$ correspondente a $f_0 = 5kHz$.

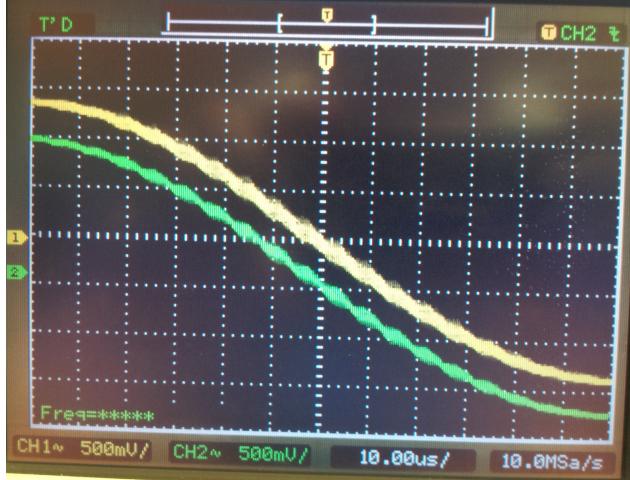


Figura 4: Sinal sinusode com interpolação(amarelo) e sem interpolação(verde).

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

8.Comparação dos espectros com e sem interpolação

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). Para estes sinais utilizou-se $\Delta_0 = 20479$ correspondente a $f_0 = 5kHz$.

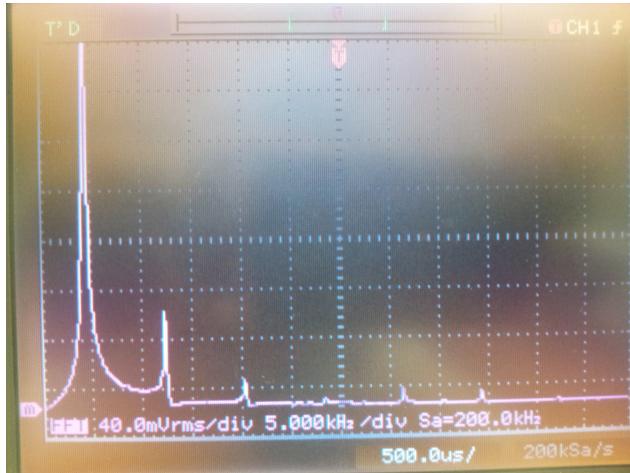


Figura 5: Espectro do sinal sinusode sem interpolação.

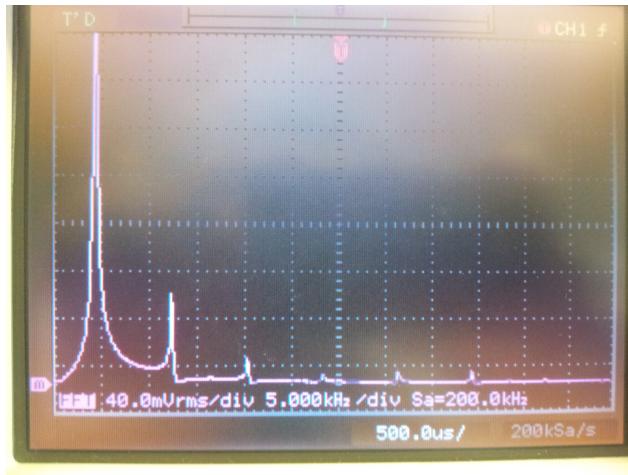


Figura 6: Espectro do sinal sinusoide com interpolação.

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, 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 da portadora $f_0 = 4$ kHz.

1. Fonte de Bits

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:

$$b_n = b_{n-1} \oplus 1 \quad (2.6)$$

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 se dividir f_b por dois pois cada meio ciclo da onda quadrada corresponde a um bit.

2.Codificador Diferencial

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:

$$c_n = c_{n-1} \oplus b_n , c_0 = 0 \quad (2.7)$$

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)

3.Mapeador

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

$$c_n = '1' \rightarrow d_n = +1 \quad c_n = '0' \rightarrow d_n = -1 \quad (2.8)$$

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:

$$d_n = 32767 * ((c_n << 1) - 1) \quad (2.9)$$

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).

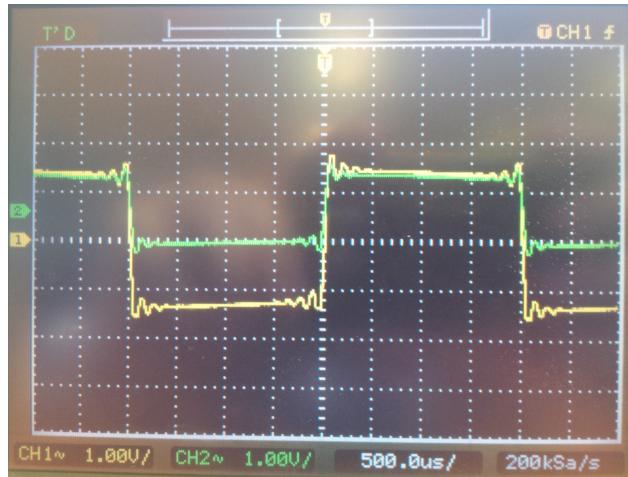


Figura 8: Formas de onda de d_n (amarelo) e c_n (verde).

4.Implementação do Modulador

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 constante (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 sinusóide, 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:

$$s_n = d_n \sin(2\pi f_0 T_s n) \quad (2.10)$$

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

5. Teste do Modulador BPSK

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).

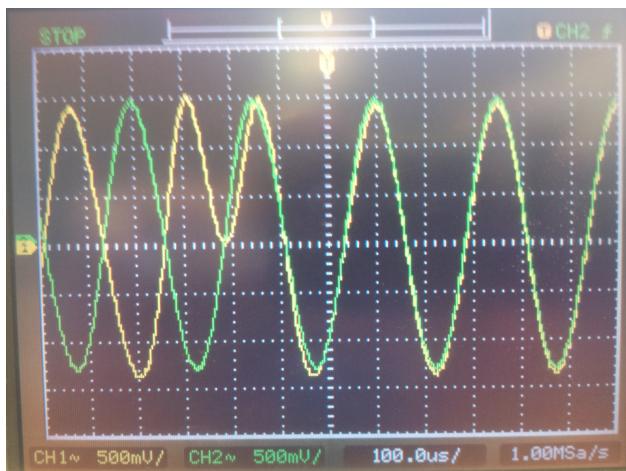


Figura 9: Onda portadora (verde) e onda modulada (amarelo)

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).

Este fenómeno 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 .

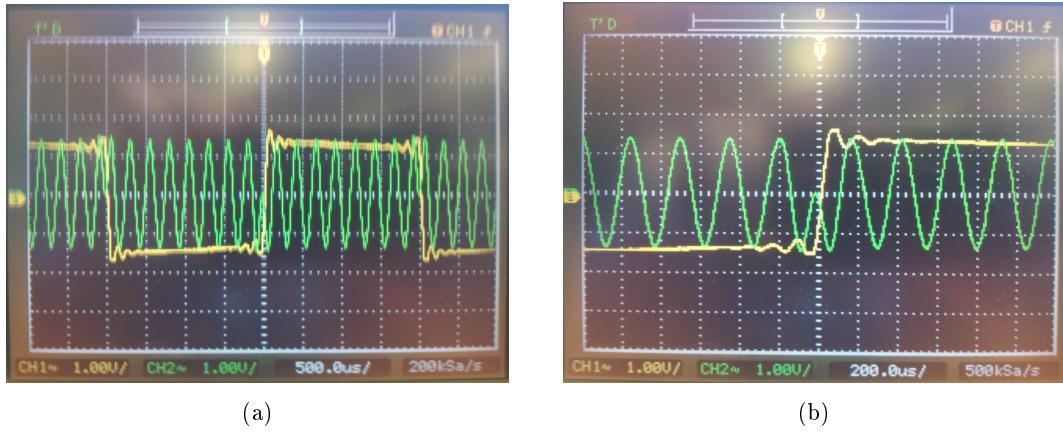


Figura 10: Formas de onda de d_n (amarelo) e s_n (verde), zoom em (b)

6. Espectro do Sinal Modulado

Para compreender melhor o sinal modulado, observou-se o espetro do mesmo (Figura 21).

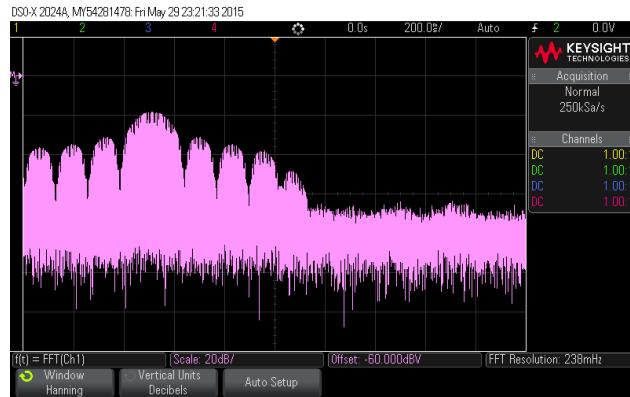


Figura 11: Espectro do sinal s_n .

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, um *sinc*, com o espetro da segunda, um impulso em f_0 . Isto resultaria numa translação do espetro da onda quadrada, ficando centrado na frequência da onda sinusoidal.

Na imagem acima vê-se o espectro de d_n centrado em 4kHz como esperado. O espetro tem o *SPAN* [0-16KHz] e apresenta a forma descrita acima, exceto a falta de réplica a 16kHz, contudo isso é explicado pelo filtro anti-aliasing do analisador de espetros.

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 espectro 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.

2.2.3 Receptor *Costas Loop*

Com o transmissor de BPSK e NCO completos, resta desenvolver o receptor com a capacidade de desmodular o sinal BPSK. Para tal, em vez de optar-mos por circuito do tipo phase-locked loop (PLL) e detetor coerente, desenvolveu-se um desmodulador com o circuito *Costas Loop*. Este é baseado em PLL, contudo tem o dobro da sensibilidade pois tem um detetor de fase para a componente em fase e para a componente em quadratura sendo estes sinais multiplicados, na malha de retroação, resultando num sinal com o dobro do erro para usar no NCO. Tem a vantagem de desmodular sinais com *Doppler shift*. Para este receptor ser utilizado com sucesso, em primeiro lugar o transmissor e o NCO têm que sofrer algumas alterações.

1. Introdução do *Scrambler*:

guilherme

Para um bom funcionamento do Costas Loop é necessário um *scrambler*. O *Scrambler* tem como função não permitir que à entrada se verifique uma longa sequência de '1's ou de '0's, pois estas podem causar uma dessincronização momentânea do sistema. Outra causa para dessincronizações momentâneas do sistema é o aparecimento de sequências periódicas à entrada. A utilização de um *Scrambler* é o método utilizado para diminuir o riscos de dessincronização momentânea.

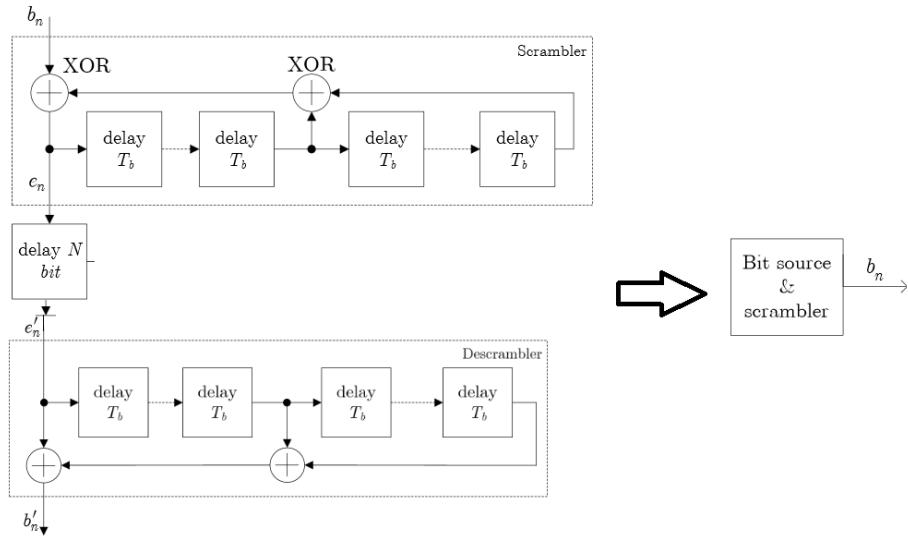


Figura 12: Modelo do *Scrambler* a construir.

A Figura 12 representa o *scrambler* a implementar, nas Equações 2.12, 2.13 e 2.14 está apresentada uma possível maneira de implementar o *Scrambler*.

$$e_n = scr(0) \oplus scr(1) \oplus b_n; \quad (2.11)$$

$$scr = [e_n, scr(7), scr(6), scr(5), scr(4), scr(3), scr(2), scr(1)]; \quad (2.12)$$

$$b_n = b_n \oplus 1; \quad (2.13)$$

$$(2.14)$$

Sendo que scr representa um vector com 8 posições (variável do tipo `char`) com uma sequência de valores "aleatórios", a Equação 2.13 simboliza a actualização dos valores do vector scr, à medida que o tempo avança a posição scr(0) é descartada e é inserida na posição scr(7) o valor de e_n . No entanto a Equação 2.12 representa a determinação de um valor "aleatório". A Equação 2.14 cria uma variável b_n alterna periodicamente entre '1' e '0'.

O *Scramble* demonstrado representa um *pseudo-scramble*, na verdade os valores não são aleatórios, mas sim construídos através de equações que não permitem uma sequência periódica de bits.

De seguida está representado o código utilizado para a implementação do *Scramble*.

```
if (cont==16){
    en=((scr & 1)^((scr & 2)>>1))^bn;      // Criacao do novo valor de scr
    scr=(scr>>1)|(en<<7);      // actualizacao do vector scramble
    bn=bn^1;                      // bn alterna entre '0' e '1'
    cont=0;                        // reinicializa a contador
}
cont=cont+1;
```

Nas Figuras 13 e 14 é possível visualizar a variação das variáveis que pertencem ao *Scrambler* em função do tempo.

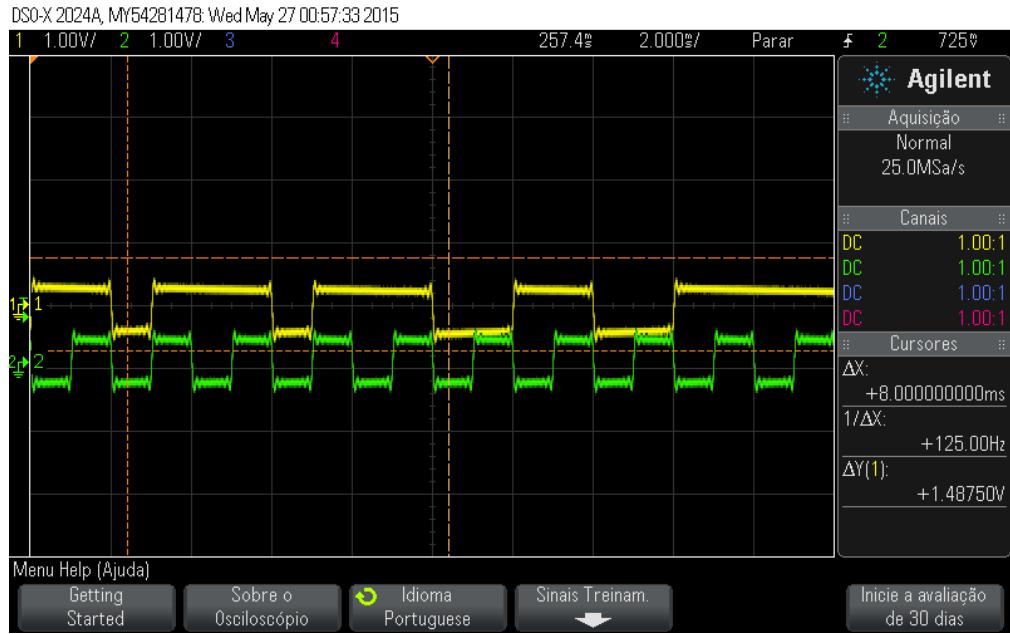


Figura 13: Visualização da variável en(amarelo) em conjunto com bn(verde).

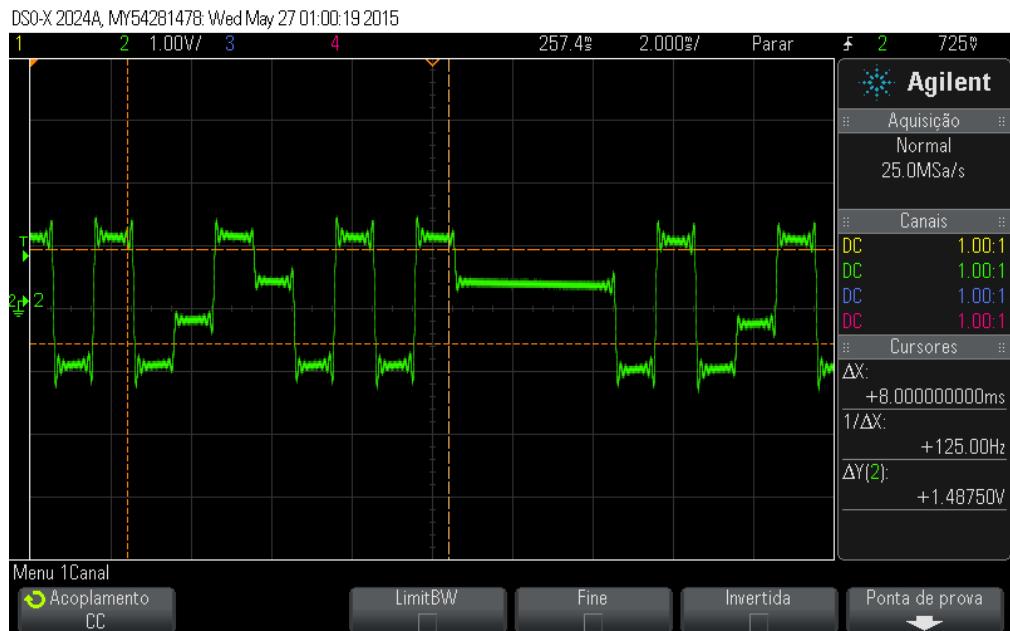


Figura 14: Visualização da variável scr.

Como se pode observar na Figura 13, é possível criar uma variável pseudo-aleatória a partir de uma variável periódica que varia entre '0' e '1'. As Figuras 13 e 14 permitem concluir que o *Scrambler* funciona como esperado.

Ainda tenho de ver o problema da pergunta

2.Implementação de NCO controlado pelo erro:

Para implementar o NCO controlado pelo erro, recorreu-se ao NCO implementado na secção joao 2.2.1, com a mesma rampa, LUT e indexação da mesma como se pode ver no código seguinte.

```
rampa=rampa+delta ;  
index=rampa>>10;  
index=31 & index ;  
aux=ampl*LUT[ index ] ; aux=aux<<1;
```

Mas agora falta a parte de controlar o NCO com o erro. Este controlo da frequência é feito da mesma maneira que na secção 2.2.1, através da variável delta (equação 2.4), este controlo observa-se na linha de código a seguir.

```
delta=16384+(erro>>2);
```

Assim, como o erro controla delta e este controla a rampa que indexa a LUT obtém-se um NCO controlado pelo erro. A seguir prosseguiu-se à obtenção das componentes em fase e em quadratura do sinal pretendido. A componente em fase já tinha sido obtida anteriormente na secção 2.2.1 pois corresponde ao sinal sinusoidal (figura 15). Como a LUT corresponde a meio ciclo do seno foi necessário implementar uma pequena lógica para criar a outra metade do ciclo, enunciada no código a seguir.

```
if (rampa<0)    seno=-aux>>16;  
else    seno=aux>>16;
```

Assim, tem-se duas zonas identificadas pelo sinal da rampa, o meio ciclo positivo e o meio ciclo negativo do seno. No caso da componente em quadratura, devido ao *offset* de 16 amostras obtém-se um sinal cosseno. Este não é tão fácil de representar como o seno, desta vez é necessário ter em conta 4 situações, 4 zonas diferentes num ciclo. Nestas zonas o que varia é a secção da LUT que é indexada e o sinal do cosseno, que dependem do sinal da rampa e da secção da LUT que estiver a ser indexada para o seno. Estas zonas estão enunciadas nos códigos seguintes.

1ª zona

```
if (rampa<0){  
    if (index<=15){  
        aux=ampl*(-LUT[(index+16) & 31]); aux=aux<<1;  
        }coseno=aux>>16;  
    }
```

A primeira zona do ciclo do cosseno corresponde quando a rampa é negativa e o seno está a descer do zero ao seu mínimo (-32767), ou seja, quando se indexa a primeira metade da LUT com o sinal invertido. Assim, quando o seno tem esse comportamento, o cosseno está a subir do seu mínimo ao

zero, sendo necessário aplicar um *offset* positivo de 16 amostras, ou seja, indexar a segunda metade da LUT, também com o sinal invertido. Foi aplicada uma máscara com 31 de modo ao *offset* nunca causar um índice superior ao permitido na LUT, 31.

2^a zona

Esta zona também só existe quando a rampa é negativa. No código seguinte pode-se observar o raciocínio implementado para a mesma.

```
else {
    aux=ampl*(LUT[(index-16) & 31]); aux=(aux<<1);
} coseno=aux>>16;
```

Continuando o comportamento do seno, este vai subir do mínimo ao zero, ou seja, indexa a segunda metade da LUT mais uma vez com o sinal invertido. Logo, o coseno sobe do zero ao máximo (32767) o que implica um *offset* negativo de 16 amostras para indexar a primeira metade da LUT, sem inverter o sinal.

Para a terceira e quartas zonas efetua-se o mesmo raciocínio de indexação só que só se inverte o sinal da LUT na quarta zona pois nestas a rampa é positiva.

3^a zona

```
if (index<=15){
    aux=ampl*LUT[(index+16) & 31]; aux=aux<<1;
} coseno=aux>>16;
```

Aqui, o seno vai subir do zero ao máximo, ou seja, primeira metade da LUT, logo, como o coseno vai descer do máximo ao zero, interessa aceder à segunda metade da LUT, aplicando um *offset* positivo de 16 amostras.

4^a zona

```
else {
    aux=-ampl*LUT[(index-16) & 31]; aux=aux<<1;
} coseno=aux>>16;
```

Por fim, chega-se à última zona do ciclo, onde o seno volta ao zero a partir do máximo, ou seja, segunda metade da LUT. Como o coseno volta ao mínimo a partir do zero, aplicando um *offset* negativo de 16 amostras e uma inversão de sinal nas posições indexadas pela LUT.

Assim obteve-se o sinal em quadratura (figura 15).

Como se pode observar ao comparar as duas componentes verifica-se que estão desfasados por um quarto de ciclo, o que significa um offset de 16 amostras.

3.Implementação dos Filtros:

guilherme

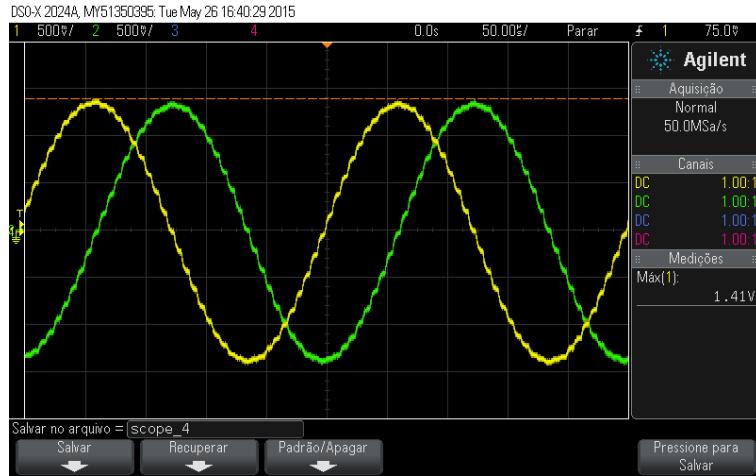


Figura 15: Componentes e fase(verde) e em quadratura do sinal(amarelo).

O Costas Loop necessita de três filtros passa baixo para o seu correcto funcionamento, sendo dois deles denominados Data Filter e o terceiro Loop Filter.

Os três filtros terão as suas equações às diferenças baseadas na Equação 2.15.

$$y(n) = \alpha y(n-1) + \gamma \frac{1-\alpha}{1-\beta} [x(n) - \beta x(n-1)] \quad (2.15)$$

Numa primeira fase o valor de β foi considerado '0', ou seja, os filtros teriam apenas um polo. Originando a função de transferência da Equação 2.16.

$$H(z) = \gamma \frac{(1-\alpha)}{1-\alpha z^{-1}} \quad (2.16)$$

Como queremos filtros com ganho de 0dB a variável γ das Equações 2.15 e 2.16 é dimensionada com o valor de '1'.

Sendo que a equação geral para uma equação às diferenças com o objectivo de implementação está representada na Figura 2.17, podemos concluir que os valores dos coeficientes são os seguintes: $b_{11} = \alpha$, $a_{01} = 1 - \alpha$ e $a_{11} = 0$.

$$y(n) = b_{11}y(n-1) + a_{01}x(n) - a_{11}x(n-1) \quad (2.17)$$

Podemos desde já concluir que a representação da saída do filtro pode ser representada no formato dos coeficientes, visto que $a_{01} + b_{11} = 1$.

Falta apenas calcular a variável α , variável esta que depende da frequência de corte. São então calculados dois valores de α , pois os Data Filter têm um valor de frequência de corte de $1kHz$ e o Loop Filter tem um de valor frequência de corte de $10Hz$.

Devido à dificuldade do cálculo dos coeficientes caso não aceitássemos aproximações foi decidido utilizar métodos de aproximação. A Equação 2.18 demonstra a fórmula utilizada para calcular uma aproximação de α em função da frequência de corte.

$$f_c = f_s \frac{1}{2\pi} \frac{1-\alpha}{\alpha} \implies \alpha = \frac{f_s + 2\pi f_c}{f_c} \quad (2.18)$$

Filtro	f_c	a_{01}	b_{11}	$a_{01} _{Q_{15}}$	$a_{01} _{Q_{15}}$
Data Filter	1kHz				
Loop Filter	10Hz				

Tabela 4: Coeficientes dos filtros e as suas respectivas representações em Q_{15} .

Os coeficientes da Tabela 4 podem ser representados em Q_{15} visto que o módulo de nenhum deles excede '1'.

De seguida segue um excerto de pseudo código que exemplifica a implementação do Loop Filter, visto que a implementação dos Data Filter é igual apenas com valores de coeficientes diferentes.

4. Loop desmodulador:

Nesta fase do projeto têm-se todos os blocos necessários do modem BPSK. O sinal de informação b_n foi misturado, e_n , codificado e mapeado, d_n , e posteriormente multiplicado à portadora com frequência $f_0 = 4kHz$, obtendo o sinal modulado BPSK, s_n . Este durante a transmissão pode sofrer variações de fase ($\Delta\phi = \phi_0 - \phi_1$) e de frequência ($\Delta f = f_1 - f_0$) tomando a forma:

$$s_n = d_n \sin(2\pi f_1 t + \phi_1) \quad (2.19)$$

Para o desmodular é necessário sincronizar o oscilador local com a frequência da portadora, estimando $\Delta\phi$ e Δf .

Usa-se, então, o *Costas Loop*, um tipo de PLL. Em semelhança aos seus familiares é composto por um detetor de fase, um *loop filter* e um oscilador local(NCO), excepto nos dois filtros adicionais em cada ramo exterior. No braço inferior o sinal BPSK é multiplicado com o sinal $\cos(2\pi f_0 t + \phi_0)$ e no superior com $\sin(2\pi f_0 t + \phi_0)$ obtendo-se a componente em quadratura e fase do sinal s_n , respectivamente:

$$s_1 = \frac{d_n}{2} [\cos(4\pi f_0 t + \Delta f + \phi_1 + \phi_0) + \cos(\Delta f + \phi_0 - \phi_1)] \quad (2.20)$$

$$s_2 = \frac{d_n}{2} [\sin(4\pi f_0 t + \Delta f + \phi_1 + \phi_0) + \cos(\Delta f + \phi_0 - \phi_1)] \quad (2.21)$$

Nestes dois multiplicadores ocorre a detecção de fase uma vez que os termos de baixa frequência têm como componentes $\Delta\phi$ e Δf , os indicadores dos erros de fase e frequência que se usará no NCO. Filtrando-se esses sinais nos *Data filters*, multiplicando-os entre si e passando no *Loop filter* obtém-se

$$\epsilon(t) = \frac{1}{8} \sin(2\Delta\phi) \approx \frac{1}{4} \Delta\phi \quad (2.22)$$

ver a
linguagem

para valores pequenos de $\Delta\phi$. Este sinal estático entra no NCO e aumenta ou diminui a frequência do oscilador reduzindo por consequência o Δf , desde que a frequência da portadora se encontre na banda de captura do dispositivo descrito.

Uma vez capturada a frequência da portadora (diferente da frequência central) o oscilador local encontra-se sincronizado à frequência da portadora ($f_1 = f_0 + \Delta f$) e em fase com a mesma ($\Delta\phi = 0$) tendo-se os sinais

$$\frac{1}{2}d_n \cos(\Delta\phi) \approx \frac{d_n}{2}, \quad \frac{1}{2}d_n \sin(\Delta\phi) \approx 0 \quad (2.23)$$

à saída do filtro da componente em quadratura e do filtro da componente em fase, respetivamente, tornando-se o braço do primeiro num detetor coerente. Como já mencionado, o espetro do sinal BPSK é o espetro do sinal de informação centrado à frequência da portadora e quando se multiplica pelo sinal local sincronizado obtém-se réplicas do espetro do sinal de informação a $f = 0$ e $f = 2f_1$, que ao passar no *Data filter* filtra as componentes superiores a 1kHz obtendo o sinal de informação.

5. Testes do *Costas Loop* com onda sinusoidal

Depois de implementar o *Costas Loop* procedeu-se à realização de testes para verificar o seu funcionamento. Primeiro testou-se com uma onda sinusoidal de entrada no *Costas Loop* variando dois parâmetros, a tensão de pico-a-pico do sinal sinusoidal, V_{pp} , a 1V e 3V e a frequência de corte do filtro de *loop*, a 10Hz e a 100Hz. Assim obteve-se quatro situações diferentes com esses parâmetros.

Estas situações foram criadas primeiro pra testar o braço superior do *Costas Loop* e depois pra testá-lo todo. Para cada situação anotou-se as bandas de captura e de seguimento, observadas através do osciloscópio no sinal do coseno mencionado anteriormente.

Ao observar o coseno de frequência central 4kHz, constata-se que há uma zona onde o sinal está em sincronismo (figura 16a), e para obter a banda de captura analisou-se quando o sinal entra em sincronismo, limite inferior e superior (figura 16a).

Para a banda de seguimento analisou-se quando o sinal sai do sincronismo, também com os limites inferior e superior (figura 16b).

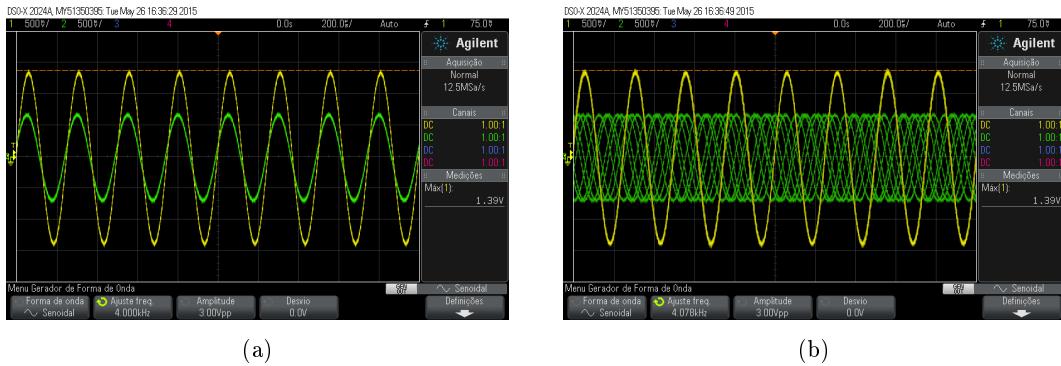


Figura 16: Em sincronismo (a) e fora do sincronismo (b), coseno (verde) e onda sinusoidal (amarelo)

Ao observar o comportamento do sinal coseno nas figuras, a entrar e a sair do sincronismo, verifica-se que este tem o comportamento de um DPLL clássico.

Antes de proceder aos resultados das bandas é necessário salientar que ao considerar um filtro de *loop* de 100 Hz foi necessário calcular os coeficientes do filtro de novo. No código a seguir observa-se os dois filtros implementados e a diferença nos seus coeficientes.

```
erro = (((32639*erro)<<1>>16)+(((128*s)<<1>>16)); //10Hz
//erro= (((31529*erro)<<1>>16)+(((1238*s)<<1>>16));//100Hz
```

Assim, podem-se observar as bandas resultantes do braço superior nas tabelas seguintes.

Vpp \ f	10 Hz	100 Hz
1V	3,936 - 4,032 kHz	3,866 - 4,131 kHz
3V	3,914 - 4,060 kHz	3,774 - 4,226 kHz

(a)

Vpp \ f	10 Hz	100 Hz
1V	3,823 - 4,144 kHz	3,831 - 4,165 kHz
3V	3,496 - 4,472 kHz	3,497 - 4,499 kHz

(b)

A tabela (a) corresponde às bandas de captura e a tabela (b) corresponde às bandas de seguimento. Como se pode ver, todas as bandas compreendem a frequência 4 kHz, oscilando sempre em relação à mesma. É possível também constatar que ao aumentar a amplitude da onda sinusoidal ou a frequência de corte do filtro *loop* a banda de captura aumenta, principalmente com a frequência de corte do filtro. No caso da banda de seguimento só a amplitude é que aumenta a mesma.

Passando ao teste do *Costas Loop* inteiro, pode-se observar os resultados na tabela seguinte, desta vez considerou-se apenas uma situação.

Parâmetros	Banda de Captura	Banda de seguimento
Vpp=3V, f=10Hz	3,965 - 4,005 kHz	3,931 - 4,039 kHz

Comparativamente ao braço superior, ambas as bandas reduziram significativamente com a adição do braço inferior.

6. Testes do *Costas Loop* com BPSK sem zero

Joao

7. Testes do *Costas Loop* com BPSK com zero

O sinal obtido do *Costas Loop* ainda apresenta um vestígio da multiplicação no detetor de fase, sendo possível observar-se uma oscilação de maior frequência somada ao sinal de informação. Tal é devido ao aliasing existente nos filtros discretos que, dadas as réplicas espectrais somarem o ganho no limite de Nyquist, $\frac{F_s}{2} = 8kHz$, o filtro não é seletivo que chegue a essa frequência resultando no observado na figura acima. Posto isto, optou-se por eliminar este ruído da maneira mais simples possível: inserir um zero na função de transferência dos Data filters à frequência do sinal de ruído.

Para tal utilizou-se a equação às diferenças do enunciado, como na alínea P.3, e determinou-se a função de transferência:

$$H(z) = \gamma \frac{1 - \alpha}{1 - \beta} \frac{1 - \beta z^{-1}}{1 - \alpha z^{-1}} \quad (2.24)$$

O ganho estático (γ) foi obtido igualando a função a 1 à frequência $\omega = 0 \Rightarrow z = 1$. O valor de α foi determinado na alínea P.3 e o valor de β foi determinado igualando a função transferência a 0 à frequência $z = e^{j\omega T_s}|_{\omega=2\pi8000}$. Desta maneira insere-se um *Notch* para os 8kHz obtendo-se um *Lowpass Notch filter* com a frequência de corte de 1kHz na mesma contudo com uma atenuação muito maior à frequência do sinal ruído. A equação às diferenças toma então a forma:

$$y_n = \alpha y_{n-1} + \gamma \frac{1 - \alpha}{1 - \beta} (x_n - \beta x_{n-1}), \quad (2.25)$$

com $\beta = -1; \alpha = 0.7180302; \gamma = 1$.

No programa alterou-se apenas os valores dos coeficientes da equação às diferenças dos *data filters*.

```
y1= (((23528*y1)<<1)>>16)+(((4620*s1_0)<<1)>>16)+(((4620*s1_1)<<1)>>16);
```

```
y2=(((23528*y2)<<1)>>16)+(((4620*s2_0)<<1)>>16)+(((4620*s2_1)<<1)>>16);
```

Observa-se na figura abaixo o sinal desmodulado sem o ruído e o sinal de informação.

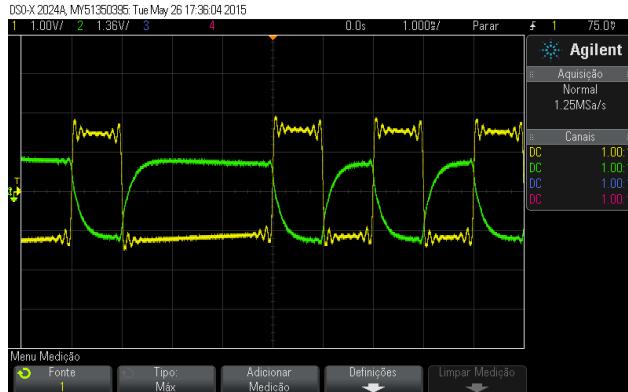


Figura 17: sinal de informação, d_n ,(amarelo) e sinal desmodulado(verde).

De seguida observa-se como previsto que, uma vez que nos encontramos em sincronismo com o transmissor os sinais resultantes dos detetores de fase, s_1 e s_2 são a onda de informação com uma componente de alta frequência somada e zero, respetivamente.

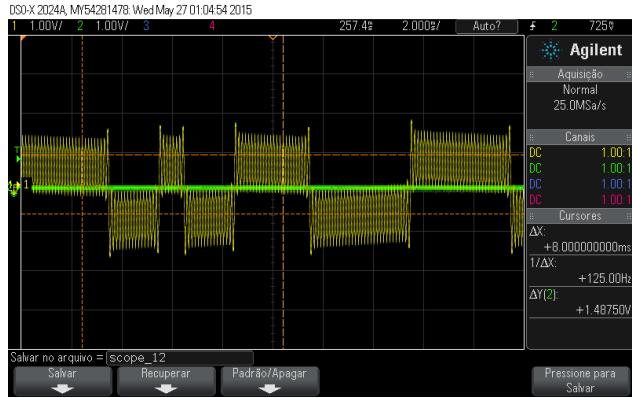


Figura 18: sinal de quadratura, s_1 ,(amarelo) e sinal de fase, s_2 (verde).

Agora tem-se uma ilustração do sinal de erro com o sinal desmodulado. O primeiro como previsto, mais uma vez porque o *Costas Loop* está sincronizado, é nulo pois não se tem $\Delta\phi$ e Δf .

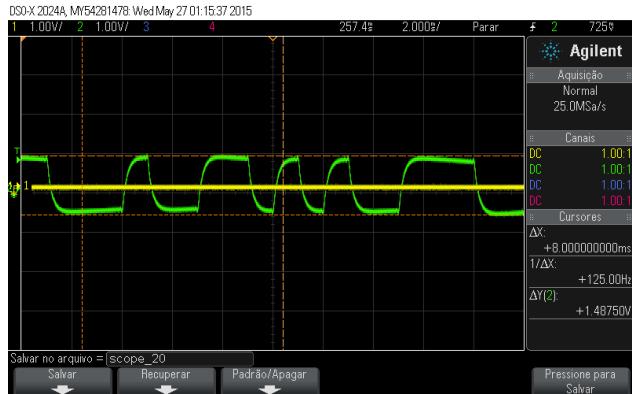


Figura 19: sinal de erro, ϵ ,(amarelo) e sinal desmodulado (verde).

Por fim representa-se os outputs dos data filters e o espetro do sinal desmodulado.

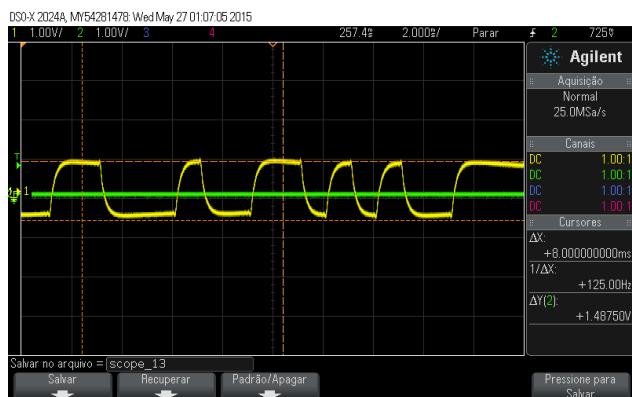


Figura 20: saída do filtro de quadratura(amarelo) e saída do filtro de fase(verde).



Figura 21: Espetro do sinal desmodulado.

Como se pode observar, o espetro trata-se de um *sinc*, com zeros nos múltiplos pares da frequência de d_n , $f_d = \frac{f_b t}{2}$, com a escala horizontal com 1kHz/divisão.

8.Transient

De forma a se poder observar o transitório de aquisição do *Costas Loop*, com a finalidade de poder analisar características adicionais do mesmo, nomeadamente o tempo para se obter sincronismo, fez-se uma análise *transient*. Esta consiste em induzir no circuito um erro de fase elevado, uma vez que é mais simples a alterar a frequência do sinal BPSK, saturando a cada 4000 amostras o valor da variável do erro ($\epsilon = 32767$). Observa-se na figura seguinte os sinais de erro e a saída desmodulada, para um *Loop filter* com frequência de corte, $f_c = 10Hz$:

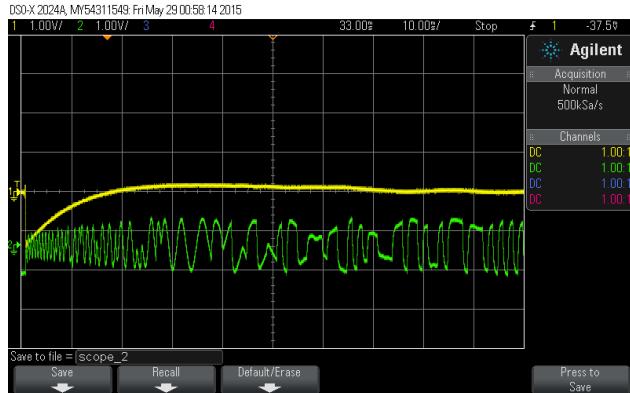


Figura 22: sinal de erro, ϵ , (amarelo) e sinal desmodulado(verde).

Observa-se o comportamento esperado do erro com algum overshoot, um curto tempo de estabelecimento e convergindo para 0. Ao mesmo tempo a forma de onda assemelha-se cada vez mais com o sinal de informação. Para este caso tem-se um tempo de sincronismo de $\sim 53ms$, tendo em conta a escala horizontal ter 10ms/divisão.

Repetiu-se este teste para um *Loop filter* com frequência de corte, $f_c = 100Hz$, e verificou-se

que este circuito sincroniza mais rapidamente com um tempo de sincronismo de $\sim 10\text{ms}$, tal como se observa na figura seguinte, com a mesma escala horizontal do caso anterior.

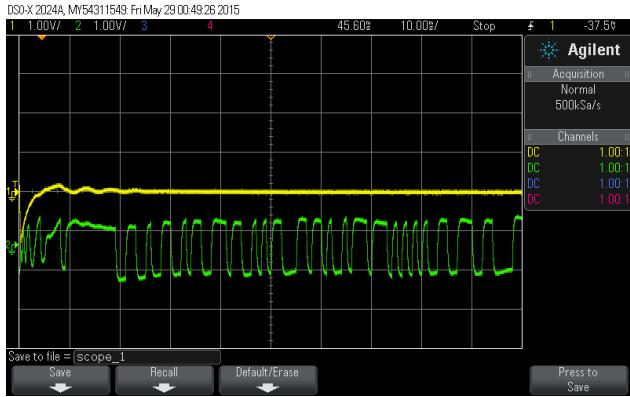


Figura 23: sinal de erro, ϵ , (amarelo) e sinal desmodulado(verde).

3 Conclusão

No projeto do oscilador, aprendeu-se como controlar um oscilador a partir de vários parâmetros e o efeito de interpolação linear no mesmo. Como já foi afirmado, a interpolação criou um sinal com maior precisão embora não tenha sido possível observar essa melhoria devido às limitações do equipamento utilizado.

No projeto do transmissor criou-se a primeira parte do modem BPSK, aproveitando o conhecimento adquirido no projeto do oscilador, e observou-se os efeitos da modulação de uma sequência de bits codificada e mapeada, tanto no tempo como na frequência, o que deu para observar pelas inversões de fase presentes no sinal modulado e nas harmónicas do espetro.

alterar
conclusao

4 Anexos

4.1 Anexo A

acrescentar
codigo da
parte 2

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, 28899, 30274, 31357, 32138, 32610, 32767,
    32610, 32138, 31357, 30274, 28899, 27246, 25330, 23170, 20788,
    18205, 15447, 12540, 9512, 6393, 3212};
    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}; //portadora
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()
{
    comm_intr(); //init DSK, codec, McBSP

    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];                      //sinal da portadora
    cont2= cont2+1;
    sn=((dn*mod)<<1)>>16;      //calculo do sinal modulado
    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 ;
}

}
}

```

4.3 Anexo C

BPSK Modem:

```

//Loop_intr.c Loop program using interrupt. Output = delayed input
// Testado na placa n. 2 - 05/10-A809

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

UInt32 fs=DSK6713_AIC23_FREQ_16KHZ;        //set sampling rate

short rampa=0;
short cont=1;
short count=1;
short delta ;
int index=0;
short seno=0;
short coseno=0;
short x[32];

```

```

short y1=0;
short y2=0;
short y3=0;
short deltaX;
short ampl;
int aux;
short inbuf=0;
short bn= 1;
short cn= 0;
short dn = -32767;
short en=0;
short aux1=0;
short aux2=0;
short aux3=0;
short aux4=0;
short mod = 0 ;
short demod = 0 ;
short erro = 0 ;
short s = 0 ;
short s1_0 = 0 ;
short s1_1 = 0 ;
short s2_0 = 0 ;
short s2_1 = 0 ;
short sn = 0 ;
short y4=0;
short cont2= 0 ;
short port[4]={0 ,32767 ,0 , -32767};
unsigned char scr=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=0;
    short LUT[32]={0,3212,6393, 9512, 12540, 15447, 18205, 20788,
    23170, 25330, 27246, 28899, 30274, 31357, 32138, 32610,
    32767, 32610, 32138, 31357, 30274, 28899, 27246, 25330,
    23170, 20788, 18205, 15447, 12540, 9512, 6393, 3212};
    comm_intr();                                //init DSK, codec, McBSP

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

    while(1){                                //infinite loop
        if(intflag != FALSE){
            intflag = FALSE;

        //bpsk modem com scrambler
        if(cont==16){
            //scrambler
            en=((scr & 1)^((scr & 2)>>1))^(bn;
            scr=(scr>>1)|(en<<7);

            cn=en^cn; //codificador
            bn=bn^1;  //novo bit
            cont=0;
            dn=32767*((cn<<1)-1); //mapeador
        }
    }

    //-----
    //transient
    if(count==4000){
        erro=32767;
        count=1;
    }
    count=count+1;
}

```

```

//-----  

        cont=cont+1;  

//modulacao do sinal  

        mod = port [ cont2 ] ;  

        cont2= cont2+1;  

        sn=((dn*mod)<<1)>>16;      //Q15 virgula fixa  

        cont2=cont2&3;  

//deltamin=8192:  

//deltamax=24576;  

        delta=16384+(erro>>2);    //controlo de frequencia  

//delta=8192;  

        rampa=rampa+delta ;  

//indexacao da LUT  

        index=rampa>>10;  

        index=31 & index ;  

//seno  

        aux=ampl*LUT[ index ] ;  

        aux=aux<<1;  

if (rampa<0){  

        seno=-aux>>16;  

} else {  

        seno=aux>>16;  

}  

//coseno  

if (rampa<0){  

//zona 1  

if (index<=15){  

        aux=ampl*(-LUT[( index+16) & 31]);  

        aux=aux<<1;  

} else{  

//zona 2  

        aux=ampl*(LUT[( index-16) & 31]);  

        aux=(aux<<1);  

}  

        coseno=aux>>16;  

} else{  


```

```

if ( index<=15){
    //zona 3
    aux=ampl*LUT[( index+16) & 31];aux=aux<<1;
} else{
    //zona 4
    aux=-ampl*LUT[( index-16) & 31];aux=aux<<1;
}
coseno=aux>>16;
}

//demodulating loop
//s1=((sn*coseno)<<1)>>16; //sem zero , beta = 0
//s2=((sn*seno)<<1)>>16;

//beta=-1:
s1_1=s1_0;
s1_0=((sn*coseno)<<1)>>16;

s2_1=s2_0;
s2_0=((sn*seno)<<1)>>16;

//-----
//filtros
//data filters
//data filter 1 (com coseno)
y1= (((23528*y1)<<1)>>16)+(((4620*s1_0)<<1)>>16)+(((4620*s1_1)<<1)>>16);
//data filter 2 (com seno)
y2=(((23528*y2)<<1)>>16)+(((4620*s2_0)<<1)>>16)+(((4620*s2_1)<<1)>>16);

s=((y1*y2)<<1)>>16; //entrada do loop filter

//upper arm
//s=y1;

//loop filter
erro = (((32639*erro)<<1)>>16)+(((128*s)<<1)>>16); //10Hz
// error= (((31529*error)<<1)>>16)+(((1238*s)<<1)>>16); //100Hz

```

```
demod=y1;

//-
// Processamento espesifico
//-
inbuf = AIC_buffer.channel[LEFT];
    AIC_buffer.channel[LEFT] =erro;
    //inbuf = AIC_buffer.channel[RIGHT];
    AIC_buffer.channel[RIGHT] = demod;
}
}
```