Instituto Superior Técnico

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

Sistemas Electrónicos de Processamento de Sinal

NCO

&

Transmissor 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, 17 de Abril de 2015

Índice

1 Introdução

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

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

2 Projecto #1 - NCO

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

As características do NCO são apresentadas na seguinte tabela.

Tabela 1: Características do NCO.

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

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

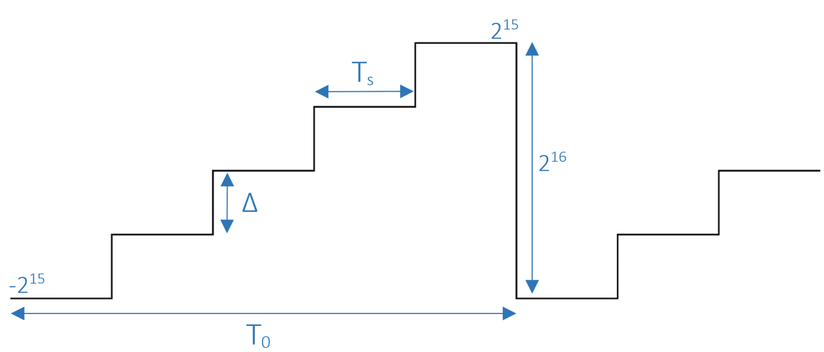


Figura 1: Esquema do oscilador de relaxação.

Com recurso à Figura 1 pode-se deduzir que

$$f_0 = \frac{\Delta}{2^{16}} \times f_s \leftrightarrow \Delta = \frac{f_0}{f_s} \times 2^{16}. \tag{2.1}$$

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 minima, 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

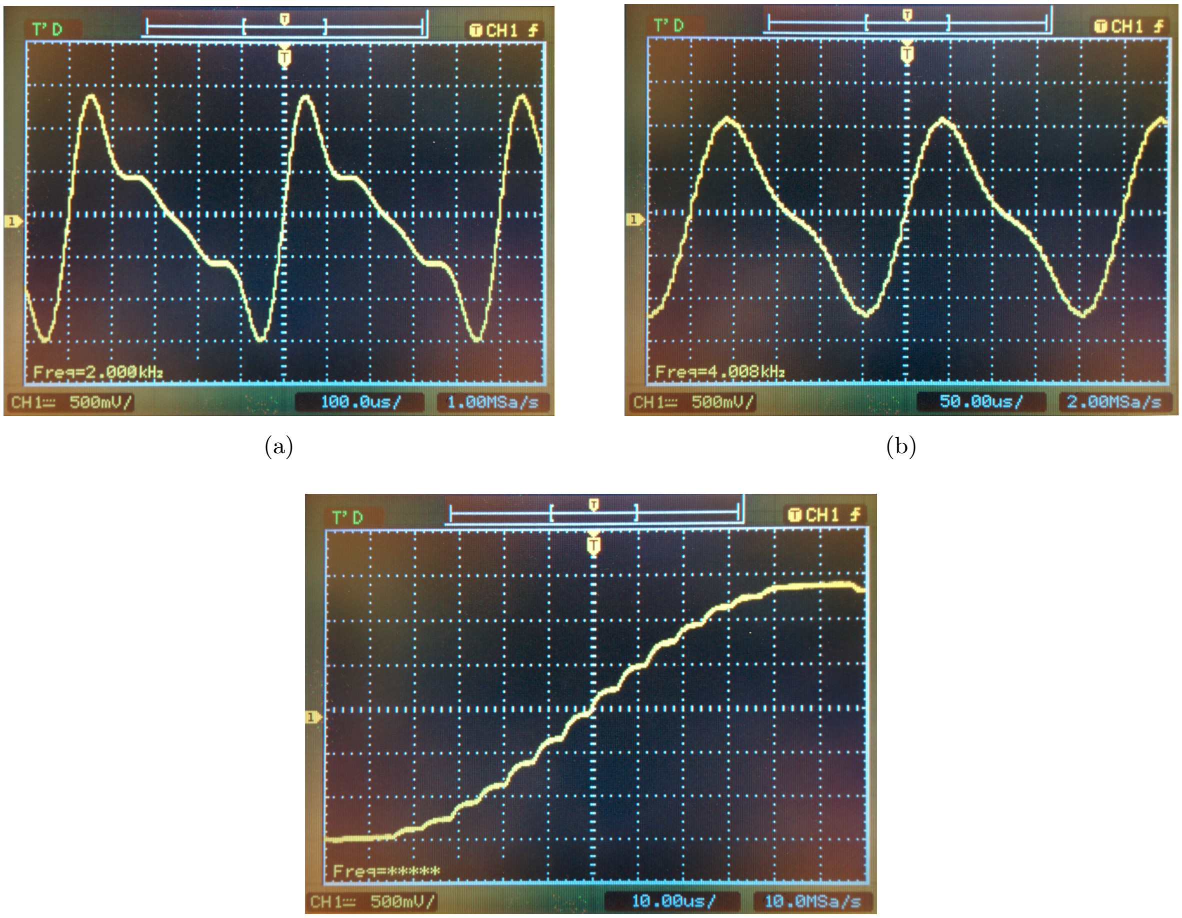
Em código, a variável de estado da rampa é `status` e a variável que representa os incrementos é `delta`. No código abaixo está a criação da rampa para um frequência de 4 kHz.

```
void main(){

  short delta = 16384;
  short status = -32767;

  while(1){
    ...
    //criacao da rampa
    status = status + delta;
    ...
  }
}
```

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



(a) (b) (c)

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

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

$$a_k = \sin\left(\frac{\pi}{32}k\right), k = 0, 1, \ldots, 32. \tag{2.2}$$

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

$$a_{k_{15}} = \text{round}\left(a_k \times 2^{15}\right), \tag{2.3}$$

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*.

```
...
  //LUT do seno
  short sine[33] =
 {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,
 15447,12540,9512,6393,3212,0};
...
```

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 `31`, 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`, com as seguintes linhas de código:

```
...
  y1 = sine[i&31];
  y2 = sine[(i+1)&31];
...
```

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.

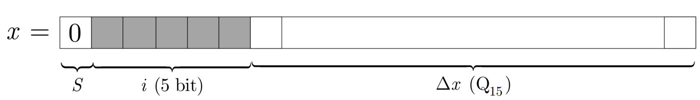


Figura 3: Representação da variável do estado do oscilador.

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

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

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.

```
void main(){
  ...
  //variavel de controlo de frequencia
  short delta = 0
  //variavel de controlo da amplitude: define um ganho de 1/2
  short amp = 16384;
  short yf = 0;
  ...
  while(1){
    if(intflag != FALSE){
    ...
    //obtencao do valor para a frequencia
    delta = 16384 + (inbuf>>2);
     ...
    //controlo da amplitude e frequencia
    yf = (y1*delta<<1)>>16);
    y = (yf*amp<<1)>>16);

    if(status < 0)
      y = -y;

    AIC_buffer.channel[LEFT] = y;
    }
  }
}
```

Analisando 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:

$$\Delta = \Delta_0 + kx. \tag{2.4}$$

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}.

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

Tendo o valor de `delta`, é simples obter a amplitude de cada amostra do sinal de saída, multiplicando `delta` por `y1`, valor obtido da LUT referente à questão 2.2. 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.

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

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.

```
...
  //controlo da amplitude e frequencia
  y = (y_f*amp<<1)>>16);
...
```

Os valores da LUT permitem aceder as 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:

```
...
if(status < 0){
  y = -y;
}
...
```

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

```
...
AIC_buffer.channel[LEFT] = y;
...
```

Com o oscilador controlado já implementado procede-se à fase de testes.

teddy

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

$$y = y_1 + (y_2 - y_1)\Delta x. \tag{2.5}$$

Na Figura 3, onde se encontra representada a variável de estado da rampa, pode-se ver que os 10 *bits* menos significativos desta correspondem à variável Δx da equação (2.5), 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`.

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

Pode-se agora computar y de acordo com a equação (2.5). Quando se faz a subtracção entre `y2` (Q_{15}) e `y1` (Q_{15}), o resultado fica no formato Q_{14} que quando é multiplicado com `delta_x`

margarida

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

```
...
  //obtencao do valor sinusoidal interpolado
  y = (amp*(y1 + ((y2-y1)*delta_x>>15))>>15);
...
```

Como se viu, foram desenvolvidos dois osciladores sinusoidais - com e sem interpolação - sendo agora importante compará-los, comparação feita inicialmente ao nível dos espectros. Quando TAL

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 ou cores mais quentes.

Na Figura 4 apresenta-se as formas de onda obtidas para as sinusoides geradas de acordo com os dois métodos.

Figura 4: Onda sinusoidal obtida sem interpolação (a verde) e com interpolação (a amarelo).

Como se pode ver, o sinal representado a verde apresenta maior dispersão relativamente

margarida

3 Projecto #2 - Transmissor BPSK

Neste projecto, pretende-se implementar o codificador de um transmissor BPSK (binary phase-shift keying), representado na Figura 5.

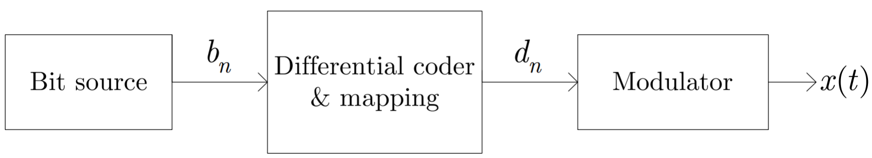


Figura 5: Esquema de um transmissor BPSK.

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:

$$c_n = \text{'0'} \rightarrow d_n = +1; \tag{3.1}$$

$$c_n = \text{'1'} \rightarrow d_n = -1. \tag{3.2}$$

Assim, o sinal modulado BPSK é dado por:

$$s(t) = \sin(2\pi f_0 t + \pi c_n) = d_n \sin(2\pi f_0 t). \tag{3.3}$$

Ou, usando tempo discreto:

$$s_n = s(nT_s) = d_n \sin(2\pi f_0 T_s n). \tag{3.4}$$

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.

```
...
  if(b_i>15){
    b_i=0;
    b_n=(b_n^1); //xor entre valor anterior de bn e o valor logico 1
    ...
  }
  b_i++;
...
```

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:

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

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 divida 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.

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:

$$c_n = c_{n-1} \oplus b_n \tag{3.5}$$

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:

```
...
  c_n = 0;
  ...
  if(b_i>15){
    b_i=0;
    b_n=(b_n^1);
```

```
      c_n=c_n^b_n; //codificador diferencial
      ...
    }
    b_i++;
  ...
```

De seguida, aplica-se o mapeamento aos *bits* do sinal c_n na constelação BPSK, de forma a obter o sinal d_n. O mapeamento segue as expressões da equação (3.1) e (3.2).

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.

```
if(b_i>15){
  ...
  c_n = c_n^b_n;
  shift15_cn = c_n<<15;    // shift de 15 posicoes para a esquerda
  not_shift15 = ~shift15_cn; // negacao do sinal anterior
  d_n = not_shift15 >>14;    // shift de 14 posicoes para a direita
  ...
}
```

figura

Pretende-se agora gerar a portadora com frequência $f_0 = 4$ 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}:

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

acabar pergunta 2 - teddy

pergunta 3 - margarida

4 Conclusões