Instituto Superior Técnico
MEFT Sistema de Aquisição de Dados 2011/2012
Paulo Coelho 65715, Teresa Jorge 65722, Tomás Cruz 65725

# 5º Trabalho: Conversão Analógica – Digital ADC do Tipo "Aproximações Sucessivas" – SAR.

Este trabalho tem como objectivo a implementação de um circuito de conversão analógico – digital do tipo aproximações sucessivas utilizando um circuito integrado DAC, um circuito integrado AmpOp e o controlo feito pelo microcontrolador dsPIC30F4011.

O esquema do ADC é o seguinte:

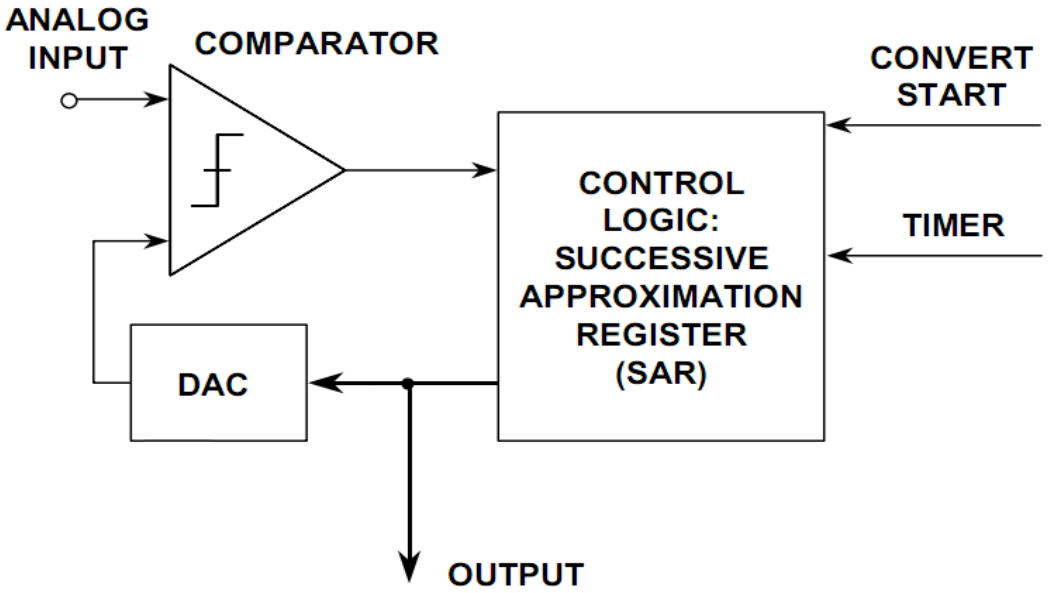



Fig. 1: Esquema do ADC do tipo SAR.

Toda a parte da direita da figura (lógica de controlo, início de conversão e timer) vai ser feita com recurso ao microcontrolador, sendo o sinal de início de conversão e o período do timer (que controla o período de conversão) dados pelo utilizador.

O DAC e o comparador utilizados são os circuitos integrados externos MCP4921 e LM385, respectivamente, sendo este último um simples amplificador operacional com alguma histerese. A comunicação entre o controlo e o DAC é SPI e foi já estudada no trabalho anterior.

Na configuração do esquema da figura 1, que é o que vai ser utilizado, a entrada analógica é um sinal constante. Para se utilizar um sinal variável na entrada teria de ser adicionado um circuito de retenção e amostragem (como o do 2º Trabalho), para se poder garantir um sinal constante na entrada do comparador em toda a conversão.

O ADC do tipo SAR tem por base o princípio de tentativa e erro, procurando a combinação binária que faz com que a saída do DAC seja o valor mais próximo possível do valor de entrada. A melhor forma de fazer estas tentativas é por aproximações sucessivas, testando de cada vez que se coloca um valor na saída se a nossa tentativa foi acima ou abaixo do valor de entrada, e assim ajustar a tentativa seguinte.

A primeira parte deste trabalho foi o teste de funcionamento das comunicações e das limitações do módulo do DAC. A maior parte das características do DAC utilizado e comunicação SPI foram já estudadas no trabalho anterior, em que se concluiu que a comunicação entre o DAC e o µP funciona bem, a curva de transferência do DAC era linear, sendo o seu declive muito próximo do ideal e o seu offset desprezável (para tensões não inferiores à centésima de Volt). O DAC pode ter alguns problemas nos 2 bits menos significativos, isto é, o valor de tensão apresentado à saída do DAC pode ter um erro da ordem da centésima do Volt.

Para este trabalho sentiu-se a necessidade de testar como se comportava o DAC se lhe fosse ordenado que alterasse a sua saída a cada "chamada" de um ciclo. De início houve alguns problemas com esta tarefa que à primeira vista parece bastante simples, pois, em algumas iteradas do ciclo, o DAC não mudava o valor da sua saída. Por essa razão fez-se um pequeno programa de teste que se apresenta de seguida. Com auxílio deste programa concluiu-se que para evitar o problema era necessário escrever no buffer do SPI e fazer clear do SPI1STATbits.SPIROV duas vezes em cada iterada.

Código de Teste:

```
/* Teste DAC */

/* Bibliotecas necessárias ao funcionamento do programa */
#include <p30F4011.h> /* biblioteca com os registos do dspic */
#include <stdio.h> /* biblioteca standard de C para IO */
#include <libpic30.h> /* biblioteca com as configurações do compilador C30 */
#include <uart.h> /* biblioteca com as funções e utilitários da UART */
#include <spi.h>

/* define da frequência de oscilação do oscilador */
#define FCY ((long) 7372)
_FOSC(CSW_FSCM_OFF & XT_PLL4); /* define oscilador com PLL 4x */
_FWDT(WDT_OFF); /* desliga watchdog timer */

/* define de variáveis de configuração da UART */
#define UART_ALTRX_ALTTX   0xFFE7
#define UART_RX_TX  0xFBE7

/* define de variáveis do módulo SPI */
#define SPICONValue FRAME_ENABLE_ON & FRAME_SYNC_OUTPUT & ENABLE_SDO_PIN & SPI_MODE16_ON
& SPI_SMP_ON & SPI_CKE_ON & SLAVE_ENABLE_OFF & CLK_POL_ACTIVE_HIGH & MASTER_ENABLE_ON &
PRI_PRESCAL_1_1 & SEC_PRESCAL_7_1
#define SPISTATValue SPI_ENABLE & SPI_IDLE_CON & SPI_RX_OVFLOW_CLR

char RXbuffer[80]; /* buffer usado para guardar os caracteres provenientes da UART */
int str_pos; /* posição no RXbuffer */
void uartconfig();
void Rxbfclear();
int main(void)
{
    int stop = 0;
    int b = 0b0111100000000000;
    double v = 2.50;
    int j = 0;
    uartconfig();
    printf("Teste DAC\n\n\r");
    Rxbfclear();
    OpenSPI1(SPICONValue, SPISTATValue);

    while(stop!=1)
    {
        b = 0b0000100000000000;
        v = 2.5;
        j = 0;
       TRISBbits.TRISB2 = 0;
        while(j < 12)
        {
            if(RXbuffer[str_pos-1] =='\r')
            {
              SPI1STATbits.SPIROV = 0; // evita overflow do buffer
              WriteSPI1(b|0b0111000000000000);
               SPI1STATbits.SPIROV = 0; // evita overflow do buffer
```

```
                WriteSPI1(b|0b0111000000000000);
                LATBbits.LATB2=0; // coloca o CS a 0 para poder passar os dados para o DAC
                 while(SPI1STATbits.SPITBF);
                 LATBbits.LATB2=1; // leva o CS a 1 o que faz com que DAC coloque na saída
o valor analógico correspondente a b1
                printf("Colocou o valor %d no DAC, que corresponde a %f V\n\n\r", b, v);
                Rxbfclear();
                 b = b >> 1;
                 v = v/2.0;
                 j++;
            }
        }
    }
    CloseSPI1(); /* desliga o SPI */
}

void uartconfig()
{
    /* variáveis auxiliares de configuração da UART */
    unsigned int UMODEvalue, U2STAvalue;
    UMODEvalue = UART_EN & UART_IDLE_CON & UART_NO_PAR_8BIT;
    U2STAvalue = UART_INT_TX & UART_TX_ENABLE & UART_INT_RX_CHAR & UART_RX_TX;
    OpenUART2 (UMODEvalue, U2STAvalue, 3); /* configura e activa a UART a 115000 bps*/
    U2STAbits.URXISEL = 1;
    _U2RXIE = 1; /* activa a interrupção da UART */
    U2MODEbits.LPBACK = 0;
    __C30_UART = 2; //define UART2 as predefined for use with stdio library, printf etc
    printf("\n\r Serial port ONLINE \n");
}
/* função de interrupção da UART */
void __attribute__((__interrupt__,auto_psv)) _U2RXInterrupt(void)
{
    /* apaga a flag da interrupção */
    IFS1bits.U2RXIF = 0;
    while(U2STAbits.URXDA)
    {
        /*guarda o valor que chega à UART no buffer */
        RXbuffer[str_pos] = U2RXREG;
        str_pos++;
        if(str_pos >= 80)
            str_pos = 0;
    }
}
void Rxbfclear()
{
    int j=0;
    str_pos = 0; /* volta à posição 0 do Rxbuffer */
    for(j=0; j<80; j++)
        RXbuffer[j] = '\0'; /* apaga o Rxbuffer */
}
```

O outro dispositivo essencial à implementação do ADC do tipo SAR estudado neste trabalho é o comparador. O comparador utilizado, como referido anteriormente, foi um simples amplificador operacional, LM385, montado sem rede de realimentação, alimentado com o VCC e o GND da placa do µP. Com esta montagem, espera-se que à saída do comparador apareça o valor VCC ou GND, que corresponde à saturação do mesmo devido à falta da malha de realimentação, consoante a diferença de tensão das suas entradas seja positiva ou negativa. O comparador apresentava alguma histerese como se pode observar na figura 2.

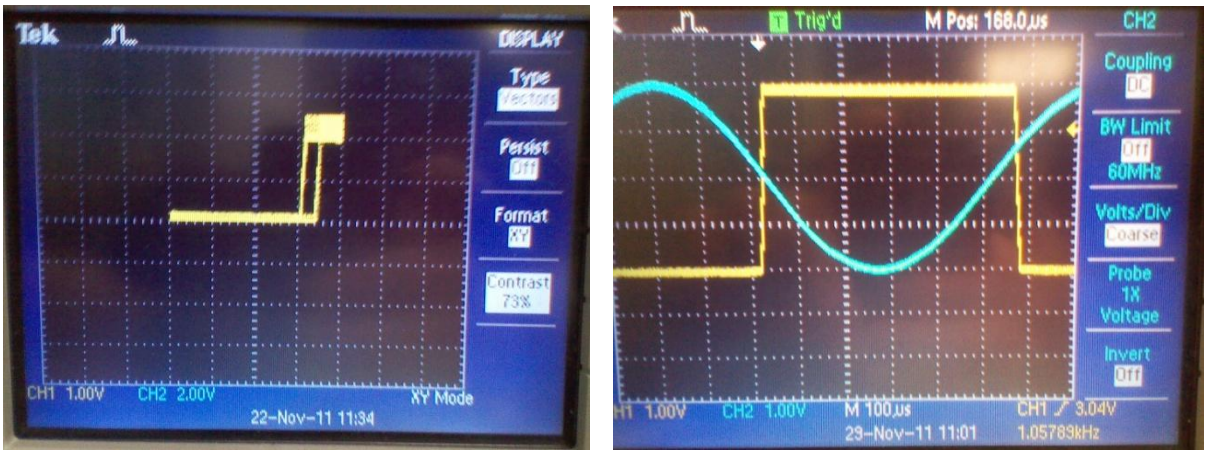



Fig. 2: Exemplo da Histerese do comparador LM385.

Para obter o sinal apresentado na figura 2 utilizamos um sinal sinusoidal. A largura da histerese era variável com a frequência do sinal sinusoidal utilizado na entrada do comparador. Esta informação pode ser relevante para o ADC tipo SAR. Na tabela seguinte apresentam-se alguns valores de histerese para algumas frequências de interesse. O ΔV apresentado foi medido entre a subida e descida da histerese.

| Frequência sinal (Hz) | ΔV Histerese (volt) |
|---|---|
| 5000 | 1,50 |
| 3000 | 1,00 |
| 1000 | 0,35 |
| 500 | 0,20 |
| 100 | 0,10 |

Mais à frente neste trabalho vamos ver se esta histerese vai ou não ter influência nos resultados obtidos. Da mesma forma observou-se que o setling time (tempo de comutação) do comparador é de 16 $\mu s$.

Outro aspecto importante acerca do comparador são as limitações da sua saturação, uma vez que se verificou que, para a tensão de alimentação VCC de 5V, este satura a cerca de 4V.

O próximo teste que foi feito ao comparador foi verificar se este comutava correctamente, para tal alimentou-se uma das suas entradas com 2.5V e a outra ligou-se a uma fonte de tensão contínua. A saída do comparador foi ligada a um pin do microcontrolador e foi utilizado o código do primeiro trabalho na sua opção de medir o estado de um PIN e reflectir o resultado num LED. Verificou-se que o LED acendia quando a tensão da fonte era superior a 2.5V e que o LED apagava quando a tensão da fonte era inferior a 2.5V, pelo que se verifica que o comparador está a ter uma resposta correcta.

Para testar a aquisição do sinal do comparador pelo microcontrolador fez-se o seguinte código de teste.

```
/* Teste Comparador */
/* Bibliotecas necessárias ao funcionamento do programa */
#include <p30F4011.h> /* biblioteca com os registos do dspic */
#include <stdio.h> /* biblioteca standard de C para IO */
#include <libpic30.h> /* biblioteca com as configurações do compilador C30 */
#include <uart.h> /* biblioteca com as funções e utilitários da UART */
#include <spi.h>

/* define da frequência de oscilação do oscilador */
#define FCY ((long) 7372)
_FOSC(CSW_FSCM_OFF & XT_PLL4); /* define oscilador com PLL 4x */
_FWDT(WDT_OFF); /* desliga watchdog timer */

/* define de variáveis de configuração da UART */
#define UART_ALTRX_ALTTX   0xFFE7
#define UART_RX_TX  0xFBE7

/* define de variáveis do módulo SPI */
#define SPICONValue FRAME_ENABLE_ON & FRAME_SYNC_OUTPUT & ENABLE_SDO_PIN & SPI_MODE16_ON
& SPI_SMP_ON & SPI_CKE_ON & SLAVE_ENABLE_OFF & CLK_POL_ACTIVE_HIGH & MASTER_ENABLE_ON &
PRI_PRESCAL_1_1 & SEC_PRESCAL_7_1
#define SPISTATValue SPI_ENABLE & SPI_IDLE_CON & SPI_RX_OVFLOW_CLR

char RXbuffer[80]; /* buffer usado para guardar os caracteres provenientes da UART */
int str_pos; /* posição no RXbuffer */
void uartconfig();
void Rxbfclear();

int main(void)
{
    int stop = 0;
    int b = 0b0000110000000000;
    TRISBbits.TRISB2 = 0;
    ADPCFGbits.PCFG5 = 1;
    TRISBbits.TRISB5 = 1;
    uartconfig();
    Rxbfclear();
    OpenSPI1(SPICONValue, SPISTATValue);
    while(stop!=1)
    {
            if(RXbuffer[str_pos-1] =='\r')
            {
                SPI1STATbits.SPIROV = 0; // evita overflow do buffer
                            WriteSPI1(b|0b0111000000000000);
                SPI1STATbits.SPIROV = 0; // evita overflow do buffer
                            WriteSPI1(b|0b0111000000000000);
                            LATBbits.LATB2=0;
                while(SPI1STATbits.SPITBF);
                LATBbits.LATB2=1; // leva o CS a 1 o que faz com que DAC coloque na
saída o valor analógico correspondente a b1
                if(PORTBbits.RB5 == 0)
                {
                    printf("Vin e menor que 3.75V\n\r");
                }
                else if(PORTBbits.RB5 == 1)
                {
                    printf("Vin e maior que 3.75V\n\r");
                }
                            Rxbfclear();
            }
    }
    CloseSPI1(); /* desliga o SPI */
}

void uartconfig()
{
    /* variáveis auxiliares de configuração da UART */
    unsigned int UMODEvalue, U2STAvalue;
    UMODEvalue = UART_EN & UART_IDLE_CON & UART_NO_PAR_8BIT;
    U2STAvalue = UART_INT_TX & UART_TX_ENABLE & UART_INT_RX_CHAR & UART_RX_TX;
    OpenUART2 (UMODEvalue, U2STAvalue, 3); /* configura e activa a UART a 115000 bps*/
    U2STAbits.URXISEL = 1;
    _U2RXIE = 1; /* activa a interrupção da UART */
    U2MODEbits.LPBACK = 0;
    __C30_UART = 2; //define UART2 as predefined for use with stdio library, printf etc
    printf("\n\r Serial port ONLINE \n");
}
```

```
void Rxbfclear()
{
    int j=0;
    str_pos = 0; /* volta à posição 0 do Rxbuffer */
    for(j=0; j<80; j++)
        RXbuffer[j] = '\0'; /* apaga o Rxbuffer */
}

/* função de interrupção da UART */
void __attribute__((__interrupt__,auto_psv)) _U2RXInterrupt(void)
{
    /* apaga a flag da interrupção */
    IFS1bits.U2RXIF = 0;
    while(U2STAbits.URXDA)
    {
        /*guarda o valor que chega à UART no buffer */
        RXbuffer[str_pos] = U2RXREG;
        str_pos++;
        if(str_pos >= 80)
            str_pos = 0;
    }
}
```

Obteve-se sempre o resultado esperado para tensões inferiores a 4V, pelo que se conclui que se está a receber bem o valor dado pelo comparador com o microcontrolador. Notou-se ainda que para tensões maiores que 4V o resultado era por vezes errado, tal deve-se à saturação do AmpOp. Este factor vai limitar a gama de entrada do nosso ADC do tipo SAR.

Adicionou-se ao código já utilizado nos trabalhos anteriores o código de controlo deste ADC do tipo SAR. Adicionou-se o seguinte código aos ficheiros de código já existentes:

No ficheiro "pttheader.h" não foi necessário fazer novas alterações, utilizou-se a mesma configuração do TIMER3 e do módulo SPI dos trabalhos anteriores.

No ficheiro "pttfunções.c" acrescentou-se novo código na função `iniciar()`:

```
void iniciar()
{
    printf( "\n\rIniciado" );
    (…)
        ADPCFGbits.PCFG5 = 1; /*Define o pin B5 com input digital, entrada do valor do comparador no uP*/
    TRISBbits.TRISB5 = 1; /*Define pin B5 como input*/
}
```

No ficheiro "pttmain.c" acrescentou-se o seguinte código:

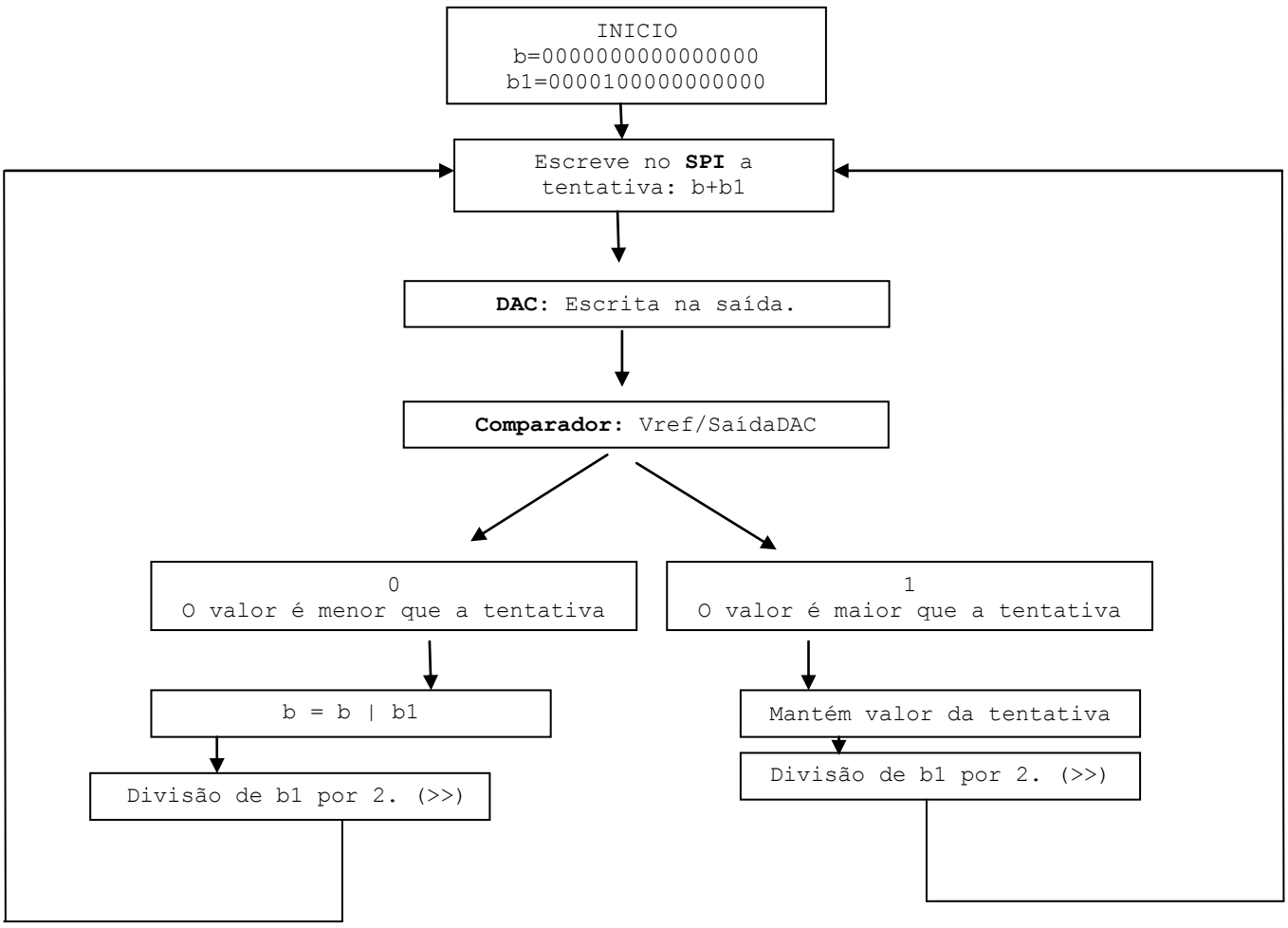
```
int b=0b0000000000000000;
int b1=0b0000100000000000;
iniciar();

if(aux1==6) // se for escolhido comunicar com o DAC a partir do módulo SPI
{
   printf("\n\r Iniciou a rotina que faz o controlo de um ADC do tipo SAR, utilizando o modulo SPI para comunicar com o DAC.\n\r");
   Rxbfclear();
   SPIconfig(); // configura o módulo SPI
   OpenTimer3(T3config, 6);
   ConfigIntTimer3(T3_INT_PRIOR_3 & T3_INT_ON); //configuração do interrupt do timer
   while(j<12)
   {
        SPI1STATbits.SPIROV = 0; // evita overflow do buffer
        WriteSPI1(b |b1  | 0b0111000000000000); //escreve no buffer de transmissão do SPI
        SPI1STATbits.SPIROV = 0; // evita overflow do buffer
        WriteSPI1(b |b1  | 0b0111000000000000); //escreve no buffer de transmissão do SPI
        CS=0; // leva o CS a 0 para fazer a transmissão
        while(SPI1STATbits.SPITBF);
        CS=1; // leva o CS a 1 o que faz com que DAC coloque na saída o valor analógico correspondente a b1
```

```
        if(r==1) //esta variável é alterada no interrupt do timer
         {
            if(PORTBbits.RB5 == 0)
             {
                b = b1 | b; // de a saída do DAC for inferior a Vin, guarda b + b1
             }
            else if(PORTBbits.RB5 == 1)
            {
                b = b;
            }
            b1 = b1 >> 0; // de a saída do DAC for superior a Vin, guarda apenas b
            j++;
            r=0;
        }
    }
    printf( "\r\r\n %d \n\r" , b); // imprime o valor final
    b=0b0000000000000000;
    b1=0b0000100000000000;
    j=0;
    aux1=0;
    stop=1;
}
```

```
void __attribute__((__interrupt__, auto_psv)) _T3Interrupt(void)
{
    IFS0bits.T3IF = 0;
       r=1;
}
```

Fluxograma do programa



O primeiro teste ao algoritmo foi feito sem recurso ao TIMER, em vez disso o programa fazia uma iterada de cada vez que se carregava ENTER (até a conversão estar completa passadas 12 iteradas). Desta forma foi possível acompanhar bem todos os passos da conversão e verificar se a aproximação do sinal à saída do DAC ao valor de Vin estava a correr correctamente. Verificou-se desta forma que tudo estava a correr como era esperado. Obtiveram-se assim os seguintes valores de tenções convertidas:

| Vin (V) | eVin (V) | Saída ADC |
|---|---|---|
| 0.025 | 0.001 | 15 |
| 0.065 | 0.001 | 48 |
| 0.119 | 0.001 | 92 |
| 0.395 | 0.001 | 326 |
| 0.554 | 0.001 | 452 |
| 0.904 | 0.001 | 745 |
| 1.272 | 0.001 | 1048 |
| 1.322 | 0.001 | 1088 |
| 1.656 | 0.001 | 1366 |
| 1.72 | 0.01 | 1436 |
| 2.14 | 0.01 | 1766 |
| 2.41 | 0.01 | 1980 |

| Vin (V) | eVin (V) | Saída ADC |
|---|---|---|
| 2.59 | 0.01 | 2138 |
| 2.83 | 0.01 | 2336 |
| 3.14 | 0.01 | 2584 |
| 3.43 | 0.01 | 2824 |
| 3.79 | 0.01 | 3123 |
| 3.82 | 0.01 | 3111 |
| 3.96 | 0.01 | 3232 |
| 4.09 | 0.01 | 4095 |
| 4.29 | 0.01 | 4095 |
| 4.63 | 0.01 | 3262 |
| 4.83 | 0.01 | 3262 |

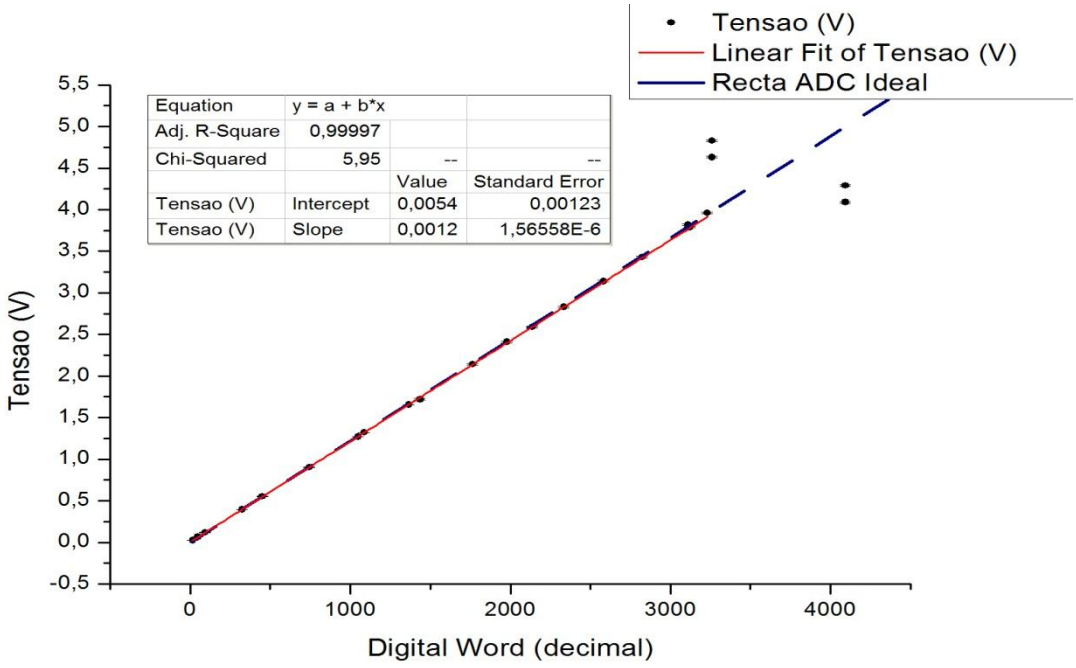
O erro do sinal de entrada é elevado em alguns casos, uma vez que é muito difícil estabilizar o sinal que sai da fonte de tensão.

Vamos então ver a recta de transferência deste ADC e comparar os resultados com a recta de transferência ideal:

$$D_{10} = \frac{V_{in}}{V_{ref}} \times 2^n \leftrightarrow V_{in} = \frac{D_{10}}{2^n} \times V_{ref}$$

$V_{ref} = 5\,V$ e $n = 12$.

O resultado obtido foi:



No ajuste não foram considerados os últimos 4 pontos, que estão obviamente mal. Pode ver-se que, tirando estes últimos pontos, todos os restantes estão quase sobre a recta de transferência ideal. Os últimos pontos estão numa gama de tensão superior à tensão de saturação do AmpOp, daí os maus resultados. Quanto aos parâmetros do ajuste podemos ver que o declive vai ser $(1.2093 \pm 0.0015) \times 10^{-3}$. Que é muito próximo do valor ideal: $1.221 \times 10^{-3}$, a diferença não é explicada pelo erro, uma possível explicação pode ser devida a alguma incerteza no valor de $V_{ref}$. Quanto ao valor de ordenada na origem, temos que o valor obtido é muito próximo de zero, no entanto não é compatível com zero, pelo que podemos estar na presença de um pequeno erro de offset neste ADC. Podemos ainda ver pela tabela (exemplo para 3,79 e 3,82V) que podemos estar na presença de alguma incerteza nos bits menos significativos.

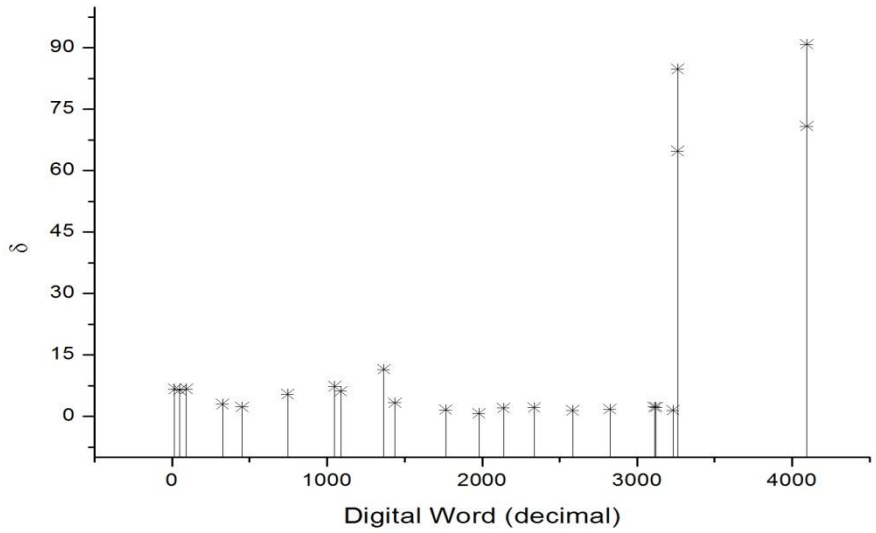
A resolução é dada pelo declive da recta de transferência. A resolução encontrada foi de:

$$(1.2093 \pm 0.0015) \times 10^{-3} \ V/bit$$

Fazendo um plot dos desvios normalizados pelo erro para os pontos obtidos,

$$\delta = \frac{|V_{teo} - V_{exp}|}{erroV_{exp}}$$

Obteve-se o seguinte gráfico:




Podemos ver que, tirando os últimos valores, os desvios normalizados são relativamente pequenos, pelo que as diferenças entre o valor teórico e o experimental são maioritariamente explicadas pelo erro experimental associado à medida de $V_{in}$.

De seguida fomos ver quanto tempo demora uma iterada do programa, com objectivo de ver qual o período mínimo que se podia fazer a conversão, para isso pôs-se o programa em loop infinito, sem timer, com o seguinte ciclo:

```
while(1)
{
     SPI1STATbits.SPIROV = 0; // evita overflow do buffer
     WriteSPI1(b | 0b0111000000000000); // escreve no buffer de transmissão do SPI
     SPI1STATbits.SPIROV = 0; // evita overflow do buffer
     WriteSPI1(b  | 0b0111000000000000); // escreve no buffer de transmissão do SPI
     CS=0; // leva o CS a 0 para fazer a transmissão
     while(SPI1STATbits.SPITBF);
     CS=1;
```

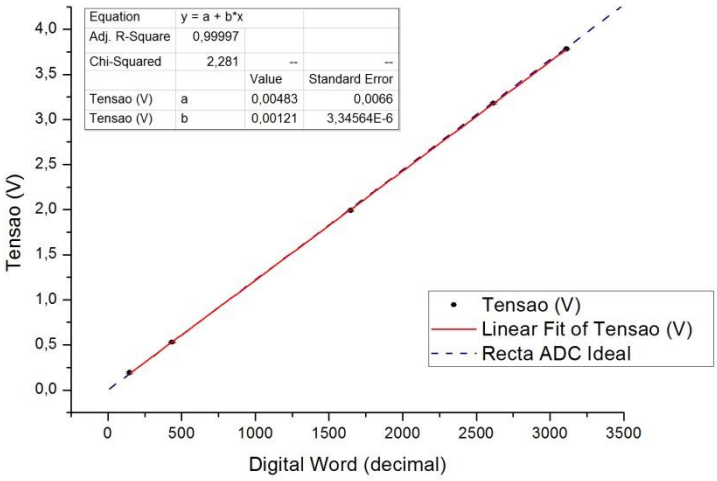
```
        if(PORTBbits.RB5 == 0)
        {
           b=0b0000100000000000;//de a saída do DAC for inferior a Vin, guarda b + b1
         }
         else if(PORTBbits.RB5 == 1)
         {
            b=0b0000000000000000;//de a saída do DAC for superior a Vin, guarda b
         }
         b1 = b1 >> 0;
         j++;
        r=0;
    }
```

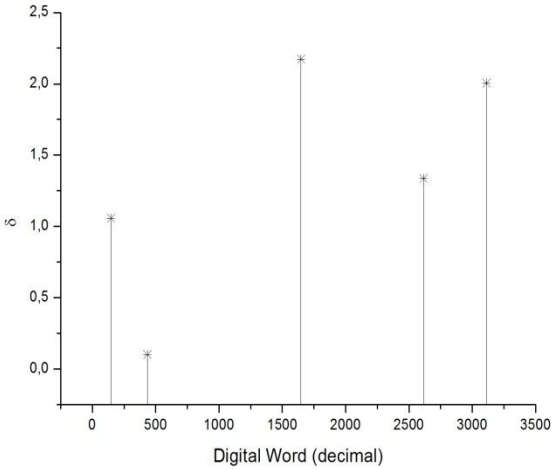
Desta forma, o período mínimo de cada iterada que se pode ter por limitações do código é de 65.6 $\mu s$. Ou seja, o período mínimo de aquisição devido a limitações do código é:

$$12 \times 65.6 = 787.2\ \mu s$$

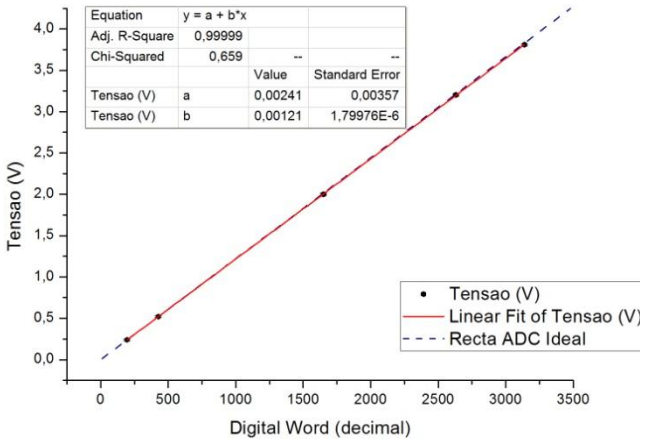
Utilizando o TIMER testaram-se ainda várias taxas de aquisição. Os resultados obtidos apresentam-se de seguida.

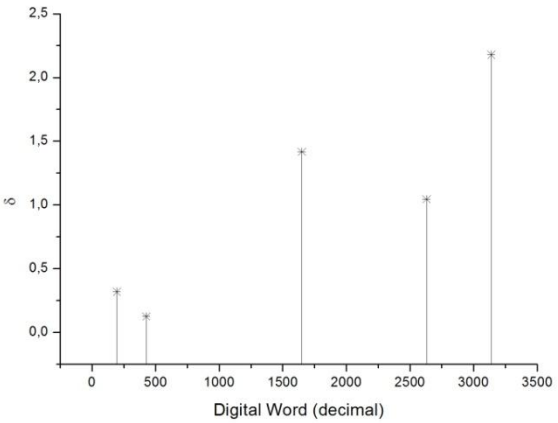
Para uma frequência do TIMER de 1000 $Hz$, ou seja, taxa de aquisição de 83.3 $Hz$:



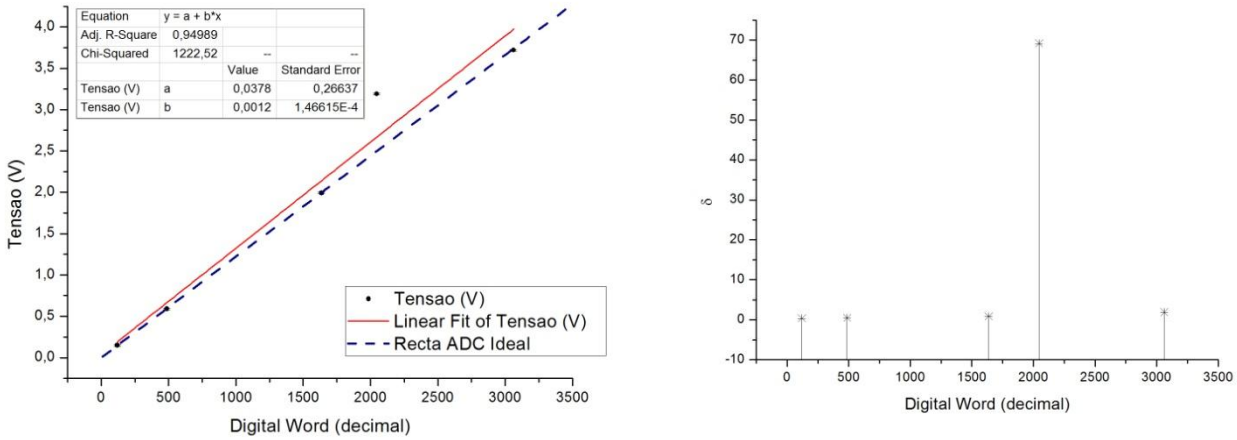




Para uma frequência de TIMER de 10000 $Hz$, ou seja, taxa de aquisição de 833.3 $Hz$:



Para uma frequência de TIMER de 100000 $Hz$, ou seja, taxa de aquisição de 8333.3 $Hz$:




Podemos ver que os resultados foram bons para as frequências de TIMER de 1000 $Hz$ e 10000 $Hz$ e começam a aparecer erros graves nas frequências de TIMER de 100000 $Hz$. Tal deve-se ao facto de esta frequência ser superior ao limite anteriormente estipulado de um período máximo, por causa do tempo mínimo de cada iterada de 65.6 $\mu s \leftrightarrow 15.24\ kHz$. Como não se encontraram problemas para a frequência de 10000 $Hz$ (os desvios são praticamente explicados pelo erro, ou seja, $\delta$ pequeno) e esta frequência é já muito próxima de 15.24 $kHz$, como esta última é inferior ao inverso do setling time do comparador, e os efeitos da histerese não se revelaram muito notáveis para estes ensaios, vamos admitir que vai ser esta frequência que nos vai dar a taxa máxima de aquisição. Logo a taxa máxima de aquisição vai ser $\frac{15.24}{12} = 1.27\ kHz$.

Quanto à gama de valores de entrada, temos que o ADC vai poder receber tensões entre os 0 e os 4V. Este limite superior é determinado pela saturação do AmpOp e pode ser alterado, uma vez que caso se alimente o AmpOp com uma tensão superior este só vai saturar a uma tensão superior.

Globalmente o ADC funcionou como esperado para tensões dentro da gama referida e frequências de aquisição menores ou iguais à máxima acima estipulada. Quanto à sua recta de transferência, temos que a resolução é muito próxima do ideal e a ordenada na origem é muito próxima de zero. No entanto estes valores não são compatíveis com o ADC ideal, pelo que podemos estar na presença de um pequeno erro de offset e de ganho no ADC. Este erro pode ser explicado por alguma incerteza na tensão de referência, esta hipótese não foi devidamente comprovada. Além disso, pelos valores obtidos temos indícios que podem haver problemas nos bits menos significativos. Verificou-se que a taxa máxima de aquisição será de 1.27 $kHz$, que é uma taxa relativamente baixa (requisito de circuito sample & hold para sinais variáveis), é imposta pela demora do código de controlo. Quanto à gama de valores de entrada, para as alimentações colocadas, ficaram-se apenas por ser dos 0 aos 4V, o que serviu para o teste, no entanto, para um correcto funcionamento, deve aumentar-se a alimentação do comparador de forma a garantir uma gama de valores de entrada dos 0 aos 5V.