

# Programação de portos de E/S e de alguns periféricos para a placa DETPIC32 usada nas aulas práticas e nos testes práticos de Arquitetura de Computadores II

# Portos de Entrada/Saída na placa DETPIC32

- Na placa das aulas práticas temos acesso aos seguintes componentes/periféricos:
  - RB0 a RB3 — [entradas] interruptores
  - RB4/AN4 — [entrada analógica] potenciômetro
  - RB8 a RB14 — [saídas] segmentos dos visores de sete segmentos
  - RC14 — [saída] led
  - RD0 a RD4/OC1 a OC5 — [saídas] sinais PWM (UART1 em OC3/OC4, não usada nos guiões 2021/2022)
  - RD5 e RD6 — [saídas] controlo dos dois visores de sete segmentos
  - RD8 e RD11 — [entradas] INT1 e INT4
  - RD9 e RD10 — I2C (não usado nos guiões 2021/2022)
  - RE0 a RE7 — [saídas] leds
  - RF4 e RF5 — UART2
  - RG6 a RG9 — SPI (não usado nos guiões 2021/2022)

# Programação de um porto como entrada ou saída digital

- Como entrada: colocar o bit apropriado do registo TRIS a 1 (por exemplo, RB2)
  - `TRISBbits.TRISB2 = 1;` ou `TRISB |= 0x0004;`
  - Nome do porto, número do porto,  $(1 \ll \text{número do porto})$
- Como saída: colocar o bit apropriado do registo TRIS a 0 (por exemplo, RC14)
  - `TRISCbits.TRISC14 = 0;` ou `TRISC &= 0xBFFF;`
  - Nome do porto, número do porto,  $\sim(1 \ll \text{número do porto})$
- Ler um porto de entrada (por exemplo, RD8)
  - `x = PORTDbits.RD8;` ou `x = (PORTD >> 8) & 1;`
  - Nome do porto, número do porto
- Escrever, por exemplo, um 1 num porto de saída (por exemplo, RB8)
  - `LATBbits.LATB8 = 1;` ou `LATB |= (1 << 8);`
  - Nome do porto, número do porto

Guiões 3 e 4

- Porto AN4 (coincide com RB4)
- Sequência de programação (interrupções opcionais):
  - `TRISBbits.TRISB4 = 1;` // configurar RB4 como entrada
  - `AD1PCFGbits.PCFG4 = 0;` // configurar AN4 como entrada analógica
  - `AD1CON1bits.SSRC = 7;`
  - `AD1CON1bits.CLRASAM = 1;`
  - `AD1CON3bits.SAMC = 16;` // tempo de conversão: 16 TAD (1TAD = 100ns)
  - `AD1CON2bits.SMPI = N - 1;` // N conversões
  - `AD1CHSbits.CH0SA = 4;` // AN4
  - `AD1CON1bits.ON = 1;` // ativar conversões A/D
  - `IPC6bits.AD1IP = 1;` // prioridade da interrupção A/D (1 a 6)
  - `IFS1bits.AD1IF = 0;` // limpar pedido de interrupção A/D
  - `IEC1bits.AD1IE = 1;` // ativar pedidos de interrupção A/D
  - `EnableInterrupts();` // o MIPS aceita pedidos de interrupção

# Conversão Analógico/Digital (continuação)

- Para despoletar uma nova sequência de conversões faz-se: `AD1CON1bits.ASAM = 1;`
- A rotina de serviço a uma interrupção A/D será da forma:

```
static int val; // média dos últimos N valores lidos
void _int_(27) isr_adc(void)
{
    val = (ADC1BUF0 + ADC1BUF1 + 1) / 2; // Para N=2
    IFS1bits.AD1IF = 0; // limpar o pedido de interrupção
}
```

- O 27 é o número do vetor da interrupção. O seu valor numérico deve ser extraído do "PIC32 Family Data Sheet", páginas 74 a 76. No nosso caso, estamos interessados na entrada "AD1 - ADC1 Convert Done" da tabela, coluna "Vector number".
- Nota: é **preciso** ler `ADC1BUF0`, `ADC1BUF1`, ... , até `ADC1BUF(N-1)`:

```
int val = N / 2;
for(int i = 0; i < N; i++)
    val += (&ADC1BUF0)[4 * i];
val /= N; // média arredondada
```

Guiões 6 e 7

# Conversão Analógico/Digital (conclusão)

- No nosso PIC32 o conversor A/D tem uma resolução de 10 bits (valores convertidos de 0 a 1023). Para converter esse valor para outra gama de valores (por exemplo 0.0 a 3.3) usa-se uma simples proporção:

- $0 \rightarrow 0$
- $1023 \rightarrow 3.3$
- $x \rightarrow y$

- Logo,

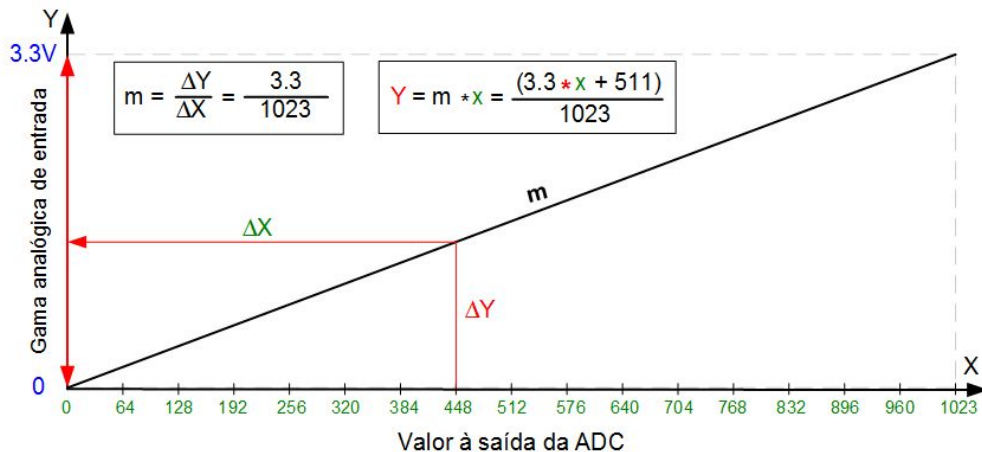
- $y = (3.3 * x) / 1023.0$

- Como o PIC32 não tem o coprocessador 1 (de vírgula flutuante) devemos fazer as contas apenas com números inteiros, pelo que em vez de 3.3 podemos usar 33 (ou mesmo 330), e teremos algo do género:

```
y = (33 * x + 511) / 1023; // com arredondamento!
```

```
y_hi = y / 10; // unidades
```

```
y_lo = y % 10; // décimas
```



# Temporizadores (configuração)

- Frequência pretendida:  $F_{out} = PBCLK / PreScaler[tckps] / (PRx+1)$
- Temporizador 1: `PreScaler[] = { 1, 8, 64, 256 }`
- Temporizadores 2 a 5: `PreScaler[] = { 1, 2, 4, 8, 16, 32, 64, 256 }`
- Sequência de programação (**x é o número do temporizador**, interrupções opcionais):

```
TxCONbits.TON = 0;
```

```
TxCONbits.TCKPS = tckps;
```

```
PRx = round(PBCLK / (Fout * PreScaler[tckps])) - 1;
```

```
TMRx = 0;
```

```
IPCxbits.TxIP = 1; // prioridade da interrupção (1 a 6)
```

```
IFS0bits.TxIF = 0; // limpar pedido de interrupção do temporizador x
```

```
IEC0bits.TxIE = 1; // ativar pedidos de interrupção do temporizador x
```

```
TxCONbits.TON = 1;
```

- Nota: em vez de `round(double)`, use

```
int round_div(int a, int b) { return (a + b / 2) / b; }
```

que arredonda a/b sem fazer contas em vírgula flutuante.

Guião 8

# Temporizadores (conclusão)

- A rotina de serviço a uma interrupção do temporizador número **x** será da forma:

```
void _int_(VETORx) isr_timer_x(void)
{
    // fazer aqui o que precisa de ser feito
    IFS0bits.TxIF = 0;    // limpar o pedido de interrupção
}
```

- O **VETORx** é o número do vetor da interrupção. O seu valor numérico deve ser extraído do "PIC32 Family Data Sheet", páginas 74 a 76. No nosso caso, estamos interessados na entrada "**Tx** - Timer**x**" da tabela, coluna "Vector number".
- Fazer uma acção de 10 em dez interrupções, por exemplo, pode ser feita da seguinte maneira (colocar o código seguinte dentro da rotina de serviço à interrupção):

```
static int counter = 0;
if(++counter == 10)
{
    // fazer aqui o que precisa de ser feito de dez em dez vezes
    counter = 0;
}
```

Guião 8



# Geração de um sinal de *Pulse Width Modulation* (PWM)

- No PIC32 existem cinco módulos de "*output compare*", OC1 a OC5. Cada um deles pode ser usado para gerar um sinal de PWM e usa ou o temporizador 2 ou o 3.
- *Duty cycle* = tempo ligado (a 1) num período dividido pelo período, e multiplicado por 100 (para que o resultado seja uma percentagem).
- Num período, o **TMR<sub>x</sub>** toma os valores 0, 1, ..., **PR<sub>x</sub>**, onde x é ou 2 ou 3.
- O sinal de saída do OC<sub>y</sub>, y=1,...,5, é obtido comparando **TMR<sub>x</sub>** com OC<sub>y</sub>RS. Se OC<sub>y</sub>RS for menor que **TMR<sub>x</sub>** a saída OC<sub>y</sub> fica a 1, caso contrário fica a zero.
- Sequência de programação:

```
// configurar o temporizador x (x=2 ou 3) com a frequência pretendida
// use o valor de PRx o maior possível, mas sem ultrapassar os 16 bits
OCyCONbits.OCM = 6;
OCyCONbits.OCTSEL = x - 2; // 0=temporizador 2, 1=temporizador 3
OCyRS = ((PRx + 1) * duty + 50) / 100; // duty=duty cycle pretendido
OCyCONbits.ON = 1;
```

Guião 9

# RS-232 (UART)

- No PIC32 da placa das aulas práticas existem 6 UARTs (*Universal Asynchronous Receiver Transmitters*).
- A **UART 2** está ligada ao cabo USB, via um circuito da FTDI que converte RS-232 para *RS-232 over USB*.
- Especificação das características do sinal RS-232 (o **com\_spec** do **pterm**):  
**baud\_rate, parity, data\_bits, stop\_bits**
  - **baud\_rate** é o inverso do tempo de duração de um bit
  - **parity** especifica a existência ou não do bit de paridade e a sua polaridade
    - **N** (*none*) — sem bit de paridade
    - **E** (*even*) — paridade par (ou exclusivo dos bits de dados)
    - **O** (*odd*) — paridade ímpar (negação do ou exclusivo dos bits de dados)
  - **data\_bits** é o número de bits de dados (8 ou 9 no PIC32, o **pterm** não aceita o 9)
  - **stop\_bits** é o número de stop bits (1 ou 2 no PIC32)
- Se não for especificado outro valor na linha de comando, o **pterm** usa **115200, N, 8, 1** para o **com\_spec**

Guião 10

# RS-232 (programação)

- Sequência de programação para a UART **x**:

```
int ovs_factor[2] = { 16,4 };  
int ovs = 0;           // 0 (x16, standard speed) ou 1 (x4, high speed)  
UxMODEbits.ON = 0;     // desativa a UART  
UxMODEbits.BRGH = ovs; // configura fator de sobre amostragem (0 ou 1)  
UxBRG = round_div(PBCLK,ovs_factor[ovs] * baud_rate) - 1;  
UxMODEbits.PDSEL = 0;  // 0 (8N), 1 (8E), 2 (8O), 3 (9N) --- ver manual  
UxMODEbits.STSEL = 0;  // 0 (1 stop bits), 1 (2 stop bits) --- ver manual  
UxSTAbits.UTXEN = 1;   // ativa transmissão (ver nota abaixo)  
UxSTAbits.URXEN = 1;   // ativa receção (ver nota abaixo)  
UxMODEbits.ON = 1;     // ativa UART
```

- Para `ovs=0` e `PBCLK=20MHz`, o maior baud rate possível (`UxBRG = 0`) é 1250000 e o menor possível (`UxBRG=65535`) é pouco mais de 19.
- Para `ovs=1` e mesmo `UxBRG` estes números são quatro vezes maiores.

Guião 10

Nota: Quando a transmissão ou a receção é ativada não é preciso configurar os pinos respetivos usando os registos **TRIS**

- Para enviar o carater `c`, armazenado numa variável do tipo `int`:

```
while(UxSTAbits.UTXBF != 0)
    ; // espera até que o buffer de transmissão deixe de estar cheio
UxTXREG = c;
```

- Para receber o carater `c`, armazenado numa variável do tipo `int`:

```
while(UxSTAbits.URXDA == 0)
    ; // espera até que o buffer de receção deixe de estar vazio
c = UxRXREG; // uma segunda leitura deste registo lerá o carater seguinte!
```

- Podem ocorrer vários tipos de erros na receção:

- erro de paridade: comparar `UxSTAbits.PERR` com 1 para ver se ocorreu
- erro de trama: comparar `UxSTAbits.FERR` com 1 para ver se ocorreu
- carater recebido com o buffer de receção cheio (erro de *overrun*): comparar `UxSTAbits.OERR` com 1 para ver se ocorreu (se sim, terá de escrever zero em `UxSTAbits.OERR` para limpar o buffer de receção, caso contrário **não receberá mais nada**)

- Sequência de programação das interrupções para a UART **x**, **y** é o número do registo apropriado (consulte a tabela das páginas 74 a 76, colunas *Flag*, *Enable* e *Priority*, do "PIC32 Family Data Sheet", respetivamente para determinar o número do registo **IFS**, **IEC** e **IPC**):

```
UxSTAbits.UTXISEL = 0; // quando se pede interrupção do lado tx (ver manual)
UxSTAbits.URXISEL = 0; // quando se pede interrupção do lado rx (ver manual)
IPCybits.UxIP = 1;      // prioridade da interrupção (1 a 6)
IFSybits.UxTXIF = 0;    // limpar pedido de interrupção do lado tx
IFSybits.UxRXIF = 0;    // limpar pedido de interrupção do lado rx
IFSybits.UxEIF = 0;     // limpar pedido de interrupção por causa de erros
IECybits.UxTXIE = 0;    // desativa pedidos de interrupção na transmissão
IECybits.UxRXIE = 1;    // ativa pedidos de interrupção na receção
IECybits.UxEIE = 0;     // desativa pedidos de interrupção em caso de erro
```

- A rotina de serviço às interrupções é comum aos lados tx e rx e a erros. Tem de se ver a(s) causa(s) do pedido de interrupção analisando os valores de **IFSybits.UxTXIF**, **IFSybits.UxRXIF** e **IFSybits.UxEIF**.