

Trabalho prático extra

Objetivos

- Compreender o princípio de funcionamento e implementar a estrutura base de um *device driver*.

Introdução

O objetivo deste trabalho prático é a implementação da estrutura básica de um *device-driver* para a UART do PIC32. Um *device-driver* é um programa que permite a uma aplicação comunicar com um dispositivo hardware sem ter de conhecer com detalhe o seu funcionamento e configuração. O *device-driver* lida com as especificidades de implementação, configuração e funcionamento do dispositivo, e fornece uma interface genérica a partir da qual a aplicação interage com o hardware - se a interface se mantiver inalterada, a eventual mudança do dispositivo hardware não acarreta qualquer mudança na aplicação. Uma outra característica de um *device-driver* reside na utilização de interrupções como modelo de transferência de informação.

No caso de um *device-driver* para a UART, a implementação baseia-se no desacoplamento da transferência de dados entre a aplicação e a UART, sendo a ligação efetuada através de FIFOs. Isto significa que a aplicação interage exclusivamente com os FIFOs, através de funções de leitura e escrita, enquanto que a transferência de informação entre os FIFOs e a UART é gerida por interrupção e é da exclusiva responsabilidade do *device-driver*. Os FIFOs podem ser implementados como *buffers* circulares, designação que será utilizada no restante texto deste trabalho.

Transmissão

A Figura 1 representa, esquematicamente, a estrutura da componente de transmissão do *device-driver*. A função `comDrv_putc()` escreve a informação a enviar para a UART no *buffer* circular de transmissão (*append*) e mantém atualizadas duas variáveis de gestão desse *buffer*: A variável `tail` que indica a posição de escrita do próximo carater e a variável `count` que contém o número de caracteres do *buffer* ainda não enviados para a UART. Estas duas variáveis são incrementadas por cada novo carater escrito no *buffer*, sendo a variável `tail` incrementada em módulo n (0, 1, 2, ..., $n-1$, 0, 1, ...), em que n é a dimensão do *buffer*.

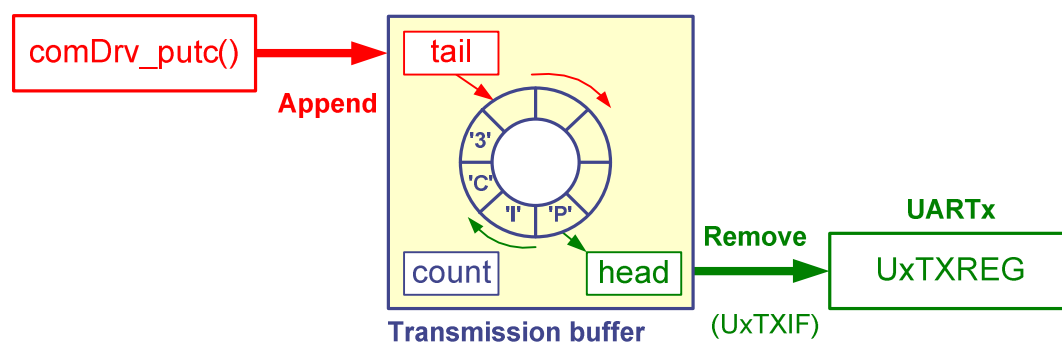


Figura 1. Representação esquemática da componente de transmissão do *device-driver* para a UART.

A informação colocada no *buffer* circular de transmissão é posteriormente enviada, por interrupção, para a UART (*remove*), isto é, a rotina de serviço à interrupção lê o carater residente na posição `head` do *buffer*, e escreve-o no registo de transmissão da UART. Adicionalmente, por cada carater transferido para a UART, a rotina de serviço atualiza as

variáveis **head** e **count**: a variável **count** é decrementada e a variável **head** é incrementada em módulo **n** (em que **n** é a dimensão do *buffer*).

Receção

A Figura 2 representa, esquematicamente, a estrutura da componente de receção do *device-driver*. Sempre que a UART recebe um novo carater gera uma interrupção e, na respetiva rotina de serviço, esse carater é copiado para o *buffer* de receção para a posição referenciada pela variável **tail** (*append*). A rotina de serviço à interrupção mantém atualizadas as variáveis **tail** e **count** de modo idêntico ao já explicado para o *buffer* de transmissão. Os caracteres presentes no *buffer* de receção são lidos pela aplicação (*remove*) através da função **comDrv_getc()** que, de cada vez que lê um carater, atualiza também as variáveis **head** e **count**.

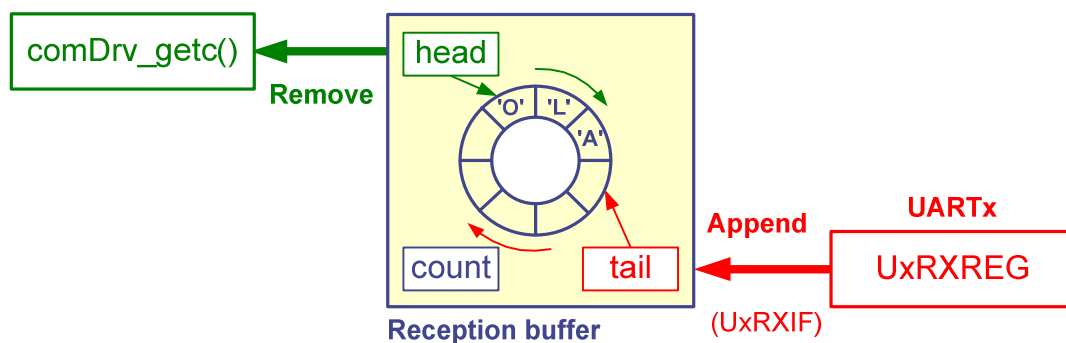


Figura 2. Representação esquemática da componente de receção do *device-driver* para a UART.

Implementação dos *buffers* circulares

Os *buffers* circulares de transmissão e receção têm uma estrutura semelhante. Para a implementação de cada um destes *buffers* é necessário ter uma área de armazenamento (um *array* linear de caracteres) e um conjunto de variáveis auxiliares de gestão: o índice do *array* onde se pode escrever o próximo carater (**tail**); o índice do *array* de onde se pode ler um carater (**head**); um contador que mantém atualizado o número efetivo de caracteres do *buffer* (**count**).

Para a definição da estrutura de um *buffer* circular podemos utilizar, em linguagem C, uma estrutura:

```
typedef struct
{
    unsigned char data[BUF_SIZE];
    unsigned int head;
    unsigned int tail;
    unsigned int count;
} t_cbuf;
```

A partir desta estrutura podem ser instanciados os dois *buffers* circulares do *device driver*:

```
volatile t_cbuf txb; // Transmission buffer
volatile t_cbuf rxb; // Reception buffer
```

Uma vez que as variáveis associadas a estes dois *buffers* circulares podem ser alteradas pelas rotinas de serviço à interrupção, tem que se acrescentar na instanciação das estruturas a palavra-chave **volatile**. Desse modo, em operações de leitura de qualquer uma dessas variáveis fora da rotina de serviço à interrupção, força-se o compilador a gerar código que

efetue o acesso à posição de memória onde a variável reside, em vez de usar uma cópia temporária residente num registo interno do CPU.

A dimensão do *array* de caracteres (**BUF_SIZE**) deverá ser ajustada em função das necessidades previsíveis de tráfego. Por uma questão de simplicidade no processamento associado aos índices de gestão do *array* deve-se, no entanto, estabelecer como dimensão um valor que seja uma potência de 2 (2, 4, 8, 16, ...). A aplicação de uma máscara com o valor da dimensão do *array* menos 1 (**BUF_SIZE-1**, i.e., 1, 3, 7, 15, ...), após uma operação de incremento, garante a rotação do valor do índice, sem qualquer teste adicional.

```
#define BUF_SIZE      32
#define INDEX_MASK    (BUF_SIZE - 1)
```

Receção – rotina de serviço à interrupção

Quando a UARTx gera uma interrupção, o carater recebido deve ser transferido para a posição **tail** do *buffer* de receção, o valor do contador de caracteres, **count**, deve ser incrementado e o índice de escrita, **tail**, deve ser incrementado em módulo **n**. No caso em que o *buffer* fica cheio (situação que ocorre quando os caracteres recebidos e colocados no *buffer* não foram lidos pela aplicação) uma solução para não perder a informação mais recente consiste em descartar o carater mais antigo. Para isso basta incrementar o índice de leitura **head** (sem incrementar o valor do contador **count**).

Receção – leitura do *buffer*

A função **comDrv_getc()**, tal como esquematizado na Figura 2, lê um carater do *buffer* de receção, da posição referenciada pela variável **head**, e atualiza as variáveis **count** e **head**. O facto de as variáveis **count** e **head** poderem ser alteradas na rotina de serviço à interrupção e na função **comDrv_getc()** faz com que elas tenham que ser encaradas como recursos partilhados. Isso tem como consequência que, fora da rotina de serviço à interrupção, uma zona de código que altere essas variáveis seja considerada uma secção crítica. Na secção crítica, não deverá ser permitido o atendimento de interrupções de receção da UARTx. Assim, a interrupção de receção deve ser desativada antes da secção crítica e reativada logo depois.

Transmissão – escrita no *buffer*

A escrita de um carater no *buffer* de transmissão é efetuada, como indicado na Figura 1, pela função **comDrv_putc()**. Esta função copia o carater a enviar para o *buffer* e atualiza as variáveis **count** e **tail**. A variável **count** pode também ser escrita na rotina de serviço à interrupção, pelo que o código de alteração desta variável fora dessa rotina constitui, à semelhança do já descrito para a receção, uma secção crítica. Nessa zona de código, deve ser efetuada a desativação das interrupções de transmissão da UARTx.

Transmissão – rotina de serviço à interrupção

Na UARTx do PIC32 uma interrupção é gerada enquanto o FIFO de transmissão tiver, pelo menos, uma posição livre¹. Na estrutura do *device-driver*, a rotina de serviço à interrupção lê do *buffer* de transmissão o carater referenciado pela variável **head** e escreve-o no registo de transmissão da UARTx. Adicionalmente, atualiza as variáveis **head** e **count** do *buffer* de transmissão e, quando a variável **count** atingir o valor 0 deve também desativar as interrupções de transmissão da UARTx. A ativação é efetuada na função **comDrv_putc()** sempre que se colocar um novo carater no *buffer* circular.

¹ Este é um dos 3 modos possíveis de geração de interrupção no PIC32. Para mais detalhes deve ser consultado o manual da UART.

Trabalho a realizar**Parte I**

1. Escreva, as macros de ativação e desativação das interrupções de receção e de transmissão da UART2:

```
#define DisableU2RxInterrupt() IEC1bits.U2RXIE = 0
#define EnableU2RxInterrupt() ...
#define DisableU2TxInterrupt() ...
#define EnableU2TxInterrupt() ...
```

2. Declare a estrutura que implementa um *buffer* circular e crie duas instâncias dessa estrutura: **rx** e **tx** (veja a introdução para mais detalhes). Defina a constante **BUF_SIZE** com o valor 8.
3. Escreva as funções **comDrv_flushRx()** e **comDrv_flushTx()** que inicializam os *buffers* circulares de transmissão e de receção:

```
void comDrv_flushRx(void)
{
    // Initialize variables of the reception buffer
}

void comDrv_flushTx(void)
{
    // Initialize variables of the transmission buffer
}
```

4. Escreva a função **comDrv_putc()** que escreve um carater no *buffer* de transmissão e atualiza as variáveis **tail** e **count**. Esta função deverá ainda esperar que haja espaço livre no *buffer* antes de copiar um novo carater:

```
void comDrv_putc(char ch)
{
    while(tx.b.count == BUF_SIZE){} // Wait while buffer is full
    tx.b.data[tx.b.tail] = ch;        // Copy character to the transmission
                                     // buffer at position "tail"
    tx.b.tail = (tx.b.tail + 1) & INDEX_MASK; // Increment "tail" index
                                     // (mod. BUF_SIZE)
    DisableU2TxInterrupt();           // Begin of critical section
    // Increment "count" variable
    EnableU2TxInterrupt();            // End of critical section
}
```

5. Escreva a função **comDrv_puts()** que evoca a função escrita no exercício anterior para enviar para a linha série uma *string* (terminada com o carater '\0'):

```
void comDrv_puts(char *s)
{
    (...)
}
```

6. Escreva a rotina de serviço à interrupção de transmissão da UART2.

```
// if U2TXIF is set
{
    // if "count" variable (transmission buffer, txb) is greater than 0
    {
        // Copy character pointed by "head" to U2TXREG register
        // Increment "head" variable (mod BUF_SIZE)
        // Decrement "count" variable
    }
    // if "count" variable is 0 then disable UART2 Tx interrupts
    // Reset UART2 TX interrupt flag
}
```

7. Escreva a função `main()` para testar as funções que escreveu nos pontos anteriores. Para a função `comDrv_config()` pode reaproveitar, com as devidas adaptações, o código escrito no guião prático anterior para a configuração da UART.

```
int main(void)
{
    comDrv_config(115200, 'N', 1); // config UART with default "pterm"
                                   // parameters, with TX and RX
                                   // interrupts disabled

    comDrv_flushRx();
    comDrv_flushTx();
    EnableInterrupts();
    while(1)
        comDrv_puts("Teste do bloco de transmissao do device driver!...");
}
```

8. Escreva a função `comDrv_getc()` que lê um carácter do *buffer* de receção. A função devolve o booleano **FALSE** se o número de caracteres no *buffer* for zero e **TRUE** no caso contrário. O carácter lido do *buffer* de receção deve ser passado através do ponteiro `pchar`:

```
char comDrv_getc(char *pchar)
{
    // Test "count" variable (reception buffer) and return FALSE
    // if it is zero

    // Disable UART2 Rx interrupts (begin of critical section)
    // Copy character pointed by "head" to *pchar
    // Decrement "count" variable
    // Increment "head" variable (mod BUF_SIZE)
    // Enable UART2 Rx interrupts (end of critical section)
    return TRUE;
}
```

9. Escreva a rotina de serviço à interrupção de receção da UART2.

```
// If U2RXIF is set
{
    rxb.data[rxb.tail] = U2RXREG; // Read character from UART and
                                   // write it to the "tail" position
                                   // of the reception buffer

    // Increment "tail" variable (mod BUF_SIZE)
    // If reception buffer is not full (e.g. count < BUF_SIZE) then
    // increment "count" variable
    // Else
    // increment "head" variable (discard oldest character)
    // reset UART2 RX interrupt flag
}
```

10. Re-escreva a função **main()** que escreveu no exercício 7 de modo a fazer o eco dos caracteres recebidos.

```
void main(void)
{
    comDrv_config(115200, 'N', 1); // default "pterm" parameters
                                   // with RX interrupts enabled
                                   // and TX interrupts disabled

    // (...)
    comDrv_puts("PIC32 UART Device-driver\n");
    while(1)
    {
        // Read character from reception buffer
        // Send character to the transmission buffer
    }
}
```

11. Altera a função anterior de modo a transmitir uma *string* com, pelo menos, 30 caracteres (à sua escolha) sempre que seja recebido o carater 's' (tenha em atenção a dimensão do buffer de transmissão).

Parte II

1. A rotina de serviço à interrupção implementada no exercício 6 apenas transfere para a UART um carater, cada vez que é executada. O procedimento de transmissão pode, contudo, ser melhorado, se se copiar mais do que 1 carater diminuindo, desse modo, o *overhead* resultante do processo de interrupção. Nesse sentido, altere a rotina de serviço à interrupção de transmissão de modo a copiar para a UART2 todos os caracteres do *buffer* circular de transmissão ou copiar até o FIFO da UART2 ficar cheio – a situação limite é a que ocorrer em primeiro lugar. Faça os testes que lhe permitam averiguar que as alterações efetuadas funcionam como pretendido.
2. Também a rotina de serviço à interrupção da receção pode ser melhorada, copiando para o *buffer* circular de receção mais do que um carater de cada vez que é executada. Faça as alterações a essa rotina de modo a copiar para o *buffer* de receção, até ao limite de espaço disponível, todos os caracteres disponíveis no FIFO de receção da UART.
3. Acrescente mais um campo à estrutura do *buffer* circular de modo a passar a ser possível determinar se existiu *overrun* na receção de caracteres.

Elementos de apoio

- Slides das aulas teóricas (Aula 19).
- PIC32 Family Reference Manual, Section 08 – Interrupts.
- PIC32 Family Reference Manual, Section 21 – UART.
- PIC32MX5XX/6XX/7XX, Family Data Sheet, Pág. 74 a 76.