

## Aula 19

- *Device drivers*
- Princípios gerais
- Caso de estudo: *device driver* para uma UART
- Princípio de operação e estruturas de dados
- Funções de interface com a aplicação
- Funções de serviço de interrupções e interface com o hardware

José Luís Azevedo, Arnaldo Oliveira, Tomás Silva, Bernardo Cunha

# Introdução

- O **número** de **periféricos** existentes é muito **vasto**:
  - Teclado, rato, placas (gráfica, rede, som, etc.), disco duro, *pen drive*, scanner, câmara de vídeo, etc.
- Estes periféricos apresentam características distintas:
  - **Operações suportadas**: leitura, escrita, leitura e escrita
  - **Modo de acesso**: carater, bloco, etc.
  - **Representação da informação**: ASCII, UNICODE, Little/Big Endian, etc.
  - **Largura de banda**: alguns bytes/s a MB/s
  - **Recursos utilizados**: portos (I/O, memory mapped), interrupções, DMA, etc.
  - **Implementação**: diferentes dispositivos de uma dada classe podem ser baseados em implementações distintas (e.g. diferentes fabricantes, diferentes modelos) com reflexos profundos na sua operação interna

# Princípios gerais - classes

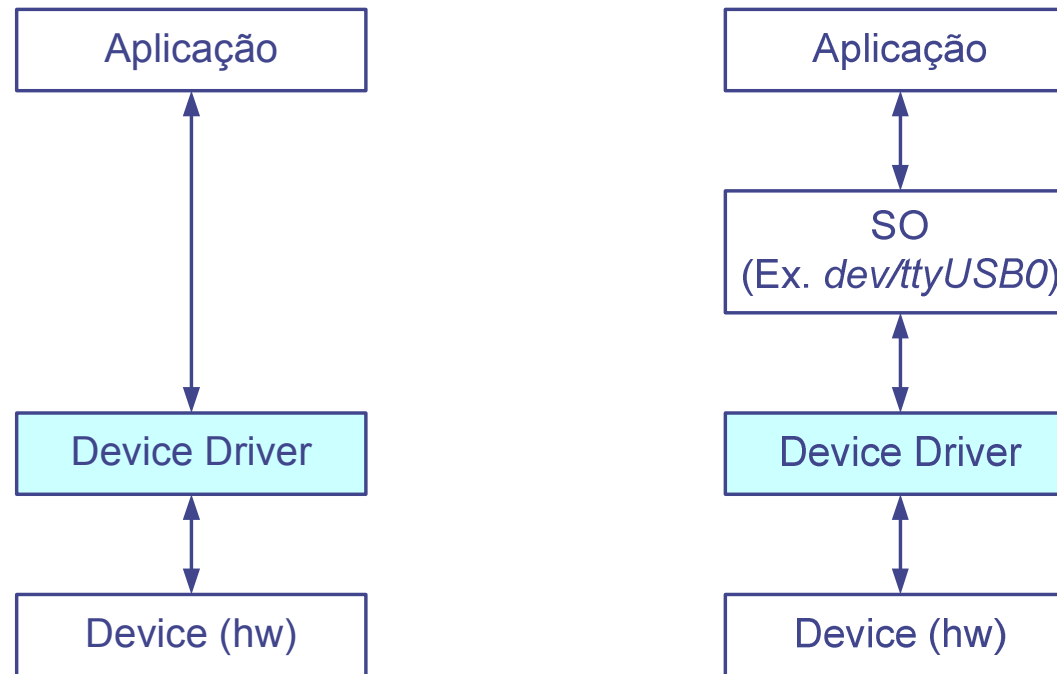
Class	Usage	Description	Examples, or exception
00h	Device	Unspecified <sup>[42]</sup>	Device class is unspecified, interface descriptors are used to determine needed drivers
01h	Interface	Audio	Speaker, microphone, sound card, MIDI
02h	Both	Communications and CDC Control	Modem, Ethernet adapter, Wi-Fi adapter, RS232 serial adapter. Used together with class 0Ah ( <i>below</i> )
03h	Interface	Human interface device (HID)	Keyboard, mouse, joystick
05h	Interface	Physical Interface Device (PID)	Force feedback joystick
06h	Interface	Image (PTP/MTP)	Webcam, scanner
07h	Interface	Printer	Laser printer, inkjet printer, CNC machine
08h	Interface	Mass storage (MSC or UMS)	USB flash drive, memory card reader, digital audio player, digital camera, external drive
09h	Device	USB hub	Full bandwidth hub
0Ah	Interface	CDC-Data	Used together with class 02h ( <i>above</i> )
0Bh	Interface	Smart Card	USB smart card reader
0Dh	Interface	Content security	Fingerprint reader
0Eh	Interface	Video	Webcam
0Fh	Interface	Personal healthcare device class (PHDC)	Pulse monitor (watch)
10h	Interface	Audio/Video (AV)	Webcam, TV
11h	Device	Billboard	Describes USB Type-C alternate modes supported by device
DCh	Both	Diagnostic Device	USB compliance testing device
E0h	Interface	Wireless Controller	Bluetooth adapter, Microsoft RNDIS
EFh	Both	Miscellaneous	ActiveSync device
FEh	Interface	Application-specific	IrDA Bridge, Test & Measurement Class (USBTMC), <sup>[43]</sup> USB DFU (Device Firmware Upgrade) <sup>[44]</sup>
FFh	Both	Vendor-specific	Indicates that a device needs vendor-specific drivers

# Introdução

- As aplicações/Sistemas Operativos (SO) **não podem conhecer todos os tipos de dispositivos** passados, atuais e futuros com um nível de detalhe suficiente para realizar o seu controlo a baixo nível!
- **Solução:** Criar uma camada de abstração que permita o acesso ao dispositivo de forma independente da sua implementação
- *Device driver*
  - Um programa que permite a outro programa (aplicação, SO) interagir com um dado dispositivo de hardware
  - Implementa a camada de abstração e lida com as particularidades do dispositivo controlado
- Aspetos-chave:
  - Abstração, uniformização de acesso, independência entre aplicações/SO e o hardware

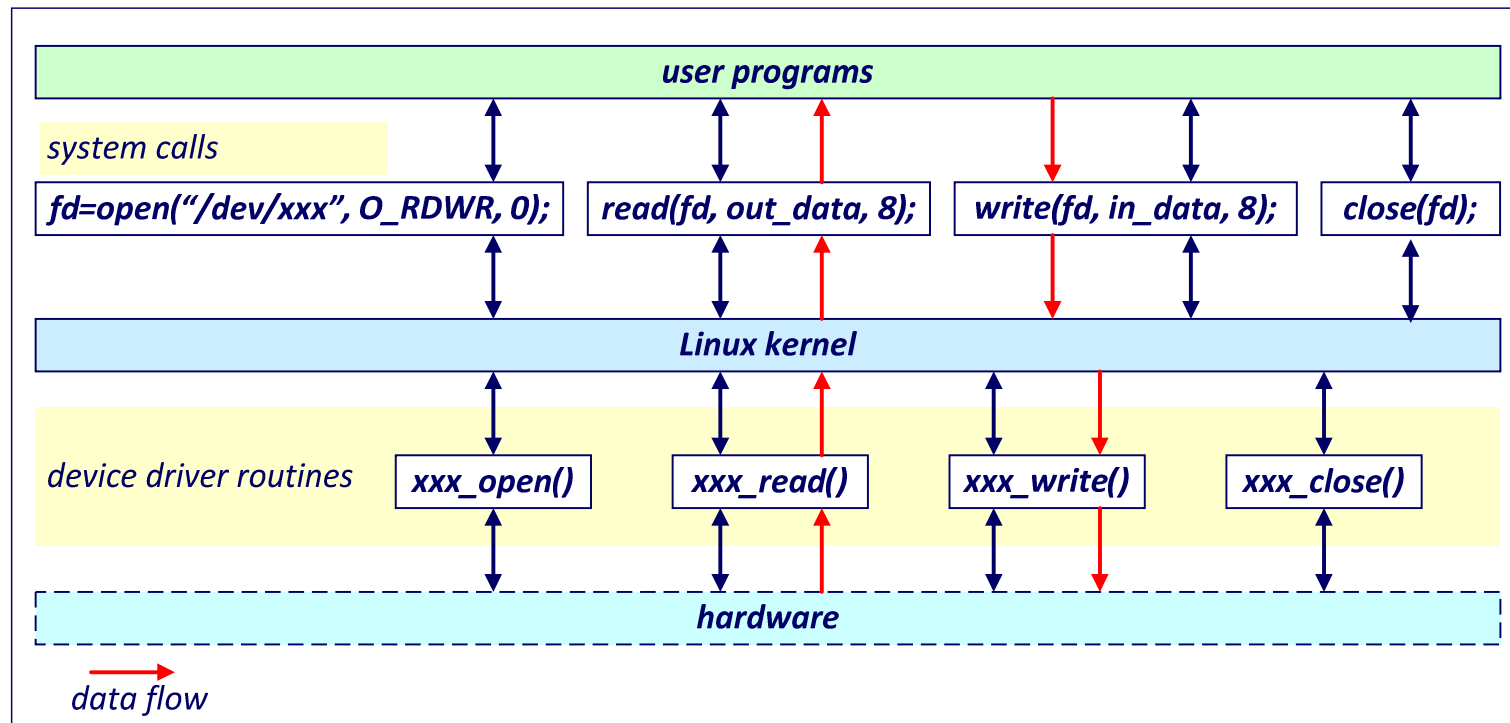
# Princípios gerais

- O acesso, por parte das aplicações, a um *device driver* é diferente num sistema embutido e num sistema computacional de uso geral (com um Sistema Operativo típico, e.g. Linux, Windows, Mac OS):
  - Aplicações em sistemas embutidos acedem, tipicamente, de forma direta aos *device drivers*
  - Aplicações que correm sobre SO acedem a funções do SO (*system calls*); o kernel do SO, por sua vez, acede aos *device drivers*



# Princípios gerais

- O Sistema Operativo especifica classes de dispositivos e, para cada classe, uma interface que estabelece como é realizado o acesso a esses dispositivos
  - A função do *device driver* é traduzir as chamadas realizadas pela aplicação/SO em ações específicas do dispositivo
  - Exemplos de classes de dispositivos: interface com o utilizador, armazenamento de massa, comunicação, ...



# Exemplo de um *device driver*: comunicação série

- Admitindo que é fornecida uma biblioteca que apresenta a seguinte interface:

- `void comDrv_init(int baudrate, char dataBits,  
char parity, char stopBits);`
- `char comDrv_getc(void);` // read a character
- `void comDrv_putc(char ch);` // write a character
- `void comDrv_close(void);`

- **Do ponto de vista da aplicação:**

- Do ponto de vista funcional é relevante qual o modelo/fabricante do dispositivo de comunicação série?
- Se o dispositivo de comunicação for substituído por outro com arquitetura interna distinta, sendo fornecida uma biblioteca com interface compatível, é necessário alterar a aplicação?

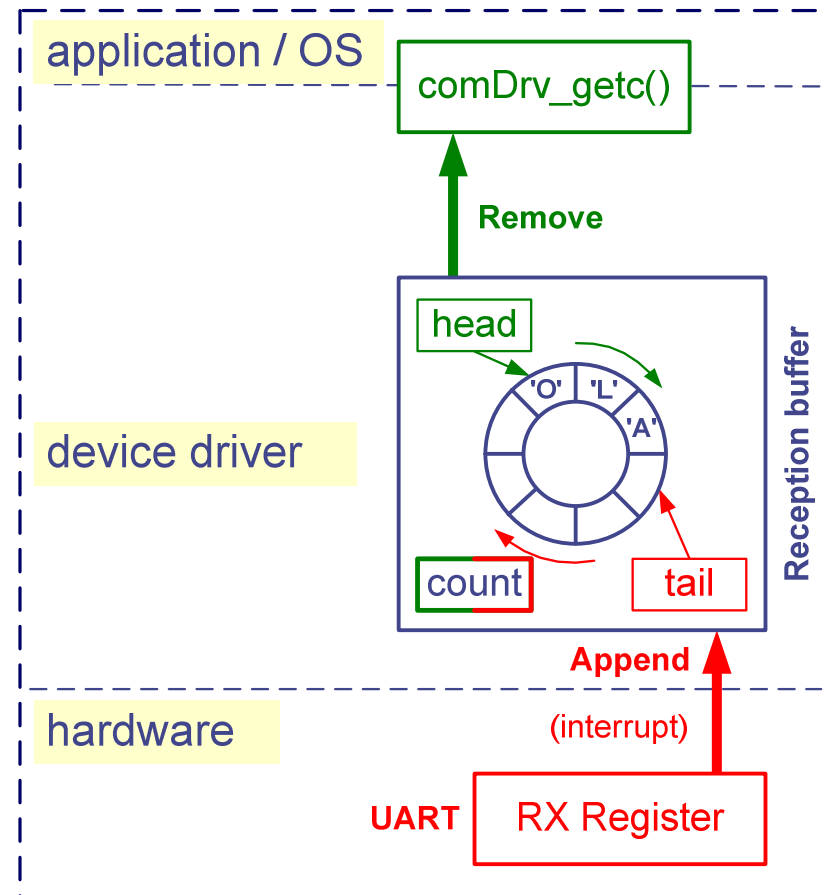
# Caso de estudo

- Realização de um *device driver* para uma UART RS232 (*Universal Asynchronous Receiver Transmitter*) para executar num sistema com microcontrolador (i.e., sem sistema operativo)
- Princípio de operação
  - Desacoplamento da transferência de dados entre a UART e a aplicação realizada por meio de FIFOs (um FIFO de transmissão e um de receção). Do ponto de vista da aplicação:
    - A **transmissão** consiste em copiar os dados a enviar para o FIFO de transmissão do *device driver*
    - A **receção** consiste em ler os dados recebidos que residem no FIFO de receção do *device driver*
  - A transferência de dados entre os FIFOs e a UART é realizada por interrupção, i.e., sem intervenção explícita da aplicação
  - Um FIFO pode ser implementado através de um *buffer* circular



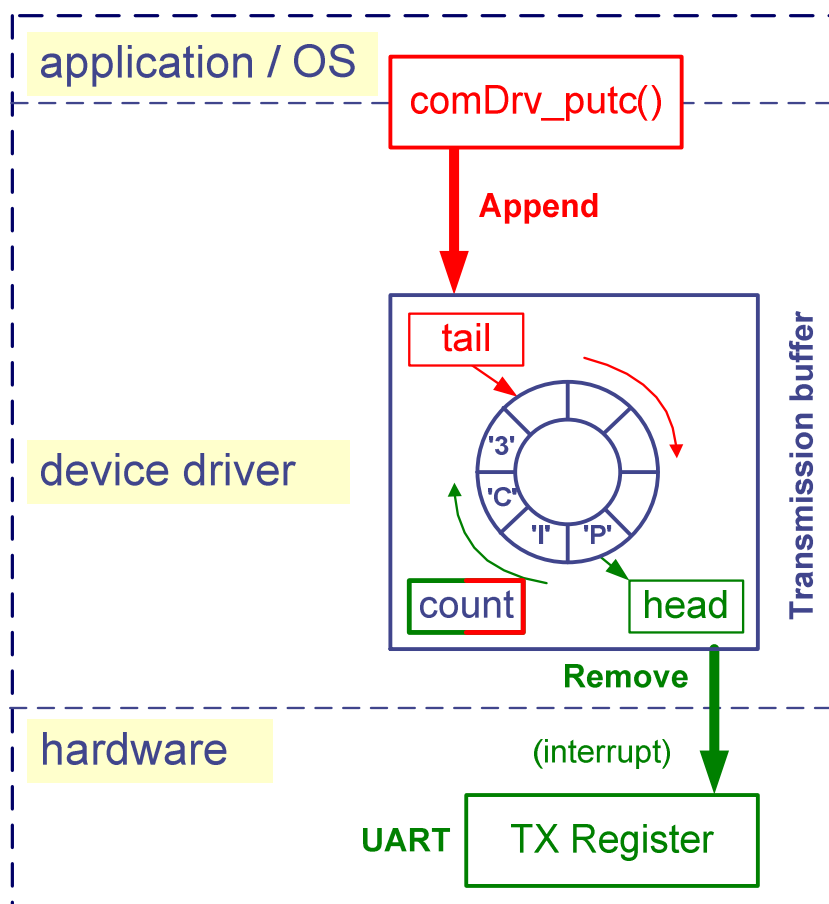
# Princípio de operação – receção

- **"tail"** – posição do buffer circular onde a **rotina de serviço à interrupção escreve** o próximo carácter lido da UART
- **"head"** – posição do buffer circular de onde a função **comDrv\_getc()** lê o próximo carácter
- **"count"** – número de caracteres residentes no buffer circular (ainda não lidos pela aplicação)
- **A variável "count" é um "recurso partilhado". Porquê?**



# Princípio de operação – transmissão

- **"tail"** – posição do buffer circular onde a função **comDrv\_putc()** escreve o próximo caracter
- **"head"** – posição do buffer circular de onde a **rotina de serviço à interrupção** lê o próximo caracter a enviar para a UART
- **"count"** – número de caracteres residentes no buffer circular (ainda não enviados para a UART)
- **A variável "count" é um "recurso partilhado". Porquê?**



# Implementação – FIFO

- FIFO - Buffer circular implementado através de um *array* linear:

```
#define BUF_SIZE 32

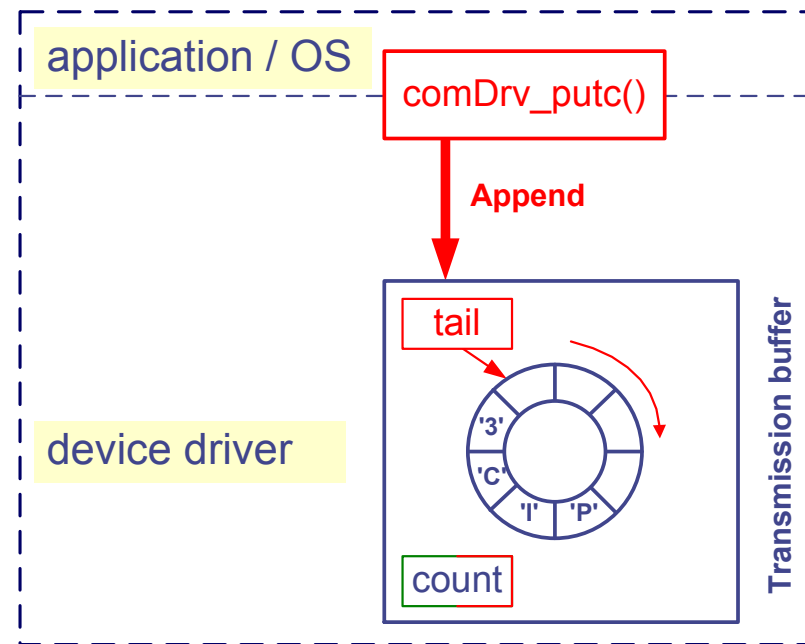
typedef struct
{
    unsigned char data[BUF_SIZE];
    unsigned int head;
    unsigned int tail;
    unsigned int count;
} circularBuffer;

circularBuffer txb; // Transmission buffer
circularBuffer rxb; // Reception buffer
```

- A constante "BUF\_SIZE" deve ser definida em função das necessidades previsíveis de pico de tráfego.
- Se "BUF\_SIZE" for uma potência de 2 simplifica a atualização dos índices do buffer circular (podem ser encarados como contadores módulo  $2^N$  e podem ser geridos com uma simples máscara)

# Implementação – Função de transmissão

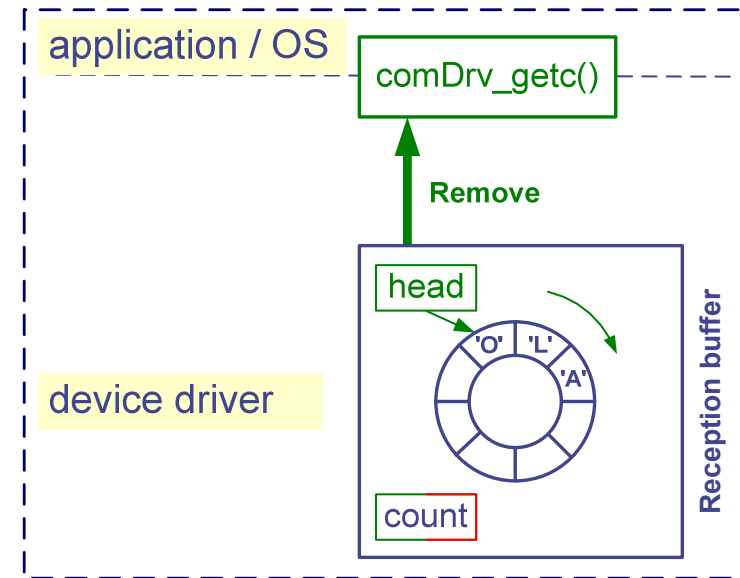
- A função de transmissão, evocada pela aplicação, copia o caracter para o **buffer de transmissão** (posição "**tail**"), incrementa o índice "**tail**" e o contador



```
void comDrv_putc(char ch)
{
    Wait while buffer is full (txb.count==BUF_SIZE)
    Copy "ch" to the buffer ("tail" position)
    Increment "tail" index (mod BUF_SIZE)
    Increment "count" variable
}
```

# Implementação – Função de receção

- A função de receção, evocada pela aplicação, verifica se há caracteres no **buffer de receção** para serem lidos e, caso haja, retorna o caracter presente na posição "**head**", incrementa o índice "**head**" e decrementa o contador



```
int comDrv_getc(char *pchar)
{
    If "count" variable is 0 then return FALSE
    Copy character at position "head" to *pchar
    Increment "head" index (mod BUF_SIZE)
    Decrement "count" variable
    return true;
}
```

# Implementação – RSI de receção

- A rotina de serviço à interrupção da receção é executada sempre que a UART recebe um novo caracter
- O caracter recebido pela UART deve então ser copiado para o **buffer de receção**, na posição "**tail**"; a variável "**count**" deve ser incrementada e o índice "**tail**" deve ser igualmente incrementado

```
void interrupt isr_rx(void)
```

```
{
```

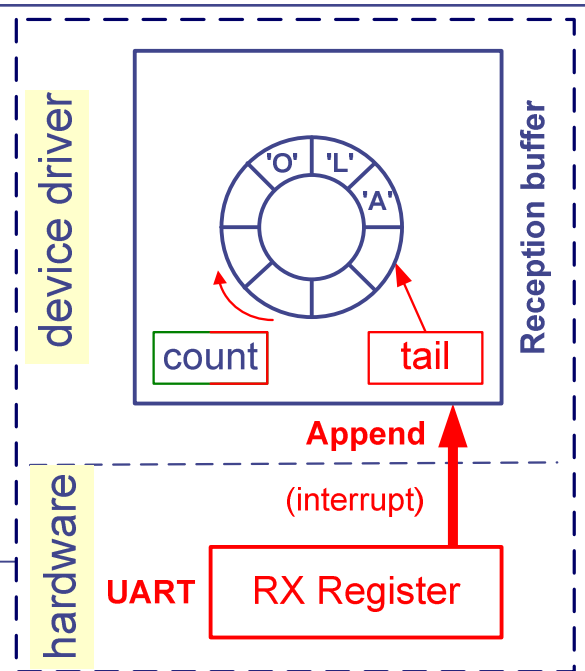
```
    Copy received character from UART to RX
```

```
        buffer ("tail" position)
```

```
    Increment "tail" index (mod BUF_SIZE)
```

```
    Increment "count" variable
```

```
}
```

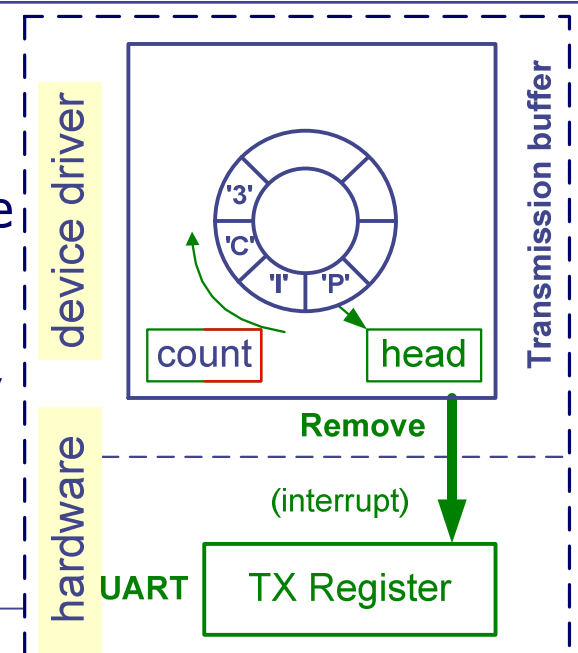


O que acontece no caso em que o *buffer* circular está cheio e a UART recebe um novo caracter? Como resolver esse problema?

# Implementação – RSI de transmissão

- A UART gera, normalmente, uma interrupção de transmissão quando tiver disponibilidade para transmitir um novo caracter
- As tarefas a implementar na respetiva rotina de serviço à interrupção são:
  - Se o número de caracteres no **buffer de transmissão** for maior que 0 ("count" > 0), copiar o conteúdo do buffer na posição "head" para a UART
  - Decrementar a variável "count" e incrementar o índice "head"

```
void interrupt isr_tx(void)
{
    If "count" > 0 then {
        Copy character from TX buffer ("head") to UART
        Increment "head" index (mod BUF_SIZE)
        Decrement "count" variable
    }
    If "count" == 0 then disable UART TX interrupts
}
```



# Atualização do TX "count" - Secção crítica

```
void comDrv_putc(char ch)
{
    Wait while buffer is full (count==BUF_SIZE)
    Copy "ch" to the transmission buffer ("tail")
    Increment "tail" index (mod BUF_SIZE)
    Disable UART TX interrupts
    Increment "count" variable
    Enable UART TX interrupts
}

void interrupt isr_tx(void)
{
    if "count" > 0 then
    {
        Copy character from TX buffer ("head") to UART
        Increment "head" index (mod BUF_SIZE)
        Decrement "count" variable
    }
    if "count" == 0 then disable UART TX interrupts
}
```

**Secção crítica**  
("count" é um recurso partilhado)

Desencadeia a imediata geração da interrupção de transmissão



# Atualização do RX "count" - Secção crítica

```
int comDrv_getc(char *pchar)
```

```
{
```

```
    If "count" variable is 0 then return FALSE
```

```
    Copy character at position "head" to *pchar
```

```
    Increment "head" index (mod BUF_SIZE)
```

```
    Disable UART RX interrupts
```

```
    Decrement "count" variable
```

```
    Enable UART RX interrupts
```

```
    return true;
```

```
}
```

**Secção crítica**  
("count" é um  
recurso partilhado)

```
void interrupt isr_rx(void)
```

```
{
```

```
    Copy received character from UART to RX buffer  
    ("tail" position)
```

```
    Increment "tail" index (mod BUF_SIZE)
```

```
    Increment "count" variable
```

```
}
```